Freie Universität Berlin

Diplomarbeit zur Erlangung des akademischen Grades
Diplom-Informatiker

Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung

Riad Djemili

Matrikelnummer: 3419442

inf@riad.de

Eingereicht bei:

Prof. Dr. Lutz Prechelt

Betreuer:

Christopher Özbek

Stephan Salinger

Berlin, 31. August 2006

Danksagung

Ich danke meinen Eltern, die mich in den Jahren meines Studiums immer unterstützt haben. Ihnen widme ich diese Arbeit.

Ich danke meinen Betreuern Christopher Özbek und Stephan Salinger für die sehr kompetente und freundliche Unterstützung und dafür, dass sie gegenüber meinen Ideen immer aufgeschlossen waren.

Ich danke Daniel Nowak und Thorsten Wenzlaff dafür sich so kurzfristig als Testkandidaten bereitgestellt zu haben. (Wir haben's geschafft!)

Ich versichere, dass ich die vorliegende Arbeit mit dem Titel „Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung“ selbstständig verfasst habe. Alle Stellen der Arbeit, die anderen Werken wörtlich oder sinngemäß entnommen sind, sind unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Berlin, den 31. August 2006
Riad Djemili

Kurzzusammenfassung

Paarprogrammierung bezeichnet eine Arbeitstechnik, bei der zwei Programmierer an einen Computer und in enger Zusammenarbeit an einem Artefakt arbeiten. Ziele dieser Technik sind unter anderem höhere Softwarequalität, schnellere Time-to-Market und verbesserte Kommunikation.

Bei verteilter Paarprogrammierung arbeiten die Entwickler in gleicher Weise an einem Artefakt, halten sich dabei jedoch an verschiedenen Arbeitsplätzen auf. Vorteile gegenüber regulärer Paarprogrammierung sind unter anderem die Unterstützung virtueller Teams und komfortablere Arbeitsplätze.

In dieser Diplomarbeit erläutere ich diese Vor- und Nachteile regulärer beziehungsweise verteilter Paarprogrammierung und vergleiche verschiedene technische Ansätze für die Realisierung von verteilter Paarprogrammierung. Meine Erläuterungen werden durch eine Umfrage, welche ich im Rahmen dieser Arbeit durchgeführt habe, ergänzt.

Der namensgebende Teil dieser Arbeit befasst sich mit der Entwicklung von *Saros*, einem Eclipse-Plugin für verteilte Paarprogrammierung. Auf Grundlage meiner vorhergehenden Erläuterungen und der Umfrage beschreibe ich meine erarbeiteten Anforderungen sowie Entwurfsentscheidungen und gebe einen Überblick über die wichtigsten Aspekte der Implementierung. Ein separates Plugin erweitert darüber hinaus den Eclipse-Sensors des Mikroprozess-Analyse-Systems ElectroCodeoGram um Mikroprozesse der Paarprogrammierung.

Das im Rahmen dieser Diplomarbeit entstandene Plugin kann unter der URL `http://sourceforge.net/projects/dpp` bezogen werden und unterliegt der GPL-Lizenz.

Inhaltsverzeichnis

1. Einleitung **11**

2. Paarprogrammierung **15**
2.1. Vorteile . . . 15
2.2. Nachteile . . . 19
2.3. Praxis . . . 21
2.4. Herausforderungen der Evaluation . . . 22

3. Verteilte Paarprogrammierung **25**
3.1. Konzept . . . 25
3.1.1. Vorteile . . . 25
3.1.2. Nachteile . . . 28
3.2. Technische Werkzeugansätze . . . 29
3.2.1. Desktop Sharing . . . 30
3.2.2. Collaboration Awareness . . . 30
3.3. Vergleich der Ansätze . . . 31
3.3.1. Übersetzbarkeit . . . 31
3.3.2. Flexibilität . . . 33
3.3.3. Technische Restriktionen . . . 35
3.3.4. Sicherheit . . . 36
3.3.5. Parallelität . . . 37
3.4. Fazit . . . 38

4. Umfrage **39**
4.1. Ankündigung . . . 39
4.2. Auswertung . . . 41
4.2.1. Demografische Daten . . . 41
4.2.2. Allgemeine Arbeitsweise . . . 43
4.2.3. Erfahrung mit verteilter Paarprogrammierung . . . 45
4.2.4. Anforderungen an DPP-Werkzeuge . . . 51
4.2.5. Abschluss und freie Anmerkungen . . . 57
4.3. Bewertung . . . 61

5. Entwurf und Implementierung eines DPP-Werkzeugs **63**
5.1. Entwurfsentscheidungen . . . 63
5.1.1. Kontaktliste . . . 63
5.1.2. Übersetzbarkeit . . . 64
5.1.3. Awareness . . . 66
5.1.4. Parallelität und Verfolgermodus . . . 67

5.2. Technologien 68
5.2.1. Java 68
5.2.2. Eclipse 69
5.2.3. Extensible Messaging and Presence Protocol 70
5.3. Implementierung 72
5.3.1. Architektur 73
5.3.2. Abfangen und Reproduzieren von Aktivitäten 74
5.3.3. Einladungen 83
5.3.4. Relevante Ressourcen 85
5.3.5. Kontaktliste 87
5.3.6. Awareness 88
5.3.7. Nutzung des Fassaden-Entwurfsmusters 90

6. Erweiterung des ECG-Eclipse-Sensors 93
6.1. Motivation 93
6.2. Actual Process und ElectroCodeoGram 94
6.3. Erweiterung um paarprogrammierspezifische Mikroprozesse . . . 95

7. Evaluation 97

8. Fazit 101
8.1. Zusammenfassung 101
8.2. Ausblick 102
8.3. Schlusswort 103

Literaturverzeichnis 105

A. Ankündigung und Fragenkatalog der Umfrage 111
A.1. Ankündigung 111
A.2. Fragenkatalog 112
A.2.1. Einleitung 112
A.2.2. Fragen 112

B. Anforderungen 117
B.1. Funktionale Anforderungen 117
B.2. Nicht funktionale Anforderungen 118

C. Anwendungsfälle 119
C.1. Veröffentlichen eines Projekts 119
C.2. Einladen von Teilnehmern 120
C.3. Übertragen von Textänderungen 121
C.4. Übertragen des aktiven Editors 122

D. Anleitungen 123
D.1. Benutzeranleitung 123
D.2. Entwickleranleitung 128

E. Schemata der ECG-Mikroprozesse 131

Abbildungsverzeichnis

4.1. Umfrage - Kürzlich verwendete Programmiersprachen 41
4.2. Umfrage - Programmier-Wochenstunden 42
4.3. Umfrage - Jahre an Programmiererfahrung 42
4.4. Umfrage - Akademische Abschlüsse 42
4.5. Umfrage - Berufe 43
4.6. Umfrage - Verwendete Versionskontrollsysteme 44
4.7. Umfrage - Anteil von PP an gesamter Programmiertätigkeit . . . 44
4.8. Umfrage - Wahrscheinlichkeit, dass vor Beginn einer PP-Sitzung eine gemeinsame Dateibasis vorliegt 45
4.9. Umfrage - Kommunikationsmittel zum Einleiten von DPP 48
4.10. Umfrage - Einsatzzwecke für verteilte Paarprogrammierung . . . 49
4.11. Umfrage - Art der Projekte 49
4.12. Umfrage - Räumliche Distanz bei DPP-Sitzungen 50
4.13. Umfrage - Durchschnittliche Teilnehmerzahl in DPP-Sitzung . . 51
4.14. Umfrage - Gewünschter Anteil von DPP an gesamter Programmiertätigkeit 57
4.15. Umfrage - Art der Kenntnisnahme 58

5.1. Server-zu-Server-Kommunikation 71
5.2. Architektur 73
5.3. Klassendiagramm der zentralen Klassen 74
5.4. Sequenzdiagramm zum Erzeugen und Senden von Aktivitäten . . 77
5.5. Sequenzdiagramm zum Empfangen und Ausführen von Aktivitäten 78
5.6. Klassendiagramm aller Aktivitäten 79
5.7. Klassendiagramm zu Einladungen 86
5.8. Kontaktliste 87
5.9. Assistenten für neue Benutzerkonten und Kontakte. 88
5.10. Arbeitsoberfläche 89

6.1. ECG Lab mit drei angeschlossenen Sensoren. 95

7.1. Sitzanordnung bei der Versuchssitzung. 97

Tabellenverzeichnis

3.1. Raum-Zeit-Taxonomie für Groupware nach Ellis et al. [EGR91]. . 29
3.3. Vor- und Nachteile von DPP-Werkzeugen 38

4.2. Umfrage - Fragenblock Kommunikationsmittel 45
4.3. Umfrage - Fragenblock zu Kommunikationsmitteln. 52
4.4. Umfrage - Fragenblock zu Driver-Observer-Interaktionen. 53
4.5. Umfrage - Fragenblock zu Awareness. 55
4.6. Umfrage - Fragenblock zur Funktionsvielfalt. 56

1. Einleitung

Williams et al. [WKCJ00] definieren Paarprogrammierung (engl. *pair programming*, kurz *PP*) als:

> Pair programming - two programmers working side-by-side at one computer collaborating on the same design, algorithm, code or test ...

Nach Williams et al. nehmen beide Programmierer dabei unterschiedliche Rollen ein. Eine Person bearbeitet das Artefakt aktiv und kontrolliert die Maus und Tastatur bzw. den Bleistift. Diese Rolle wird üblicherweise als *Pilot* oder *Driver* bezeichnet und entspricht weitgehend der des klassischen Einzelprogrammierers: Der Driver bedient den Computer bzw. schreibt das Dokument.

Ihm zur Seite steht der *Navigator*, auch *Observer* oder *Kopilot* genannt. Der Navigator hat keine aktive Kontrolle über das Artefakt, aber betrachtet die Aktionen des Drivers. Er überprüft die unmittelbaren Eingaben des Drivers auf Defekte, wie Syntaxfehler oder falsche Operationsaufrufe, hat aber gleichzeitig auch eine übergeordnete Sichtweise, die ihm erlaubt mehr über die allgemeine Problemdomäne und grundlegendere Entwurfsentscheidungen nachzudenken. Die Rollenverteilung innerhalb einer derartigen Sitzung muss nicht festgelegt sein; es können situative Wechsel zwischen den Rollen stattfinden.

Im Gegensatz zur klassischen Paarprogrammierung, bei dem die Gegebenheiten weitgehend deutlich definiert sind, ist die verteilte Paarprogrammierung (engl. *distributed pair programming*, kurz *DPP*) weniger klar definiert. Handelt es sich bereits um verteilte Paarprogrammierung, wenn zwei Entwickler sich während der Programmierung über einen Chat unterhalten und dabei Dateien hin und her schicken? Wohl kaum. Handelt es sich um DPP, wenn zwei Entwickler mit einem speziellen Texteditor arbeiten, der die Änderungen an der Datei auf dem System des anderen Nutzers repliziert? Womöglich ja oder müssen Artefakte wie beispielsweise kompilierte Programme ausführbar sein? Sind Gestiken und Gesichtsausdrücke per Videoübertragung notwendig für eine gute Kollaboration?

Man hat es hier also eher mit unterschiedlichen Graden eines Spektrums verteilter Paarprogrammierung zu tun. Eine allgemeine Definition, auf welche ich mich folgend stützen möchte, liefern Schröder und Schröder [SS01]:

> During a distributed pair programming session, programmers must be able to communicate; they have to see the same part of code not on one shared screen but on their individual screens. At least one of them has to be able to change this code. As both pair members focus on the code they work on and communicate verbally, they must be able to hear but not necessarily to see each other.

Stotts et al. [SWN+03] liefern eine Definition, die eher technisch orientiert ist:

> By dPP we mean that two members of the team (which may consist solely of these two people) synchronously collaborate on the same design or code from different locations. This means that both must view a copy of the same screen, and at least one them should have the capability to change the contents on the screen. To be able to do this, they require technological support for sharing desktops and verbal conversation, and perhaps even of video conferencing capabilities.

Da letztere Definition stärker auf *Desktop-Sharing*-Ansätze abzielt (siehe Abschnitt 3.2.1), habe ich mich im Rahmen dieser Arbeit für die zuerst genannte Definition entschieden. Insbesondere gilt dabei, folgende Feststellung zu beachten: Während die Definition von Stotts et al. explizit eine räumliche Verteilung fordert (wenn auch nicht ganz klar ist, ab wann diese gegeben ist), stellt die Definition von Schröder und Schröder nur die Forderung nach unterschiedlichen Arbeitsplätzen. Damit ist ein Szenario mit zwei Entwicklern, die von benachbarten Arbeitsplätzen aus in einer gemeinsamen verteilten Paarprogrammier-Sitzung arbeiten, ebenfalls Teil der Definition (siehe dazu auch Abschnitt 3.1.1).

Ein Aspekt, der von keiner der beiden Definitionen explizit erwähnt wird, ist der der Wechsel der Driver-Rolle zwischen den Teilnehmern. Ich denke, dass der situative und schnelle Wechsel der Rollenverteilung, ebenfalls als elementarer Aspekt verteilter Paarprogrammierung behandelt werden sollte und daher auch ein expliziter Teil der Definition sein sollte.

Diese Arbeit behandelt den Entwurf und die Implementierung eines Eclipse-Plugins für verteilte Paarprogrammierung. Die Basis für meine dabei erarbeiteten Anforderungen und Entwurfsentscheidungen bilden umfassende Erläuterungen zur regulären und verteilten Paarprogrammierung sowie eine Umfrage, die ich im Rahmen dieser Diplomarbeit ebenfalls durchgeführt habe. Dieses Dokuments ist folgendermaßen aufgebaut:

- Kapitel 2 behandelt reguläre Paarprogrammierung. Ich gehe hier auf Stärken und Schwächen der Arbeitstechnik ein und charakterisiere die Verbreitung von Paarprogrammierung auch als Teil von *Extreme Programming*.

- Kapitel 3 erläutert die grundlegende Motivation für verteilte Paarpro-

grammierung. Ich beschreibe die beiden grundlegenden technischen Ansätze - *Desktop Sharing* und *Collaboration Awareness* - und vergleiche sie anhand verschiedener Kriterien.

- Kapitel 4 ergänzt meine Ausführungen zur verteilten Paarprogrammierung mit einer von mir durchgeführten Umfrage. Ihre zentrale Forschungsfrage lautet: „In welcher Weise setzen Nutzer verteilte Paarprogrammierung ein und welche Funktionen eines DPP-Werkzeugs werden dabei als wichtig erachtet.“.

- Kapitel 5 behandelt den Entwurf und die Implementierung eines DPP-Werkzeugs als Eclipse-Plugin. Der Name des Werkzeugs lautet *Saros*. Ich erläutere die Anforderungen, Entwurfsentscheidungen sowie die wichtigsten Details der Implementierung.

- Kapitel 6 beschreibt das *ElectroCodeoGram*-System (ECG), welches zur Protokollierung von Mikroprozessen dient. Teil dieser Diplomarbeit ist auch eine Erweiterung des Eclipse-ECG-Sensors um neue Mikroprozesse. Die Implementierung dieser Funktionalität wird in diesem Kapitel beschrieben.

- Kapitel 7 stellt eine Testsitzung dar, die ich im Zuge einer ersten Evaluation der Software vollzogen habe.

- Kapitel 8 enthält ein zusammenfassendes Fazit sowie einen Ausblick über zukünftige Weiterentwicklungen.

2. Paarprogrammierung

Jegliche Betrachtung verteilter Paarprogrammierung muss auf einem Verständnis von regulärer Paarprogrammierung fußen. Ohne dieses Wissen ist es unmöglich zu verstehen, in welchen Aspekten reguläre und verteilte Paarprogrammierung voneinander abweichen und wie diese Unterschiede zu bewerten sind. Dies ist wichtig, da andernfalls die Gefahr besteht, sich mit seinem Ansatz zur verteilten Paarprogrammierung zu weit von den Kernprinzipien und Vorteilen der Paarprogrammierung zu entfernen.

Das folgende Kapitel befasst sich mit Stärken und Schwächen regulärer Paarprogrammierung. Ich erläutere, wie weit Paarprogrammierung in der Praxis verbreitet ist, und beschreibe Herausforderungen bei der Evaluation von Paarprogrammierung.

2.1. Vorteile

In der Literatur wird Paarprogrammierung vor allem eine Verbesserung der Faktoren Softwarequalität, Time-to-Market und Kommunikation nachgesagt. Diese Gesichtspunkte sind dabei stark miteinander verwoben. Im Folgenden versuche ich die einzelnen Stärken dezidiert darzulegen:

Weniger Defekte Eine Technik zur Verbesserung der Codequalität ist die sogenannte Durchsicht (engl. *inspection* oder *review*). Durchsichten gehören zu den statischen Verfahren der analytischen Qualitätssicherung. Wie alle Maßnahmen der analytischen Qualitätssicherung zielen sie darauf ab, mittels prüfender Maßnahmen entstandene Mängel nachwirkend aufzudecken und zu korrigieren.

Durchsichten sind dabei ein manuelles Verfahren, bei dem die Artefakte von Menschen auf Mängel untersucht werden. Wie dies genau geschieht, kann sich in mehreren Freiheitsgraden, zum Beispiel in der Anzahl der Beteiligten, dem Formalisierungsgrad oder der Lesetechnik, unterscheiden. Üblicherweise findet die Durchsicht in bestimmten Abständen, beispielsweise nach bestimmten Zeiteinheiten oder Programmiterationen, statt und wird von einer anderen Person als dem ursprünglichen Autor durchgeführt.

Untersuchungen haben gezeigt, dass Durchsichten ein wirkungsvolles Mittel zur Verbesserung der Codequalität sein können [Lai02]. Ihre größte Stärke liegt in der Tatsache, dass sie Defekte in frühen Phasen des Entwicklungsprozesses finden können. Dadurch, dass sie auf alle möglichen Artefakte angewendet werden können, eignen sie sich beispielsweise auch zur Überprüfung von Anwendungsfällen. Je früher ein Defekt gefunden wird, als desto billiger gilt seine Behebung [Hum97].

Williams et al. [WKCJ00] verstehen die Paarprogrammierung dabei als eine Art sofortige und stetige Durchsicht. Da bei der Paarprogrammierung schon die direkte Erstellung und Bearbeitung eines Artefakts unter der Aufsicht einer zweiten Person geschieht, hat man es hier mit einer extremen Form der Durchsicht zu tun, womit eine verbesserte Qualität in Aussicht gestellt werden soll. Defekte sollen leichter erkannt und die Defektrate gesenkt werden.

Williams et al. beobachteten diesen Effekt bei einem Versuch mit Studenten. Die Programme der 14 Paare, welche Paarprogrammierung einsetzten, bestanden im Schnitt 14% mehr Testfälle als die der Kontrollgruppe mit 13 Einzelprogrammierern. Bei einem Test mit professionellen Programmierern kam Nosek [Nos98] zu einem vergleichbaren Resultat. Die Programme von fünf Paaren schnitten in der Metrik *Funktionalität*[1] im Vergleich zu denen der Kontrollgruppe mit fünf Einzelprogrammierern um circa 33% besser ab.

Williams und Kessler [WK00] vermuten darüber hinaus, dass Paarprogrammierung auch mit Durchsichten assoziierte Abneigungen mindern würde, da die Defektaufspürung innerhalb des Paars geschieht und nicht mehr durch eine „außenstehende“ Person offenbart wird.

Besserer Entwurf Während der Zweck von Durchsichten hauptsächlich das Aufspüren von Defekten ist, soll Paarprogrammierung darüber hinaus auch zu einem insgesamt besseren Entwurf führen. Gemeint ist damit, dass zwei Entwickler gemeinsam eher zu einem guten Entwurf gelangen sollen als Einzelprogrammierer. Mehr und unterschiedlichere Ansätze sollen in der Diskussion besser durchdacht werden.

Dieser Aspekt ist in Versuchen nur schwer nachzuweisen und wird daher oft eher argumentativ und anekdotenhaft begründet. Constantine [Con95] ist einer der Ersten, der diesen Punkt nach Beobachtung von Paaren anführte. Williams und Kessler [WK00] argumentieren ebenfalls ausgehend von ihren persönlichen Erfahrungen:

> Experiences show that, together, a pair will come up with more

[1] Die Untersuchung unterschied die Leistungsmetriken READIBILTY, FUNCTIONALITY, SCORE und TIME sowie die Zufriedenstellungsmetriken CONFID und FUN.

> than twice as many possible solutions that the two would have working alone. They will then proceed to more quickly narrow in on the „best“ solution and will implement it more quickly and with better quality.

Schnellere Time-to-Market Brooks' Gesetz [Bro95] besagt: „Adding manpower to a late software project makes it later“. Gründe dafür sind ein erhöhter Kommunikationsbedarf mit jedem Mitarbeiter und der Einarbeitungsaufwand in das bereits fortgeschrittene Projekt, durch den die produktive Leistung bisheriger Mitarbeiter geschmälert wird.

Nosek [Nos98] weißt darauf hin, dass Paarprogrammierung Freiräume für das Hinzufügen von neuen Programmierern schaffen soll. Neue Programmierer könnten eher an den kritischen Modulen arbeiten, anstatt wie üblich neu geschaffenen Teilaufgaben zugeordnet zu werden. Geplante Aufgaben müssten nicht feiner zerlegt werden, so könne die *Time-to-Market* wirksam reduziert werden.

Problemlösung Cockburn und Williams [CW00] postulieren auch eine verbesserte Problemlösung. Mit Problem sei eine Situation gemeint, in der ein unerwarteter Fehler oder ein komplexes Entwurfsproblem auftritt. Cockburn und William schöpfen hier den Begriff des „pair relaying“. Damit beschreiben sie einen Prozess, bei dem zwei Entwickler abwechselnd die Initiative übernehmen und abwechselnd die Problemlösung vorantreiben.

Meiner Meinung nach setzt sich das pair-relaying aus zwei Aspekten zusammen. Zum einem motivieren sich die Partner eines Paars gegenseitig. Auf diesen Aspekt gehe ich im nächsten Punkt weiter ein. Zum Anderen könnte die verbesserte Problemlösung auch daher rühren, dass bei der Paarprogrammierung alle Gedankengänge und Überlegungen klar formuliert und ausgesprochen werden müssen, um sie dem Partner zu kommunizieren. Jeder Programmierer hat wohl schon die Erfahrung gemacht, dass häufig schon das einfache, verbale Beschreiben eines Problems völlig unvermutet zur Lösung führte.

In einer unter über 500 Software-Entwicklern durchgeführten Umfrage im Jahr 1995 gaben die Teilnehmer „Diskussionen mit Kollegen“ als zugleich wertvollste und meist genutzte Technik an [KS95]. Ein wesentlicher Grund könnte die Problemlösung im Kollegenkreis dargestellt haben.

Das Gummienten-Phänomen

Bei meiner Anstellung als Java-Entwickler bei Exozet konnte ich beobachten, dass Kollegen häufig andere Kollegen um Hilfe baten, obwohl diese weder am gleichen Projekt noch mit der gleichen Technologie arbeiteten. Häufig half nämlich schon die reine Formulierung des Problems, um den Hilfesuchenden auf die richtige Spur zu schicken.

Andrew Hunt und David Thomas bezeichnen dieses Phänomen in ihrem Buch *The Pragmatic Programmer* [HT99] als „Rubber Ducking“:

> A very simple but particularly useful technique for finding the cause of a problem is simply to explain it to someone else. The other person should look over your shoulder at the screen, and nod his or her head constantly (like a rubber duck bobbing up and down in a bathtub). The do not need to say word; the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap of the screen and announce itself.

Höhere Motivation Paarprogrammierung soll zu einer höheren Motivation beitragen. Williams et al. [WKCJ00] prägen den Begriff des *pair pressure*, einer positiven Form von Gruppenzwang. Programmierer sollen bei Paarprogrammierung ein größeres Anliegen haben, besser und disziplinierter zu programmieren. So sollen beispielsweise Code-Konventionen und Termine besser eingehalten werden. Williams et al. beobachteten beispielsweise, dass bei ihrem Versuch mit 41 Studenten die Programme von Paaren immer pünktlich abgegeben wurden, während die Programme von Einzelprogrammierern teilweise verspätet oder gar nicht eintrafen.

Ein weiterer häufig genannter Vorteil lautet, dass diese Art von Programmieren mehr Spaß mache und motivierender sei, da die Last der Verantwortung von mehreren Schultern getragen würde. Umfragen zeigen, dass Paarprogrammierer ein höheres Sicherheitsgefühl haben und dadurch mutiger gegenüber Codeänderungen und insbesondere Refactorings werden [Nos98, CW00]. Die intensive Kommunikation, die Paarprogrammierung fordert, soll den Entwicklern helfen auch ihre allgemeine Kommunikationsfähigkeit zu verbessern und eine gesunde Team-Kultur aufzubauen.

Wissensaustausch In der Literatur wird auch betont, dass Paarprogrammierung eine starke Kommunikation zwischen den Teilnehmern erfordere, womit ein stetiger Wissensaustausch gefördert werden würde. Dieser Aus-

tausch soll sich sowohl auf Wissen über den eigenen Projektcode, als auch auf allgemeines technisches Wissen und Erfahrung beziehen [WKCJ00, SWN+03].

In Bezug auf den eigenen Projektcode soll das Wissen über Implementierungsdetails und etwaige undokumentierte Eigenschaften besser im Team verteilt werden. Dies würde eine Kultur der gemeinsamen Codeverantwortlichkeit fördern, bei der jeder Programmierer in jedem Modul des Produktionscodes zumindest über Grundkenntnisse verfügt. Durch wechselnde Paare würde sich das Wissen allmählich über das gesamte Team verteilen. Die Abhängigkeit von einzelnen Programmierern und das Risiko durch Teamabgänge[2] würden reduziert.

2.2. Nachteile

Im Folgenden fasse ich einige der üblicherweise mit Paarprogrammierung assoziierten Nachteile zusammen. Als größter Kritikpunkt wird häufig der Kostenaspekt angegeben.

Erhöhte Kosten Untersuchungen in Bezug auf die reine Programmiereffizienz kommen teilweise zu sehr unterschiedlichen Ergebnissen. Insgesamt scheinen die meisten Untersuchungen jedoch davon auszugehen, dass die Leistung eines Programmierpaares zwischen etwa 100% und 170% der Leistung eines üblichen Einzelprogrammierers liegt [Nos98, CW00, WKCJ00, MP04]. Dies würde bedeuten, dass Paarprogrammierung im Vergleich zum normalen Programmieren als ineffizient anzusehen wäre, da gegenüber zwei Einzelprogrammierern ein Leistungsverlust zwischen 30% und 100% entstehen würde.

Dazu muss aber bemerkt werden, dass sich die bei diesen Untersuchungen betrachteten Metriken meist nur auf oberflächliche Maßzahlen, wie Codezeilen oder die Zeitdauer in der eine Funktionalität umgesetzt werden kann, beziehen. Befürworter von Paarprogrammierung argumentieren, dass die höheren Kosten vor allem durch die verbesserte Wartbarkeit, welche im Allgemeinen einen erheblichen Anteil an den Gesamtkosten haben, überkompensiert werden.

In einem Gedankenspiel mit einem Programm mit 50.000 Codezeilen errechnen Cockburn und Williams [CW00] beispielsweise eine Überkompensierung von Faktor 15 zwischen erhöhtem Programmieraufwand durch Paarprogrammierung und dem Aufwand, den eine Qualitätssicherung gebraucht hätte, um die durch Paarprogrammierung vermiedenen Defekte

[2] Im Jargon manchmal auch als Truck-Faktor oder Truck Number bezeichnet: „How many or few people would have to be hit by a truck (or quit) before the proejct is incapaciated?“

aufzuspüren. Ohne Qualitätssicherung soll sogar ein Faktor von 60 zwischen erhöhtem Aufwand durch Paarprogrammierung und dem Aufwand der Wartung beim Kunden liegen.

Ein anderer Aspekt ist der Vergleich mit Durchsichten. Da die Ziele von Durchsichten und Paarprogrammierung sich teilweise überdecken, ist es Gegenstand der Forschung festzustellen, unter welchen Aspekten welches Instrument vorzuziehen ist. Müller und Padberg [PM03] kommen hier nach Entwicklung eines Kostenmodells beispielsweise zu dem Resultat, dass Paarprogrammierung vor allem dann vorzuziehen wäre, wenn ein hoher Marktdruck vorliegen würde und eine schnelle Time-to-Market von hoher Bedeutung wäre. Andernfalls seien zumindest der bessere Entwurf und die Defektsenkung auch mittels kostengünstigerer Durchsichten zu erreichen.

Räumliche Nähe Paarprogrammierung benötigt eine räumliche Nähe der beteiligten Programmierer. Dies stellt für verteilt arbeitende Teams ein erhebliches Hindernis bei der Adaptierung von Paarprogrammierung dar.

Unpassende Arbeitsplätze Paarprogrammierung benötigt Arbeitsplätze, an denen mehrere Entwickler gleichzeitig arbeiten können. Dies beinhaltet vor allem die Möglichkeit, an einem Arbeitsplatz bequem mit zwei Stühlen Platz nehmen zu können und beiden Entwicklern einen guten Blick auf den Bildschirm zu gewähren. Eventuell sind dazu Umgestaltungen von Tischplätzen notwendig. Kent Beck geht in seinem Buch *Extreme Programming Explained: Embrace Change* [Bec99] stärker auf dieses Thema ein.

Unpassende Charaktertypen Ein weiterer Kritikpunkt ist, dass viele Programmierer insgesamt eher als introvertiert gelten würden und sich daher davor scheuen würden ihren lieb gewonnenen Einzelarbeitsplatz einem Umfeld zu öffnen, in dem sie sich so eng auf die Zusammenarbeit mit einer anderen Person einlassen müssen [CW00, RS02].

Williams et al. [WKCJ00] sehen hier gerade eine zu große Dominanz („my way or the highway“) beziehungsweise eine zu große Passivität als die kritischsten Konstellationen für Paare. Potenziell drohe hier auch die Gefahr der *Freifahrt*, wenn ein Programmierer die Paarung nicht ernst nehmen würde und die gesamte Arbeit seinem Partner überließe.

Fixierung auf Plattformen Ein weiterer Aspekt stellt meiner Meinung nach die Festlegung auf bestimmte Plattformen dar. Paarprogrammierung bedeutet auch immer sich auf eine gemeinsame Entwicklungs-Plattform einigen zu müssen. Dies fängt bei unterschiedlichen Betriebssystemen (zum Beispiel Windows, Linux oder Mac) an und betrifft unter anderem Editoren, Syntaxhervorhebungen, Tastaturkürzeln und Bildschirmaufteilungen.

Im Umgang mit diesem Problem identifiziere ich zwei Ansätze: Zum einem kann man sich mit den Unterschieden abfinden, in der Annahme, dass die Effektivität eines Programmierers auch mit einem für ihn ungewohnten Editor oder Betriebssystem nicht relevant geschmälert wird.

Ein anderer Ansatz wäre eine Standardisierung, beispielsweise die allgemeine Festlegung auf eine bestimmte IDE. Der Grad der Standardisierung ist hier graduell. Im Extremfall könnten in einem Unternehmen gar keine persönlichen Arbeitsplätze mehr eingesetzt werden und stattdessen allgemeine, für jeden zugängliche Arbeitsplätze eingerichtet werden. Auf diesen Arbeitsplätzen könnte dann eine starke Standardisierung verschiedener Einstellungen vorliegen. Dieser Ansatz wird von Kent Beck [Bec99] weiter erläutert.

2.3. Praxis

Paarprogrammierung wird nicht nur programmatisch sondern vor allem auch spontan und nach Bedarf genutzt. Meist ist dies der Fall, wenn besonders wichtige Entwurfsentscheidungen zu treffen sind, komplexe Programmteile zu bearbeiten sind oder ein Entwickler an der Expertise eines mit dem System oder Framework besser vertrauten Entwicklers teilhaben will. In dieser Form ist Paarprogrammierung relativ weit verbreitet, wird allerdings nicht immer als eigene Technik erkannt, geschweige denn als Paarprogrammierung bezeichnet.

Paarprogrammierung wurde formal in den 1990ern definiert. 1995 beschrieb Larry Constantine in seinem Buch *Constantine on Peopleware* sogenannte „dynamic duos“, welche er bei einem Besuch einer Firma namens Whitesmiths Ltd. bemerkt hatte [Con95]. Im gleichen Jahr beschrieb Jim Coplien das „Developing-in-Pairs“-Organisationsmuster in seinem Buch *Pattern Languages of Program Design* [CS95].

Besondere Bekanntheit hat Paarprogrammierung aber erst im Umfeld des *Extreme Programming* (XP) erlangt. XP ist ein Prozessmodell, welches von Kent Beck [Bec99] (mit Ward Cunningham und Ron Jeffries) entwickelt wurde und bereits bekannte und als sinnvoll anerkannte Techniken (dies wird zumindest von Kent Beck postuliert) kombiniert und in extremer, das heißt grundsätzlicher und disziplinierter Weise anwendet. Paarprogrammierung gehört in XP zu den zwölf Grundprinzipien und wird nicht nur gelegentlich, sondern als alleinige Entstehungsart für Code angewendet.

XP hat sich zu einem der bekanntesten Prozessmodelle entwickelt und ist heute der am weitesten verbreitete agile Prozess. Bei Umfragen unter XP-Anwendern gilt Paarprogrammierung als eine ihrer wertvollsten Techniken [RS02].

Vonseiten klassischer und nicht-agil geplanter Projekte herrscht eine größere Skepsis gegenüber dieser ausgeprägten Form der Anwendung und obwohl Paarprogrammierung bereits seit über 10 Jahren erforscht wird, hat sie sich in der allgemeinen Programmierpraxis noch nicht durchgesetzt. Gerade Verantwortlichen scheint oft schwer zu vermitteln, warum zwei Programmierer, die sich zu um einen Bildschirm scharen, mehr leisten können sollen, als zwei Einzelprogrammierer, die jeweils gleichzeitig tippen könnten [CW00].

Die 80:20-Regel im Extreme Programming

Laut Beck soll Paarprogrammierung innerhalb von XP auch deshalb ihr ganzes Potenzial ausschöpfen, weil die üblicherweise assoziierten Kritikpunkte (siehe Abschnitt 2.2) durch den gleichzeitigen Einsatz der anderen XP-Praktiken gelindert würden. Kent Beck spricht dabei von der Anwendung der 80:20-Regel: Werden nur 80% seiner insgesamt 12 vorgeschlagenen Techniken angewendet, ist nur mit einer Leistung von 20% im Vergleich zum Gesamtpotenzial von XP zu rechnen [Bec99].

Folgende Gründe werden dafür von ihm aufgezählt:

- Programmierstandards reduzieren Reibereien und Frustration, die durch unterschiedliche Arbeitsstile entstehen können.
- Die 40-Stunden-Woche sorgt dafür, dass jeder frisch und ausgeruht ist, wodurch die Wahrscheinlichkeit verringert wird, dass es zu unfruchtbaren Diskussionen kommt.
- Die Paare schreiben gemeinsam Tests, was den einzelnen Programmierern die Möglichkeit gibt, ihr Verständnis mit dem anderer Programmierer abzustimmen, bevor sie mit der Implementierung beginnen.
- Die Paare können ihre Entscheidungen bezüglich der Benennung und des grundlegenden Designs an einer Metapher ausrichten.
- Die Paare arbeiten mit einem einfachen Entwurf, sodass beide Programmierer wissen, was vor sich geht.

2.4. Herausforderungen der Evaluation

Untersuchungen von Paarprogrammierung haben oft anekdotenhaften Charakter und unterscheiden sich stark im Aufbau, womit die genaue Evaluation der oben genannten Aspekte erschwert wird. Insbesondere ist nicht nur zu klären, wie erstrebenswert Paarprogrammierung ist, sondern vor allem auch wann und unter welchen Umständen [GAD03]. Hulkko and Abrahamsson weisen beispielsweise darauf hin, dass bei ihren Beobachtungen die Paarprogrammierung vor allem in anfänglichen Phasen angewendet wurde und sich vor allem für die Paar-Analyse beziehungsweise den Entwurf eignet [HA05].

Auch William et al. [WKCJ00] bestätigen, dass die gemeinsame Analyse und Entwurf anscheinend als wichtiger erachtet wird, als die Implementierung im Paar. Mitunter soll auch zuerst zusammen analysiert und anschließend getrennt implementiert werden. Dies solle besonders häufig dann geschehen, wenn die Implementierung nicht sehr schwer erscheine und eher stupiden Charakter habe.

Problematisch an Evaluationen von Paarprogrammierung ist, dass die Teilnehmer der Versuche häufig keine vorhergehenden Erfahrungen mit Paarprogrammierung haben und dadurch eventuell nicht ihre volle Effizienz innerhalb des Versuchs erreichen können. Verschiedene Untersuchungen berichten davon, dass Paare eine gewisse Einstimmungszeit benötigen. Diese wird von Williams et al. [WKCJ00] als *pair-jelling* bezeichnet. Williams et al. berichten beispielsweise von Einstimmungszeit von wenigen Tagen bis Stunden in der Industrie je nach Individuum. Bei ihrem Versuch mit Studenten sank der durch Paarprogrammierung zusätzliche entstandene Zeitaufwand von anfangs 60% auf 15% nach einer Eingewöhnungszeit von einigen Sitzungen.

3. Verteilte Paarprogrammierung

Dieses Kapitel beschäftigt sich mit verteilter Paarprogrammierung. Ich erläutere zuerst konzeptionelle Vor- und Nachteile dieser Programmiertechnik. Anschließend führe ich eine Taxonomie von technischen Werkzeugansätzen an, die ich aus dem Forschungsfeld der *Groupware* entlehne. Anhand verschiedener Kriterien vergleiche ich die Ansätze des *Desktop Sharing* und der *Collaboration Awareness*. Abgeschlossen wird das Kapitel durch ein Fazit, in welchem ich meine Erläuterungen zusammenfasse.

3.1. Konzept

In diesem Abschnitt beschreibe ich verteilte Paarprogrammierung als Konzept. Bei meinen Erläuterungen zu Vor- und Nachteilen stellt die reguläre Paarprogrammierung meinen Bezugspunkt dar.

3.1.1. Vorteile

Unterstützung virtueller Teams Mit steigender Verfügbarkeit von Netzwerk-Technologie werden die Möglichkeiten zur Fernarbeit größer. Mit den Möglichkeiten neuartiger verteilter Teamstrukturen befasst sich der Forschungsbereich der *Virtual Teams*. Townsend et al. [TDH98] definieren virtuelle Teams folgend:

> Groups of geographically and/or organizationally dispersed co-workers that are assembled using a combination of telecommunications and information technologies to accomplish an organizational task.

Die Definition betont sowohl die räumliche Verteilung als auch die Nutzung elektronischer Kommunikationsmittel. Für diese Art von Teams stellt verteilte Paarprogrammierung womöglich nicht nur eine Alternative, sondern sogar die einzige Möglichkeit zur Anwendung von Paarprogrammierung dar.

Für verteilte Teams ist eine gute Kommunikation üblicherweise schwerer zu realisieren als bei räumlicher Nähe. Von besonderem Interesse könnte daher auch dieser Aspekt der Paarprogrammierung sein. Die enge Kommunikation, die verteilte Paarprogrammierung verlangt, könnte den Aufbau einer „virtuellen Kultur“ fördern [McM01].

Maurer [Mau02] argumentiert außerdem, dass Extreme Programming zwar gegenüber traditionelleren Prozessen einige Vorteile aufweisen könne, jedoch in seiner ursprünglichen Form [Bec99] auch zwei schwerwiegende Schwächen besäße: Erstens skaliere es aufgrund der Bevorzugung von direkter Kommunikation gegenüber Dokumentation nicht gut für große Teams; zweitens erfordere es eine gemeinsame räumliche Anwesenheit des Teams, um allen Prinzipien nachkommen zu können.

Es gibt daher verschiedene Bemühungen verteilte Prozessvarianten des Extreme Programming zu entwickeln. Ein Beispiel für eine solche Bemühung ist das Software-System *MILOS* (Minimally Invasive Longterm Organizational Support) [MM02]. MILOS dient zur Realisierung von *distributed extreme programming*, verteilte Paarprogrammierung stellt eines ihrer Kernprinzipien dar.

Bequemere Arbeitsplätze Wie von Beck [Bec99] beschrieben, kann der Aufbau eines Büros die Paarprogrammierung erschweren, da diese häufig nicht auf Paarprogrammierung ausgelegt sind. Auf diesen Aspekt bin ich bereits unter Abschnitt 2.2 als Nachteil der Paarprogrammierung eingegangen. Stotts et al. [SWN+03] weisen darauf hin, dass diese Problematik durch verteilte Paarprogrammierung reduziert wird. Im Gegensatz zur regulären Paarprogrammierung verfügen alle Teilnehmer der Sitzung über separate Bildschirme und Eingabegeräte. Die Sichtbarkeit und die Bequemlichkeit der Arbeitsplätze wird verbessert[1].

Höhere Formalität Stotts et al. [SWN+03] geben außerdem an bei Versuchen mit verteilter Paarprogrammierung eine höhere Formalität im Umgang mit Dokumenten beobachtet zu haben. Bei klassischen Paarprogrammiersitzungen entstehen häufig spontan Dokumente, die bei der Vermittlung eines bestimmten Klassenaufbaus oder einer Idee helfen. Während diese bei einer gemeinsamen Anwesenheit womöglich beiläufig auf einem Stück Papier oder einer Whiteboard festgehalten werden würden und so leicht verloren gingen, erlaube die elektronische Form, die durch die verteilte Paarprogrammierung erzwungen wird, sie leichter zu speichern und später wieder zu finden.

Denkbar wäre beispielsweise die Nutzung eines grafischen Editors um Ide-

[1] Denkbar werden dadurch auch Sitzungen mit deutlich mehr als zwei Programmierern. Ob in Sitzungen mit vielen Teilnehmern noch die Prinzipien der Paarprogrammierung umgesetzt werden können, bleibt allerdings zu untersuchen.

en schnell zu skizzieren. Welche Arten von Editoren dabei genau zur Verfügung stehen, hängt im wesentlichen vom technischen Ansatz des DPP-Werkzeugs ab (siehe Abschnitt 3.2). Abzuwägen bleibt jedoch, ob der zusätzliche Aufwand in der Benutzung dieser elektronischen Werkzeuge sich womöglich hemmend auf den Kommunikationsfluss auswirkt.

Mehr Parallelität In der regulären Paarprogrammierung hat nur der Driver Kontrolle über die Arbeitsfläche und bestimmt den einzigen Aufmerksamkeitsbereich des Paares. In der verteilten Paarprogrammierung können Werkzeuge dagegen einen gewissen Grad an Parallelität aufweisen. Mit Parallelität meine ich, dass der Observer die Möglichkeit hat, aktiver in die Sitzung einzugreifen. Durch Parallelität könnte die Arbeitsleistung des Paares gesteigert werden.

Stotts et al. [SWN+03] berichten, dass bei einem Versuch die Teilnehmer die Möglichkeit nutzten, Programme parallel zum DPP-Werkzeug zu betreiben. Dadurch sei die Geschwindigkeit des Paares gestiegen, denn der Observer konnte beispielsweise Dokumentationen nachzuschlagen oder im Bugtracker Informationen ablegen, während der Driver mit der Implementierung fortfuhr.

Im Folgenden identifiziere ich drei Stufen der Parallelität, die man meiner Meinung nach mindestens unterscheiden kann:

Parallelität auf Programmebene - Diese Parallelität ist gegeben, wenn der Observer während einer Sitzung die Möglichkeit hat, das Fenster des DPP-Werkzeugs in den Hintergrund zu schieben, um mit anderen Programmen zu interagieren.

Parallelität auf Sichtebene - Diese Parallelität ist gegeben, wenn eine *Parallelität auf Programmebene* gilt und der Observer die Möglichkeit hat von dem Aufmerksamkeitsbereich des Drivers abzuweichen und eigenständig Dateien eines Projekts zu durchstöbern.

Parallelität auf Schreibebene - Diese Parallelität ist gegeben, wenn eine *Parallelität auf Sichtebene* gilt und es mehr als einen Driver geben darf, das heißt mehrere Personen bekommen gleichzeitig Schreibrechte für die Dateien.

Die Grade an Parallelität bauen aufeinander auf. Ein gewisser Grad an Parallelität scheint in einem DPP-Werkzeug dabei nahezu unausweichlich. So ergibt sich eine *Parallelität auf Programmebene* fast automatisch.

Dabei ist meiner Meinung nach zu beachten, dass nicht jeder Grad an Parallelität erwünscht sein kann. Wie in Abschnitt 2.1 erläutert, stellt die stetige Durchsicht ein wichtiges Grundprinzip von Paarprogrammierung

dar. Auch bei der verteilten Paarprogrammierung sollte dieses Prinzip nie außer acht gelassen werden. Insofern ist es wichtig zu untersuchen, welcher Grad an Parallelität zu einer Effizienzsteigerung von Paaren beitragen kann, aber auch ab wann die Prinzipien der Paarprogrammierung verletzt werden. Ich ergründe diesen Aspekt noch weiter in der von mir durchgeführten Umfrage (siehe Abschnitt 4.2.4).

3.1.2. Nachteile

Workspace Awareness Dem Begriff *Awareness* kommt im Bereich der *Computer supported cooperative work* (siehe auch Abschnitt 3.2) eine besondere Bedeutung zu. Dourish und Bellotti [DB92] definieren Awareness als:

> An understanding of the activities of others, which provides a context for your own activity.

Diese Definition ist sehr allgemein gehalten. Eine etwas enger gefasste Definition liefern Gutwin und Greenberg [GG99] unter dem Begriff *Workspace Awareness*:

> The up-to-the-moment understanding of another person's interaction with the shared space.

Diese Definition stellt in zweierlei Hinsicht eine Verschärfung der Definition von Dourish und Bellotti dar. Zum einem impliziert „up-to-the-moment“ eine synchrone, das heißt echtzeitliche Aktualität des Bewusstseins. Zum anderen liegt das Bewusstsein nur in Bezug auf die Nutzung des gemeinsamen Arbeitsplatzes vor.

In Bezug auf verteilte Paarprogrammierung ist die Motivation für die Beschäftigung mit Workspace Awareness klar: Es müssen nicht nur die Änderungen an den gemeinsamen Artefakten nachgebildet werden, sondern teilweise auch das Verhalten der Programmierer. Dazu können beispielsweise Sprache, Gestiken, Blicke, Gesichtsausdrücke und die Körpersprache gehören. Welche dieser Aspekte von besonderer Bedeutung sind, erforsche ich ebenfalls im Rahmen der von mir durchgeführten Umfrage (siehe Abschnitt 4) genauer.

Auch wenn DPP-Werkzeuge in der Lage sind, die bei einer großen räumlichen Trennung[2] verloren gegangene Aufmerksamkeit teilweise zu kompensieren, so ist insgesamt doch von einem Verlust an Awareness auszugehen. Im schlimmsten Fall kann dies sogar soziale Auswirkungen haben: Die

[2] Liegt nur eine geringe räumliche Trennung (zum Beispiel Sichtreichweite) vor, fällt der Verlust an Aufmerksamkeit natürlich entsprechend geringer aus.

	Gleiche Zeit	**Verschiedene Zeiten**
Gleicher Ort	Gesicht-zu-Gesicht Interaktion	Asynchrone Interaktion
Verschiedene Orte	Synchrone verteilte Umgebung	Asynchrone verteilte Umgebung

Tabelle 3.1.: Raum-Zeit-Taxonomie für Groupware nach Ellis et al. [EGR91].

Vorteile der verbesserten Kommunikation und Teamkultur kommen bei verteilter Paarprogrammierung womöglich nicht im gleich Umfang zum Tragen wie bei regulärer Paarprogrammierung.

Werkzeugeinsatz - Zur Realisierung von verteiler Paarprogrammierung müssen spezielle Werkzeuge eingesetzt werden. Eine zusätzliche Einarbeitungszeit, neue Investitionskosten und Risiken durch Werkzeugausfälle sind Nachteile, die entstehen können.

3.2. Technische Werkzeugansätze

Zum Aufspüren einer Taxonomie für technische Werkzeugansätze für verteilte Paarprogrammierung habe ich den Forschungsbereich der sogenannten *Groupware* herangezogen. Ellis et al. [EGR91] definieren Groupware folgendermaßen:

> computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.

Systeme, die als Groupware bezeichnet werden können, sind vielfältig. Entsprechend der Definition gehören beispielsweise auch E-Mails, Wikis, Bugtracker und Konferenzsysteme in diese Kategorie. Eine von Ellis et al. vorgeschlagenes System zur Unterscheidung unterschiedlicher Groupware, stellt die Raum-Zeit-Taxonomie dar (siehe Tabelle 3.1), welche die Programme in vier Kategorien aufteilt. Innerhalb dieser Kategorisierung gehören Werkzeuge für verteilte Paarprogrammierung in die linke untere Ecke (gleiche Zeit/verschiedene Orte). Groupware stellt eine Erscheinungsform der *Computer Supported Cooperative Work* (CSCW) dar, welche sich mit dem allgemeinen Erforschen der theoretischen Grundlagen von computergestützter Gruppenarbeit beschäftigt.

Es lassen sich zwei Ansätze unterscheiden, die Groupware-Entwickler in den vergangenen 15 Jahren zur Umsetzung synchroner Zusammenarbeit gewählt haben: *Desktop Sharing* und *Collaboration Awareness*. Diese beiden Ansätze

entsprechen den Typen heutige DPP-Werkzeuge [Han05, HRGW04]. Im Folgenden möchte ich diese beiden Ansätze kurz einführend erklären.

3.2.1. Desktop Sharing

Beim *Desktop Sharing* (auch als *Screen Sharing* oder *Application Sharing* bezeichnet) wird die Ansicht eines Desktops (auf Deutsch Arbeitsfläche) über ein Netzwerk auf den Schirm eines oder mehrerer anderer entfernter Systeme übertragen. Dem Benutzer des entfernten Systems ist es dabei üblicherweise möglich die Kontrolle über den Computer zu übernehmen und Maus und Tastatur zu kontrollieren, indem er an seinen eigenen Arbeitsplatz agiert und seine Aktionen über das Netzwerk übertragen werden. Häufig wird diese Funktionalität zur Fernadministration und für Demonstrationen eingesetzt.

Erstmals wurde Desktop Sharing bereits 1969 von Engelbert und English demonstriert [Han05]. Seitdem hat sich diese Funktionalität in einer Vielzahl von kommerziellen und Open-Source-Applikationen niedergeschlagen. Bekannte kommerzielle Produkte sind beispielsweise Symantec pcAnywhere [PCA] und Microsoft NetMeeting [NM].

Die bekannteste Open-Source-Plattform stellt das plattformunabhängige *Virtual Network Computing* (VNC) beziehungsweise ihr *Remote Framebuffer Protocol* dar [RSFWH98]. VNC war ursprünglich von AT&T entwickelt und unter der *GNU General Public License* [GPL] veröffentlicht worden. Inzwischen basiert eine Vielzahl ebenfalls quelloffener und größtenteils kompatibler Ableger auf dem gleichen Code. Die bekanntesten Vertreter sind RealVNC[3] [RVN] und UltraVNC [UVN].

3.2.2. Collaboration Awareness

Im Gegensatz zum Desktop Sharing bei der die schließlich auf der Arbeitsfläche verwendeten Programme nichts von ihrem „verteilten“ Einsatz wissen, ist die Mehrbenutzer-Unterstützung bei Werkzeugen mit *Collaboration Awareness* direkt integriert. In welcher Form dies geschieht, kann sehr unterschiedlich sein. Typische Programme dieser Art sind beispielsweise kollaborative Texteditoren, kollaborative Zeichenprogramme oder Konferenzsysteme. Für meine Untersuchung sind dabei zwei Arten von kollaborations-bewussten Werkzeugen von Interesse: Kollaborative Texteditoren und integrierte Entwicklungsumgebungen (IDEs) mit Mehrbenutzerunterstützung.

Kollaborative Texteditoren sind Texteditoren, die mehreren Autoren ermöglichen über eine meist serverbasierte Architektur, gleichzeitig an einem gemeinsa-

[3] Vom originalen Entwicklungsteam entwickelt.

men Textdokument zu arbeiten. Ein bekanntes Programm ist *SubEthaEdit* [SEE], welches für Mac Systeme verfügbar ist und nach einer anfänglichen quelloffenen Entwicklung zu einem geschlossenen, kommerziellen Modell übergegangen ist. Im Open-Source-Bereich existieren auch weitere wenn auch weniger bekannte und ausgereifte kollaborative Texteditoren. Zu nennen wären hier beispielsweise Gobby [Gob] und MoonEdit [Dob].

Eine weitere Kategorie stellen IDEs mit integrierter Unterstützung für verteilte Zusammenarbeit dar. Sie arbeiten ähnlich wie die gerade genannten Texteditoren, bieten darüber hinaus aber auch weitergehende Funktionalitäten, wie zum Beispiel entferntes Übersetzen oder entferntes *Debugging*. Auf die technische Umsetzung einer Fernübersetzbarkeit gehe in Abschnitt 3.3 ein.

Die beiden bekanntesten IDEs mit diesen Funktionen sind Sun NetBeans/Java Studio [RS05][4] und Borland JBuilder [JB]. Eine Möglichkeit zur Erweiterung der Eclipse-Plattform stellt das *Sangam*-Plugin [HRGW04] dar. Dieses scheint allerdings seit etwa 2003 nicht mehr gepflegt zu werden. Versuche es auf meinem System zum Laufen zu bekommen, scheiterten.

3.3. Vergleich der Ansätze

Im Folgenden vergleiche ich die beiden charakterisierten Ansätze anhand verschiedener Kriterien.

3.3.1. Übersetzbarkeit

Die Möglichkeit, dass alle Teilnehmer einer verteilten Paarprogrammiersitzung das gemeinsame Projekt übersetzen können, ist eine elementare Anforderung. Sie stellt jedoch auch eine besondere Herausforderung dar, denn zur Übersetzung einer Javadatei müssen alle im Code verwendeten Datentypen bekannt sein, das heißt ihre Klassendatei vorliegen.

Desktop Sharing

Wie oben beschrieben, finden beim Desktop Sharing alle Aktionen beim *Host*[5] der Sitzung statt. Wird ein Gast zum Driver, so kann er alle Funktionen der auf der Arbeitsfläche geöffneten IDE (beispielsweise Übersetzung) nutzen, da die

[4] Erhältlich in Form eines kostenlos herunterladbaren Plugins von der Sun Internetseite.

[5] Host bezeichne den Benutzer, der die Desktop-Sharing-Sitzung eingeleitet hat und dessen Arbeitsfläche übermittelt wird. Ein Gast sei jemand, der an der Sitzung teilnimmt, aber nicht der Host ist.

Anweisungen beim Host ausgeführt werden, wo alle zur Übersetzung benötigten Dateien vorliegen.

Vorteilhaft ist darüber hinaus, dass die Sitzungen keinerlei aufwendige Einleitungsphase benötigen. Außerdem ergeben sich keine Probleme mit eventuellen externen Abhängigkeiten (zum Beispiel Datenbankanbindungen, die nur für den Host der Sitzung erreichbar sind oder Dateien, die sich außerhalb des Projektverzeichnisses befinden).

Collaboration Awareness

Bei Collaboration Awareness identifiziere ich zwei unterschiedliche Ansätze, die ich mit *Virtueller Zugriff* beziehungsweise *Replikation* bezeichnen möchte.

Virtueller Zugriff - Mit diesem Begriff meine ich einen Ansatz, der bei verschiedenen IDEs mit DPP-Erweiterung (beispielsweise Sun NetBeans [RS05] und Borland JBuilder [JB]), zum Einsatz kommt. Zu Beginn der Sitzung wird eine Dateiliste des entfernten Projekts übertragen[6].

Öffnet der Host eine Datei des Projekts, so wird der Inhalt aus der lokalen Datei geladen. Öffnet allerdings ein Gast eine Datei, so liegt ihm lokal keine Datei vor. Statt eine lokale Ressource zu öffnen, sendet er dem Host eine Anfrage nach dem Inhalt der Datei. Hat er den Inhalt der Datei empfangen, wird dieser in seinem Editor dargestellt und zwar so als ob er eine lokale Datei geöffnet hätte. Er arbeitet also mit einem virtuellen Dateisystem.

Das Übersetzen des Projekts oder das Abrufen von Codeunterstützungen (beispielsweise Codevervollständigung) geht analog von statten: Der Gast schickt eine entsprechende Nachricht an den Host, der Host vollzieht die Aktion lokal und meldet das Resultat zurück.

Dieser Ansatz hat den Vorteil, dass er eine vergleichsweise geringe Anlaufzeit benötigt, da zu Beginn der Sitzung nur rudimentäre Dateilisten übertragen werden müssen, nicht aber der Inhalt der Dateien selber. Ein Nachteil ist, dass viele Vorgänge aus Sicht eines Gastes mit einer deutlich spürbaren Verzögerung verbunden sind. Darüber hinaus kann der Gast ausführbare Artefakte nicht unmittelbar ausführen (beispielsweise kann er in einem Java-Projekt ein erstelltes Java-Programm nicht ausführen). Dazu müsste das Programm mitsamt aller Abhängigkeiten übertragen werden, was einen sehr hohen Aufwand bedeuten würde. Eine einfache Alternative ist zumindest die Übertragung von reinen Textinformationen. Auf diese Art können beispielsweise die Laufresultate eines Komponententests übertragen werden.

[6] Diese Dateiliste kann bereits alle Dateien des Projekts umfassen oder zunächst nur das Grundverzeichnis enthalten, mit der Option nachträglich weitere Verzeichnise zu laden.

Replikation Ein weiterer Ansatz ist die komplette Replikation des Projekts. Bei allen Gästen liegt dann eine komplette Kopie des gemeinsamen Projekts vor. Änderungen an den Ressourcen werden über das Netzwerk kopiert, sodass sie jeweils an allen lokalen Kopien vorgenommen werden.

Dieser Ansatz hat gegenüber dem *Virtuellen Zugriff* den Vorteil, dass Übersetzungen, Code-Inspektion, Suchen und andere IDE-Funktionen lokal und damit ohne eine netzwerkabhängige Verzögerung ausgeführt werden. Dadurch wird die Programmierumgebung wesentlich ansprechbarer. Zudem kann der Gast ein gemeinsam entwickeltes Programm ausführen, sofern dies nicht durch externe Abhängigkeiten verhindert wird.

Ein weiterer Vorteil liegt in der Tatsache begründet, dass beim Collaboration Awareness jede einzelne zu unterstützende Funktionalität der Java-IDE explizit angepasst beziehungsweise erweitert werden muss. Beim Ansatz der Replikation funktionieren dagegen sämtliche sonstigen Funktionalitäten der IDE auch ohne Anpassung.

Problematisch ist hier der teilweise erheblich Aufwand für die vorhergehende Synchronisation. Ich gehe in Abschnitt 5.1.2 weiter auf diese Problematik ein.

3.3.2. Flexibilität

Desktop Sharing

Der Ansatz des Desktop Sharing ist sehr flexibel, da alle Arten von Programmen ohne Anpassung von mehreren Benutzern benutzt werden können [Gre90]. Damit können in einer Sitzung eine Vielzahl von Artefakttypen bearbeitet werden. Beispielsweise kann der Anwender von einem Texteditor problemlos zu einem beliebigen Malprogramm wechseln, um eine Funktionalität als Skizze zu erläutern.

Dies ist meiner Meinung nach, nicht nur in Bezug auf die Übertragung der Artefakte interessant, sondern auch in Bezug auf die Interaktion mit den Programmen selber. Damit meine ich die genaue Art und Weise, wie ein Benutzer das Programm bedient, das heißt beispielsweise die Art wie er die Maus führt, welche Menüpunkte er verwendet oder welche Tastatureingaben er tätigt[7]. Zum Anderen könnten auch bestimmte Vorgänge demonstriert werden, die beispielsweise zu einem unerwünschten Verhalten eines erstellten Programms führen.

[7] Aus meiner eigenen Erfahrung kann ich beispielsweise berichten, dass ich einige Techniken der Eclipse IDE (siehe Abschnitt 5.2.2) erst durch die Observer-Tätigkeit in Paarprogrammier-Sitzungen kennengelernt habe.

Hank [Han05] argumentiert, dass Desktop Sharing auch deshalb zu bevorzugen sei, weil es unwahrscheinlich scheint, dass sich alle Programmierer auf einen gemeinsamen Editor einigen könnten. Dies erscheint mir nicht nachvollziehbar, da auch beim Desktop Sharing die Wahl schlussendlich auf einen Editor fallen muss. Man kann allerdings argumentieren, dass dann zumindest einer der Entwickler die freie Wahl des Editors hat. Damit entspricht die Situation also weitgehend der bei der regulären Paarprogrammierung.

Collaboration Awareness

Werkzeuge mit Collaboration Awareness sind unflexibel und nur streng auf den Verwendungszweck ausgelegt, für den sie konstruiert worden sind. So scheitert die Bearbeitung exotischer Artefakte häufig an der fehlenden Unterstützung. Zur Minderung dieses Nachteils müssen möglichst viele Bearbeitungsfunktionen in das DPP-Werkzeug integriert werden. Ein Mindestkriterium für die Paarprogrammierung bildet die kollaborative Textbearbeitung. Darüber hinaus erscheint eine Whiteboard-Funktionalität, in der schnell Klassendiagramme und Skizzen gestaltet werden können, womöglich ebenfalls sinnvoll.

Im Gegensatz zum Desktop Sharing werden bei Collaboration Awareness die Artefakte und die Veränderungen der Artefakte übertragen, nicht jedoch die tatsächlichen Mausbewegungen oder Tastenkombinationen, die zum Ausführen der Operation vollzogen wurden. Statt Ursachen werden Effekte übertragen. Dies mindert Lerneffekte in Bezug auf die Verwendung des gemeinsamen Editors. Es verhindert darüber hinaus die Möglichkeit zur Demonstration bestimmter Interaktionen mit dem Programm.

Dafür kann ein Werkzeug dieser Art spezielle Unterstützungen für die verteilte Paarprogrammierung direkt integrieren. Sie versteht den Kontext und den Zusammenhang der Vorgänge. Denkbar wäre beispielsweise eine Historie der letzten Aktionen des Drivers, die dem Benutzer ermöglichen würde verpasste Aktionen noch mal einzusehen oder sogar zurückzuspulen. Eine weitere Einsatzmöglichkeit findet sich in der systematischen Erforschung der Vorgänge innerhalb von Paarprogrammier-Sitzungen. Auf diesen Aspekt gehe ich in Abschnitt 6.2 ausführlicher ein.

Bei Collaboration Awareness müssen beide Entwickler zumindest für die Zeitdauer der Sitzung den gleichen Editor benutzen, da der Editor und das Werkzeug zur Bereitstellung der DPP-Sitzung eins sind. Damit haben diese Werkzeuge einen umfassenderen Einfluss auf die Sitzung als das Desktop Sharing, wo die Wahl des Editors unabhängig bleibt von der Wahl des Desktop-Sharing-Werkzeugs.

3.3.3. Technische Restriktionen

Desktop Sharing

Beim Desktop Sharing wird die übertragene Arbeitsfläche, wie eine Art Video gespiegelt. Dadurch kann es in Bezug auf die Auflösung des Desktops zu Problemen kommen. Üblicherweise wird der entfernte Desktop in einem Fenster gespiegelt, dessen Größe exakt der Auflösung des entfernten Desktops entspricht. Sind die Auflösungen des Senders und Empfängers identisch, so nimmt das Fenster den kompletten Bildschirm des Empfängers ein. Ist sie größer, kann nicht mehr alles auf einen Blick erfasst werden und es werden Bildlaufleisten nötig, um alles zu erfassen.

Insofern ist es empfehlenswert, dass sich die Teilnehmer einer Sitzung zuerst bestimmte Auflösungen verabreden. Stotts et al. [SWN+03] empfehlen zum Beispiel, dass der Empfänger eine größere Auflösung haben sollte als der Sender. So kann der gespiegelte Desktop in einem Fenster dargestellt werden, welches nicht den ganzen Bildschirm einnimmt. Maurer et al. [MM02] und Ho et al. [HRGW04] empfehlen zumindest ähnliche Auflösungen zu wählen.

Die Art der Übertragung bedeutet darüber hinaus einen hohen Aufwand in Bezug auf die Datenrate, die zur Spiegelung benötigt wird. Der Belastungsgrad wird unter anderem von drei Faktoren bestimmt: Der Bildschirmauflösung, der Farbauflösung und der Bildfrequenz. Die Reduktion der jeweiligen Einstellungen kann jedoch negative Auswirkungen haben. Bei niedriger Auflösung kann weniger Text dargestellt werden und es muss eventuell auf Hilfsanzeigen verzichtet werden. Bei einer geringen Farbauflösung kann die Arbeitsfläche schwerer zu erkennen sein. Von besonderer Bedeutung ist allerdings die Bildfrequenz: Eine hohe Frequenz resultiert in einer hohen Netzlast. Bei einer zu niedrigen Frequenz entgehen einem Observer dagegen Vorgänge. Ho et al. [HRGW04] beschreiben folgendes Szenario:

> A low display-refresh rate can sometimes be confusing. Something significant may be lost at the remote display. For example, if the driver copies some text from an editor and pastes it into another editor, the navigator may only see the new content of the latter editor, without knowing where it came from.

Collaboration Awareness

Wie bereits erwähnt, werden bei kollaborationsbewussten Werkzeugen nicht die Ursachen für Effekte (beispielsweise Mausklicks oder Tastaturkürzel) übertragen, sondern nur deren Effekte selber. Das Werkzeug kennt den Sinn jeder Aktion und kann sie selbstständig nachbilden. Das DPP-Werkzeug kann so von jedem Teilnehmer der Sitzung eigens in Größe und Konfiguration angepasst

werden. So könnten beide Entwickler beispielsweise unterschiedliche Bildaufteilungen, Tastaturkürzel oder Texteinstellungen verwenden.

Bei Werkzeugen dieser Art fällt darüber hinaus weniger Netzlast an als beim Desktop Sharing. Zum einem werden nur relevante Daten übertragen (beispielsweise keine Mausbewegungen oder Fensterverschiebungen[8]). Zum anderen haben die Aktionen ein wesentlich geringeres Datenvolumen als die Bilder, welche beim Desktop Sharing übertragen werden.

3.3.4. Sicherheit

Desktop Sharing

Computer sind meiner Meinung nach häufig, unabhängig davon ob zuhause oder an einer beruflichen Arbeitsstätte, auch ein Ort, an dem persönliche Daten aufbewahrt werden. Gerade wenn sich die Paarprogrammierer nicht persönlich kennen - wie beispielsweise bei verteilten Teams im Open-Source-Bereich denkbar - und daher noch wenig Vertrauen zueinander aufgebaut haben, kann es ein Gefühl des Unbehagens und der Unsicherheit auslösen, seinen Computer so freigeben zu müssen, wie dies beim Desktop Sharing der Fall ist[9].

Ganz konkret könnte man sich einen Fall vorstellen, bei dem ein Programmierer beispielsweise ein relevantes Codefragment aus einer E-Mail laden möchte, aber zögert das E-Mail-Programm zu öffnen, weil sich dort auch persönliche Korrespondenz befindet.

Collaboration Awareness

Bei Collaboration Awareness existiert eine klare Grenze zwischen Informationen, die übertragen werden und solchen, die nicht übertragen werden. Für die Sitzung irrelevante und womöglich sogar private Daten werden keinem Sicherheitsrisiko ausgesetzt. Eventuelle Ängste vor einer Paarsitzung könnten so reduziert werden.

[8] Dies kann in Bezug auf Demonstrationen jedoch auch als Nachteil gewertet werden (siehe Abschnitt 3.3.2).

[9] Diese Gefahr bezieht sich beim Desktop Sharing natürlich nur auf den Host der Sitzung.

Paarprogrammierung und Privatsphäre

Bei meinen eigenen Erfahrungen mit Paarprogrammierung habe ich immer wieder Situationen erlebt, in denen der Besitzer des Computers plötzlich aufspringende Instant-Messenger-Meldungen hastig wegklickte oder Browser-Fenster vor Beginn der Sitzung beendete. Dies hat meiner Meinung nach nicht unbedingt mit dem unangenehmen Inhalt der betreffenden Medien zu tun, sondern viel eher mit einem allgemeinen Unbehagen gegenüber dem Kontrollverlust der auftritt, wenn jemand am eigenen Computer zu Werke geht.

Kent Becks [Bec99] Auffassung von einer perfekten PP-Umgebung unterscheidet vielleicht ja auch aus diesem Grund zwischen öffentlichen Arbeitsstationen für die Paarprogrammierung und ergänzenden privaten Arbeitsnischen, in die sich Programmierer zurückziehen können sollen, um beispielsweise ungestört Telefonate erledigen zu können.

3.3.5. Parallelität

Den Begriff Parallelität habe ich in Abschnitt 3.1.1 eingeführt.

Desktop Sharing

Beim Desktop Sharing ist teilweise eine *Parallelität auf Programmebene* gegeben. Jeder Gast sieht die Sitzung nur als Fenster. Es steht ihm daher frei das Fenster in den Hintergrund zu schieben. Für den Host der Sitzung ist üblicherweise keinerlei Parallelität gegeben. Er stellt seine ganze Arbeitsfläche zur Verfügung: Wird er zum Observer, übernimmt der Driver die Maussteuerung und der Host hat keine Möglichkeit parallel andere Arbeiten auszuführen.

Collaboration Awareness

Bei Collaboration Awareness liegt nahezu immer eine *Parallelität auf Programmebene* vor. Beispiele für IDEs mit Unterstützung für *Parallelität auf Sichtebene* sind Borland JBuilder [JB] und Sangam [HRGW04]. SubEthaEdit [SEE] und die meisten anderen kollaborativen Texteditoren ermöglichen zudem eine *Parallelität auf Schreibebene*, allerdings sind sie meist auf die Arbeit mit einigen wenigen Dateien ausgelegt. Eine IDE, welche eine Schreibparallelität mit vollständiger Projektunterstützung ermöglicht, ist Sun NetBeans [RS05].

Desktop Sharing	**Collaboration Awareness**
+ Unterstützt alle Artefakte	- Unterstützt bestimmte Artefakte
+ Für Demonstrationen geeignet	- Für Demonstrationen nicht geeignet
+ Guter Lerneffekt bzgl. Editor-Bedienung	- Schlechterer Lerneffekt bzgl. Editor-Bedienung
+ Keine Probleme mit externen Abhängigkeiten	- Evtl. Probleme mit externen Abhängigkeiten
- Geringe Parallelität	+ Hohe Parallelität
- Keine spezielle PP-Unterstützung	+ Spezielle PP-Unterstützung
- Darstellungsprobleme	+ Immer optimale Darstellung
- Hohe Netzlast	+ Geringe Netzlast
- Gesamter Desktop wird frei gegeben	+ Mehr Datenschutz

Tabelle 3.3.: Gegenüberstellung von Vor- und Nachteilen der beiden technischen Ansätze zur verteilten Paarprogrammierung.

3.4. Fazit

Tabelle 3.3 stellt die wichtigsten von mir erläuteten Punkte übersichtlich gegenüber. Die Tabelle erhebt natürlich keinen Anspruch auf Vollständigkeit; zudem sind nicht alle Punkte gleich gewichtig zu werten. Welcher Werkzeugansatz für die eigene Programmierpraxis nützlicher ist, hängt immer auch von den eigenen Bedürfnissen und Programmierumständen ab.

Die wichtigsten Vorteile der Desktop-Sharing-Ansätze scheinen mir ihre Flexibilität in Bezug auf die Bearbeitung von allen möglichen Artefakttypen und ihre Fähigkeit zu Demonstrationen. Es gibt Anwendungsfälle, in denen diese Aspekte ausschlaggebend sind. Findet beispielsweise ein wesentlicher Teil der Entwicklungstätigkeit mittels eines speziellen Artefakteditors statt, so müsste dieser bei gewünschter Collaboration Awareness angepasst werden, was finanziell unrentabel sein kann oder aufgrund von fehlenden Rechten und Quelldateien sogar unmöglich. In solch einem Fall wäre der Einsatz von Desktop-Sharing sinnvoll.

In Situationen, in denen die Artefakttypen stark definiert und textbasiert sind, wie es beispielsweise in den meisten Java-Entwicklungen der Fall ist, scheinen mir die Vorteile der Collaboration Awareness zu überwiegen. Insbesondere die Möglichkeiten von DPP-spezifischen Funktionalitäten halte ich für sehr reizvoll. Ich habe mich daher entschlossen, das von mir zu entwickelnde DPP-Werkzeug mit Collaboration Awareness zu realisieren.

Ein ideales DPP-Werkzeug würde womöglich den Anatz der Collaboration Awareness, mit der Möglichkeit zum Screen-Sharing bestimmter frei wählbarer Fenster, kombinieren. Ein derartiges Werkzeug würde die Vorteile beider Ansätze kombinieren.

4. Umfrage

Zur Ergänzung meiner Erläuterungen zu Werkzeugen verteilter Paarprogrammierung habe ich eine Umfrage entwickelt. Die Forschungsfrage der Umfrage lautete:

> In welcher Weise setzen Benutzer verteilte Paarprogrammierung heute ein und welche Funktionen eines DPP-Werkzeugs werden dabei als besonders wichtig erachtet?

Ziel der Umfrage war somit auch von mir getroffene Annahmen zur verteilten Paarprogrammierung (siehe Abschnitt 3) tendenziell zu bestätigen oder zu widerlegen. Zudem sollten mir die Resultate als Entscheidungshilfe bei der Priorisierung von Anforderungen für die Entwicklung eines DPP-Werkzeugs dienen (siehe Abschnitt 5.1). Mir ist klar, dass eine Umfrage in diesem Rahmen bestenfalls ein zusätzliches Indiz, aber kein allein stehender Beweisgang sein kann.

Zur Implementierung der Umfrage habe ich das freie Online-Umfragen-System phpESP [ESP] verwendet, welches auf einem Projektserver des Instituts für Informatik installiert wurde. Das Erscheinungsbild der Umfrage wurde von mir so angepasst, dass es thematisch mit dem Design der Webseiten der Arbeitsgruppe Software-Engineering übereintraf. So sollte der akademische Charakter der Umfrage unterstrichen werden.

4.1. Ankündigung

Der genaue Text der Ankündigung kann in Anhang A.1 nachgelesen werden. Eine Herausforderung stellte die Kontaktierung geeigneter Teilnehmer dar, da verteilte Paarprogrammierung im Vergleich zur regulären Paarprogrammierung weniger verbreitet ist und noch nicht wie beispielsweise das Extreme Programming über ausgewiesene Internetgemeinschaften verfügt.

Meine Ankündigung ging gezielt über Kanäle, über die ich hoffte, möglichst viele Programmierer mit Erfahrung in verteilter Paarprogrammierung anzusprechen:

- Zum einem habe ich meine Umfrage über die Mailinglisten und Foren

von unter Abschnitt 3.2 genannten DPP-Werkzeugen angekündigt. Unter anderem hoffte ich die Nutzer der Werkzeuge Sun NetBeans [RS05], Borland JBuilder [JB], Gobby [Gob], SubEthaEdit [HRGW04] und MoonEdit [Dob] anzusprechen.

- Zum anderem wählte ich Internetgemeinschaften, die sich im Allgemeinem mit Extreme Programming oder im Speziellen mit der Paarprogrammierung beschäftigten. Dies beinhaltete unter anderem das Anschreiben der Yahoo-Gruppe *Extreme Programming*[1] mit etwa 8000 Mitgliedern sowie der Usenet-Gruppe `comp.software.extreme-programming`.

- Ergänzend habe ich die Umfrage an die Usenet-Gruppe der Eclipse-Plattform[2] geschickt. Dies geschah in Hinblick auf die Tatsache, dass ich zu diesem Zeitpunkt bereits Eclipse (siehe Abschnitt 5.2.2) als Implementierungsplattform vorsah und damit vorhandene Eclipse-Benutzer eine wesentliche Zielgruppe für das DPP-Werkzeug darstellen würden.

Mit Aussicht auf eine größere Zielgruppe habe ich die Umfrage auf Englisch gestaltet. Die Umfrage wurde am 22.6.2006 durch das erste Aussenden der Ankündigung eröffnet und war über einen Zeitraum von vier Wochen verfügbar. Der Fragebogen wurde vor der Veröffentlichung mithilfe verschiedener Kommilitonen erprobt und anhand dieser Erfahrungen verbessert.

Was gehört in eine Ankündigung?

Nach dem Studium von Ankündigungstexten verschiedener Umfragen erstellte ich eine Liste von Fragen, die der potenzielle Teilnehmer nach dem Lesen einer guten Ankündigung beantworten können sollte. Diese Punkte waren:

- Wer hat die Ankündigung geschrieben?
- Welche Organisation/Institution steht hinter der Umfrage?
- Was ist die allgemeine Motivation der Umfrage?
- Was habe ich persönlich von einer Teilnahme?
- Wie anonym werde meine Angaben behandelt?
- Wie lange dauert das Ausfüllen der Umfrage?
- Wer soll daran teilnehmen?
- Wie lautet die URL?
- Wie kann ich mit dem Autor Kontakt aufnehmen?
- Wie lange ist die Umfrage verfügbar?

[1] http://groups.yahoo.com/group/extremeprogramming/
[2] Gruppe `eclipse.platform` des Eclipse-Usenets (`news.eclipse.org`).

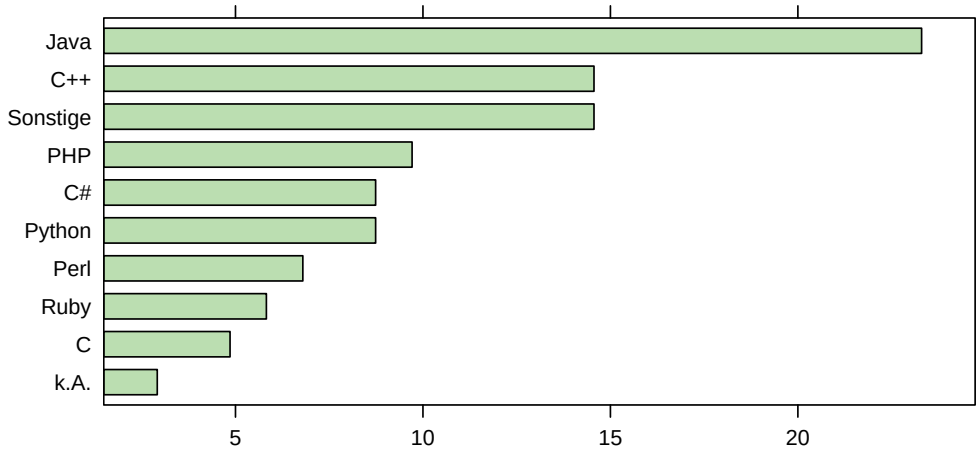

Abbildung 4.1.: Kürzlich verwendete Programmiersprachen (in Prozent).

4.2. Auswertung

Der gesamte Fragebogen kann in Anhang A.2 eingesehen werden. Insgesamt nahmen 44 Personen an der Umfrage teil. Im Folgenden erläutere ich alle Fragen und analysiere die gegebenen Antworten und ihre Relevanz. Alle Fragen waren dabei optional, mussten also nicht beantwortet werden, um die Umfrage abzuschließen. Zur Auswertung habe ich die freie Statistik-Software R [R] verwendet.

4.2.1. Demografische Daten

Die einleitenden Fragen beschäftigten sich mit allgemeinen demografischen Angaben und sollten dabei helfen die Zusammensetzung der Teilnehmer besser einzuschätzen.

Meine erste Frage galt der Erfahrung mit Programmiersprachen („What programming language(s) did you use in the last 6 months?“). Es lagen mehrere Antwortmöglichkeiten mit der Möglichkeit zur Ergänzung vor, Mehrfachantworten waren erlaubt. Die meisten Programmierer antworteten mit Java (23%), gefolgt von den Programmierern, die C++ (15%) und PHP (10%) angaben. Ein nicht unwesentlicher Teil (15%) gab verschiedene Einzelnennungen an, die ich in Darstellung 4.1 unter *Sonstige* zusammenfasse. Der Überhang zu Java-Programmierern könnte sich aus der in Abschnitt 4.1 erwähnten Kontaktaufnahme in der Eclipse Usenet-Gruppe erklären.

Aus meiner zweiten („How many hours per week do you usually spend programming?“) und dritten Frage („How many years of experience in software development do you have?“) folgt, dass der Teilnehmer durchschnittlich über 12,85

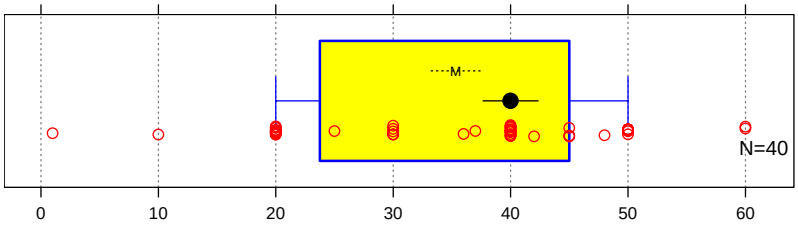

Abbildung 4.2.: Programmier-Wochenstunden.

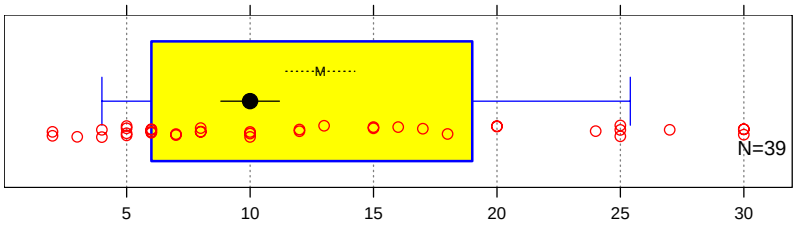

Abbildung 4.3.: Jahre an Programmiererfahrung.

(Standardabweichung: 8,74) Jahre Erfahrung als Software-Entwickler verfügte und im Schnitt 35,3 (Standardabweichung: 13,34) Wochenstunden programmierte. Abbildung 4.2 und 4.3 zeigen diese Verteilungen als Box-Whisker-Plots.

Aus der Frage nach dem Abschluss („Which of these academic degrees is closest to your own?“) ging hervor, dass der Großteil der Teilnehmer über einen Bachelor oder ihm gleichwertigen Abschluss (40%) verfügte. Danach folgten die Teilnehmer mit einem Master oder ihm gleichwertigen Abschluss (25%). Die weitere prozentuale Verteilung der Antworten kann in Abbildung 4.4 betrachtet werden.

Die diesen Bereich abschließende Frage untersuchte die aktuelle Arbeitsbezeichnung. Aufgrund der vielen möglichen Antworten wählte ich hier ein Freitextfeld. Sechs Teilnehmer befanden sich noch in der Schule (7%) oder im Studium (7%).

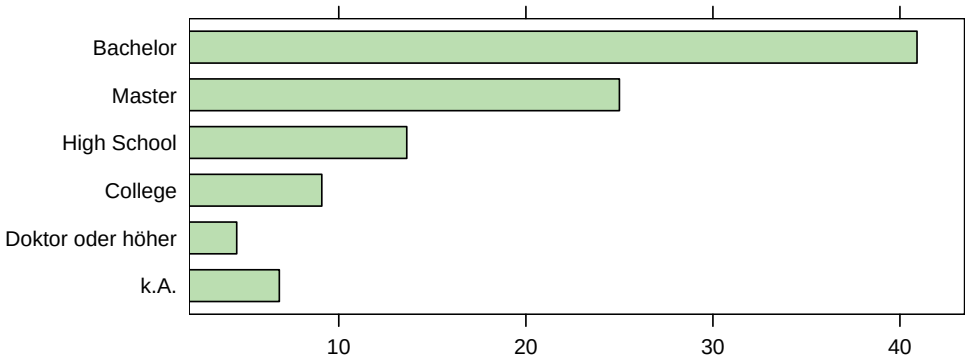

Abbildung 4.4.: Akademische Abschlüsse (in Prozent).

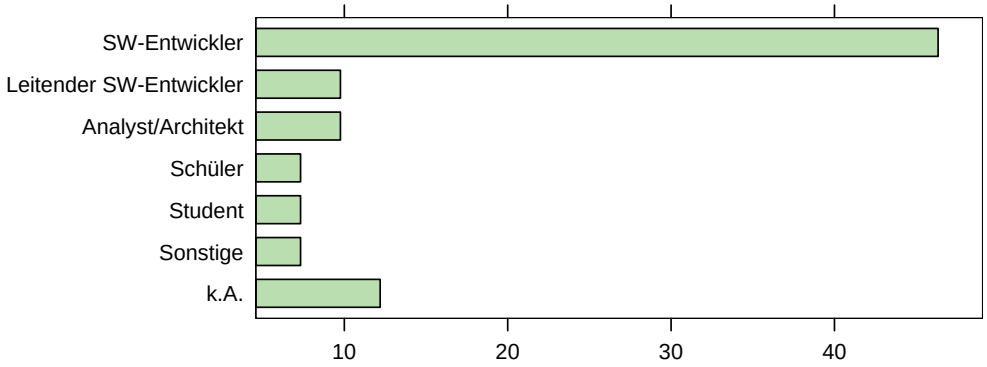

Abbildung 4.5.: Berufe (in Prozent).

Alle anderen 32 Antworten deuteten auf eine professionelle Beschäftigung als Software-Entwickler. Die Verteilung wird in Abbildung 4.5 dargestellt.

4.2.2. Allgemeine Arbeitsweise

Die folgenden Fragen beschäftigen sich mit allgemeinen Aspekten der Arbeitsweise. Meine erste Frage aus diesem Bereich bezog sich auf verwendete Versionsverwaltungssysteme („Which revision control system(s) do you use?“). Verschiedene Antworten mit Möglichkeit eigener Ergänzungen und Mehrfachwahl waren vorgegeben. Motivation dieser Frage war die tendenzielle Bestätigung oder Widerlegung meiner Annahme, dass eine Vielzahl der Projekte mit mehreren beteiligten Entwicklern über eine Versionskontrolle verfügt.

Aus den Antworten resultiert, dass alle Teilnehmer Versionsverwaltungssysteme einsetzten. Die Antwort „I use no revision control system“ wurde nur in der Nennung mit anderen Antworten gewählt. Am Verbreitetsten waren bei den Teilnehmern SVN (Subversion) (38%) und CVS (Concurrent Versions System) (33%). Abbildung 4.6 zeigt eine genauere Aufstellung. Dies scheint meine genannte Annahme zu bestätigen. Ich werde in Abschnitt 5.1.2 nochmals auf diesen Aspekt zurückkommen.

Meine nächste Frage ermittelte, wie viel Anteil das Programmieren im Paar am gesamten Entwicklungsalltag der Person hat („Which percentage of your total programming time, do you spend with pair programming ...“). Abbildung 4.7 zeigt die Verteilung der gegebenen Antworten. Der Durchschnittswert betrug 16.95% (Standardabweichung: 20,76). Nur zwei Teilnehmer gaben an, Paarprogrammierung als vorwiegende Entwicklungsart (jeweils 90% und 80%) zu nutzen. Die Mehrzahl der Teilnehmer schien Paarprogrammierung dagegen nur in bestimmten Phasen zu nutzen. Dies ist insofern überraschend, da die Umfrage nicht unwesentlich auch an Anwender von Extreme Programming herangetragen wurde und die systematische Paarprogrammierung innerhalb von XP zu

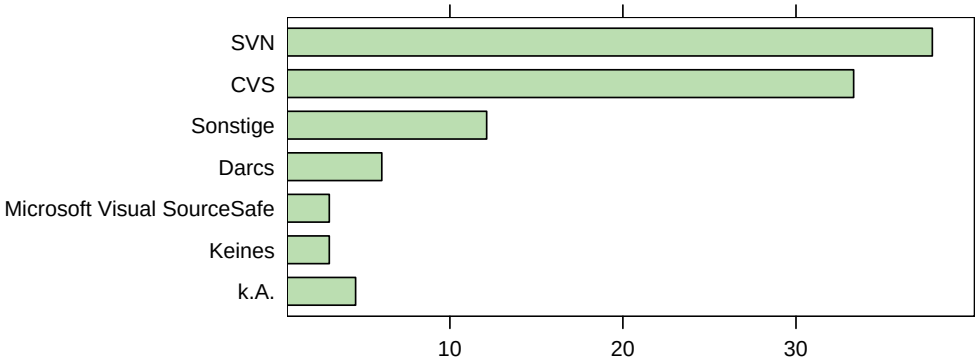

Abbildung 4.6.: Verwendete Versionskontrollsysteme (in Prozent).

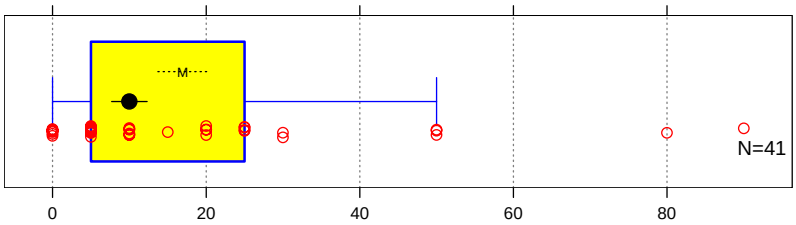

Abbildung 4.7.: Anteil von Paarprogrammierung an gesamter Programmiertätigkeit (in Prozent).

einer der am meisten verbreiteten Techniken zählt (siehe Abschnitt 2.3).

Meine nächste Frage sollte klären, wie wahrscheinlich es ist, dass der Teilnehmer der Umfrage den Code eines Projekts, für welches er zu einer Paarprogrammier-Sitzung eingeladen wurde, bereits auf dem eigenen System installiert hat („If somebody asks you to join a pair programming session how likely is it that you have the project already installed yourself for development? In other words, how likely is it that you both have the code on your own machines?“). Insbesondere hoffte ich mit der Frage meine Erfahrung zu bestätigen, dass Paarprogrammier-Sitzungen sich meist auf gemeinsame Projekte beziehen, die man in irgendeiner Form bereits auf dem eigenen System installiert hat.

Abbildung 4.8 zeigt die prozentuale Verteilung der Antworten. Die 39 Antworten wiesen im Durchschnitt einen Prozentsatz von 72,5% (Standardabweichung: 37,36) auf. Ein Drittel der Teilnehmer gab dabei 100% an. Dieses Resultat betrifft unmittelbar eine wichtige Entwurfsentscheidung, welche sich auf die Übersetzbarkeit von entfernten Projekten in DPP-Sitzungen bezieht. Ich werde diese in Abschnitt 5.1.2 genauer erläutern.

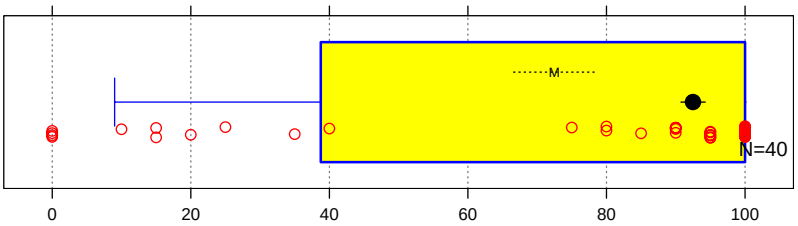

Abbildung 4.8.: Wahrscheinlichkeit, dass vor Beginn einer PP-Sitzung eine gemeinsame Dateibasis vorliegt (in Prozent).

Software	k.A.	S1	S2	S3	S4	S5
Virtual Network Computing (VNC)	16	30	16	16	11	11
SubEthaEdit	14	57	11	0	7	11
Gobby	18	59	5	5	5	9
Emacs (using make-frame-on-display)	16	30	39	9	2	5
Sun Java Studio/NetBeans	16	23	34	20	5	2
Borland JBuilder	14	32	34	14	5	2
MoonEdit	18	59	7	9	5	2
Symantec pcAnywhere	20	34	34	5	5	2
Microsoft NetMeeting	16	23	32	25	5	0
Sangam (Eclipse-Plugin)	18	68	7	7	0	0
DocSynch	20	59	16	5	0	0
ACE	20	64	11	5	0	0

Tabelle 4.2.: Fragenblock Kommunikationsmittel absteigend geordnet nach Wertung (in Prozent).

4.2.3. Erfahrung mit verteilter Paarprogrammierung

Die folgenden Fragen beschäftigen sich mit der eventuell vorhandenen Erfahrung im Bereich der verteilten Paarprogrammierung. Frage 9 bat den Teilnehmer der Umfrage aus einer festgelegten Liste von DPP-Werkzeugen, diejenigen Werkzeuge auszuwählen, die er selber kennen oder einsetzen würde. Dabei sollten die Teilnehmer ihre Bewertung anhand folgender Skala vornehmen:

S1 Don't know
S2 Know but haven't tried
S3 Have tried, but did not like
S4 Have tried and liked it
S5 Use it regulary

Tabelle 4.2 zeigt die gegebenen Antworten, die Einträge sind nach der Höhe der Wertungen geordnet. Bei den Teilnehmern wurden die Werkzeuge *VNC*, *SubEthaEdit* und *Gobby* am höchsten eingestuft. Es fällt allerdings auf, dass auch sie von nur bis zu 11% der Teilnehmer regelmäßig genutzt wurden. Nur

bei den gerade genannten Werkzeugen sowie bei dem Werkzeug *pcAnywhere* überwiegten die Teilnehmer, die das Werkzeug probierten und als gut befanden (S3+S4). Bei allen anderen Werkzeugen überwiegten die Teilnehmer, die nicht zufrieden waren (S5).

Von besonderem Interesse für mich waren auch die Antworten zu dem Sangam-Plugin [HRGW04], da dieses sich ebenfalls an die Elicpse-Umgebung richtet. 68% aller Teilnehmer gaben an noch nicht von diesem Werkzeug gehört zu haben. 14% der Teilnehmer kanten es, von den 7% die es eingesetzt hatten, bewertete es aber niemand als gut (S5).

Das von mir verwendete Umfragen-System phpESP [ESP] erlaubte bei Bewertungsfragen keine eigenen Ergänzungen durch den Teilnehmer. Daher beschäftigte sich die zehnte Frage mit Werkzeugen, die womöglich in der vorherigen Liste gefehlt hatten. Ziel dieser Frage war einerseits festzustellen, wie zufrieden stellend und vollständig die Antwortmöglichkeiten der vorherigen Frage waren und gleichzeitig auf bisher noch nicht von mir bedachte Ansätze zu stoßen („If you know of any other tools that can be used for distributed pair programming please provide them here“).

Durch das Freitextfeld hoffte ich den Teilnehmer auch zu komplexeren Vorschlägen und Antworten zu ermutigen. Neun Teilnehmer machten von der Möglichkeit einer Angabe Gebrauch. Ich gebe ihre Antworten unverändert an:

> Croquet & Opentalk
>
> GNU screen (Have tried, and liked it, before Gobby was created; there is the restriction of only one at one time coding, so it is similar to real pair programming)
>
> Remote Desktop, SMART Board
>
> screen - the unix command line tool that lets you keep a shell session running over long periods of time has a multi-user mode where two people can be using the same shell, connected from two different terminals. iChat - using text, audio and video chat, we especially use video chat during distributed pair programming IRC - using a local irc server
>
> Store in VisualWork VisualStudio
>
> UNA
>
> vim + ssh vnc is nice but not for programming just because the delay through the net Though i use gobby regulary i doenst like it

> that much ;)
>
> We use a modified *Visual Age for Smalltalk* that keeps n development images in sync.
>
> I did distributed shared screen programming for about 2 years two startups back. The entire team was distributed, but was also heavily into XP. We did not pair program as much as we should have, but when we did, we used a screen session with :multiuser-on set and VOIP. In my opinion voice is essential for a distributed pair programming session; without it, you have to stop to type in a chat window, and it's not obvious to the other person whether you are thinking about editing or not. Screen has advantages and disadvantages; the advantage is that it works, and works everywhere. The main disadvantage is that there's only one cursor, unlike SubEtha-Edit. IMO, a solution for distributed pair programming will not be successful unless there is an implementation for at least Emacs, perhaps vi, and a native Mac (probably also windows) editor.

Besonders die letzte Antwort war hier von Interesse: Sie beinhaltete mehrere Aspekte, die ich bereits bei meinen Erläuterungen zur verteilten Paarprogrammierung angeführt hatte (siehe Abschnitt 3). Der Teilnehmer bestätigte unter anderem die hohe Flexibilität des Desktop Sharing und den höheren Grad an Parallelität bei Collaboration Awareness. Die Wichtigkeit einer Stimmübertragung wird durch eine noch folgende Frage ergründet.

Frage 11 beschäftigte sich mit den Kontaktmöglichkeiten, über die verteilte Paarprogrammier-Sitzungen eingeleitet werden können („If you've already used distributed pair programming: Which mean(s) of communication do you usually use to tell your partner that you want to initiate a distributed pair programming session?"). Es waren mehrere Antwortmöglichkeiten vorgegeben, der Teilnehmer hatte die Möglichkeit zur Ergänzung und Mehrfachwahl. Die Teilnehmer der Umfrage gaben hauptsächlich *Instant Messaging* (30%) und *Directly in person* (16%) an. Abbildung 4.9 illustriert die Verteilung noch genauer.

Die Beliebtheit von Instant Messaging zum Verabreden von Sitzungen erscheint nicht überraschend, da sie ein gutes Medium zum schnellen Austausch von Informationen darstellen. Für die Nutzung bei der verteilten Paarprogrammierung scheint darüber hinaus ihre Anwesenheitsanzeige von besonderem Vorteil zu sein.

Die Tatsache, dass die Antwort *Direkt in Person* („Directly in person") an zweiter Position folgt, bestätigt dass verteilte Paarprogrammierung anscheinend auch bei vorhandener räumlicher Nähe[3] eingesetzt wird. Auf diesen Aspekt geht auch die folgende Frage 14 genauer ein.

[3] Im Englischen auch als *colocation* bezeichnet.

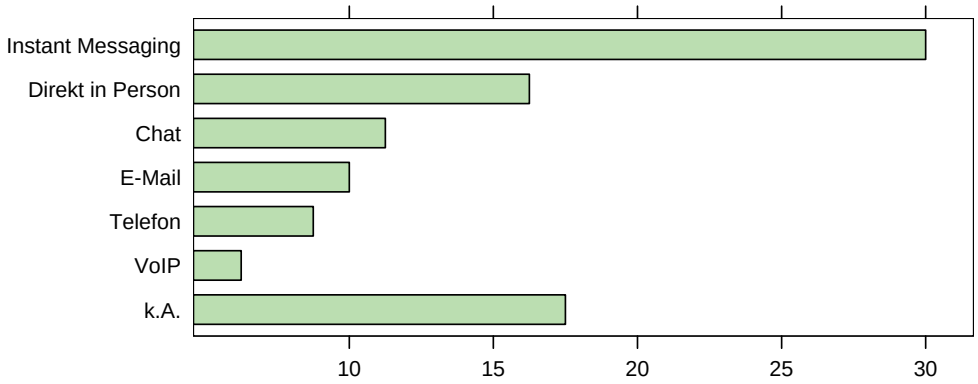

Abbildung 4.9.: Kommunikationsmittel zum Einleiten einer verteilten Paarprogrammier-Sitzung (in Prozent).

Die Möglichkeit zur Ergänzung nutzten 5 Teilnehmer (6%), um die Antwort *VoIP* (Voice over IP) anzugeben. Aus dieser Anzahl schließe ich, dass diese Antwort von Anfang an bei den vorgegebenen Antworten hätte vorhanden sein sollen. Schlimmstenfalls könnte das Fehlen dieser Antwortmöglichkeit auch zu einer Unklarheit bezüglich des Punkts *Telefon* geführt haben. Es ist denkbar, dass Teilnehmer diese Antwort gewählt haben, obwohl sie eigentlich *VoIP* meinten.

Meine nächste Frage erforschte die Entwicklungsphasen, in denen die Teilnehmer der Umfrage hauptsächlich die verteilte Paarprogrammierung einsetzen („If you've already used distributed pair programming: For which kind of task(s) do you usually use distributed pair programming?"). Eine Liste mit vorgegebenen Antwortmöglichkeiten mit der Möglichkeit zur Erweiterung und Mehrfachauswahl war vorgegeben.

Eine genauere Illustration der Verteilung kann in Abbildung 4.10 betrachtet werden. Die drei meist genannten Antworten entfielen auf *Schwerer Code* (23%), *Debugging* (18%) und *Entwurf* (16%). Auf den Aspekt *Entwurf* bin ich schon in Abschnitt 2.4 eingegangen. Er wurde dort bereits als potenziell guter Einsatzzweck für Paarprogrammierung identifiziert.

Darüber hinaus ist zu sehen, dass die Teilnehmer der Umfrage DPP mehr für komplexen und schwierigen Code (23%) als für einfachen Code (12%) einsetzten. Dies bestätigt ebenfalls die in Abschnitt 2.4 angeführte Beobachtung, dass anscheinend eher schwerer und komplexer Code im Paar implementiert wird als einfacher Code. Interessant ist außerdem, dass *Debugging* (18%) einen relativ hohen Anteil erhalten hat. Die gute Eignung zum Finden von Problemstellen wurde ja bereits von Cockburn und Williams postuliert [CW00] (siehe auch Abschnitt 2.1). Zwei Teilnehmer ergänzten die Angabe der Phasen außerdem noch mit den Einzelnennungen „distribution of experiences" und „fun". Beide Aspekte hatte ich ebenfalls schon in dem Abschnitt zur Paarprogrammierung

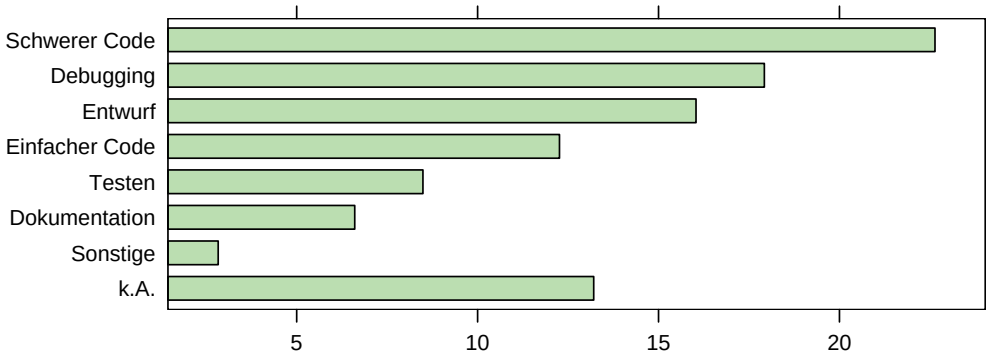

Abbildung 4.10.: Einsatzzwecke für verteilte Paarprogrammierung (in Prozent).

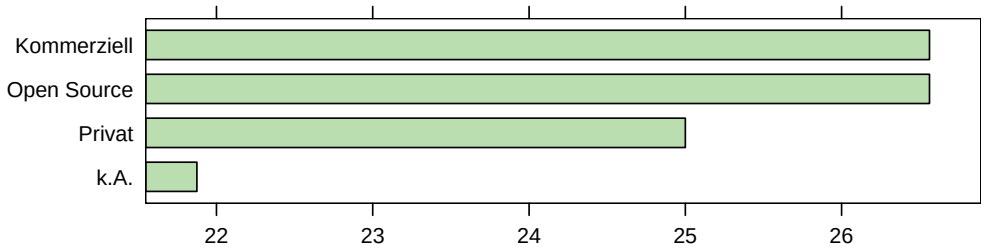

Abbildung 4.11.: Art der Projekte, bei denen verteilte Paarprogrammierung eingesetzt wird (in Prozent).

angeführt.

Die nächste Frage beschäftigte sich mit dem Umfeld der Projekte in denen DPP eingesetzt wird („If you've already used distributed pair programming: On which kind of projects do you use distributed pair programming?"). Mehrfachwahl war hier erlaubt. Die Antworten entfielen hier gleichmäßig auf die drei Antwortmöglichkeiten *kommerziell* (27%), *open-source* (27%) und *privat* (25%).

Im Vorfeld hatte ich hier eher einen Hang zu Open-Source-Projekten erwartet, da diese häufig von virtuellen Teams realisiert werden und somit eine potenziell interessante Zielgruppe für die verteilte Paarprogrammierung darstellen (siehe auch Abschnitt 3.1.1). In diesem Fall konnte diese Vermutung nicht bekräftigt werden. In Abbildung 4.11 wird die Verteilung nochmals dargestellt.

Frage 14 ermittelte, wie weit entfernt der Partner einer verteilten Paarprogrammier-Sitzung üblicherweise sitzen würde („If you've already used distributed pair programming: How far is your distributed pair programming partner usually away?"). Wie in Abschnitt 3.1.1 erläutert, impliziert verteilte Paarprogrammierung nicht, dass die Partner weiträumig getrennt sein müssen. Es kann zum Beispiel aufgrund von unpassenden Arbeitsplätzen auch attraktiv sein, verteil-

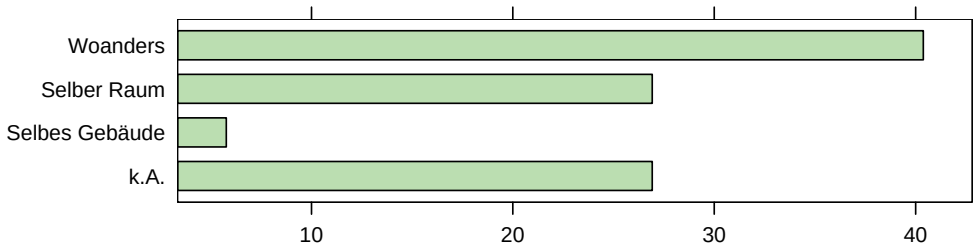

Abbildung 4.12.: Räumliche Distanz zwischen den Partnern der DPP-Sitzung (in Prozent).

te Paarprogrammierung bei vorhandener räumlicher Nähe zu betreiben (siehe auch Abschnitt 2.2).

Der Zweck von Frage 14 war daher zu klären, welches der beiden Szenarios - räumliche Nähe oder Trennung - bei den Teilnehmern der Umfrage häufiger vorlag. Dabei entfielen 40% der Stimmen auf die Wahl *Woanders*, 27% der Stimmen auf *Selber Raum* und 6% der Stimmen auf *Selbes Gebäude*. Abbildung 4.12 illustriert diese Verteilung. Wenn man die Antwortmöglichkeiten *Selber Raum* und *Selbes Gebäude* als leicht herzustellende räumliche Nähe wertet (colocation), dann bestätigt das Resultat anscheinend die Verbreitung verteilter Paarprogrammierung bei räumlicher Nähe.

Die nächste Frage betraf die übliche Anzahl von Partnern während einer verteilten Paarprogrammier-Sitzung („If you've already used distributed pair programming: How many people besides yourself are usually part of one distributed pair programming session?"). Während an regulären Paarprogrammier-Sitzungen alleine schon durch die praktischen Gegebenheiten der Arbeitsplätze (siehe auch Abschnitt 2.2) üblicherweise nur zwei Entwickler beteiligt sind, bieten viele Werkzeuge zur verteilten Paarprogrammierung die Unterstützung einer höheren Anzahl von Teilnehmern. Mit dieser Frage wollte ich klären, ob diese Funktionalität auch häufig angenommen wird oder ob die Mehrzahl der Sitzungen sich dennoch aus zwei Teilnehmern zusammensetzt (siehe auch Abschnitt 3.1.1).

20 Antworten (46,5% bezogen auf die Gesamtmenge aller 43 Teilnehmer) gaben Sitzungen mit einfachen Paaren an, 7 Antworten (14,0% bezogen auf die Gesamtmenge) wiesen auf Sitzungen mit drei Teilnehmern hin und zwei Antworten (4,7% bezogen auf die Gesamtmenge) deuteten auf Sitzungen mit vier Teilnehmern hin. Im Durchschnitt entspricht dies 1,58 Teilnehmern (Standardverteilung: 1,07). Dies bestätigt anscheinend meine Vermutung, dass in der verteilten Paarprogrammierung trotz anderweitiger Möglichkeiten hauptsächlich die klassische Paarkonstellation gewählt wird (siehe auch Abschnitt 1).

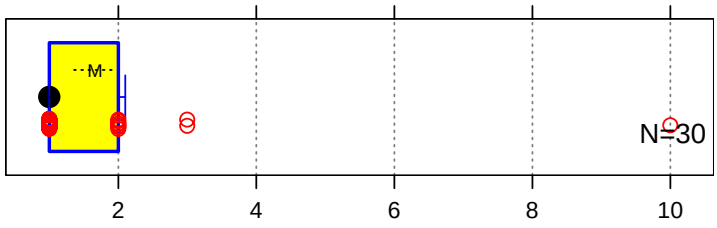

Abbildung 4.13.: Durchschnittliche Teilnehmerzahl in einer DPP-Sitzung.

Es bleibt jedoch anzumerken, dass mit 29 von 43 Teilnehmern im Vergleich zu den anderen Fragestellungen ein auffällig kleiner Teil diese Frage beantwortet hat. Ich schließe daraus, dass die Frage womöglich nicht genau genug formuliert war und daher von einigen Teilnehmern übergangen wurde. Ein potenzielles Problem mit dieser Frage war mir bereits bei einem früheren Probedurchgang aufgefallen. Trotz der darauf folgenden Umformulierung scheint die Frage allerdings noch immer nicht ideal formuliert gewesen zu sein. So bleibt auch unklar, ob die einzelne Stimme für die Wahlmöglichkeit *10 (besides yourself)* tatsächlich im Sinne der Frage gemeint war oder aus einem Missverständnis herrührte. Da niemand auf die Antwortmöglichkeit *More than 10 (besides yourself)* zurückgriff, benutze ich einen Box-Whisker-Plot als Abbildung 4.13.

4.2.4. Anforderungen an DPP-Werkzeuge

Die folgenden Fragen ermittelten Anforderungen, die der Teilnehmer an ein optimales Werkzeug für verteilte Paarprogrammierung stellen würde. Dazu sollte er mittels folgender Skala seine Präferenzen für verschiedene Funktionalitäten angeben:

R1 Counter-Productive
R2 Unnecessary
R3 Nice to have
R4 Important
R5 Essential

Um die Teilnehmer besser durch die Umfrage zu führen, habe ich die Fragen in Blöcke zusammengefasst. Darüber hinaus hoffte ich so, dass der Teilnehmer bei seiner Antwort alle Fragen des Blocks in Relation zueinander beantworten würde.

Ich greife im Folgenden auf Tabellen zurück, um die Antworten zu erläutern. Obwohl die Antworten nummeriert sind, muss ich hier auf Box-Whisker-Plots verzichten, da die Unterschiede zwischen den Rängen nicht normierbar sind, das

Funktionalität	k.A.	R1	R2	R3	R4	R5
Chat	7	5	7	20	16	45
Stimme	7	0	7	9	36	41
Whiteboard	7	0	11	45	27	9
Video	7	14	45	23	9	2

Tabelle 4.3.: Fragenblock zu Kommunikationsmitteln (in Prozent). Die Funktionalitäten sind nach der Summe der Nennungen von R4 und R5 absteigend geordnet.

heißt, es lässt sich nicht bestimmen, ob zum Beispiel der Unterschied zwischen den Angaben R2 und R3 genau so groß zu werten ist wie der Abstand zwischen R3 und R4.

Kommunikationsmittel

Dieser Block beschäftigte sich mit Kommunikationsaspekten während einer verteilten Paarprogrammier-Sitzung. Ich stellte drei Fragen, die sich mit Chats („You can chat with your partner (e.g. text chatting, instant messaging etc.“), Stimmübertragung („You can talk to your partner by voice (e.g. by using a head set, telephone etc.)“), Videoübertragung („You can see your partner via video (e.g. by using a web cam).“) und Whiteboard-Funktionalitäten („You have a whiteboard editor which allows you to draw geometric shapes and diagrams, which your partner can see.“) beschäftigten.

Tabelle 4.3 zeigt die von den Teilnehmern vorgenommene Bewertung. Am häufigsten wurde die höchste Wertung an *Chat* und *Stimme* vergeben, beiden Funktionalitäten scheint eine besondere Bedeutung zuzukommen. An dritter Stelle folgte eine *Whiteboard*-Funktionalität, am wenigsten wurde Wert auf *Video*-Übertragung gelegt. Zur Begründung einer Entwurfsentscheidung komme auf dieses Resultat nochmal in Abschnitt 5.1.1 zurück.

Driver-Observer-Interaktion

Dieser Fragenblock behandelte Benutzerrollen der verteilten Paarprogrammierung sowie verschiedene Grade der Parallelität, welche ich in Abschnitt 3.1.1 definiert habe. Die Resultate sind in Tabelle 4.4 dargestellt.

Meine erste Frage galt einer Funktionalität, bei der dem Observer erlaubt sein soll eigenständig im Code zu stöbern und den Sichtbereich des Drivers zu verlassen („Observers are allowed to browse the code base independently, even while the driver is coding at another part [in contrast to regular pair programming, where two programmers must share one view]“). Dies entspricht der von mir definierten *Parallelität auf Sichtebene*, in der Tabelle ist diese Frage unter dem Begriff „Sichtparallelität“ angegeben. Diese Funktionalität wurde größtenteils

Funktionalität	k.A.	R1	R2	R3	R4	R5
Rollenwechsel	14	7	11	14	23	32
Sichtparallelität	14	5	5	34	36	7
Schreibparallelität	14	18	14	32	14	9

Tabelle 4.4.: Fragenblock zu Driver-Observer-Interaktionen (in Prozent). Die Funktionalitäten sind nach der Summe der Nennungen von R4 und R5 absteigend geordnet.

als „wichtig“ (36%) und „nice to have“ (34%) angesehen.

Die zweite Frage galt der Möglichkeit von Rollenwechseln zwischen Driver und Observer („One participant is allowed to be driver at a time and switching driver state is allowed.“). Wie in der Einleitung (Abschnitt 1) angegeben, ist dies nicht Teil der von mir verwendeten Definition für verteilte Paarprogrammierung. Meine Vermutung ist, dass dies dennoch ein wesentlicher Aspekt der Paarprogrammierung ist. Die Antworten auf diese Frage scheinen dies zu bestätigen. Mit 32% der Wahlen für R5 findet diese Funktionalität einen deutlichen Zuspruch.

Die dritte Frage galt der Möglichkeit, dass mehrere Nutzer gleichzeitig Schreibrechte annehmen können („Multiple participants are allowed to be drivers at the same moment, i.e. they can enter text concurrently at the same time.“). Dies entspricht der von mir definierten *Parallelität auf Sichtebene* und wird in der Tabelle als „Schreibparallelität“ angegeben. 9% der Teilnehmer halten diese Funktion für „essenziell“. Die meisten Teilnehmer (32%) bewerteten diese Funktion nur als „nice to have“.

Im Rahmen meiner Entwicklung eines DPP-Werkzeugs halte ich ausgehend von diesen Resultaten den Rollenwechsel und die Unterstützung einer *Parallelität auf Sichtebene* für wichtige Anforderungen. Ich gehe in Abschnitt 5.1.4 näher auf die diesbezüglichen Entwurfsentscheidungen ein.

Awareness

Der größte Fragenblock erforschte den Aspekt der Awareness (siehe auch Abschnitt 3.1.2). Die Antworten der Teilnehmer sind in Tabelle 4.5 dargestellt.

Die ersten Fragen sollten eruieren wie viele zusätzliche Informationen nach Meinung der Umfrageteilnehmer, während einer Sitzung übertragen werden sollten, um die Awareness zu verbessern. Die ersten drei Fragen befassten sich dabei mit Informationen über den Driver, die an alle Observer weitergegeben werden könnten. Dies betraf die folgenden drei Aspekte:

- Allen Observern wird mitgeteilt, welche Datei der Driver aktuell geöffnet und fokussiert hat („You can see which file a driver has currently opened

and focused [even while he currently isn't editing the file].“).

- Allen Observern wird mitgeteilt, wo sich der Cursor des Drivers aktuell befindet („You can see the cursor position and text selections of the driver [even while he currently isn't editing the file].“).

- Allen Observern wird mitgeteilt, was der Sichtbereich des Drivers ist („You can see at the exact viewport [i.e. the top and bottom code lines] the driver is currently looking at.“). Mit Sichtbereich ist der Teil einer Datei gemeint, der in einem Editor sichtbar ist (beispielsweise Zeile 10 bis 40).

Aus den Antworten zu diesen Fragen ließ sich eine Favorisierung (30% der Stimmen für „essenziell“) der Anzeige der fokussierten Datei ablesen. Auf die Anzeige des Cursors und des Sichtbereichs entfielen zwar weniger Wertungen, jedoch liegt auch bei ihnen insgesamt eine positive Tendenz vor. Wenn man R1 und R2 negativ bewertet, R3 als neutral und R4 und R5 als positiv, so liegt jeweils ein Hang zur positiven Seite vor. Bei der Anzeige der Datei stehen 64% positive Wertungen 16% negativen Wertungen gegenüber und bei der Anzeige des Cursors stehen 62% positive Wertungen gegenüber 16% negativen Wertungen. Bei der Anzeige des Sichtbereichs stehen 41% positive Wertungen, 16% negativen Wertungen gegenüber. Ausgehend von diesen Daten bewerte ich diese Funktionalitäten als wichtig bis sehr wichtig.

Die drei nächsten Fragen betrachteten diese Funktionalitäten aus umgekehrter Sicht: Wie viele Awareness-Informationen sollten von einem oder mehreren Observern an den oder die Driver übermittelt werden? In diesem Bereich wurde die Übertragung des Cursors favorisiert. Interessant ist hier aber vor allem eine Tendenz zu den eher negativen Wertungen. Ordnet man die Wertungen wie zuvor in positive und negative Tendenzen, so liegt bei allen Funktionen ein deutlicher Überhang zur negativen Seite vor.

Ausgehend von diesem Resultat halte ich die zuerst genannten Funktionalitäten für eine sinnvolle Anforderung an das von mir zu entwickelnde DPP-Werkzeug. Die anderen Funktionalitäten erscheinen mir zumindest für die Entwicklung einer ersten Version als weniger wichtig. Ich gehe auf meine Entwurfsentscheidungen zur Awareness in Abschnitt 5.1.3 genauer ein.

Die letzten drei Fragen aus diesem Bereich galten Funktionalitäten, die im Gegensatz zu den vorherigen Fragen nicht auf zusätzlichen Informationen basieren, sondern auf der Art der Verwertung von Daten. Die erste Frage galt hier einer Verfolgermodus-Funktionalität, bei der ein Observer automatisch dem Sichtbereich eines Drivers folgen können soll („You can enable a *follow mode* which allows you to automatically follow the navigation and view of a driver.“). Dies kommt der Durchsicht nahe, welche bei regulärer Paarprogrammierung vollzogen wird. 30% der Stimmen entfielen auf die höchste Wertung und auch

Funktion	k.A.	R1	R2	R3	R4	R5
Awareness über den Driver						
Aktuelle Datei	16	0	11	9	34	30
Cursor	16	0	9	14	39	23
Sichtbereich	16	0	16	27	27	14
Awareness über den Observer						
Aktuelle Datei	18	0	32	32	14	5
Cursor	16	2	30	27	14	11
Sichtbereich	16	2	36	30	11	5
Andere Aspekte						
Verfolgermodus	16	0	5	20	30	30
Historie	16	0	16	52	11	5
Sitzungsliste	16	0	23	55	5	2

Tabelle 4.5.: Fragenblock zur Awareness (in Prozent).

insgesamt lag eine Tendenz zu den höheren Wertungen vor. Aufgrund dieses Resultats erscheint mir die Implementierung dieser Funktionalität ebenfalls als eine sinnvolle Anforderung.

Die nächste Frage befasste sich mit der Funktionalität einer Historie, in der die Aktionen des Drivers protokolliert werden sollten, damit sie auch im Nachhinein betrachtet werden können. Sie würde erlauben Aktionen des Drivers, die man verpasst hatte, nachträglich noch einmal betrachten zu können („The actions of a driver are recordable so that you can jump back in time and replay events (e.g. text changes in files) if you missed something.“). Wie in Abschnitt 3.3.3 erläutert, droht gerade beim Desktop Sharing die Gefahr, dass Aktionen des Drivers übersehen werden. Die Antworten waren hier insgesamt ausgeglichen. Es entfielen 5% der Stimmen auf die höchste Wertung und 6% der Stimmen auf die zweihöchste Wertung, allerdings auch 13% Stimmen auf die Wertung *unwichtig*. Insgesamt deuten die Antworten auf keine große Wichtigkeit dieser Funktionalität. Daher habe ich mich gegen die Aufnahme einer entsprechenden Anforderung für das von mir zu entwickelnde DPP-Werkzeug entschieden.

Die letzte Frage in diesem Bereich eruierte die Attraktivität einer Sitzungsliste, in der alle in der Umgebung[4] laufenden Sitzungen angezeigt werden sollten („You can see a list of all currently running distributed pair programming sessions in your environment.“). Die Antworten deuteten ebenfalls auf eine eher geringe Priorität dieser Funktionalität. Die meisten Stimmen (55%) entfielen auf „nice to have“. Ich komme hier ebenfalls zu dem Entschluss diese Funktionalität eher niedrig zu priorisieren.

[4]Gemeint sind zum Beispiel alle Sitzungen in einem lokalen Firmen-Netzwerk oder eines bestimmten Jabber-Servers.

Funktion	k.A.	R1	R2	R3	R4	R5
Lauf	14	2	7	23	18	36
Übersetzung	14	2	7	25	23	30
Demonstration	16	5	11	43	20	7

Tabelle 4.6.: Fragenblock zur Funktionsvielfalt (in Prozent). Die Funktionalitäten sind nach der Summe der Nennungen von R4 und R5 absteigend geordnet.

Übersetzen und Ausführen

Der letzte Fragenblock galt den Fähigkeiten des Übersetzens und Ausführens von erstellten Artefakten. Tabelle 4.5 zeigt die Resultate.

Die erste Funktionalität betraf die Möglichkeit, dass jeder Teilnehmer der Sitzung selbstständig in der Lage sein sollte das Projekt zu übersetzen und Übersetzungsfehler angezeigt zu bekommen („Every participant of the session (and not only the session host) is allowed to compile the project and can see compile time errors.“). Diese Funktion wurde von den Teilnehmern der Umfrage stark bestätigt. Ein deutlicher Überhang zu den hohen Wertungen ist zu erkennen.

Als Nächstes sollte eine Funktionalität abgefragt werden, die jedem Teilnehmer der Sitzung erlauben würde das erstellte Programm nach der Übersetzung selbstständig auszuführen („Every participant of the session (and not only the session host) can run the project.“). Der Hang zu hohen Wertungen fällt bei dieser Frage noch höher aus als bei der vorherigen Frage. Dies ist insofern bemerkenswert, da die Lauffähigkeit eines Programms eine Übersetzbarkeit des Selbigen eigentlich schon voraussetzen müsste. Bei nachträglicher Betrachtung der beiden Fragen muss ich feststellen, dass diese klarer formuliert werden müssen, um diese Unklarheiten zu beseitigen. Insgesamt halte ich die deutliche Tendenz der Antworten dennoch für relevant.

Als Drittes prüfte ich die Möglichkeit zu Demonstrationen („You can see how your partner is running the project (e.g. what buttons he/she is clicking on the desktop application that you are developing). “). Wie in Abschnitt 3.3.2 erläutert, ist dies eine Stärke des Ansatzes *Desktop Sharing*. Die Bestätigung dieser Funktionalität fällt geringer aus als die der beiden zuvor Genannten. Während dort 36% bzw. 30% Stimmen auf die höchste Wertung entfielen, fielen hier 7% der Stimmen auf diese Wertung. Die meisten Stimmen entfielen auf „nice to have“.

Anhand dieser Ergebnisse erscheinen mir die Funktionalitäten zur Übersetzung und zum Lauf als wichtige Anforderungen für ein DPP-Werkzeug. Die Fähigkeit zur Demonstration priorisiere ich anhand dieser Daten niedriger. Dieses Resultat schlägt sich auch in einer Entwurfsentscheidung wider, welche ich in Abschnitt 5.1.2 erläutere.

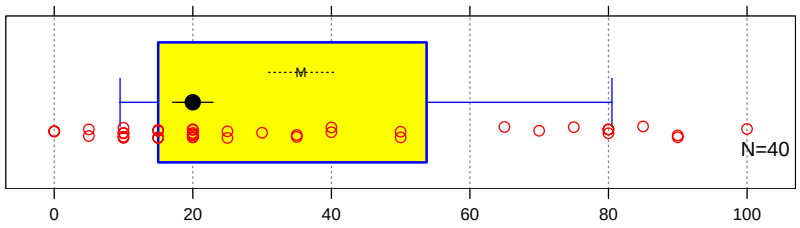

Abbildung 4.14.: Gewünschter Anteil verteilter Paarprogrammierung an gesamter Programmiertätigkeit (in Prozent).

4.2.5. Abschluss und freie Anmerkungen

Eine erste abschließende Frage sollte klären, wie stark der Teilnehmer DPP nutzen würde, wenn ein Werkzeug alle von ihm als wichtig bewerteten Funktionen bieten würde. Abbildung 4.14 zeigt die Antworten als Box-Whisker-Diagramm.

Der Durchschnitt der Antworten betrug 35% (Standardabweichung 29). Dieser Wert liegt deutlich über dem Wert, denn die Teilnehmer für ihre aktuelle Nutzung von Paarprogrammierung angegeben hatten (17% bei einer Standardabweichung von 21). Dies deutet anscheinend darauf hin, dass für die Teilnehmer eine verteilte Paarprogrammierung zumindest potenziell attraktiver ist als reguläre Paarprogrammierung.

In gewisser Weise war mit dieser Angabe zu rechnen, da sich der Fragebogen besonders an Programmierer mit Erfahrung in verteilter Paarprogrammierung richtete. Bei einer solchen Zielgruppe war davon auszugehen, dass diese wohl besondere Vorteile im verteilten Paarprogrammieren sieht. Ob dieser Vorteil im Fehlen einer Möglichkeit zur regulären Paarprogrammierung begründet liegt (siehe auch Abschnitt 3.1.1) oder andere Gründe hat, kann der Fragebogen nicht klären.

Frage 21 ermittelte, auf welche Art der Teilnehmer von der Umfrage erfahren hatte. Die Motivation der Frage war vor allem die Optimierung zukünftiger Veröffentlichungen von Fragebögen. Die meisten Teilnehmer (50%) gaben an über eine Mailingliste von der Umfrage erfahren zu haben. Abbildung 4.15 zeigt die genaue Verteilung.

Die letzten beiden Fragen waren als freie Textfelder gestaltet und sollten den Teilnehmer dazu ermutigen Kritik und Bemerkungen zum Fragebogen und dem Forschungsfeld zu äußern.

Frage 22 galt allgemeinen Anmerkungen zu verteilter Paarprogrammierung („Are there any aspects of distributed pair programming that you find to be especially important and that weren't checked on the former questions? If yes,

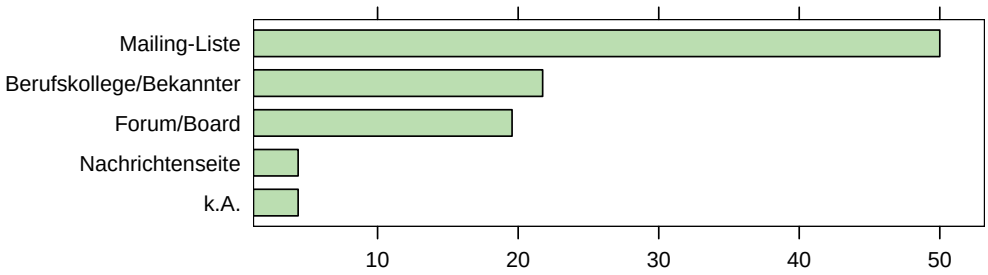

Abbildung 4.15.: Art der Kenntnisnahme der Umfrage (in Prozent).

which are these?“). Da die Antworten sehr unterschiedlich ausfielen, erschien eine Kategorisierung nicht sinnvoll. Ich gebe sie daher nachfolgend unverändert wider.

Die folgenden Antworten bezogen sich vor allem auf technische Aspekte und auf das *Desktop-Sharing*. Die Aspekte Flexibilität und verschiedene technische Probleme der Ansätze habe ich in Abschnitt 3.3 erläutert:

> For me the most essential feature is rapid screen refreshes especially in a transnational distributed pair session. One aspect not touched on which is not limited to distributed pairing is screen size and resolution. Basically the more the better as this allows requirements, issue tracking, etc. to be available as well as the IDE.

> Overall usability of the editor application - Easy setup, generic application suitable for all kind of editing scenarios

Die folgende Antwort schilderte eine Anforderung, welche von mir aber nicht weiter berücksichtigt wurde:

> No central server for distributed pair programming should be required but installing one in cases it is helpful should be possible, though.

Die folgenden und damit meisten Antworten galten sozialen Aspekten:

> about the difficulties (bariers, obstacles) or expactations of each partner the need of administrating the pair programming (some times we done it forced, now we only do it spntaneously)

> do you think it could come to a driver-battle, when only one person is allowed to edit the current file and all other are observers, who want to be the driver? are the others able to change their view or is their views focussed to the driver's one. on the other hand, when

> everyone can edit the files and all of them see the changes from eachother is it possible that they still group together to watch all on the same piece of code to solve a more difficult problem or are they trying to „compete“ to themselves to solve it or more dramatically ignore it? distributed programming seems useful to me, but the programmers have to be trained or „scheduled“ to program in a distributed environment. maybe some problems can occur (of a social kind?) that one is not aware of. i think there have to be a simple set of rules, that all agree with. maybe it's useful to configure the tool to let the users or a supervisor decide, which of the possible environments is the most productive. -> example: a lead programmer could be an observer and some others too, some of the participants can be observers and can not directly contribute to the code, and have to ask permission to break a current session to get the driver status temporarily. distributed programming could be a more useful to actually teach others to code, when the code session can be replayed at a later point. most of the problems can be easily solved with communication tools i guess. should be enough for now :D

> How difficult it is to find two programmers that will work together in a pair programming environment.

> human brain limits: differences in focus/distraction level when programming alone and with peers (partially covered by „hard/simple programming task“)

> Management buy in and encouragement i.e. an environment where it is both encouraged and developed

> Personal relationship, trust and respect between the people working together. I.e. „that social thing“.

> Time span to switch between driver/observer status.

Frage 23 fragte abschließend nach Kritik zum Umfragebogen selber („If you have any comments/opinion on the quality of this survey or on questions that you missed, please provide them here“). Ich gebe im Folgenden alle Antworten wider.

Zwei Teilnehmer kritisierten eine zu hohe Konzentration auf Dateien, als zentrale Ressource:

> The nature of some questions assumes that file based development is the only alternative. This gives me great doubts regarding the

> objectivity of your research. It certainly appears your research is biased owards filed based environments. In my world, there is no direct correlation between files, code modules, garanularity, meta-object models and their file system representations, certainly not to the extent where the file system dictates the code model.
>
> We do NOT look at files if we are coding, cause we deal with classes and objects in a development tool (in Smaltalk it is a part of the language, for C# we used Visual Studio). So a lot of questions are not confirm with the real development environment.

Zwei Teilnehmer hätten gerne mehr Fragen zur allgemeinen Paarprogrammierung gesehen:

> Nice quality on the question. Maybe some questions in the problematic fields are missing in my point of view (egg. about problematic getting some foreign to pairprogramming into this stuff, a partner that does not behave well on direct critique on his/her code, etc).
>
> No questions about why we are not pair programming. Having tried it in our archive group it failed to increase productivity.

Fragen dieser Art hätten meiner Meinung nach, allerdings den Rahmen der Umfrage gesprengt und von ihrer Forschungsfrage abgelenkt.

Ein Teilnehmer kritisierte die technische Umsetzung der Umfrage:

> You didn't take into account, that as a non-pair programmer, was not able to answer all the questions, and thus they should have been left out! You should have asked the question: Do you have experience with pair-programming? and based on the answer changed the survey questions. Ok, hope you can use my answers anyway!

Die Idee bestimmte Fragen abhängig von den Antworten auf vorherige Fragen auszuwählen, war bereits während eines Pilottests der Umfrage aufgekommen, scheiterte jedoch an der Umsetzung im verwendeten phpESP-System. In Angesicht des Zeitdrucks, des erheblichen Aufwands für die Neuaufsetzung eines anderen Systems und der Tatsache, dass dieser Aspekt nur einen sehr kleinen Teil der Fragen betraf, hatte ich mich für die Beibehaltung der aktuellen Implementierung entschieden.

Die beiden folgenden Fragen befassten sich mit unterschiedlichen Aspekten:

> Question 17 was a stumper. SubEthaEdit has shown us that sometimes the fact that you can edit two different files is useful, and

> sometimes a hinderance.

Auf diese Kritik bin ich ebenfalls in Abschnitt 3.1.2 eingegangen:

> I forgot to mention before that we also used a shared VNC session to observe the actual application under test (a web application viewed using firefox on linux under vnc)

In Abschnitt 3.3 bin ich unter dem Unterpunkt *Flexibilität* bereits darauf eingegangen, dass Desktop Sharing nicht nur die Übertragung jeglicher Programme ermöglicht, sondern nicht zuletzt auch die Demonstration der Interaktion mit diesen Programmen ermöglicht. Dieser Teilnehmer schildet einen entsprechenden Anwendungsfall.

Ein Teilnehmer lobte die Zusammenstellung der Umfrage:

> the survey is well balanced, i am curious about the result

4.3. Bewertung

Der Start der Umfrage hatte sich aufgrund von technischen Problemen im Institut um mehrere Wochen verschoben, sodass ich aus zeitlichen Gründen verschiedene Basisfunktionalitäten bereits vor Ende der Umfrage angefangen hatte zu programmieren. Dennoch hatte die Umfrage starken Einfluss auf die Anforderungen und Entwurfsentscheidungen des zu entwickelnden DPP-Werkzeugs. Auch Fragen, die zwar weniger direkten Einfluss auf die Implementierung hatten, haben insgesamt dazu beigetragen meine Erläuterungen zu unterstützen oder mich auf neue Aspekte aufmerksam zu machen.

Für eine erneute Durchführung der Umfrage sollten verschiedene Fragen, wie jeweils von mir erläutert, präzisiert werden. Ich denke jedoch, dass der Fragenkatalog einen sehr guten Ausgangspunkt für weitere Umfragen in diesem Bereich darstellt. Für diese Arbeit war er in jedem Fall von erheblichem Nutzen.

5. Entwurf und Implementierung eines Werkzeugs zur verteilten Paarprogrammierung

Im Folgenden gehe ich auf den namensgebenden Teil dieser Diplomarbeit ein: den Entwurf und die Implementierung eines Eclipse-Plugins zur verteilten Paarprogrammierung. Der Name des Werkzeugs lautet *Saros*[1].

In Abschnitt 5.1 erläutere ich zunächst verschiedene Entwurfsentscheidungen. In Abschnitt 5.2 widme ich mich den genutzten Technologien und tauche anschließend in Abschnitt 5.3 tiefer in verschiedene Implementierungsdetails ein.

5.1. Entwurfsentscheidungen

Nach einer ersten Recherchephase wurde in Abstimmung meiner Betreuer und mir, eine Liste der wichtigsten Anforderungen ausgearbeitet. Die aktuelle Liste der Mindestanforderungen kann in Anhang B eingesehen werden. Einige wichtige Anwendungsfälle, die aus diesen Anforderungen und den folgenden Entwurfsentscheidungen ergeben, können in Anhang C nachgelesen werden.

Das folgende Kapitel befasst sich mit den Entscheidungen, die sich direkt oder indirekt aus diesen Anforderungen ergaben. Um meine Annahmen zu untermauern, habe ich in den vorherigen Abschnitten bereits verschiedene Aspekte der regulären und verteilten Paarprogrammierung erläutert. Meine Untersuchungen wurden durch eine Umfrage ergänzt. Zusammen bilden diese Erkenntnisse die Grundlage meiner Entwurfsentscheidungen, von denen ich die Wichtigsten folgend erläutere.

5.1.1. Kontaktliste

Teil der Anforderungen war die Möglichkeit andere Benutzer mittels eines Bezeichners benennen zu können und über diesen Sitzungen einleiten zu können.

[1] Sarosperioden sind astronomische Zeitspannen nach denen sich Sonnenfinsternisse (Eklipsen) und Mondfinsternisse wiederholen.

Ich habe mich dazu entschlossen, diese Anforderung mithilfe einer Kontaktliste zu realisieren. Die Kontaktliste sollte dabei drei Funktionalitäten realisieren:

1. Über die Anzeige der Anwesenheit soll der Anwender sehen können, welche Kontakte überhaupt für eine etwaige Paarprogrammier-Sitzung zur Verfügung stehen. Jeder Klient meldet sich dazu an einem speziellen Server an und teilt diesem den aktuellen Status mit (beispielsweise „verfügbar“). Der Server meldet diesen neuen Status an alle Mitglieder, die zurzeit ebenfalls angemeldet sind und den betreffenden Kontakt auf ihrer Kontaktliste führen.

2. Über eine Instant-Messaging-Funktionalität soll der Anwender in der Lage sein, anderen Anwendern kurze Nachrichten zu schicken. Die von mir realisierte Umfrage (siehe Abschnitt 4) bestätigte meine Annahme, dass Instant Messaging ein häufig genutztes Kommunikationsmedium zur Verabredung von DPP-Sitzungen ist. Die Antwort „Instant Messaging“ war in der betreffenden Frage die meist genannte Antwort gewesen (siehe Tabelle 4.3).

   Bezogen auf die Kommunikation während Sitzungen, brachte die Umfrage jedoch auch die Wichtigkeit einer Sprachunterstützung zum Ausdruck. Daher sollten die Kurznachrichten vor allem vor beziehungsweise nach Sitzungen zum Einsatz kommen. Während einer Sitzung sollte auf eine ergänzende VoIP-Funktionalität zurückgegriffen werden. Diese unmittelbar in Saros zu integrieren hätte allerdings den Rahmen dieser Diplomarbeit gesprengt, weshalb ich an dieser Stelle auf andere Produkte, wie beispielsweise Skype oder Google Talk verweisen muss.

3. Da die Kontaktlisten Vermittler zwischen zwei an einer DPP-Sitzung interessierten Anwendern darstellen, können Sitzungen eingeleitet werden, indem der gewünschte Partner über seine einfache Nutzerkennung adressiert wird. Ein händischer Austausch von IP-Adressen wäre aufgrund seiner Umständlichkeit und Fehlerträchtigkeit deutlicher weniger benutzerfreundlich.

Zur Realisierung dieser Funktionalitäten entschied ich mich für die XMPP-Plattform. Auf dieses Protokoll und die genauen Gründe für dessen Wahl gehe ich in Abschnitt 5.2.3 detailliert ein. Auf die genaue Implementierung der Kontaktliste gehe ich in Abschnitt 5.3.5 ein.

5.1.2. Übersetzbarkeit

Eine elementare Anforderung an das DPP-Werkzeug war, dass alle Teilnehmer der verteilten Paarprogrammier-Sitzung in der Lage sein sollten, auf alle Dateien eines gemeinsamen Projekts zugreifen zu können und Übersetzungsfehler

angezeigt zu bekommen. Wie in Abschnitt 3.3.1 bereits erläutert, können bei *Collaboration Awareness* zwei Ansätze unterschieden werden: *Virtueller Zugriff* und *Replikation*.

Nach eigenen Nutzungstests mit NetBeans und JBuilder (siehe Kasten „Erfahrungen mit NetBeans und JBuilder“), bemerkte ich, dass die Nutzung verschiedener IDE-Funktionen (beispielsweise Code-Inspektionen[2]) mit erheblichen Verzögerungen verbunden waren. Ungeachtet der Tatsache, dass ich bereits eine identische Version des Projekts auf meinem System installiert hatte, wurden alle Daten immer von dem entfernten Host der Sitzung übertragen.

Dieser Entwurfsentscheidung scheint die Annahme zugrunde zu liegen, dass die meisten DPP-Anwender vor Beginn einer Sitzung, nicht bereits die gleiche Version eines Projekts auf ihren Systemen vorliegen hätten. Dies widersprach jedoch meiner persönlichen Erfahrung: Betrieb ich beispielsweise auf einer Arbeitsstelle Paarprogrammierung, dann mit Kollegen, die an dem gleichen Projekt arbeiteten, und mit denen mein Projektstand abgesehen von minimalen Änderungen seit dem letzten Übertragen in das Versionskontrollsystem übereinstimmte.

Um meine eigene Annahme zu überprüfen, fragte ich in der von mir realisierten Umfrage (siehe Abschnitt 4), wie wahrscheinlich es sei, dass die Anwender bereits vor Beginn einer DPP-Sitzung das zu bearbeitende Projekt auf ihrem System installiert hätten. Die Teilnehmer der Umfrage gaben im Durchschnitt 72,5% bei einer Standardabweichung von 37,36 an. Ein Drittel der Teilnehmer gab dabei sogar 100% an (siehe Abschnitt 4.2.2).

Da ich meine Annahme tendenziell bestätigt sah, entschloss mich daher für die Nutzung des Replikations-Ansatzes. Der Nachteil dieses Ansatzes, die Vorbedingung, dass alle Teilnehmer identische Versionen des Systems auf ihrem System installiert haben mussten, sah ich durch meine Annahme abgeschwächt. Wird in Saros nämlich eine verteilte Paarprogrammier-Sitzung eingeleitet, so sollen auf dem System des Gastes alle lokalen Projekte mit dem entfernten Projekt verglichen werden. Anhand meiner Annahme ist hier meist mit dem Finden eines nahezu identischen Projekts zu finden. Dieses Projekt sollte die Ausgangsbasis für die folgende restliche Synchronisation bilden, welche dadurch drastisch abgekürzt werden sollte. Auf die genaue Implementierung dieses Ansatzes gehe ich in Abschnitt 5.3.3 ausführlich ein.

Wie in Abschnitt 4.2.2 angegeben, nutzten alle Teilnehmer der Umfrage Versionskontrollsysteme. Öffentliche Repositorien (auf Englisch *repositories*) sind ein hervorragender Mechanismus, um den aktuellen Projektstand zur Veröffentlichung zu stellen. Ein möglicher Anwendungsfall bei der Benutzung von Saros wäre auch sich den Projektcode vom Repositorium zu laden und dann auf Basis dieser Version die Sitzung einzuleiten.

[2] Damit meine ich die Möglichkeit sich alle Methoden eines Typs mitsamt Dokumentation in einer Liste anzeigen zu lassen.

Durch die Verwendung des Ansatzes Replikation ist Saros zudem nicht nur auf die Unterstützung der Programmiersprache Java ausgelegt. Funktioniert die Replikation korrekt, muss nicht jede zusätzliche IDE-Funktionalität einzeln einprogrammiert werden, sondern wird automatisch unterstützt. Aus der Sicht einer IDE-Funktionalität liegt ein ganz normales lokales Projekt vor.

Erfahrungen mit NetBeans und JBuilder

Im Rahmen meiner Diplomarbeit habe ich Tests mit verschiedenen vorhandenen DPP-Werkzeugen durchgeführt. Dabei waren für mich vor allem die Tests mit Sun NetBeans [RS05] und Borland JBuilder [JB] von besonderem Interesse, da diese in gewisser Weise direkte Konkurrenten zu Saros darstellen.

Beide Werkzeuge erschienen zum Zeitpunkt meines Tests, März 2006, noch unausgereift. Betrat ich bei NetBeans als Gast eine verteilte Paarsitzung, so wurden in der ganzen Datei Übersetzungsfehler angezeigt. Dazu gesellten sich andere Tücken, wie ein unkontrolliertes Umschalten des Textcursors zwischen Chatfenster und Texteditor jedes Mal, wenn mein Sitzungspartner mir eine Kurznachricht schickte. JBuilder erschien mir im Vergleich robuster und ausgereifter. Allerdings waren Dateien auch hier nicht vollständig frei von falschen Übersetzungsfehlern.

Beide Werkzeuge boten die Möglichkeit von Code-Vervollständigungen. Diese waren aber teilweise mit deutlichen Verzögerungen verbunden, sodass ich während meiner Testsitzungen anfing, so weit wie möglich darauf zu verzichten, während ich diese Funktionalitäten in meiner regulären Programmiertätigkeit sehr häufig einsetze.

5.1.3. Awareness

Innerhalb von Saros wird Awareness (siehe auch Abschnitt 3.1.2) auf einer Vielzahl von Ebenen realisiert. Dies beginnt mit der in Abschnitt 5.1.1 erläuterten Kontaktliste, durch die Anzeige der Anwesenheit der Kontakte kann der Benutzer bereits abwägen, wer für eine eventuelle DPP-Sitzung zur Verfügung steht. Ist eine Sitzung zustande gekommen, wird innerhalb der Sitzung eine Liste aller Teilnehmer der Sitzung angezeigt. In dieser Liste wird auch vermerkt, wer der aktuelle Driver ist (siehe Abschnitt 5.1.4).

Darüber hinaus realisiert die Nachbildung der Ressourcenänderungen bereits eine gewisse Awareness. Dieser Informationsgrad kann bei der Bearbeitung kleinerer Projekte mit wenigen Ressourcen, bereits zur effektiven Zusammenarbeit ausreichen. Da Saros allerdings auch für die Arbeit mit normalen Java-Projekten genutzt werden sollte, reicht dieser Grad an Awareness nicht aus. Die

Vielzahl und Größe der Ressourcen eines solchen Projekts erfordern zusätzliche Maßnahmen. In Saros sollen folgende Funktionalitäten realisiert werden:

1. In Dateiübersichten sollen die durch den Driver gerade bearbeiteten Dateien farbig markiert werden.

2. Innerhalb von Editoren soll, abgesehen von Textänderungen, auch die Position des Cursors des Drivers übertragen werden. Dies erscheint mir in zweierlei Hinsicht wichtig: Zum einem kann ein Observer durch die Position des Cursors besser abschätzen, auf welchen Punkt des Editors sich der Driver gerade konzentriert und wo er seine nächste Änderung vornehmen wird. Zum anderen kann der Driver das Auswählen von Text als eine Art Gestik benutzen und sich innerhalb von Diskussionen unmittelbar auf den von ihm ausgewählten Text beziehen.

3. Der aktuelle Sichtbereich des Drivers soll übertragen und angezeigt werden. Mit Sichtbereich ist der Teil einer Datei gemeint, die der Driver in seinem Editor gerade sehen kann. Bei einer kleinen Datei umfasst der Sichtbereich womöglich den ganzen Inhalt, bei größeren Java-Klassen kann der Editor aber normalerweise nur einen Teil der Datei darstellen. Der Sichtbereich bezeichnet dann den Bereich zwischen der obersten und untersten Zeile der Datei, die der Driver in seinem Editor gerade noch sehen kann.

4. Die Textänderungen des Drivers sollen mit einer bestimmten Hintergrundfarbe gekennzeichnet werden. So entgehen dem Observer auch im Nachhinein keine Änderungen, die er ansonsten eventuell übersehen hätte.

Abschnitt 5.3.6 beschäftigt sich mit der Implementierung dieser Funktionalitäten.

5.1.4. Parallelität und Verfolgermodus

In Abschnitt 3.1.1 habe ich den Begriff der *Parallelität* eingeführt. Parallelität ist eine Form von nebenläufiger Paararbeit. In diesem Aspekt weichen verteilte Paarprogrammierung und reguläre Paarprogrammierung voneinander ab. In der durchgeführten Umfrage bewerteten die Teilnehmer Rollenwechsel und eine *Parallelität auf Sichtebene* als deutlich wichtiger als die Unterstützung von *Schreibparallelität* (siehe Abschnitt 4.2.4).

Für Saros habe ich mich daher dazu entschlossen, in Bezug auf die Rolle des Drivers regulärer Paarprogrammierung möglichst nah zu kommen. Es soll zu jedem Zeitpunkt nur einen Driver geben. Für den Observer sind keine Eingaben in den Editor möglich. In der ersten Version von Saros wird zudem nur

ein Observer unterstützt[3]. Der aktuelle Driver soll auf seinen Wunsch zu jedem Zeitpunkt einen Rollentausch mit dem Observer durchführen können. Der Initiator der Sitzung kann darüber hinaus zu jedem Zeitpunkt der Sitzung die Driver-Rolle einfordern, er benötigt dazu keine Bestätigung durch den Gast. Außerdem soll Saros eine *Parallelität auf Sichtebene* und auf *Programmebene* ermöglichen.

Zusätzlich soll auch ein Verfolgermodus implementiert werden. Ist dieser aktiviert, so folgt der Sichtbereich des Observers automatisch dem des Drivers. Öffnet er eine Datei, so wird diese auch beim Observer geöffnet. Der Verfolgermodus soll vom Observer selbstständig aktiviert und deaktiviert werden.

5.2. Technologien

Zur Implementierung dieser Anforderungen und der Entwurfsentscheidungen greife ich vor allem auf drei Technologien zurück: Java als Implementierungssprache, XMPP als Kommunikationsprotokoll und Eclipse als zu erweiternde IDE. Im Folgenden führe ich diese Technologien ein und begründe jeweils meine Wahl.

5.2.1. Java

Die Java-Plattform setzt sich aus einer Menge von Software-Komponenten und Spezifikationen zusammen. Die Kernkomponenten sind dabei die Programmiersprache Java und die virtuelle Java-Laufzeitumgebung (engl. *Java Runtime Environment*). Die Java-Programmiersprache ähnelt in ihrer Syntax stark der Programmiersprache C++, ist aber im Gegensatz zu ihr grundlegend objektorientiert[4]. Ein besonderer Schwerpunkt aller Java-Komponenten liegt auf einer hohen Maschinenunabhängigkeit. Darüber hinaus stellt die Java-Umgebung eine automatische Speicherbereinigung (*Garbage Collection*) sowie eine umfangreiche Klassenbibliothek bereit.

Die erste Version von Java wurde in den frühen 1990er Jahren von *Sun Microsystems* entwickelt. Java wurde seitdem in mehreren Versionen stark erweitert

[3] In Anbetracht des engen Zeitplans musste ich mich zu einem späten Zeitpunkt dazu entscheiden auf die Unterstützung mehrerer Observer verzichten. Die verabredeten Anforderungen sahen ohnehin nur einen Observer voraus. Dennoch sieht die Implementierung diese Funktionalität bereits an allen Punkten bereits vor. So arbeiten Methoden, die Observer betreffen mit Listen, anstatt mit einzelnen Objekten. Auch die GUI und Benutzerführung ist bereits vollständig auf die Unterstützung mehrerer Teilnehmer ausgerichtet. Ich rechne daher mit einem vergleichsweise geringen Aufwand für spätere Vollendung dieser Funktionalität.

[4] Eine Ausnahme bilden die primitiven Literale wie zum Beispiel `int`, `float` oder `long`.

und hat sich zu einer erfolgreichen Software-Entwicklungs-Plattform entwickelt. Meine Wahl für Java als Implementierungssprache hatte drei Gründe:

1. In Hinblick auf potenzielle Weiterentwicklungen des DPP-Werkzeugs an der Universität lag die Wahl einer möglichst weit verbreiteten Sprache nahe. Java genießt im akademischen Bereich eine hohe Akzeptanz und wird am Institut für Informatik der FU-Berlin häufig eingesetzt.

2. Eclipse benutzt Java als Implementierungssprache (siehe Abschnitt 5.2.2).

3. Ich verfügte bereits über sechs Jahre Programmiererfahrung mit Java und hatte damit schon verschiedenste Projekte umgesetzt.

Zum Einsatz kommt die aktuelle stabile 1.5 Version der Java-Plattform.

5.2.2. Eclipse

Eclipse ist ein Open-Source-Rahmenwerk, welches ursprünglich als stark erweiterbare Entwicklungsumgebung (*integrated development environment*, kurz IDE) konzipiert worden war. Über Jahre wurde das Framework so stark erweitert, dass es mittlerweile als sogenannte *Rich Client Platform* (RCP) als Basis für jegliche Desktopapplikationen dienen kann.

Eclipse ist in Java implementiert, setzt allerdings nicht dessen Standard-GUI ein. Statt *Java Swing* kommt das eigens entwickelte *Standard Widget Toolkit* (SWT) zum Einsatz, welches im Gegensatz zu Swing die nativen grafischen Elemente des jeweiligen Betriebssystems einsetzt. Dies sorgt für einen natürlicheren „Look“ der SWT-Applikationen und für eine teilweise höhere Effizienz.

Eclipse wurde zuerst als Nachfolger von *IBM Visual Age for Java 4.0* und als kommerzielles Produkt entwickelt. 2001 gab IBM den gesamten Quellcode frei und übertrug die Führung des Projekts der neu gegründeten *Eclipse-Foundation*, welche 2003 unabhängig von IBM wurde. Dennoch ist IBM noch stark an der Entwicklung von Eclipse beteiligt [Ecl]. Die Wahl von Eclipse hatte drei Gründe:

1. Eclipse verfügt über ein sehr flexibles und mächtiges Plugin-System, welches ermöglicht Eclipse als IDE für verschiedenste Sprachen zu verwenden. Die *Java Development Tools* (JDT) stellen dabei die bekanntesten IDE-Funktionalitäten bereit und werden von einer Vielzahl von Java-Programmierern eingesetzt. Durch die Wahl von Eclipse hoffte ich eine möglichst große Zahl von potenziellen Benutzern ansprechen zu können, sowohl innerhalb der Gemeinschaft der Java-Programmierer, als auch außerhalb.

2. Eclipse ist komplett quelloffen. Zusammen mit der Flexibilität des Plugin-Systems ging ich davon aus, dass die gewünschte Funktionalität so tatsächlich mit Eclipse zu realisieren wäre.

3. Verschiedene Projekte des Instituts für Informatik der Freien Universität und im Speziellen der Arbeitsgruppe Software-Engineering, setzen Eclipse bereits als Plattform ein. Um potenzielle Anbindungen an eines dieser Projekte zu erleichtern, lag die Wahl derselben Plattform nahe[5].

Ein Aspekt, der gegen die Wahl von Eclipse sprach, war, dass ich zu Beginn der Diplomarbeit zwar über viel Erfahrung im Umgang mit der JDT verfügte, aber keinerlei Erfahrung mit dem Rahmenwerk selber hatte. In Anbetracht der Komplexität und Größe des Eclipse-Rahmenwerks stellte dies ein ernsthaftes Hemmnis dar. Insgesamt schienen mir die Vorteile der Plattform jedoch zu überwiegen.

Als Zielplattform wurde Eclipse 3.2 als Anforderung ausgewählt (siehe auch Anhang B).

5.2.3. Extensible Messaging and Presence Protocol

Das *Extensible Messaging and Presence Protocol* (kurz XMPP) [SA05] ist ein offenes und XML-basiertes Protokoll für Instant Messaging und Anwesenheitsvermittlung. Unter Instant Messaging versteht man einen Dienst, der das schnelle Verschicken und Zustellen von kurzen Textnachrichten an andere Benutzer erlaubt. Der Austausch von Anwesenheitsinformationen ist dabei stark mit dieser Funktionalität verwoben, sie erlaubt Informationen über die eigene Verfügbarkeit zu kommunizieren. XMPP definiert dazu sogenannte Roster, persönliche Kontaktlisten, welchen Benutzer des Netzwerks hinzugefügt werden können und über deren Anwesenheit man, nach einer eventuellen Autorisierung, fortan informiert wird.

Das Protokoll basiert auf einem dezentralisierten Serververbund. Die XML-Nachrichten werden grundsätzlich über Server vermittelt. Jeder Benutzer verfügt über einen eindeutigen Bezeichner, die Jabber-ID[6] (JID), welche sich aus dem Domänennamen des Servers und einem Nutzernamen zusammensetzt .

Wird eine Nachricht an einen Teilnehmer eines anderen Servers geschickt, so wird die Nachricht auf Serverebene zuerst an den korrekten Server geleitet, der die Nachricht dann an seinen lokalen Teilnehmer übermitteln kann. Abbildung 5.1 zeigt ein entsprechendes Szenario. *Clarks* Nachricht wird zuerst an seinen Server `metropolis.com` gesendet. Ist der Server `gotham.net` auf

[5] Diese Entscheidung ermöglichte mir in einer späten Phase der Implementierung den Eclipse-Sensor des ElectroCodeGram zu erweitern (siehe Abschnitt 6).

[6] Zum Beispiel `riad@jabber.org`.

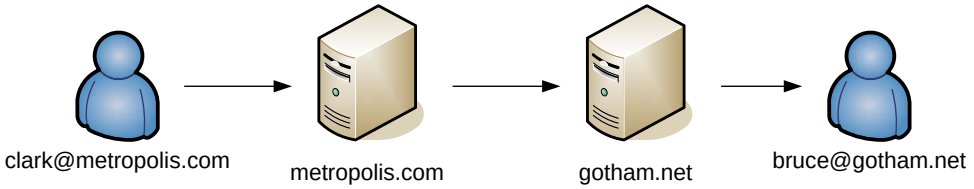

Abbildung 5.1.: Server-zu-Server-Kommunikation im XMPP-Netzwerk.

`metropolis.com` gesperrt, so verfällt die Nachricht. Andernfalls wird die Nachricht an den richtigen Server weitergeleitet. Ist `metropolis.com` auf `gotham.net` gesperrt, so wird die Nachricht ebenfalls verworfen. Andernfalls wird versucht die Nachricht zuzustellen. Sollte *Bruce* zu Zeit nicht angemeldet sein, so wird die Nachricht auf dem Server verwahrt, um sie ihm bei seiner nächsten Einwahl zuzustellen.

Eine zentrale Einheit, die den gesamten Serververbund kontrolliert oder verwaltet, existiert nicht. Auf diese Art erlaubt XMPP jeder Person dem Serververbund beliebig neue Server hinzuzufügen.

XMPP ging aus dem Jabber-Projekt hervor, welches 1998 durch Jeremie Miller, als eine offene und erweiterbare Alternative zu proprietären Instant Messengern wie beispielsweise Mirabilis ICQ oder MSN Messenger, entwickelt wurde. Resultat dieser Anstrengung war unter anderem das Protokoll *XML Streaming for instant messaging and presence*, welches die Basis von Jabber bildet. Das Protokoll wurde durch die *Internet Engineering Task Force* (IETF) als das unabhängige XMPP standardisiert und bei der *Internet Engineering Steering Group* (IESG) eingebracht. Erweiterungen an XMPP werden seitdem durch die *Jabber Software Foundation* angeführt.

Die Bezeichnungen Jabber und XMPP werden meist synonym benutzt. Die Gründe, aus denen ich XMPP wählte, sind Folgende:

1. XMPP stellt mit Instant Messaging, Anwesenheitsvermittlung, Dateiübertragung und Chat bereits viele Funktionalitäten bereit, die ich bei der Implementierung von Saros benötigte. Das Protokoll wird zudem bereits von einer Vielzahl von Software-Bibliotheken implementiert.

2. Die Verwendung von XMPP birgt ein relativ kleines Lieferantenrisiko. XMPP befindet sich seit Jahren in einer stetigen Entwicklung und ist ein etablierter Standard. Jabber-URIs[7] gehören mittlerweile zu den durch die IESG anerkannten Standard-URIs. Die Gefahr eines potenziellen Entwicklungsstillstands oder einer unpassenden Entwicklung der Standardisierung

[7] Zum Beispiel `xmpp:riad@jabber.org`

schien unwahrscheinlich.

3. XMPP eignet sich ausgesprochen gut für Erweiterungen. Nachrichten können XML-Elemente angehangen werden, womit Klienten schnell um neue Funktionalitäten erweitert werden können. Auf diesen Aspekt gehe ich in Abschnitt 5.3.1 umfassender ein.

4. Firewalls und *Network Address Translation* (NAT) stellen für netzbasierte Software erhebliche Herausforderungen dar. Durch die Nutzung eines etablierten Protokolls und Standardports scheint es wahrscheinlicher, dass die jeweiligen Barrieren bereits auf den XMPP-Verkehr eingestellt wurden.

5. 2005 veröffentlichte Google in Zusammenarbeit mit der Jabber Software Foundation das *Jabber Enhancement Proposal* (JEP) für *Jingle* [LBSA+06]. Jingle dient zur Einleitung von Peer-to-Peer-Sitzungen und wird von Google in der Voice-Over-IP-Software (kurz VoIP-Software) *Google Talk* eingesetzt.

   Die Jingle-Erweiterung wurde in der Jabber-Gemeinschaft positiv aufgenommen und in den nächsten Monaten ist mit VoIP-Implementierungen für verschiedene XMPP-Bibliotheken zu rechnen. In Anbetracht der Wichtigkeit von Stimmübertragung für verteilte Paarprogrammierung (siehe auch Abschnitt 4.2.4), stellte dies eine viel versprechende Aussicht dar.

6. Die Jabber-Gemeinschaft verfügt in Form der *Jabber Enhancement Proposals* über einen ausgereiften Standardisierungsprozess für Erweiterungen. Bei einem Erfolg von Saros wäre langfristig denkbar, eine Version des Saros-Netzwerkprotokolls für verteile Paarprogrammierung als Jabber-Erweiterung vorzuschlagen.

Nach kleineren Tests mit verschiedenen Softwarebibliotheken für XMPP fiel meine Wahl auf *Smack* [Sma], welches von *Jive Software* quelloffen entwickelt wird.

5.3. Implementierung

In diesem Abschnitt gehe ich auf die Implementierung der in Abschnitt 5.1 erläuterten Entwurfsentscheidungen ein. Ich gebe zunächst einen allgemeinen Überblick über die Architektur. Im Anschluss erläutere ich die wichtigsten Implementierungsdetails. Das Herzstück von Saros bildet dabei das System der „Aktivitätsobjekte“, welches ich am Ausführlichsten erläutere.

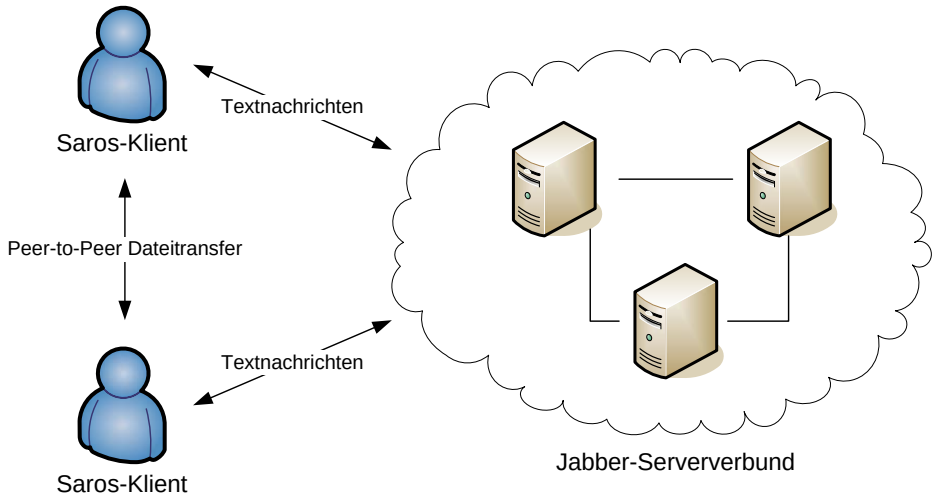

Abbildung 5.2.: Einfacher Überblick über die Architektur von Saros.

5.3.1. Architektur

Da Saros auf XMPP (siehe Abschnitt 5.2.3) basiert, nutzt es eine serverbasierte Architektur. Wie bereits erwähnt, verfügt XMPP über ein mächtiges Erweiterungskonzept: Den XML-Nachrichten können beliebige XML-Elemente angehangen werden. Dabei müssen die Server in keiner Weise angepasst werden. Sie prüfen bei den Nachrichten nur die Konsistenz bestimmter Kerndaten wie beispielsweise des Absendereintrags. Dies verringert die Komplexität von Saros erheblich und steigert die Erfolgsaussichten des Plugins, da bereits vom Start weg ein großer Serververbund zur Verfügung steht.

Eine Ausnahme der serverbasierten Kommunikation bilden Dateiübertragungen [MME04]. Aus Effizienzgründen versucht Saros zuerst eine direkte *Peer-to-Peer*-Verbindung (auf Basis des SOCKS5-Protokolls [SMSA04]) zwischen den beiden Teilnehmern der Sitzung herzustellen. Sollte die Sitzung in einem lokalen Netzwerk stattfinden, so ist immer mit einem Zustandekommen dieser Verbindung zu rechnen. Bei Verbindungen zu anderen Netzwerken (über das Internet) kann diese Verbindung allerdings aufgrund einer Firewall oder einer NAT (Network Address Translation) scheitern. In diesem Fall fällt die Dateiübertragung auf ein Ausweichprotokoll zurück: Die Datei wird wie sonstige Nachrichten auch über den Umweg der Server übermittelt [Kar03]. Aufgrund der resultierenden Netzlast ist dieses Verfahren allerdings wesentlich langsamer als das erst genannte Verfahren. Abbildung 5.2 zeigt einen Überblick über diese Architektur.

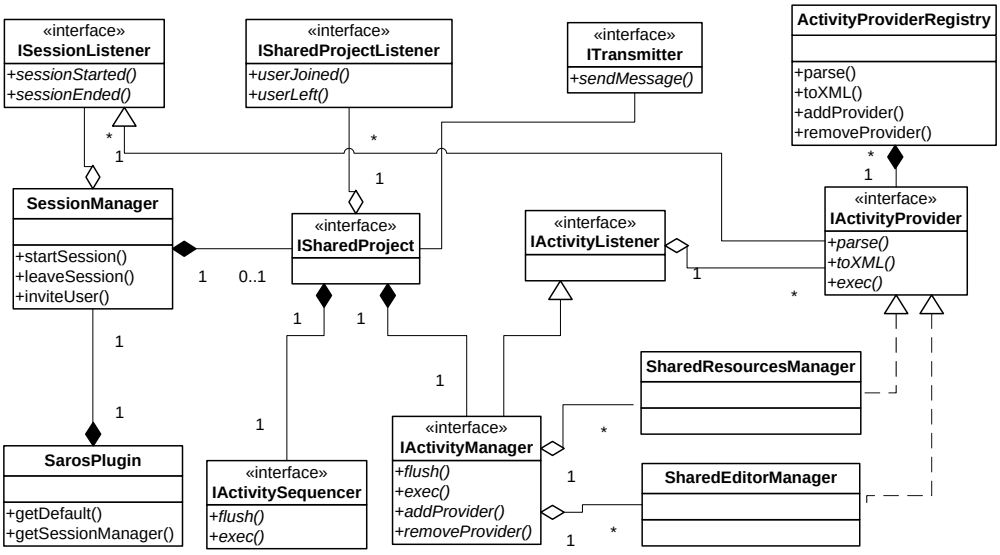

Abbildung 5.3.: Klassendiagramm einiger zentraler Klassen und Schnittstellen im Saros-Plugin.

5.3.2. Abfangen und Reproduzieren von Aktivitäten

Die Kernfunktionalität von Saros beschäftigt sich hauptsächlich mit dem Abfangen bestimmter relevanter Vorgänge, wie beispielsweise dem Einfügen von Text oder dem Selektieren eines Textblocks, und dem anschließenden Reproduzieren dieser Vorgänge auf dem Klienten des anderen Teilnehmers der DPP-Sitzung. Dies ist vor allem bei der Veränderung von Ressourcen wichtig. Bei Beginn der Sitzung liegt bei jedem Teilnehmer eine identische Kopie des Projekts vor (siehe Abschnitt 5.1.2). Wird eine Ressource während der Sitzung verändert, muss die gleiche Änderung auch auf der Gegenseite vollzogen werden.

Die wichtigsten Klassen im Überblick

Abbildung 5.3 zeigt einen Überblick über wichtige Klassen und Schnittstellen, die an der Verarbeitung von sogenannten Aktivitäten beteiligt sind. Ich werde diese im Folgenden kurz einführend erläutern:

SarosPlugin Diese Klasse ist über die Plugin-Konfigurationsdatei als Basisklasse des Saros-Plugins festgelegt. Dadurch wird sie beim Start der IDE automatisch aufgerufen und kann sich um die Erzeugung aller für Saros benötigten Objekte kümmern. Gemäß den Eclipse-Konventionen für Plugin-Basisklassen ist sie als Singleton realisiert und verfügt über die Zugriffsmethode `getDefault()`.

SessionManager Diese Klasse ist für den Lebenszyklus von DPP-Sitzungen zuständig. Über sie werden Sitzungen eingeleitet und beendet sowie Einladungen verschickt (siehe Abschnitt 5.3.3). Sie hält eine Liste von `ISessionListener`-Exemplaren, welche über den Lebenszyklus von Sitzungen informiert werden.

ISharedProject Diese Schnittstelle repräsentiert eine DPP-Sitzung. Sie bildet den zentralen Treiber für alle Abläufe, die innerhalb einer Sitzung kontrolliert werden müssen.

IActivity Jede Klasse, die diese Schnittstelle implementiert, steht für ein Verhalten, das gespeichert und reproduziert werden kann. Auf die unterschiedlichen Arten von Aktivitäten (Exemplare vom Typ `IActivity`) gehe ich in Abschnitt 5.3.2 näher ein.

IActivityProvider Das Erzeugen und Verarbeiten von `IActivity`-Exemplaren wird durch Klassen vom Typ `IActivityProvider`[8] geregelt. Jeder Provider ist für eine oder mehrere Arten von Vorgängen zuständig. Er beobachtet die Umgebung durch den Gebrauch von Standardbeobachtern der Eclipse-API. Geschieht ein relevanter Vorgang, reagiert er mit dem Erzeugen eines passenden Exemplars vom Typ `IActivity`. Die Attribute des Exemplars erhalten alle zur Reproduktion des Vorgangs benötigten Daten.

Auf diese Art wandelt ein Provider Verhalten in `IActivity`-Exemplare um. Umgekehrt muss er das `IActivity`-Exemplar aber auch wieder in Verhalten umwandeln können. Dazu dient die Methode `exec(IActivity)`, welche per Vertrag alle Aktivitätstypen unterstützen muss, die vom Provider erzeugt werden können.

ActivityProviderRegistry Die `IActivityProvider`-Exemplare haben eine längere Lebenszeit als Sitzungen, sie existieren unabhängig von ihnen. Die `ActivityProviderRegistry` erzeugt bei Plugin-Start alle `IActivityProvider`-Exemplare. Über die Implementierung von `ISessionListener` registrieren sich die Provider als Beobachter an dem `SessionManager`. Wird eine Sitzung erzeugt, meldet der Provider sich selbstständig an dem `ActivityManager` des `ISharedProject` an. Dadurch bekommt der `ActivityManager` jedes Erzeugen einer IActivity mitgeteilt.

Momentan existieren drei Klassen, welche die Registratur automatisch erzeugt. Dies sind der `SharedResourcesManager` (zuständig für Aktivitäten am Dateibaum), der `EditorManager` (zuständig für Aktivitäten in einem Editor) und der `RoleManager` (zuständig für Rollenwechsel zwischen Driver und Observer).

[8] Die Namensendung auf „Provider“ erfolgte gemäß einer Namenskonvention in Eclipse für bestimmte Steuerklassen (Controller).

IActivityManager Diese Schnittstelle empfängt alle durch `IActivityProvider` erzeugten Aktivitätsexemplare und sammelt diese in einer Liste. Durch das Aufrufen der Methode `flush()` wird der aktuelle Inhalt der Liste zurückgeliefert und geleert.

IActivitySequencer Laut Sun und Ellis [SE98] versteht man bei kollaborativen Texteditoren unter *Divergenz* eine fehlerhafte Ordnung von Operationen. Das Problem entsteht dann, wenn Operationen in einer anderen Reihenfolge eintreffen, als sie abgeschickt wurden.

Da XMPP (siehe Abschnitt 5.2.3) keine Beibehaltung einer bestimmten Reihenfolge forciert, muss Saros diese Garantie sicherstellen. Divergenz droht in Saros vor allem bei Dateiübertragungen. Kopiert der Anwender beispielsweise eine größere Textdatei in das Projekt und bearbeitet sie gleich, so muss sichergestellt werden, dass die Datei zuerst komplett übertragen wurde, bevor die Textänderung repliziert werden kann.

Der `IActivitySequencer` annotiert Aktivitäten mit Zeitstempeln, indem er die `IActivity`-Exemplare in `TimedActivity`-Exemplaren kapselt. Bei ankommenden Aktivitäten stellt er durch die Überprüfung der Zeitstempel die korrekte Ordnung sicher. Dabei arbeiten `IActivityProvider` immer nur mit „zeitlosen“ `IActivity`-Exemplaren, für welche die korrekte Ordnung garantiert ist. Das Entwurfsgeheimnis der Zeitstempel wird durch den `IActvitySequencer` vor den Providern verborgen.

ITransmitter und IEditorAPI Diese Schnittstellen realisieren das Fassadenmuster [GHJV95] für die Subsysteme der XMPP-Funktionalität und der Behandlung von Eclipse-Editoren. Ich erläutere sie in Abschnitt 5.3.7 ausführlicher.

Erzeugen und Senden von Aktivitäten

Nach dem groben Überblick über die Klassen möchte ich nun ihr Zusammenspiel genauer erläutern. Ich erkläre im Folgenden zuerst welchen Weg ein exemplarisches `IActivity`-Exemplar von der Erzeugung, bis zum Verschicken über XMPP, durchläuft. Abbildung 5.4 zeigt diesen Vorgang als Sequenzdiagramms:

1. Der Benutzer vollführt in einem Editor einen Tastenanschlag. Der `EditorManager` (implementiert `IActivityProvider`) wird durch den zuvor angemeldeten `IDocumentListener` der Eclipse-Umgebung von diesem Vorgang unterrichtet. Er erzeugt ein Exemplar vom Typ `TextEditActivity`, welches alle zur Reproduktion benötigten Daten enthält (siehe auch Abschnitt 5.3.2).

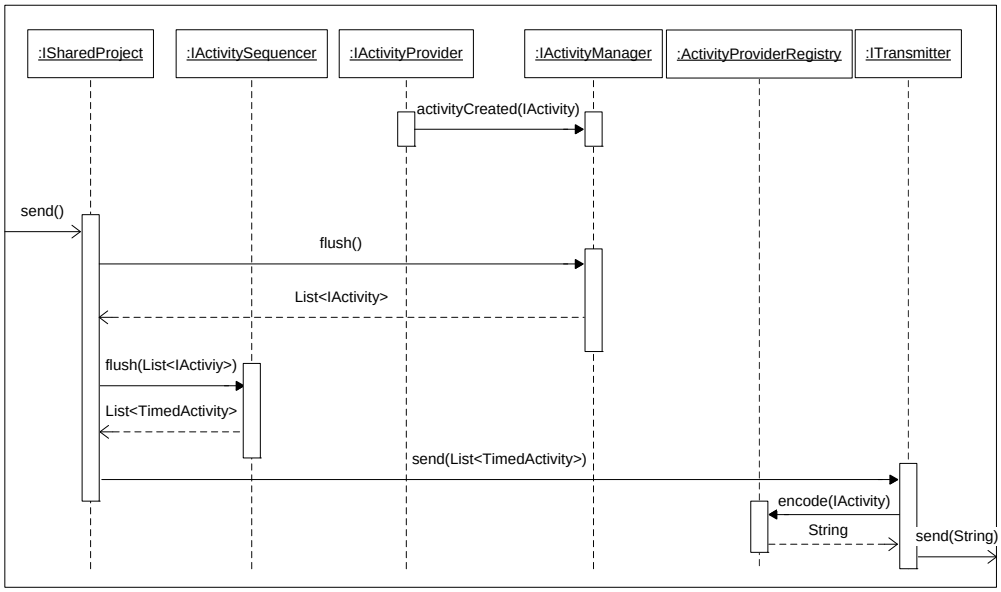

Abbildung 5.4.: Sequenzdiagramm, welches das Erzeugen und Senden von Aktivitäten zeigt.

2. Der `IActivityManager` empfängt das `IActivity`-Exemplar durch seine Beobachtung des Providers. Er nimmt die erzeugte Aktivität in seiner chronologisch geordneten Liste auf.

3. Das System beauftragt das Senden aller vorliegenden Aktivitäten. Dies passiert in Saros in festen Zeitabständen. Komplexere Algorithmen, die beispielsweise erst nach einer gewissen Verzögerung neue Aktivitäten verschicken, wären allerdings genauso denkbar.

4. Das `ISharedProject`, welches die laufende DPP-Sitzung repräsentiert, leert durch Aufruf der `flush`-Methode die Liste der aufgelaufenen Aktivitäten beim `IActivityManager`.

5. Die Liste der Aktivitäten wird der Aktivitätsablaufsteuerung (`IActivitySequencer` übergeben, welche jede Aktivität mithilfe eines lokalen Zählers mit einem Ganzzahl-Zeitstempel ergänzt. Dazu kapselt sie die Aktivitäten in `TimedActivity`-Exemplare. Der interne Zeitstempel der Ablaufsteuerung wird für jedes Exemplar inkrementiert.

6. Das `ISharedProject` übergibt die erhaltene Liste von `TimedActivity`-Exemplaren an den aktuellen `ITransmitter`.

7. Der `ITransmitter` wandelt alle Aktivitäten in XML-Repräsentationen um. Dazu bedient er sich der `IActivityProviderRegistry`. Diese leitet die Anfrage an die angemeldeten `IActivityProvider` weiter. Der für

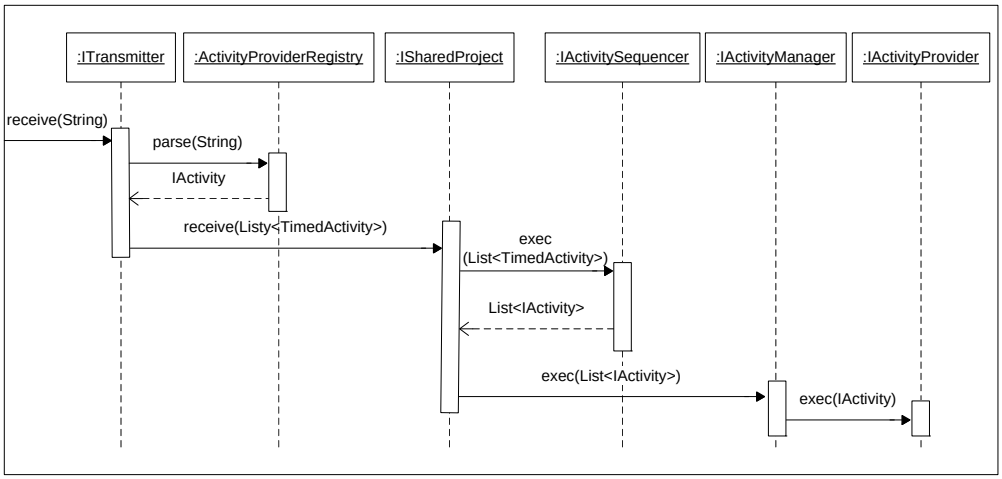

Abbildung 5.5.: Sequenzdiagramm, welches das Empfangen und Ausführen von Aktivitäten zeigt.

die `IActivity` zuständige Provider liefert eine Zeichenkette, welche die Aktivität im XML-Format repräsentiert. Der `ITransmitter` kombiniert alle Zeichenketten der Aktivitäten in einer XML-Nachricht, die zusätzlich auch den Zeitstempel enthält und sendet die Nachricht dann über das XMPP-Netzwerk.

Empfangen und Verarbeiten von Aktivitäten

Um den Vorgang vollständig zu erläutern, muss noch das Empfangen und Verarbeiten von Aktivitäten erläutert werden. Abbildung 5.5 zeigt ein Sequenzdiagramm dieses Prozesses. Dabei laufen folgende Phasen ab:

1. Der `ITransmitter` des Empfängers nimmt die Nachricht entgegen. Er erstellt aus dem XML-Inhalt der Nachricht zunächst wieder `IActivity`-Exemplare. Dazu lässt er jedes XML-Element, welches eine Aktivität repräsentiert, von der `ActivityProviderRegistry` parsen. Die Registratur hat Kenntnis über alle Provider und leitet die Anfrage an sie weiter. Der zuständige Provider wandelt die Zeichenkette, die er ja zuvor ebenfalls erstellt hat, wieder in ein `IActivity`-Exemplar um. Das Entwurfsgeheimnis, wie eine `IActivity` als Zeichenkette dargestellt wird, wird von dem Provider gehalten, der auch für die Erzeugung und Ausführung der Aktivität zuständig ist.

   Der `ITransmitter` sammelt all diese Aktivitäten und kapselt sie ausgehend von dem Zeitstempel, der ebenfalls Teil der Nachricht ist, in `TimedActivity`-Exemplare. Die Liste dieser Exemplare leitet er nun an das `ISharedProject`.

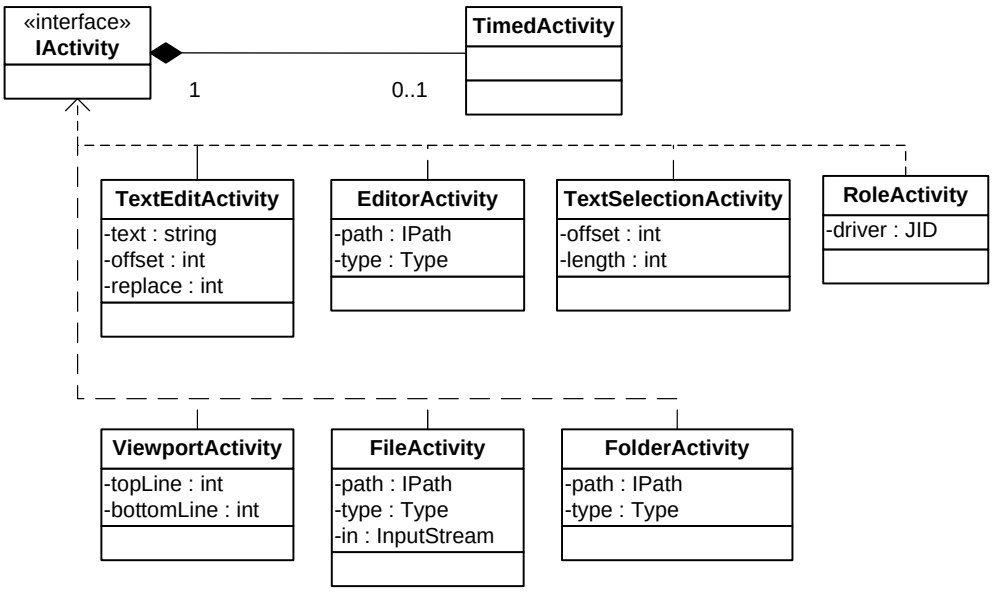

Abbildung 5.6.: Klassendiagramm aller Aktivitäten.

2. Das `ISharedProject` fungiert als Treiber der Sitzung. Es leitet die Liste an die Aktivitätsablaufsteuerung weiter.

3. Die Aktivitätsablaufsteuerung (`IActivitySequencer`) vergleicht die Zeitstempel der eingehenden Aktivitäten mit dem internen Zähler. Alle Aktivitäten, die durch ihren Zeitstempel auf eine rechtmäßige Ordnung hinweisen, werden als Listenresultat geliefert. Deutet der Zeitstempel jedoch auf eine Divergenz hin, so werden die unzulässigen Aktivitäten nicht als Teil des Resultats zurückgeliefert. Stattdessen werden sie so lange vorgehalten, bis bei einem neuen Aufruf die fehlenden Aktivitäten übergeben wurden, erst dann liefert die Methode die aufgestauten `IActivity`-Exemplare.

4. Die Liste mit nun korrekt geordneten Aktivitäten wird dem Aktivitätsverwalter (`IActivityManager`) übergeben.

5. Der Aktivitätsverwalter leitet die Aktivitäten an alle `IActivityProvider` weiter. Die Provider sind für das Ausführen der Aktivitäten zuständig.

Liste der Aktivitäten

Im Mittelpunkt der DPP-Sitzung steht die `IActivity`-Schnittstelle. Ihre implementierenden Klassen kapseln die replizierbaren Vorgänge. Abbildung 5.6 zeigt ein Klassendiagramm aller Aktivitäten. Diese sind im Einzelnen:

EditorActivity Behandelt Wechsel der aktuellen Driver-Editoren. Driver-Editoren

sind die Editoren, welcher der aktuelle Driver gerade geöffnet hat (siehe auch Abschnitt 5.1.3). Die Klasse behandelt das Aktivieren[9], Schließen und Speichern von Editoren.

TextEditActivity Behandelt Textänderungen im gerade aktiven Driver-Editor. Dies beinhaltet sowohl das Einfügen von neuem Text, als auch das Löschen von vorhandenem Text. Die Attribute der Klasse sind, die Zeichenposition an der dieser Vorgang beginnt, die Länge des zu löschenden Textes und der einzufügende Text.

TextSelectionActivity Behandelt Textselektierungen. Die Attribute entsprechen der neuen Anfangsposition des Cursors sowie der Länge der neuen Selektierung. Die Länge kann dabei auch null sein, womit diese Klasse auch zur Angabe einfacher Cursorpositionen benutzt werden kann.

ViewportActivity Behandelt Änderungen am Sichtbereich des Drivers. Die Klasse enthält Attribute für die Zeilennummer der obersten und untersten Zeile einer Datei, die der Driver in seinem Editor gerade sehen kann.

RoleActivity Behandelt das Übergeben der Driver-Rolle an einen anderen Teilnehmer der Sitzung. Das einzige Attribut dieser Klasse entspricht der Jabber-ID des neuen Driver[10].

Die folgenden Klassen beschäftigen sich mit dem Hinzufügen und Entfernen von Ressourcen. Grundsätzlich wird dabei nur zwischen den atomaren Operationen *Hinzufügen*[11] und *Löschen* unterschieden. Mittels dieser Grundoperationen lassen sich alle Arten von Ressourcenänderungen realisieren. Das Bewegen von Dateien resultiert beispielsweise in einer Löschung an der alten Position und einem Einfügen an der neuen Position.

FileActivity Behandelt Dateioperationen. Die Attribute der Klasse entsprechen dem Pfad der Datei (relativ zum Projektpfad) sowie dem Typ (*Hinzufügen* oder *Löschen*) des Vorgangs. Darüber hinaus wird beim hinzufügenden Typ ein `InputStream` mit dem Inhalt der neuen Datei übertragen.

Wie bereits in Abschnitt 5.3.1 erläutert, stellt dieser Vorgang innerhalb von Saros eine Besonderheit dar, weil er nicht wie alle anderen Vorgänge nur als Textnachricht verschickt wird. Ich gehe auf die genaueren Implementierungsaspekte dieser Tatsache in Abschnitt 5.3.2 näher ein.

FolderActivity Behandelt Veränderungen an Verzeichnissen. Sie gleichen bis auf das Fehlen eines `InputStream` der `FileActivity`-Klasse.

[9] Das Aktivieren von Editoren impliziert deren vorhergehende Öffnung.

[10] Da im Moment nur zwei Teilnehmer einer Sitzung unterstützt werden, ist das Attribut eigentlich unnötig. In Anbetracht des nächsten Entwicklungsschritts habe ich es dennoch bereits vorgesehen.

[11] *Hinzufügen* und *Ersetzen* stellen die gleiche Operation dar.

Bis auf drei Ausnahmen werden für alle Dateiänderungen entsprechende `FileActivity`-Objekte erzeugt. Die erste Ausnahme bilden Dateiänderungen, die indirekt aus einer Textänderung folgen, die bereits durch eine entsprechende `TextEditActivity` abgehandelt wurde. Auf die beiden weiteren Ausnahmen gehe ich in Abschnitt 5.3.4 ein.

Aktivitäten als XML-Nachrichten

Das Senden und Empfangen der Aktivitäten wird über die `ITransmitter`-Schnittstelle realisiert (siehe Abschnitt 5.3.7). Nahezu alle Aktivitäten werden dabei als XML-Elemente notiert und den Textnachrichten als zusätzlicher Bestandteil angehangen. Wie dies unter Benutzung der `ActivityProviderRegistry`-Klasse geschieht, habe ich bereits in Abschnitt 5.3.2 und 5.3.2 erläutert.

Ein Beispiel für eine solche XML-Nachricht kann man in Listing 5.1 sehen. Das Listing zeigt drei XMPP-XML-Nachrichten, welche mit DPP-spezifischen Daten und Zeitstempeln ergänzt wurden. Die Aktivitäten werden in dem `activities`-tag eingeklammert. Jeder `activities`-tag ist mit einem Zeitstempel versehen und kann mehrere Aktivitäten aufnehmen. Namenskonflikte werden durch den Namensraum `de.fu_berlin.inf.dpp` vermieden.

```
<!-- Message 1 -->
<message from='donald@jabber.org/Smack' type='chat'
    to='scrooge@jabber.org/Smack' id='Ejspq-16'>

    <thread>2nfgI0</thread>
    <activities xmlns='de.fu_berlin.inf.dpp'>
        <timestamp>1</timestamp>
        <!-- open editor and set selection -->
        <editor path='src/IConnectionListener.java'
            type='Activated'/>
        <textSelection offset='159' length='0'/>
    </activities>
</message>

<!-- Message 2 -->
<message from='donald@jabber.org/Smack' type='chat'
    to='scrooge@jabber.org/Smack' id='Ejspq-16'>

    <thread>2nfgI0</thread>
    <activities xmlns='de.fu_berlin.inf.dpp'>
        <timestamp>3</timestamp>
        <!-- set new viewport -->
        <viewport top='5' bottom='48'/>
    </activities>
</message>
```

```
<!-- Message 3 -->
<message from='donald@jabber.org/Smack' type='chat'
    to='scrooge@jabber.org/Smack' id='Ejspq-16'>

    <thread>2nfgI0</thread>
    <activities xmlns='de.fu_berlin.inf.dpp'>
        <timestamp>4</timestamp>
        <!-- insert text 'public␣voi' -->
        <edit offset='59' replace='0'/>public voi</edit>
    </activities>
</message>
```

Listing 5.1: Beispiel einer Nachrichtenübertragung im XML-Format während einer DPP-Sitzung.

Eine Ausnahme dieser Art von XML-Darstellung stellen die `FileActivity`-Objekte dar, welche Dateierzeugungen repräsentieren. Sie werden nicht als Textnachrichten, sondern als Dateiübertragungen realisiert. Diese Funktionalität wird vollständig vom `ITransmitter` übernommen, der die betreffenden Dateiaktivitäten gesondert sendet und empfängt. Außerhalb des Übermittlers treten diese Objekte dagegen als normale `IActivity`-Objekte auf.

Erweiterbarkeit

Das gerade beschriebene System der `IActivity`-Typen ist in zweierlei Hinsicht auf Erweiterbarkeit ausgelegt: Zum Ersten können die bestehenden `IActivity`-Typen einfach durch das Hinzufügen neuer `IActivityProvider`-Implementierungen ergänzt werden. Dazu muss das neue Provider-Exemplar nur an der Singleton-Klasse `ActivityProviderRegistry` bekannt gemacht werden. Der Provider kann dann beliebig selbst erzeugte `IActivity`-Implementierungen verwenden. Durch das mächtige Plugin-System von Eclipse ist es dabei sogar problemlos möglich einen neuen Provider in einem separaten Plugin zu realisieren, welches auf das Saros-Plugin zugreift.

Ein anderer Erweiterungspunkt ist der `IActivitySequencer`. Dieser beschränkt sich in Saros nur auf die Verhinderung von Divergenz. Architektonisch bildet er aber bereits die Grundlage für komplexere Ablaufsteuerungen. Dadurch, dass alle `IActivity`-Exemplare durch die zentrale Verwaltungsstelle geschleust werden, wäre es hier beispielsweise auch möglich einen Algorithmus zu realisieren, der mehreren Benutzern gleichzeitig Schreibrechte zubilligt.

5.3.3. Einladungen

Wie in Abschnitt 5.1.2 bereits erläutert, wird das gemeinsame Projekt vor Beginn einer DPP-Sitzung zunächst auf einen gemeinsamen Stand gebracht. Die Sitzung darf erst beginnen, wenn alle relevanten Dateien (siehe Abschnitt 5.3.4) in allen Kopien identisch sind. Ich nenne den Vorgang, der dies sicherstellt, Synchronisation. Er ist Teil des Einladungsprozesses.

Im folgenden Abschnitt erläutere ich die den Prozess und die Implementierung der Einladungen genauer.

Phasen einer Einladung

Eine Einladung wird immer durch einen Benutzer initiiert. Der Benutzer sucht sich den gewünschten DPP-Partner aus seiner Kontaktliste heraus und wählt über das Kontextmenü den Eintrag „Invite user...“ (siehe auch Abschnitt 5.3.5.

Im Folgenden beschreibe ich die Phasen der Einladung. Ich verwende dabei die Rollenbezeichnungen „Einlader“ und „Eingeladener“. Die Begriffe meinen die Saros-Klienten der betreffenden Benutzer. Ist der Benutzer selber gemeint, so wird dies aus der Beschreibung klar. Einlader und Eingeladener stehen während des Einladungsvorgangs in einem wechselseitigen Dialog. Die Bezeichnungen „lokales“ und „entferntes“ Projekt sind relativ zur handelnden Rolle zu verstehen.

Die Benutzer haben während jeder Phase die Möglichkeit den Prozess über die Abbrechen-Schaltfläche des gerade sichtbaren Assistenten abzubrechen. In diesem Fall schickt der Auslöser der Unterbrechung eine entsprechende Nachricht an den anderen Benutzer. Kommt es während des Einladungsvorgangs zu einem Fehler, wird dieser ebenfalls über eine solche Nachricht mitgeteilt.

1. **Einlader sendet Einladung** - Es wird die Einladungsnachricht an den einzuladenden Benutzer verschickt. An Informationen trägt diese Nachricht nur den Namen des Projekts, welches von dem Einlader ausgewählt wurde.

2. **Eingeladener sendet Bestätigung** - Beim eingeladenen Benutzer öffnet sich ein Assistent, der ihn über den Eingang der Einladung informiert. Bestätigt der Benutzer die Einladung, wird eine Bestätigungsnachricht an den Einlader gesendet.

3. **Einlader sendet Dateiliste** - Der Einlader erhält die Einladungsbestätigung. Es wird nun für das Projekt, welches Inhalt der Sitzung sein soll, eine Dateiliste erzeugt. Diese Liste enthält alle Pfade der relevanten Da-

teien und Verzeichnisse des Projekts[12]. Alle Pfade werden relativ zum Pfad des Projekts angegeben. Außerdem wird für jede Datei zusätzlich zu dem Pfad eine Prüfsumme gespeichert, die aus dem Inhalt der Datei berechnet wird. Die Dateiliste wird an den Eingeladenen geschickt.

4. **Eingeladener sendet Dateiliste** - Der Eingeladene empfängt die Dateiliste. Für jedes lokale Eclipse-Projekt wird nun ebenfalls eine Dateiliste erstellt. Diese Listen werden mit dem Vorbild der empfangenen Dateiliste verglichen und jeweils eine Prozentzahl berechnet, welche angibt wie viele Dateien des entfernten Projekts mit dem lokalen Projekt übereinstimmen. Das lokale Eclipse-Projekt mit der größten Übereinstimmung zum entfernten Projekt wird als die Basis für die folgende Synchronisation ausgewählt.

   Der Einladungs-Assistent zeigt dem Benutzer die errechnete prozentuale Übereinstimmung an. Der Benutzer gibt nun einen Projektnamen ein. Unter dem Namen wird ein neues Projekt erstellt und alle Ressourcen des zuvor gefundenen lokalen Projekts in das neue Projekt kopiert. Auf diese Weise wird der Inhalt des ursprünglichen Projekts nicht gefährdet.

   Die Dateiliste des ausgewählten lokalen Projekts wird an den Einlader gesendet. Existiert keine Übereinstimmung zwischen dem entfernten Projekt und einem lokalen Projekt, so wird dementsprechend eine leere Dateiliste gesendet und lokal ein leeres Projekt erzeugt.

5. **Einlader sendet fehlende Ressourcen** - Der Einlader empfängt die Dateiliste des Eingeladenen. Durch den Vergleich der Listen hat er nun genaue Kenntnis über die Unterschiede zwischen seinem lokal ausgewählten Projekt und dem entfernten Projekt des Eingeladenen, welches noch in Übereinstimmung gebracht werden muss. Der Einlader sendet alle Dateien, die im entfernten Projekt noch fehlen oder sich in der Prüfsumme unterscheiden. Ein Fortschrittsbalken und ergänzende Textinformationen zeigen dem Benutzer den Stand des Vorgangs an (zum Beispiel „4 of 15 files sent.“. Wurden alle Ressourcen erfolgreich verschickt, ist der Einladungsprozess abgeschlossen für den Einlader abgeschlossen.

   Während dieser Phase kann der Benutzer keinerlei Interaktion mit dem Projekt vornehmen. Andernfalls würden eventuell Änderungen an Dateien vorgenommen, die schon gesendet wurden. Diese Änderungen würden verloren gehen und die Sitzung mit nicht identischen Projekten beginnen.

6. **Eingeladener empfängt fehlende Ressourcen** - Der Eingeladene wartet solange, bis er alle fehlenden Ressourcen empfangen hat. Da er bereits über die Dateiliste des entfernten Projekts verfügt, hat er genaue Kennt-

[12] Mit Ausnahme der in Eclipse als *team private* oder *derived* bezeichneten Ressourcen (siehe Abschnitt 5.3.4).

nis darüber, welche Dateien zur vollständigen Synchronisation übertragen werden müssen und damit auch darüber, wann der Prozess abgeschlossen ist. Ein Fortschrittsbalken und ergänzende Textinformationen geben dem Benutzer während des Vorgangs zusätzlich Auskunft.

Nach dem Empfang der benötigten Dateien werden alle lokalen Dateien, die nicht zu dem entfernten Projekt gehören, gelöscht. Der Einladungsprozess ist damit abgeschlossen.

Anstatt zuerst nur eine Einladungsnachricht zu verschicken, könnte der Einlader in der ersten Phase auch direkt die Dateiliste seines Projekts mitschicken. Da die Dateiliste jedoch per Dateitransfer übertragen wird, erschien es mir ein schlechter Stil zu sein, Dateiübertragungen durchzuführen, solange noch kein beiderseitiges Interesse bestätigt ist.

Klassen

Abbildung 5.7 zeigt die Klassenhierarchie für Einladungen. Die Basis wird durch die Schnittstelle `IInvitationProcess` und ihrer Unterschnittstellen `IIncoming-InvitationProcess` und `IOutgoingInvitationProcess` gebildet. Zugriff auf Netzwerkfunktionen nehmen ihre konkreten Implementierungen durch ein Objekt vom Typ `ITransmitter`.

Die Einladungsklassen realisieren einfache lineare Zustandsmaschinen. Für jede Phase in dem das jeweilige Exemplar der Akteur ist, wird dabei ein eigener Zustand definiert. Bei der Erzeugung wird das Exemplar in einen Startzustand versetzt. Durch den Aufruf von Methoden schreitet der Zustand linear voran, solange bis der letzte Zustand in dem Abschluss der Einladung und dem Beginn der Sitzung mündet.

Die Überprüfung des Zustands und der richtigen Methodenaufrufe erfolgt mittels der Hilfsmethode `assertState(State state)`, welche in der abstrakten Oberklasse `AbstractInvitiationProcess` implementiert ist. Die Methode wird als Wächter in den Methoden der Unterklassen eingesetzt und überprüft, ob das Aufrufen der jeweiligen Methode mit dem aktuellen Zustand überhaupt zulässig ist. Falls nicht, wird eine Ausnahme ausgelöst, der Zustand auf `FAILED` gesetzt und eine Fehlernachricht an den Partner der Einladung geschickt.

5.3.4. Relevante Ressourcen

Saros stellt sicher, dass sich das Projekt bei allen Teilnehmern einer Sitzung gleicht. Grundsätzlich werden dabei alle Dateien beachtet, es gibt allerdings zwei Ausnahmen:

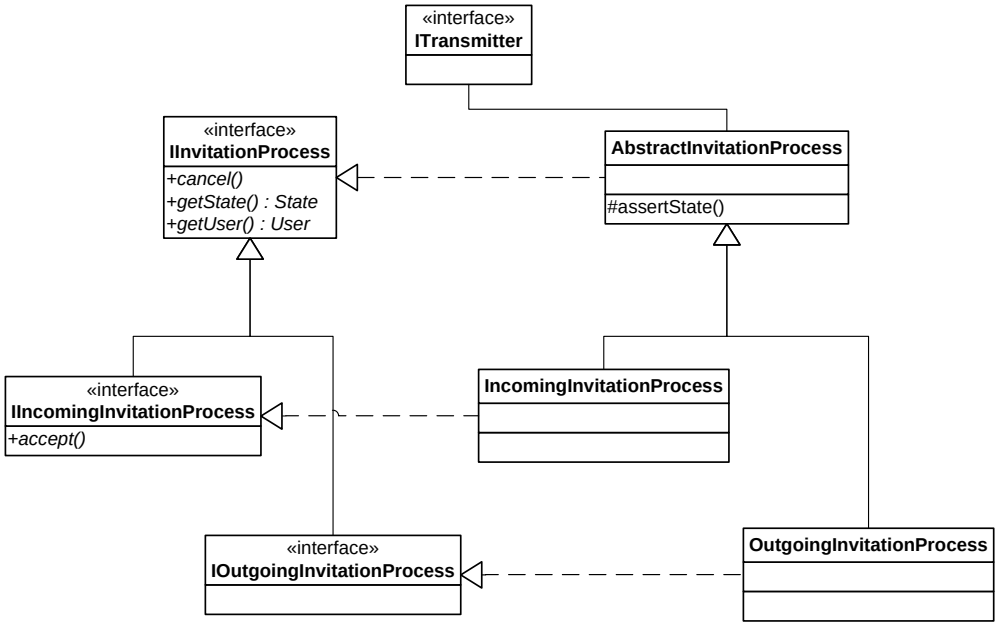

Abbildung 5.7.: Klassendiagramm zu den an Einladungen beteiligten Klassen.

Wird in einem Java-Projekt Code verändert, so ändert sich mit der Javadatei auch die entsprechende Klassendatei. In diesem Fall wäre es unsinnig die veränderte Klassendatei zu übertragen. Nachdem die Änderung der Javadatei übertragen wurde, wird von dem Java-Übersetzer des jeweiligen Systems ohnehin automatisch eine Neuübersetzung der Javadatei durchgeführt, welche auch die entsprechende Klassendatei überschreibt.

Ressourcen, die sich auf diese oder eine vergleichbare Art aus anderen Ressourcen beispielsweise im Zuge einer Übersetzung ableiten lassen, werden in der Eclipse-Umgebung als *derived* bezeichnet. Durch Abfrage des entsprechenden Attributs kann man leicht feststellen, ob es sich um eine relevante übertragungswürdige Datei handelt oder um eine Datei, die automatisch lokal und selbstständig abgeleitet wird. Abgeleitete Dateien sind ein Konzept des allgemeinen Ressourcen-Systems von Eclipse. Daher funktioniert diese Funktionalität auch mit anderen Plugins und Sprachen und ist nicht unmittelbar an Java gebunden.

Eine weitere Ausnahme sind Dateien, bei denen das Attribut *team-private* gesetzt ist. Dateien dieser Art werden von Versionsverwaltungssystemen benutzt, um Informationen über den lokalen Entwicklungsstand zu speichern. Für die verteilte Paarprogrammierung sind sie nicht relevant.

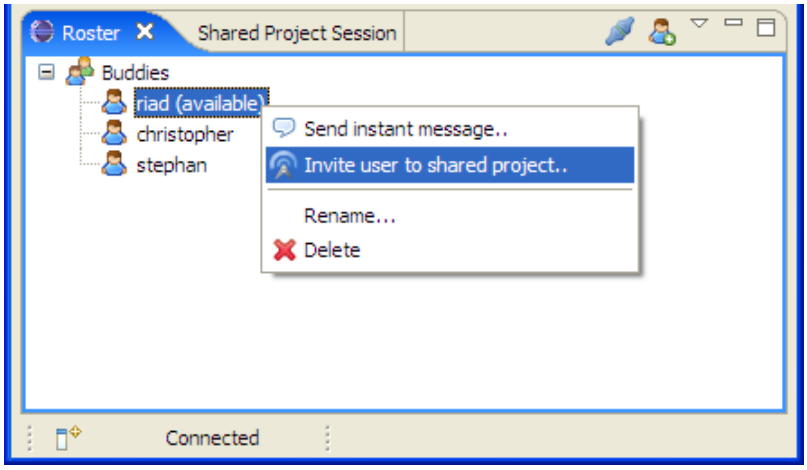

Abbildung 5.8.: Die Kontaktliste mit drei Kontakten. Über das Kontextmenü werden verschiedene Aktionen abgerufen. Die Verbindung zum Server kann durch das Icon, welches ein stilisiertes Kabel zeigt (das erste Piktogramm in der oberen Leiste), beendet beziehungsweise hergestellt werden. Die Statuszeile zeigt zusätzlich den aktuellen Stand der Verbindung an.

5.3.5. Kontaktliste

Die Kontaktliste (siehe Abschnitt 5.1.1) wird durch eine eigene sogenannte *View* visualisiert. Views sind in der grafischen Benutzerschnittstelle der Eclipse-Plattform Fensterabschnitte, die verschiedenste Inhalte anzeigen können und vom Benutzer beliebig angeordnet beziehungsweise geöffnet und geschlossen werden können. Im *Roster*-View werden alle bekannten Kontakte sowie ihr aktueller Status (beispielsweise „anwesend“ oder „nicht anwesend“) angezeigt. Abbildung 5.8 zeigt ein Bildschirmfoto dieser Komponente.

Über das Programmmenü und das Kontextmenü kann der Anwender auf die Funktionen der Kontaktliste zurückgreifen. Dazu gehört vor allem die Möglichkeit Nachrichten zu verschicken. Zu diesem Zweck öffnet sich ein Chat-Fenster welches, wie bei Programmen dieser Art üblich, eine Historie der letzten Nachrichten anzeigt und per Textfeld neue Nachrichten entgegen nimmt. Auch Einladungen zu Paarprogrammier-Sitzungen werden über das Menü getätigt.

Zu den Bearbeitungsmöglichkeiten der Liste gehören das Hinzufügen neuer Kontakte, das Umbenennen vorhandener Kontakte und das Löschen von Kontakten. Für das Erzeugen von Benutzerkonten steht ebenfalls ein grafischer Assistent bereit (siehe Abbildung 5.9).

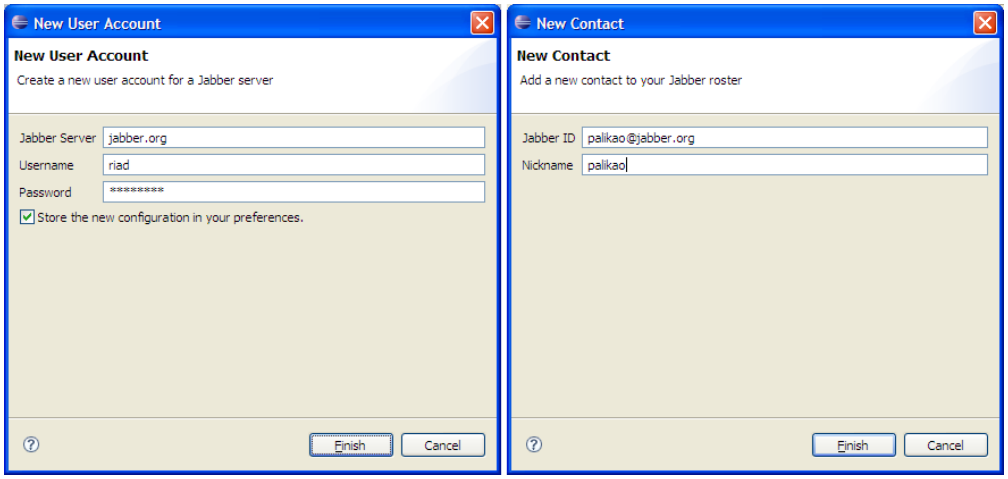

Abbildung 5.9.: Assistenten zur Erzeugung neuer Benutzerkonten und dem Hinzufügen neuer Kontakte.

5.3.6. Awareness

Abbildung 5.10 zeigt, wie ich die in Abschnitt 5.1.3 getroffenen Entwurfsentscheidungen umgesetzt habe. Ich erläutere die interessantesten Implementierungsdetails folgend:

1. In Dateilisten werden die aktuell durch den Driver bearbeiteten Dateien mit farbigen Symbolen markiert. Im Dateinavigator und dem *Java Package Explorer* stehen gelbe Kugeln für Dateien, die der Driver in seiner Eclipse-Umgebung derzeit geöffnet hat. Eine grüne Kugel markiert die Datei, welche der Driver gerade fokussiert hat.

   Wie alle an DPP-Sitzungen interessierte GUI-Komponenten meldet sich das Dekorator-Objekt[13] dabei als Beobachter am `SessionManager` (siehe Abschnitt 5.3.2) an und bekommt so Zugriff auf den Verweis auf die jeweils aktuelle Sitzung.

2. Der Texteditor zeigt die Textauswahlen des Drivers. Die Textauswahlen werden dabei einfach über die Schnittstelle `IEditorAPI` gesetzt (siehe Abschnitt 5.3.7).

3. Die Funktionalität der Anzeige von Sichtbereichen erschien mir gerade für große Codedateien mit womöglich hunderten Zeilen von Code für sinnvoll. Die Funktionalität wird durch das Einführen einer `ViewportAnnotation` realisiert, welche sich immer über den aktuellen Sichtbereich des Drivers erstreckt. Annotationen sind getypte Anmerkungen, die einem Dokument

[13] Nutzung des Eclipse Extension-Points `org.eclipse.ui.decorators`.

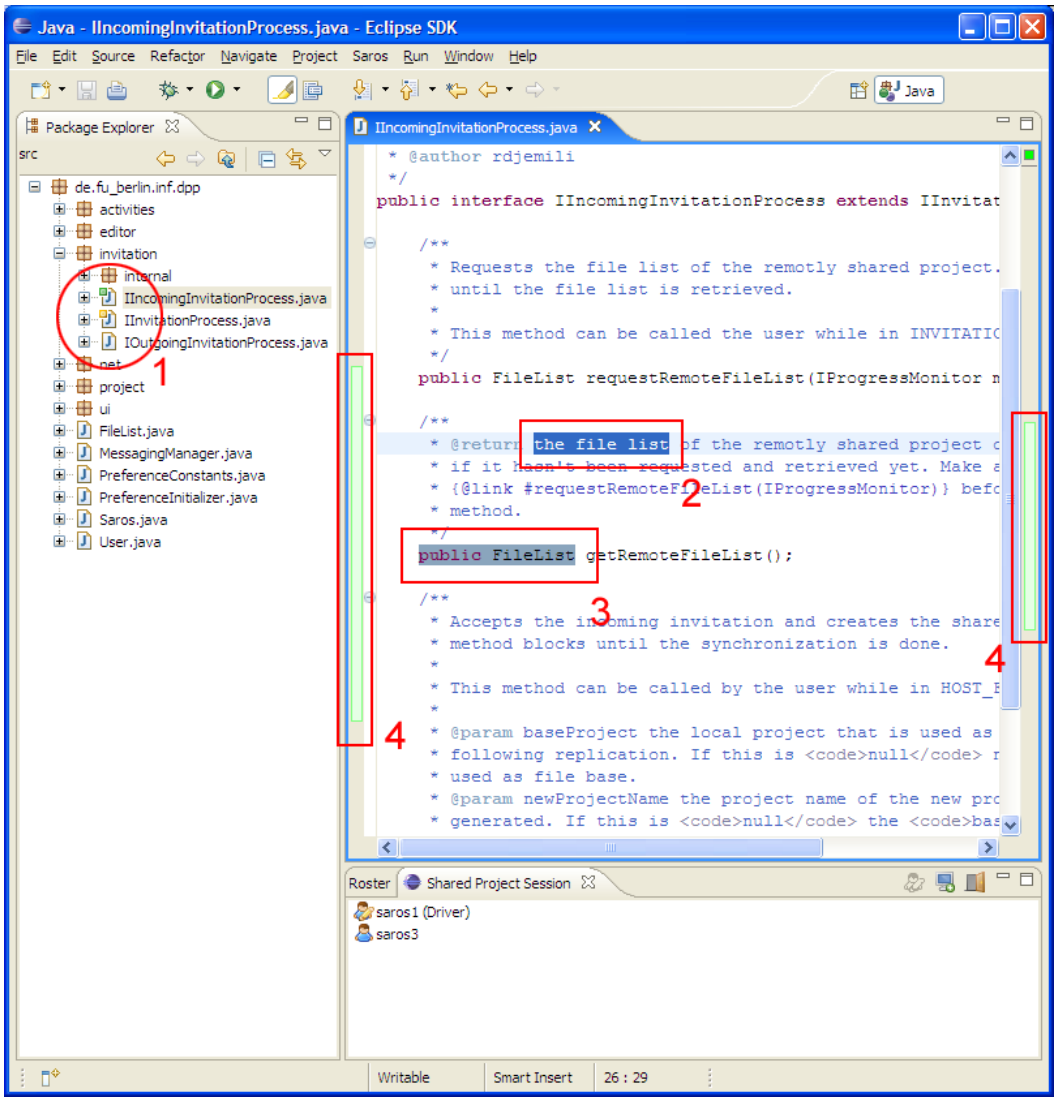

Abbildung 5.10.: Das Saros-Plugin im Einsatz. Das Bildschirmfoto zeigt verschiedene Awareness-Hilfen. Die Zahlen bezeichnen: 1=Dateimarkierungen, welche die geöffneten Dateien des Drivers anzeigen. 2=Die aktuelle Textauswahl des Drivers. 3=Die Textänderung des Drivers (mit einer Hintergrundfarbe versehen). 4=Den aktuellen Sichtbereich des Drivers.

zeichengenau zugeordnet werden können. Eclipse benutzt Annotationen beispielsweise zum Markieren von Übersetzungsfehlern, Lesezeichen und Fundstellen von Suchaktionen. Der Ansatz der Annotationen hat dabei zwei Vorteile:

Zum Ersten werden Annotationen nicht nur im Text grafisch dargestellt, sondern auch im *overview ruler* und *vertical ruler*. Diese Balken zeigen links und rechts von Editoren jeweils Annotationen der geöffneten Ressource an. Der *overview ruler* zeigt dabei alle Annotationen der Datei, während der *vertical ruler* nur die Annotationen zeigt, die im aktuellen Ausschnitt des Editors zu sehen sind. Durch das Überschreiben der Zeichenmethode der Annotation wird die Annotation, wie im Bildschirmfoto zu sehen, als Balken dargestellt wird.

Zum Zweiten kann die Anzeige von Annotationen über die Standardeinstellungen der Eclipse-Umgebung bearbeitet werden. So kann der Benutzer den Sichtbalken beispielsweise in der Art verändern oder sogar ganz abschalten.

4. Um die Textänderungen des Drivers mit einer Farbe zu hinterlegen, habe ich wieder auf eine Annotation zurückgegriffen, die auf die Stelle gelegt wird, an der Eingaben erfolgen. Auch die Darstellung dieser `ContributionAnnotation` kann vom Benutzer in der Art und Farbe verändert werden.

5.3.7. Nutzung des Fassaden-Entwurfsmusters

Eine Herausforderung bei der Implementierung von Saros war die große Abhängigkeit von den beiden Technologien: Eclipse und XMPP (siehe Abschnitt 5.2). Bei beiden Plattformen müssen komplexe Objektstrukturen erzeugt und verwaltet werden. Dies erhöhte die Komplexität des Quellcodes beträchtlich und erschwerte das Schreiben von Komponententests, da das Bereitstellen der für einen Test benötigten Objekte teilweise sehr aufwendig sein kann. Ein typisches Beispiel ist das Testen von Netzwerk-Funktionalitäten für die erst eine teure Netzverbindung hergestellt werden muss.

Eine Lösung für dieses Problem ist die Benutzung des Fassaden-Entwurfsmusters nach *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]. Ihr Zweck wird dort definiert als:

> Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.[14]

[14] Übersetzung aus der deutschen Auflage.

Die Vorteile, die mich im Rahmen von Saros besonders interessierten, waren Folgende:

- Die einheitliche Schnittstelle kann komplexe Methoden bereitstellen, die intern auf eine Vielzahl von Klassen des Subsystems zurückgreifen. Vor dem Aufrufer wird diese Komplexität weitgehend verborgen. Der Code, welcher auf die Funktionalitäten des Subsystems zurückgreift, wird deutlich einfacher. Dabei kann die Schnittstelle perfekt an die Bedürfnisse der eigenen Anwendung angepasst werden.

- Die Funktionalität des Subsystems wird nur noch über die einheitliche Schnittstelle abgerufen. Zwar verbietet das Fassaden-Entwurfsmuster nicht den direkten Zugriff auf Klassen des Subsystems durch Klienten, wird jedoch eine klare Trennung eingehalten, so können die Implementierungen von kompletten Subsystemen problemlos ausgetauscht werden.

  Die Entkopplung kommt auch Komponententests zugute: Innerhalb eines Tests kann beispielsweise die echte und aufwendige Netzwerk-Fassade durch eine einfache Pseudo-Netzwerk-Klasse ausgetauscht werden, die auf Aufrufe einfach mit einem festgelegten und ausreichenden Verhalten reagiert (Stub-Verhalten).

Im Folgenden erläutere ich die beiden Stellen, an denen ich diese Fassaden-Schnittstellen eingeführt habe:

ITransmitter Behandelt die Netzwerkkommunikation. Die Implementierung dieser Schnittstelle lautet `XMPPChatTransmitter` und basiert auf sogenannten „Chatthreads“. Zur Realisierung einfacher Sitzungen mit wenigen Teilnehmern sind sie gut geeignet. Wollte man jedoch zu einem späteren Zeitpunkt große Anzahlen von Sitzungsteilnehmern unterstützen, so könnte man zum Beispiel eine Implementierung auf Basis der *Mutli User Chat Extension* [SA06] realisieren, im Idealfall ohne den Rest des Programms anpassen zu müssen.

IEditorAPI Behandelt einen Teil des sehr komplexen Editor-Subsytems von Eclipse. Eclipse stellt für Tests von grafischen Elementen einen speziellen Testmodus bereit, bei dem eine Eclipse-Umgebung automatisch gestartet wird und alle Komponententests ausführt. Auf diese Art lässt sich beispielsweise die Funktion eines Editors oder einer anderen grafischen Komponente korrekt testen. Allerdings ist dieser Vorgang auch mit einem gewissen Aufwand verbunden, da das Ausführen von grafischen Tests grundsätzlich mehr Zeit kostet als einfache Komponententests ohne grafische Benutzerführung.

Eine andere Möglichkeit des Testens besteht hier darin, zum einem Komponententests zu programmieren, die die Implementierung der Schnitt-

stelle testen. Diese Tests können dann wie beschrieben aufwendig über die grafische Testumgebung getestet. In den Komponententests für Code, der die Fassade benutzt, kann das korrekte Funktionieren der Fassade dann angenommen werden und die Schnittstelle durch eine einfache Pseudo-Implementierung ersetzt werden, die keinerlei grafische Oberfläche benötigt. Die Ausdrucke 5.2 und 5.3 zeigen exemplarisch eine echte Implementierung und eine Testimplementierung.

```
public IEditorPart getActiveEditor() {
    IWorkbenchWindow window = getActiveWindow();
    if (window != null) {
        IWorkbenchPage page = window.getActivePage()
            ;
        if (page != null) {
            return page.getActiveEditor();
        }
    }

    return null;
}
```

Listing 5.2: Die „echte“ Implementierung einer Methode der Schnittstelle `IEditorAPI` in der Klasse `EditorAPI`.

```
public IEditorPart getActiveEditor() {
    return activeEditor; // previously set by
        openEditor(..)
}
```

Listing 5.3: Die Pseudo-Implementierung einer Methode der Schnittstelle `IEditorAPI` in der Klasse `EditorAPIStub`.

6. Erweiterung des ElectroCodeGram-Eclipse-Sensors

In diesem Kapitel erläutere ich Motivation und Implementierung der Erweiterung des ElectroCodeGram-Eclipse-Sensors um neue DPP-spezifische Mikroprozesse.

6.1. Motivation

Wie in Abschnitt 2.4 bereits erläutert ist die Erforschung von Paarprogrammierung mit verschiedenen Herausforderungen behaftet. Untersuchungen zu regulärer und verteilter Paarprogrammierung konzentrieren sich vor allem auf die Gesichtspunkte Effektivität, Codequalität und Kommunikation. Diese Aspekte werden meist durch den Vergleich einer Gruppe von Paarprogrammieren mit einer Kontrollgruppe - zum Beispiel bestehend aus Einzelprogrammierern - gemessen. Üblicherweise lösen beide Gruppen die gleichen Programmieraufgaben.

Gemein haben diese Versuche, dass sie sich auf einfach zu erfassende Metriken stützen. Übliche Metriken sind beispielsweise die benötigte Zeit, die erbrachten Lines-of-Code oder die bestandenen Komponententests. Zusätzlich werden häufig auch Daten über vor- und nachgelagerte Umfragen erhoben. Diese Informationen sind relativ einfach zu erfassen und erlauben den Forschern Studien mit großen Teilnehmerzahlen, die Aussagekraft der Untersuchung wird somit potenziell gestärkt.

Nachteilig an diesen Verfahren ist, dass sie nur Zusammenhänge zwischen den einfach abzulesenden Metriken untersuchen, aber keine genaueren Einblicke in die Sitzungen vornehmen. So gibt es beispielsweise Studien zu Zusammenhängen zwischen den Persönlichkeitstypen der beteiligten Programmierer und der Effizienz ihrer Paarprogrammierung [MP04, KWW+04]. Jedoch arbeiten auch diese Untersuchungen nur über Fragebögen und Metriken, wie der benötigten Zeit zum Umsetzen einer Funktionalität. Wie viel während der Sitzung kommuniziert wurde, wie oft die Driver-Rolle gewechselt wurde, wer die Wechsel initiiert hat, wer wie lange Driver war, wer wie viel Code geschrieben hat und andere detailreichere Maßzahlen gehen nicht in die Untersuchungen ein.

Dies liegt womöglich an dem Aufwand, den eine solche Untersuchung benötigen würde: Die Sitzung müsste mittels Video aufgezeichnet und nachträglich manuell annotiert beziehungsweise ausgewertet werden, ein Aufwand der nur bei einer kleinen Teilnehmerzahl und einer kurzen Laufzeit aufzubringen ist. Eine Lösung der gerade beschriebenen Problematik könnte im Forschungsbereich des *Actual Process* liegen, ich beschreibe diesen Ansatz im folgenden Abschnitt.

6.2. Actual Process und ElectroCodeoGram

Prechelt, Jekutsch und Johnson [LP06] weisen darauf hin, dass die meisten Untersuchungen von Prozessen vereinfachende Annahmen und Abstraktionen vornehmen müssen, um den jeweiligen Sachverhalt untersuchen zu können. Dies sei problematisch, da im Zuge dieser Abstraktion eventuell wichtige Details des Prozesses vernachlässigt werden würden. Der *Actual Process* ist eine alternative Betrachtungsweise, die Vorgänge ausgehend von ihren kleinsten Teilen untersucht.

Im Kern des *Actual Process* stehen daher die sogenannten *Mikroprozesse*. Sie werden von Jekutsch [SJ06] folgend definiert:

> A micro-process describes the sequence of programming activities on a very fine-grained level. The activities studied in micro-process research are of very short duration: Opening of a file, running of the program, renaming of a method, copying and pasting of code blocks, etc.

Im *Actual Process* werden Ströme von Mikroprozessen analysiert, um *Episoden* aufzuspüren. Episoden sind charakteristische Muster bestimmter Abfolgen von Aktivitäten. Ein möglicher Einsatzweck für die Erkennung von Episoden ist die Defektvermeidung: Durch eine aktive Episodenerkennung könnte eine IDE den Programmierer auf fehlerträchtige Entwicklungsepisoden hinweisen. Eine solche Episode könnte beispielsweise viele „Copy and Paste“-Vorgänge beinhalten [ERR].

Da Mikroprozesse sehr fein-granuläre Aktivitäten beschreiben, können in einer Stunde normaler Programmiertätigkeit Hunderte von ihnen anfallen. Daher ist eine automatische Erfassung dieser Mikroprozesse von großer Wichtigkeit. 2005 hat Frank Schlesinger im Rahmen seiner Diplomarbeit unter Betreuung von Sebastian Jekutsch das ElectroCodeoGram (kurz ECG) entwickelt [Sch05]. Das ECG setzt sich aus zwei Bestandteilen zusammen: Aus einem verteilten System zur Erfassung der Mikroprozesse über Sensoren und aus einem System zur zentralisierten Datenerfassung und -analyse, das ECG-Lab.

Die Sensoren werden unmittelbar in verschiedene IDEs und Texteditoren in-

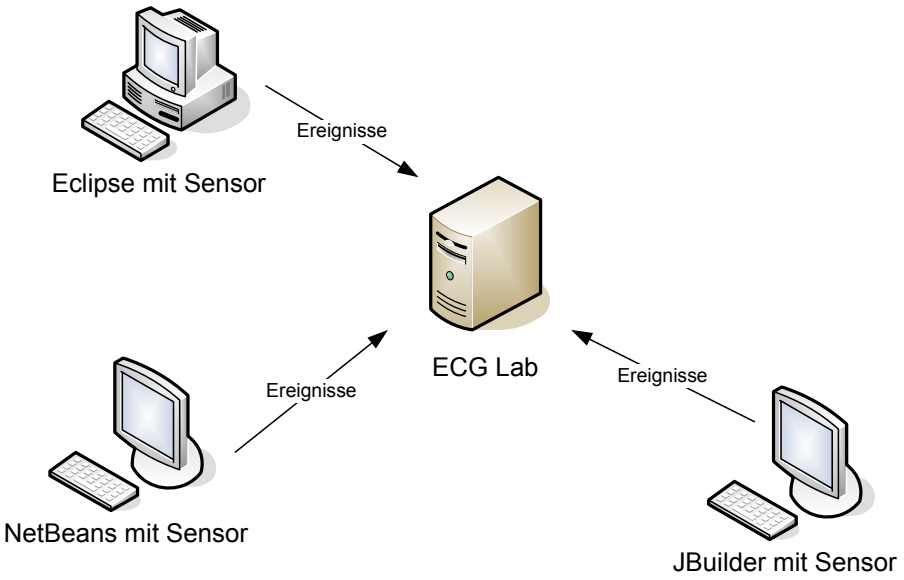

Abbildung 6.1.: ECG Lab mit drei angeschlossenen Sensoren.

tegriert, wo sie die Aktionen des Benutzers in definierte Mikroprozesse übersetzen. Die Mikroprozesse werden dann durch ein Netzwerkprotokoll an das ECG-Lab übermittelt, wo sie mithilfe von sogenannten Modulen verarbeitet werden[1]. Das Plugin-System von ECG erlaubt Module einfach einzufügen. Sie ermöglichen beispielsweise die Mikroprozesse in verschiedenen Datensenken zu speichern oder eine Episodenerkennung vorzunehmen.

Im Rahmen von Schlesigners Diplomarbeit entstand das ECG-Lab sowie ein Sensor für die Eclipse-Plattform, der verschiedene Vorgänge wie beispielsweise Textänderungen oder Fensterselektionen meldet. Die Abbildung 6.1 zeigt die Architektur des ECG mit exemplarischen Sensoren[2].

6.3. Erweiterung um paarprogrammierspezifische Mikroprozesse

Mikroprozesse sind für die Erforschung von Paarprogrammierung deshalb interessant, weil sie ermöglichen die Vorgänge innerhalb der Paarprogrammierung zu untersuchen. Im Gegensatz zu den erwähnten manuellen Verfahren, welche sehr

[1] Sensor und ECG-Lab können sich natürlich auch auf dem gleichen System befinden.
[2] Die Abbildung ist einer Abbildung in der ECG-Diplomarbeit [Sch05] nachempfunden.

aufwendig sind, könnten die relevanten Mikroprozesse durch einen ECG-Sensor auf automatisierte Weise erfasst werden.

ECG eignet sich bereits jetzt zur Untersuchung von Paarprogrammierung und wurde in dieser Weise auch schon eingesetzt [SJ06]. Durch die Erweiterung des aktuellen Eclipse-Sensors durch paarprogrammierspezifische Mikroprozesse könnten zukünftige Untersuchungen weiter erleichtert und verbessert werden.

Während einer Paarprogrammierung fallen Aktionen an, denen im momentanen Sensor keinerlei Mikroaktivität gegenübersteht. Beispielsweise wird das Beginnen oder Beenden einer DPP-Sitzung nicht erkannt. Auch ein Rollenwechsel wird als Mikroprozess nicht erfasst. Um diese Daten momentan zu erfassen, müsste der Mikroprozess-Strom des Eclipse-Sensors von Hand zusätzlich annotiert werden. Würden alle Ereignisse dagegen automatisch erfasst werden, so könnte die Paarprogrammierung ohne besondere Vorbereitung des Arbeitsplatzes untersucht werden.

Im Rahmen der Diplomarbeit habe ich den Eclipse-Sensor des ElectroCodeoGram erweitert. Diese zusätzliche Funktionalität wurde als separates Plugin realisiert, welches sowohl das Saros-Plugin als auch das ECG-Sensor-Plugin voraussetzt. Motivation dieser Entscheidung war die Minderung eventueller Sicherheitsbedenken vonseiten der Benutzer, die Senkung der Saros-Plugingröße und die Verringerung der Codekomplexität des ursprünglichen Saros-Plugins.

Der ECG-Eclipse-Sensor wurde um folgende Mikroprozesse erweitert:

- Das Beginnen und Beenden von DPP-Sitzungen.
- Das Wechseln der Driver-Rolle.
- Das Senden und Empfangen von Kurznachrichten.
- Das Aktivieren/Deaktivieren des Verfolgungsmodus.
- Das Öffnen und Schließen von Dateien durch den Driver.

Während der Entwicklung von Saros hatte ich bereits verschiedenste Beobachter-Entwurfsmuster [GHJV95] implementiert, um eine lose Kopplung zwischen der Kontroll-/Modellebene und der grafischen Benutzerdarstellung zu erreichen. Das SarosECG-Plugin stellt in diesem System nur einen weiteren Beobachter dar. Die Implementierung des Plugins erwies dadurch als einfach.

Um neue Mikroprozesse im ECG bekannt zu machen, müssen sie als XML Schema Definitionen (XSD) deklariert werden. Die Schemata der von mir hinzugefügten Mikroprozesse können in Anhang E eingesehen werden.

7. Evaluation

Nach dem Abnahmetest durch meine Betreuer habe ich Saros am 17.8.2006 einem weiteren Nutzungstest unterzogen. Mein Ziel war weitere Erfahrungen beim tatsächlichen Einsatz des Werkzeugs zu gewinnen und die Robustheit des der Datenübertragung zu prüfen.

Der Test wurde von mir organisiert und mit zwei Kommilitonen des Informatik-Studiengangs durchgeführt. Der Versuchsaufbau war so gestaltet, dass beide Programmierer von gegenüberliegenden Computern aus mittels Saros Paarprogrammieren sollten. Abbildung 7.1 zeigt die Anordnung der Arbeitsplätze[1]. Die räumliche Nähe wurde von mir gewählt, um beide Programmierer gleichzeitig beobachten zu können und um auf zusätzliche VoIP-Software verzichten zu können. Bis auf die mündliche Kommunikation durfte die Zusammenarbeit nur über das Saros-Werkzeug stattfinden.

Da beide Programmierer an keinem tatsächlichen Projekt zusammenarbeiteten, stellte ich ihnen eine Programmieraufgabe. Die Aufgabe entnahm ich der Aufgabensammlung[2] des *Berlin Programming Contest 2005*[3]. Bei der gewählten Textaufgabe handelte es sich um ein relativ einfaches Rucksackproblem (auch Knapsackproblem genannt), welches ein bekanntes Optimierungsproblem der Kombinatorik darstellt.

Die Sitzung wurde von mir beobachtet, um Aufschluss über den Gebrauch des

[1] Die Sitzung fand in den Computerräumen des Instituts für Informatik statt.
[2] http://contest.informatik.hu-berlin.de/2005/
[3] Dieser fand am 25.06.2005 in der Humboldt Universität zu Berlin statt.

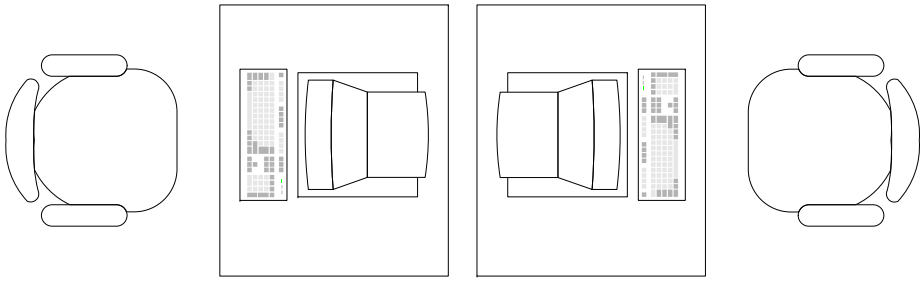

Abbildung 7.1.: Sitzanordnung bei der Versuchssitzung.

Werkzeugs zu sammeln. Sie dauerte etwa 45 Minuten, dabei bemerkte ich folgende interessante Punkte:

- Ein Teilnehmer kritisierte unmittelbar zu Beginn der Sitzung das Fehlen der Möglichkeit gleichzeitig Code schreiben zu können (siehe Abschnitt 5.1.4). Nachdem die Sitzung allerdings weiter vorangelaufen war und beide Teilnehmer mehrmals die Rolle gewechselt hatten, kam diese Kritik nicht mehr auf und wurde auch nach Ende des Versuchs nicht mehr geäußert.

  Ich hatte den Eindruck, dass der vorhandene Schreibmodus beiden Teilnehmern half, sich in den aus der regulären Paarprogrammierung bekannten Rollen einzufinden. Ich fühlte mich in meiner diesbezüglichen Entwurfsentscheidung tendenziell bestätigt.

- Wie von mir vermutet, wurde die Textauswahl des Drivers auch benutzt, um auf bestimmte Codeteile hinweisen zu können und bildete damit einen einfachen Ersatz für Gestik. Auf der anderen Seite hätte der Observer ebenfalls gerne über diese Möglichkeit verfügt und musste stattdessen umständlich verbale Zeilenpositionen erklären. Für die nächste Version von Saros erscheint eine Gestikulierungsfunktion für Observer wünschenswert.

- Wie erwartet machten beide Teilnehmer sehr viel Gebrauch von mündlicher Kommunikation. Hätten die Arbeitsplätze der Programmierer nicht so nah beieinander gelegen, wäre der Einsatz einer VoIP-Software essenziell gewesen.

- Teil der Aufgabenstellung, welche die Programmierer lösen sollten, war das Auslesen einer vorgegebenen Textdatei. Diese Textdatei war zum Anfang der Sitzung nicht in das Eclipse-Projekt geladen worden, sondern wurde erst in das Projekt kopiert, als die Sitzung schon mehrere Minuten vorangeschritten war. Saros erkannte den Vorgang korrekt und kopierte die Datei auch zum zweiten Teilnehmer.

- Beide Teilnehmer machten immer wieder von Möglichkeit Gebrauch das Programm zu starten und sich dessen Ausgabe anzeigen zu lassen. Die Tatsache, dass beide Teilnehmer die Möglichkeit zur Programmausführung hatten, war bezogen auf die Aufgabenstellung sehr nützlich.

- Beide Teilnehmer wechselten mehrmals die Rollenverteilung. Dabei ging der Wunsch des Wechsels meist vom aktuellen Driver aus[4]. Es wäre interessant zu untersuchen, ob sich das Wechselverhalten durch die Imple-

[4] Zum Zeitpunkt der Sitzung konnte nur der gegenwärtige Driver den Rollenwechsel eingeben. Dennoch hätte der Observer den Driverwechsel indirekt anfordern können, indem er verbal seinen Wunsch geäussert hätte.

mentierung einer Driver-Anforderungs-Funktionalität ändern würde.

Natürlich war dies nur ein erster Erfahrungsbericht. In der Sitzung wurde nicht die Arbeit an einem großen Projekt getestet. Dadurch spielten bestimmte Awareness-Hilfen wie beispielsweise die Anzeige der geöffneten Dateien des Drivers oder die Markierung des Sichtbereichs keine Rolle. Umfangreichere Nutzungstests und Erfahrungsberichte wären daher wünschenswert, hätten aber in Anbetracht des Umfangs dieser Diplomarbeit den hier zur Verfügung stehenden Zeitrahmen gesprengt.

8. Fazit

8.1. Zusammenfassung

In dieser Arbeit habe ich die Grundlagen der Paarprogrammierung erläutert (siehe Abschnitt 2). Dazu habe ich in der Literatur genannte Vor- und Nachteile aufbereitet und die Verbreitung in der Praxis skizziert. Als eines der Grundprinzipien von Paarprogrammierung habe ich dabei den Aspekt der *stetigen Durchsicht* identifiziert.

Das Wissen über Paarprogrammierung bildete die Grundlage für meine Erläuterungen zur verteilten Paarprogrammierung (siehe Abschnitt 3). Bei meinem Überblick über grundlegende Vor- und Nachteile dieser Programmiertechnik habe ich auch die Grade an *Parallelität* angeführt, die den Teilnehmern einer verteilten Paarprogrammier-Sitzung zur Verfügung stehen können. Dabei ist zu beachten, dass diese teilweise in direktem Widerspruch zum oben genannten Aspekt der *stetigen Durchsicht* stehen.

Außerdem habe die beiden Werkzeugansätze *Desktop Sharing* und *Collaboration Awareness* anhand verschiedener Kriterien gegeneinander abgewogen. Dabei folgerte ich, dass Desktop Sharing vor allem bei exotischen Artefakttypen Vorteile habe. Bei Projekten mit klar definierten Artefakttypen sei insgesamt eine Collaboration Awareness zu bevorzugen.

Meine gesammelten Erkenntnisse zur regulären und verteilten Paarprogrammierung flossen in eine Umfrage ein, welche untersuchte in welcher weise Softwareentwickler verteilte Paarprogrammierung einsetzen und was für Anforderungen sie dabei an ein Werkzeug stellen würden (siehe Abschnitt 4). Die Teilnehmer gaben unter anderem an, dass sie eher schwierigen, komplexen Code als einfachen Code im Paar bearbeiten würden. Ein weiterer interessanter Aspekt war, dass 27% der Teilnehmer angaben, verteilte Paarprogrammierung trotz räumlicher Nähe zu betreiben.

Die Umfrage half mir bei der Priorisierung von Anforderungen und bei Entwurfsentscheidungen, die ich bei der Implementierung von Saros treffen musste (siehe Abschnitt 5). Saros ist eine Eclipse-Erweiterung für verteilte Paarprogrammierung. Das Plugin basiert auf der *Replikation* von Eclipse-Projekten. Es bietet neben der reinen Übertragung von Textänderungen, auch eine Projektunterstützung und vielfältige Awareness-Hilfen. Die Netzwerk-Kommunikation

wird durch das XMPP-Protokoll realisiert.

Als Vorteil der verteilten Paarprogrammierung habe ich auch die Aufzeichenbarkeit von Vorgängen identifiziert. Dementsprechend lag die Erweiterung des Eclipse-Sensors des ElectroCodeoGrams um neue paarprogrammier-spezifische Mikroprozesse nahe. Diese habe ich in Abschnitt 6 beschrieben.

8.2. Ausblick

Auch wenn sich Saros zu diesem Zeitpunkt bereits in einem einsatzfähigen Zustand befindet, so sehe ich die aktuelle Fassung erst als eine erste Version, die zukünftig noch weiter ausgebaut werden soll. Im Folgenden nenne ich einige der Aspekte, die für zukünftige Weiterentwicklungen von Interesse sind:

Komplexe Vorgänge Bei nahezu allen reproduzierbaren Aktivitäten handelt es sich momentan um atomare Vorgänge[1]. Sämtliche Dateioperationen werden beispielsweise über die beiden Aktionen *Hinzufügen* und *Entfernen* realisiert. Wird eine Datei auf Seite des Drivers verschoben, so wird die Datei zuerst am alten Standort gelöscht und dann eine neue Datei am neuen Ziel erzeugt. Bei Anlegen der neuen Datei findet dann unnötigerweise eine Dateiübertragung statt. Für zukünftige Versionen wäre es zur Optimierung wünschenswert, wenn dies stattdessen in einer *Verschiebe*-Aktion resultieren könnte.

Ähnlich verhält es sich mit Refactorings: Momentan werden Refactorings einfach durch atomare Textänderungen realisiert. Denkbar wäre auch hier, dass anstatt einer Vielzahl von einzelnen Textänderungen, einfach die Refactoring-Anweisung als solche übertragen werden würde.

Problematisch könnten dabei die unterschiedlichen lokalen Einstellungen der Benutzer sein. Bei atomaren feingranulären Aktivititäten kann die korrekte Reproduktion leicht sichergestellt werden. Bei einer komplexen Refactoring-Anweisung hängt das Resultat dagegen von einer Vielzahl von Einstellungen, wie beispielsweise der lokalen Codeformatierung ab.

Weitere Vorgänge Momentan wird vor allem das gemeinsame Bearbeiten von Texten unterstützt. Wünschenswert wäre die Unterstützung von weiteren Java-Funktionalitäten wie beispielsweise einer Form von *remote debugging*. Die dafür benötigten `IActivityProvider` könnten sogar als separate Plugins realisiert werden, ohne das ursprüngliche Plugin anpassen zu müssen (siehe Abschnitt 5.3.2).

[1] Eine Ausnahme bilden Textänderungen. Eine Textoperation, die die „xy“ einfügt, könnte auch in zwei Textoperationen aufgeteilt werden, die „x“ beziehungsweise „y“ einfügen.

Schreibparallelität Obwohl eine *Parallelität auf Schreibebene* teilweise kritisch zu bewerten ist, so stellt sie in jedem Fall ein interessantes Feld für weitere zukünftige Forschungen dar (siehe Abschnitt 3.1.1). Wie diese Funktionalitäten technisch umgesetzt werden könnten, habe ich ebenfalls in Abschnitt 5.3.2 erläutert.

Whiteboard Funktionalität Die Umfrage ergab auch, dass eine Whiteboard-Funktionalität von den Benutzern gewünscht wird. Eine triviale Umsetzung könnte mit einem grafischen Editor erreicht werden, der das grafische Modell zusätzlich als eine Textrepräsentation speichert (beispielsweise im XML-Format). Würde die Replikation auf diese Datei angewandt, könnte sehr einfach ein grafischer Editor realisiert werden.

Voice over IP Es wäre sehr sinnvoll eine VoIP-Funktionalität direkt in Saros zu integrieren. Dies erscheint im Moment noch relativ unwahrscheinlich, da (freie) VoIP-Bibliotheken für Java praktisch nicht existieren. Mit dem Erscheinen von Jingle gibt es aber Grund zur Hoffnung, dass sich daran in absehbarer Zeit noch etwas ändern könnte (siehe Abschnitt 5.2.3).

8.3. Schlusswort

Eine Herausforderung dieser Diplomarbeit war, dass die Anforderungen zu Beginn zwar grundsätzlich gegeben waren, aber bezüglich der Entwurfsentscheidungen sehr viel Entscheidungsspielraum ließen. Insofern stand am Anfang dieser Arbeit zuerst eine umfassende Einarbeitung in das Feld der Paarprogrammierung. Dies ist meiner Meinung nach auch notwendig, wenn man ein Werkzeug für den tatsächlichen Programmieralltag entwickeln möchte.

Die Umfrage machte mir anfangs bedenken, da ich in der empirischen Forschung über keine Erfahrungen verfügte. Sie hat sich jedoch als sehr hilfreich erwiesen und ich freue mich, dass ich hier die Gelegenheit hatte, eine für mich neue Forschungsmethode anzuwenden.

Der Ansatz der Replikation hat mir erlaubt in dem relativ kurzen Zeitrahmen dieser Arbeit doch ein Werkzeug zu erstellen, welches tatsächlich einsetzbar ist. Bei der Realisierung eines *virtuellen Zugriffs* (siehe Abschnitt 3.3.1) hätte ich wahrscheinlich nur einen kleinen Teil an IDE-Funktionalität umsetzen können.

Ich hoffe mit dieser Arbeit einen kleinen Beitrag zur Erforschung und Anwendung von verteilter Paarprogrammierung beigetragen zu haben.

Literaturverzeichnis

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[Bro95] Brooks, Frederick P.: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley Professional, August 1995.

[Con95] Constantine: *L. L. Constantine on Peopleware.* Yourdon Press, Englewood Cliffs, NJ, 1995.

[CS95] Coplien, James O. und Douglas C. Schmidt (Herausgeber): *Pattern languages of program design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[CW00] Cockburn, Alistair und Laurie Williams: *The Costs and Benefits of Pair Programming*, jun 2000.

[DB92] Dourish, P. und V. Bellotti: *Awareness and coordination in shared workspaces.* In: *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Seiten 107–114, Toronto, Ontario, 1992. ACM Press.

[Dob] Dobrowolski, Tomasz: *MoonEdit.* http://me.sphere.pl/indexen.htm (geprüft am 20.7.2006).

[Ecl] Eclipse Foundation: *„About Us"-Webseite des Eclipse-Projekts.* http://www.eclipse.org/org/ (geprüft am 12.8.2006).

[EGR91] Ellis, Clarence A., Simon J. Gibbs und Gail Rein: *Groupware: some issues and experiences.* Commun. ACM, 34(1):39–58, 1991.

[ERR] *Research project on „Micro-process analysis for avoiding programming errors".* http://projects.mi.fu-berlin.de/w/bin/view/SE/ErrorHome (geprüft am 12.8.2006).

[ESP] Team Butterfat: *phpESP (php Easy Survey Package).* http://sourceforge.net/projects/phpesp/ (geprüft am 8.8.2006).

[GAD03] Gallis, Hans, Erik Arisholm und Tore Dybå: *An Initial Framework for Research on Pair Programming.* In: *ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering*, Seite 132, Washington, DC, USA, 2003. IEEE Computer Society.

[GG99] Gutwin, Carl und Saul Greenberg: *The effects of workspace awareness support on the usability of real-time distributed groupware.* ACM Transactions on Computer-Human Interaction, 6(3):243–281, 1999.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gob] 0x539 dev group: *Gobby - a collaborative text editor.* http://gobby.0x539.de/ (geprüft am 20.7.2006).

[GPL] Free Software Foundation, Inc.: *GNU GENERAL PUBLIC LICENSE.* http://www.gnu.org/copyleft/gpl.html (geprüft am 20.7.2006).

[Gre90] Greenberg, S.: *Sharing views and interactions with single-user applications.* In: *Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, Seiten 227–237, New York, NY, USA, 1990. ACM Press.

[HA05] Hulkko, Hanna und Pekka Abrahamsson: *A multiple case study on the impact of pair programming on product quality.* In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, Seiten 495–504, New York, NY, USA, 2005. ACM Press.

[Han05] Hanks, B.: *Empirical Studies of Distributed Pair Programming.* Doktorarbeit, University of California, Santa Cruz, 2005.

[HRGW04] Ho, Chih-Wei, Somik Raha, Edward Gehringer und Laurie Williams: *Sangam: a distributed pair programming plug-in for Eclipse.* In: *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, Seiten 73–77, New York, NY, USA, 2004. ACM Press.

[HT99] Hunt, Andrew und David Thomas: *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Professional, October 1999.

[Hum97] Humphrey, Watts S.: *Introduction to the personal software process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[JB] Borland Software Corporation: *JBuilder 2006.* http://www.borland.com/us/products/jbuilder/index.html (geprüft am 26.8.2006).

[Kar03] Karneges, Justin: *Jabber Enhancement Proposal 0047: In-Band Bytestreams.* Jabber Software Foundation, December 2003. http://www.jabber.org/jeps/jep-0047.html (geprüft am 12.8.2006).

[KS95] Kraut, Robert E. und Lynn A. Streeter: *Coordination in software development.* Commun. ACM, 38(3):69–81, 1995.

[KWW+04] Katira, Neha, Laurie Williams, Eric Wiebe, Carol Miller, Suzanne Balik und Ed Gehringer: *On understanding compatibility of student pair programmers.* In: *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, Seiten 7–11, New York, NY, USA, 2004. ACM Press.

[Lai02] Laitenberger, Oliver: *A Survey of Software Inspection Technologies.* Handbook on Software Eng. and Knowledge Eng., Vol. 2, 2002. 517-555.

[LBSA+06] Ludwig, Scott, Joe Beda, Peter Saint-Andre, Joe Hildebrand und Sean Egan: *Jabber Enhancement Proposal 0166: Jingle.* Jabber Software Foundation, July 2006. http://www.jabber.org/jeps/jep-0166.html (geprüft am 12.8.2006).

[LP06] Lutz Prechelt, Sebastian Jekutsch, Philip Johnson: *Actual Process: A Research Program.* Technischer Bericht B-06-02, Institut für Informatik, Freie Universität Berlin, March 2006.

[Mau02] Maurer, Frank: *Supporting Distributed Extreme Programming.* In: *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, Seiten 13–22, London, UK, 2002. Springer-Verlag.

[McM01] McMahon, Paul E.: *Virtual Project Management: Software Solutions for Today and the Future.* Boca Raton: St. Lucie Press, An Imprint of CRC Press LLC, 2001.

[MM02] Maurer, F. und S. Martel: *Process Support for Distributed Extreme Programming Teams*, 2002.

[MME04] Muldowney, Thomas, Matthew Miller und Ryan Eatmon: *Jabber Enhancement Proposal 0096: File Transfer*. Jabber Software Foundation, April 2004. http://www.jabber.org/jeps/jep-0065.html (geprüft am 12.8.2006).

[MP04] Muller, Matthias M. und Frank Padberg: *An Empirical Study about the Feelgood Factor in Pair Programming*. In: *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04)*, Seiten 151–158, Washington, DC, USA, 2004. IEEE Computer Society.

[NM] Microsoft Corporation: *NetMeeting*. http://www.microsoft.com/windows/netmeeting/ (geprüft am 20.7.2006).

[Nos98] Nosek, John T.: *The case for collaborative programming*. Commun. ACM, 41(3):105–108, 1998.

[PCA] Symantec Corporation: *pcAnywhere*. http://www.symantec.com/ (geprüft am 20.7.2006).

[PM03] Padberg, Frank und Matthias M. Müller: *Analyzing the Cost and Benefit of Pair Programming*. In: *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, Seite 166, Washington, DC, USA, 2003. IEEE Computer Society.

[R] R Foundation: *The R Project for Statistical Computing*. http://www.r-project.org/ (geprüft am 8.8.2006).

[RS02] Rumpe, B. und A. Schröder: *Quantitative Survey on Extreme Programming Projects*, 2002.

[RS05] Rao, Ashwin und Marina Sum: *Developer Collaboration in Our Shrinking World of an Expanding Workforce*. Sun Microsystems, Inc., October 2005.

[RSFWH98] Richardson, Tristan, Quentin Stafford-Fraser, Kenneth R. Wood und Andy Hopper: *Virtual Network Computing*. IEEE Internet Computing, 2(1):33–38, 1998.

[RVN] RealVNC Ltd.: *RealVNC*. http://www.realvnc.com/ (geprüft am 20.7.2006).

[SA05] Saint-Andre, Peter: *Streaming XML with Jabber/XMPP*. IEEE Internet Computing, 9(5):82–89, 2005.

[SA06] Saint-Andre, Peter: *Jabber Enhancement 0045: Multi-User Chat*. Jabber Software Foundation, May 2006. http://www.jabber.org/jeps/jep-0045.html (geprüft am 22.8.2006).

[Sch05] Schlesinger, Frank: *Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung*. Diplomarbeit, Freie Universität Berlin, Institut für Informatik, November 2005.

[SE98] Sun, Chengzheng und Clarence A. Ellis: *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements*. In: *Computer Supported Cooperative Work*, Seiten 59–68, 1998.

[SEE] TheCodingMonkeys - Martin Ott, Martin Pittenauer, Dominik Wagner GbR: *SubEthaEdit*. http://www.codingmonkeys.de/subethaedit/ (geprüft am 20.7.2006).

[SJ06] Schlesinger, Frank und Sebastian Jekutsch: *ElectroCodeoGram: An Environment for Studying Programming*. In: *Workshop on Ethnographies of Code*, March 2006.

[Sma] Jive Software: *Smack API*. http://www.jivesoftware.org/smack/ (geprüft am 12.8.2006).

[SMSA04] Smith, Dave, Matthew Miller und Peter Saint-Andre: *Jabber Enhancement Proposal 0065: SOCKS5 Bytestreams*. Jabber Software Foundation, November 2004. http://www.jabber.org/jeps/jep-0065.html (geprüft am 12.8.2006).

[SS01] Schümmer, Till und Jan Schümmer: *Support for Distributed Teams in eXtreme Programming*. In: Succi, G. und M. Marchesi (Herausgeber): *eXtreme Programming Examined*. Addison Wesley, 2001.

[SWN+03] Stotts, P. David, Laurie A. Williams, Nachiappan Nagappan, Prashant Baheti, Dennis Jen und Anne Jackson: *Virtual Teaming: Experiments and Experiences with Distributed Pair Programming*. In: *XP/Agile Universe*, Seiten 129–141, 2003.

[TDH98] Townsend, Anthony M., Samuel M. DeMarie und Anthony R. Hendiickson: *Virtual teams: Technology and the*

workplace of the future. Academy ol Management Executive, 12(3):17–29, 1998.

[UVN] *UltraVNC.* http://ultravnc.sourceforge.net/ (geprüft am 20.7.2006).

[WK00] Williams, Laurie A. und Robert R. Kessler: *All I really need to know about pair programming I learned in kindergarten.* Commun. ACM, 43(5):108–114, 2000.

[WKCJ00] Williams, Laurie, Robert R. Kessler, Ward Cunningham und Ron Jeffries: *Strengthening the Case for Pair Programming.* IEEE Software, 17(4):19–25, /2000.

A. Ankündigung und Fragenkatalog der Umfrage

A.1. Ankündigung

Es folgt der Originaltext, denn ich zur Ankündigung der Umfrage (siehe Abschnitt 4) verwendet habe.

> Hello,
>
> My name is Riad Djemili. I'm a computer science student at the Freie Universität Berlin in Germany. I'm working on a graduation thesis, which aims at examining distributed pair programming.
>
> For this I've compiled a short questionnaire, which I would like to get as much programmers as possible to answer. The survey is completely academic and has no commercial aims. This questionnaire is mainly aimed at programmers that have already worked with pair programming, ideally even distributed pair programming. It takes about 10 minutes to complete it. The URL is as following:
>
> http://survey.mi.fu-berlin.de/dpp
>
> Why should you take the questionnaire? You would help me in understanding in which ways distributed pair programming is currently used and therefore help me to research how distributed pair programming tools could be improved in general.
>
> If you're interested in further information about the results of my work, you'll be able to leave your e-mail after answering the questionnaire. Your e-mail won't be used for any other reason than this and will be deleted as soon as you've been contacted on the results of the survey.
>
> This survey will be up until the 17th of July 2006. Please feel free to notify other potentially interested participants of this survey.
>
> If you have any comments on this announcement, on the questions

or on the topic in general, please don't hesitate to contact me at my e-mail address:

djemili@inf.fu-berlin.de

Many thanks in advance, Riad Djemili

A.2. Fragenkatalog

A.2.1. Einleitung

By participating in this survey, you'll help us to find out in which ways software developers are using distributed pair programming tools and even more importantly, how these could be improved in the future.

First some terminology so that we know that we are talking about the same things: By the term „distributed pair programming“ we mean an activity where multiple programmers are working in a tight collaborative way on the same task (e.g. code) in real-time.

In contrast to regular „pair programming“ every participant is seated at his own computer/work place. This can be achieved by using collaborative text editors like SubEthaEdit or by using application sharing tools like Virtual Network Computing (VNC). Version control systems like Concurrent Versions System (CVS) or Subversion (SVN) do not quality as „distributed pair programming“.

The distance between you and your programming partner is also not important. Only the fact that you aren't both using the same computer like in regular pair programming. If you sit next to each other, but both of you use separate computers, then this is considered distributed pair programming too for this survey.

If you have no opinion about a question or chose not to answer, just leave the question blank.

A.2.2. Fragen

1. What programming language(s) did you use in the last 6 months? *Java, C++, C#, PHP, Ruby, Python, Perl, Other* ____.

2. How many hours per week do you usually spend programming?

3. How many years of experience in software development do you have?

4. Which of these academic degrees is closest to your own? *High school, Associate degree, College degreem Bachelor degree, Master's degree, Doctoral degree or higher, Other ____.*

5. What is your current occupation? *CVS (Concurrent Versions System), SVN (Subversion), Darcs, Microsoft Visual SourceSafe, I use no revision control system, Other ____.*

6. Which revision control system(s) do you use?

7. Which percentage of your total programming time, do you spend with pair programming (two programmers seated at one computer)? (0% being never and 100% being all the time)

8. If somebody asks you to join a pair programming session how likely is it that you have the project already installed yourself for development? In other words, how likely is it that you both have the code on your own machines?

9. Please rate the following tools in regards of whether you know them and how much you use them for distributed pair programming. (1 = Don't know, 2 = Know but haven't tried, 3 = Have tried, but did not like, 4 = Have tried and liked it, 5 = Use it regulary) *Borland JBuilder, Sun Java Studio/NetBeans, Sangam (Eclipse plugin), SubEthaEdit, Gobby, DocSynch, MoonEdit, ACE, Virtual Network Computing (VNC), Microsoft NetMeeting, Symantec pcAnywhere, Emacs (using make-frame-on-display).*

10. If you know of any other tools that can be used for distributed pair programming please provide them here.

11. If you've already used distributed pair programming: Which mean(s) of communication do you usually use to tell your partner that you want to initiate a distributed pair programming session? *E-Mail, Instant messaging, Chat, Telephone, Directly in person, Other ____.*

12. If you've already used distributed pair programming: For which kind of task(s) do you usually use distributed pair programming? *Design, Writing hard code, Writing simple code, Debugging, Testing, Documentation, Other ____.*

13. If you've already used distributed pair programming: On which kind of projects do you use distributed pair programming? *Commercial, Open Source, Private.*

14. If you've already used distributed pair programming: How far is your distributed pair programming partner usually away? *In same room, In same building, Somewhere else.*

15. If you've already used distributed pair programming: How many people besides yourself are usually part of one distributed pair programming session? *1 (besides yourself), 2 (besides yourself), . . . , 10 (besides yourself), more than 10 (besides yourself).*

How important or unimportant do you consider each of the following features of a distributed pair programming tool to be for a successful and pleasing distributed pair programming experience?

16. Communication with your programming partner

    The meaning of the numbers is as following: 1 = Counter-Productive, 2 = Unnecessary, 3 = Nice to have, 4 = Important, 5 = Essential

    a) You can chat with your partner (e.g. text chatting, instant messaging etc.)

    b) You can talk to your partner by voice (e.g. by using a head set, telephone etc.).

    c) You can see your partner via video (e.g. by using a web cam).

    d) You have a whiteboard editor which allows you to draw geometric shapes and diagrams, which your partner can see.

17. Driver-Observer Interaction

    For the following questions we'll refer to participants that have read/write privileges as drivers and to participants, who only have read privileges as observers.

    The meaning of the numbers is as following: 1 = Counter-Productive, 2 = Unnecessary, 3 = Nice to have, 4 = Important, 5 = Essential

    a) Observers are allowed to browse the code base independently, even while the driver is coding at another part (in contrast to regular pair programming, where two programmers must share one view).

    b) One participant is allowed to be driver at a time and switching driver state is allowed.

c) Multiple participants are allowed to be drivers at the same moment, i.e. they can enter text concurrently at the same time.

18. Awareness

The meaning of the numbers is as following: 1 = Counter-Productive, 2 = Unnecessary, 3 = Nice to have, 4 = Important, 5 = Essential

a) You can see which file a driver has currently opened and focused (even while he currently isn't editing the file).

b) You can see the cursor position and text selections of the driver (even while he currently isn't editing the file).

c) You can see at the exact viewport (i.e. the top and bottom code lines) the driver is currently looking at.

d) You can see which file an observer has currently opened (even while he currently isn't editing the file).

e) You can see the cursor position and text selections of the observer (even while he currently isn't editing the file).

f) You can see at the exact viewport (i.e. the top and bottom code lines) an observer is currently looking at.

g) You can enable a „follow mode“ which allows you to automatically follow the navigation and view of a driver.

h) The actions of a driver are recordable so that you can jump back in time and replay events (e.g. text changes in files) if you missed something.

i) You can see a list of all currently running distributed pair programming sessions in your environment.

19. Compiling and Running

In the following questions we use the term host for the participant that initiated the session and that provides the code base on which is worked on.

The meaning of the numbers is as following: 1 = Counter-Productive, 2 = Unnecessary, 3 = Nice to have, 4 = Important, 5 = Essential

a) Every participant of the session (and not only the session host) is allowed to compile the project and can see compile time errors.

b) Every participant of the session (and not only the session host) can run the project.

c) You can see how your partner is running the project (e.g. what buttons he/she is clicking on the desktop application that you are developing).

Closing Questions

20. If you had a distributed pair programming tool that would provide all your wanted features: Which percentage of your total programming time, would you spend with distributed pair programming? (0% being never and 100% being all the time)

21. How were you referred to this survey? *Via a forum/board, Via a mailing list, Via a news site, Via a colleague/friend, Other.*

22. Are there any aspects of distributed pair programming that you find to be especially important and that weren't checked on the former questions? If yes, which are these?

23. If you have any comments/opinion on the quality of this survey or on questions that you missed, please provide them here.

24. If you want to be notified when the results of this survey become available, you can leave your email here and we will contact you. The results will also be available at this site.

B. Anforderungen

Im folgenden die zwingenden Anforderungen, welche vor Beginn der Diplomarbeit festgelegt wurden. Im Laufe der Arbeit haben sich teilweise neue Anforderungen als sinnreich erwiesen. Manche ältere wurde dafür in Priorität gesenkt.

B.1. Funktionale Anforderungen

- Das Projekt MUSS eine Möglichkeit zur Kommunikation zwischen den Parteien bieten.
- Das Projekt MUSS die Kollaboration von zwei Entwickler in einer Code-Datei unterstützen.
- Das Projekt MUSS das Browsing durch das freigegebene Projekt durch den Guest ermöglichen.
- Das Projekt MUSS es dem Observer ermöglichen zu sehen, wo der Driver ist und diesen automatisch zu verfolgen.
- Das Projekt MUSS es dem Host ermöglichen auf Projektebene Freigaben für die verteilte Paarprogrammierung zu erteilen.
- Das Projekt MUSS es dem Observer ermöglichen zur momentanen Position des Drivers zu springen.
- Das Projekt MUSS auf der Guest-Seite Syntax-Highlighting haben.
- Das Projekt DARF NICHT auf der Guest-Seite Fehler anzeigen, die auf der Host-Seite nicht aufgetreten sind.
- Das Projekt MUSS einen ECG-Sensor besitzen, der zusätzlich zu den existierenden Eclipse-Sensoren DPP-relevante Daten aufzeichnet.
- Das Projekt MUSS damit umgehen können, wenn die Internetverbindung verloren geht oder der andere Teilnehmer die Sitzung verlässt.

B.2. Nicht funktionale Anforderungen

- Das Projekt MUSS einen Bug/Issuetracker, eine öffentliche Mailingliste, eine Projektseite und eine Eclipse-Updatesite haben.
- Alle externen Dokumente des Projektes MÜSSEN in English erstellt werden, d.h. alles außer der Aufgabenstellung und Diplomausarbeitung.
- Der Quellcode des Projektes MUSS dokumentiert sein auf Methodenebene, Klassenebene und Paketebene.
- Das Projekt MUSS für folgende Zielgruppen dokumentiert sein: Eine Benutzerdokumentation, eine Installierungsdokumentation und eine Entwicklerdokumentation.
- Zwei Benutzer, die sich bereits über irgendeinen Bezeichner kennen (Chataccount und ähnliches) kennen und die Software installiert haben, können sich weniger als einer Minute miteinander arbeiten.
- Das Projekt MUSS sich bei bereits vorhandenem Eclipse in weniger als 5 Minuten installieren lassen.
- Das Projekt MUSS sowohl unter Linux Eclipse als auch Windows Eclipse lauffähig sein.
- Das Projekt MUSS Eclipse 3.2 unterstützen.
- Das Projekt MUSS damit umgehen können, wenn ECG nicht installiert ist.
- Das Projekt MUSS mindestens einen Usabilitytest durchlaufen haben (z.B. eine zwei stündige Session mit Christopher und Stephan als Entwicklerpaar).
- Dass Projekt MUSS mindestens mit JDK 1.5 lauffähig sein.

C. Anwendungsfälle

Im Folgenden liste ich die wichtigsten Anwendungsfälle für Saros auf.

C.1. Veröffentlichen eines Projekts

Hauptakteur

Projekt-Host

Vorraussetzungen

- Benutzer hat Saros gestartet
- Benutzer hat eine Jabber-Verbindung hergestellt
- Benutzer leitet bisher kein anderes Projekt

Beteiligte und Interessen

Projekt-Host: Will eine neue Saros-Sitzung mit einem lokalen Projekt einleiten.

Mindestzusicherung

Beim Fehlschlagen der Einleitung bleibt das Projekt lokal immer unverändert.

Haupt-Erfolgsszenario

1. Benutzer öffnet ein Kontextmenü auf dem betroffenen Projekt (entweder im File Navigator oder Java Package Viewer)
2. Benutzer wählt den Menüpunkt „Share project“.

3. Es öffnet sich ein Sitzungsfenster, in dem der Benutzer als einziger Teilnehmer und aktueller Driver vermerkt ist.

C.2. Einladen von Teilnehmern

Hauptakteur

Projekt-Host

Vorraussetzungen

1. Beide Benutzer haben Saros gestartet
2. Beide Benutzer haben die Jabber-Verbindung hergestellt
3. Projekt-Host hat eine Sitzung eröffnet
4. Eingeladener gehört bisher zu keiner Sitzung
5. Eingeladener gehört zum Roster des Projekt-Hosts

Beteiligte und Interessen

- Projekt-Host: Will eine neuen Teilnehmer in das Projekt einladen, damit dieser an der Sitzung teilhaben kann.
- Eingeladener: Will vom Host in das Projekt eingeladen werden.

Haupt-Erfolgsszenario

1. Host öffnet das Roster
2. Host wählt den Eingeladenen aus dem Roster aus und wählt „Invite user to session...“ (entweder über Kontextmenü oder Actionbutton).
3. Beim Eingeladenen öffnet sich ein Fenster mit der Meldung, dass man zu einem Projekt eingeladen wurde. Zusätzliche Informationen: Name (Jabber-ID) des Projekt-Hosts und der Name des Projekts.
4. Der Eingeladene nimmt die Einladung an und gibt den Namen ein, unter welchem das Projekt bei ihm erzeugt werden soll.

5. Saros bestimmt mit welchem lokalen bereits bestehenden Projekt das entfernte Projekt die größte Übereinstimmung hat. Der Inhalt des Projekts wird kopiert, so das die Synchronisation abgekürzt werden kann.

6. Das neue lokale Projekt wird mit dem entfernten Projekt abgeglichen, d.h. es werden alle Dateien des entfernten Projekts kopiert, die nicht im lokalen Projekt vorhanden sind oder überein stimmen.

7. Die Einladung ist beendet. Die Teilnehmerliste der Sitzung führt jetzt beide Benutzer.

Erweiterungen

- 5a) Der Eingeladene lehnt ab. Beim Projekt-Host erscheint eine Meldung, dass die Einladung abgelehnt wurde.

- 6a) Es existiert zu keinem vorliegenden Projekt eine Übereinstimmung. Das neue Projekt wird dann komplett neu erstellt.

- 7a) Während der Replikation läuft etwas schief. Beide Benutzer kriegen eine entsprechende Fehlermeldung angezeigt.

C.3. Übertragen von Textänderungen

Hauptakteur

Driver

Vorraussetzungen

Beide Benutzer sind Teil einer gemeinsamen Sitzung

Beteiligte und Interessen

- Driver: Will in seinem Editor eine Textdatei bearbeiten.

- Observer: Will die Aktionen des Drivers übermittelt kriegen.

Haupt-Erfolgsszenario

1. Der Driver tippt etwas ein. Er kann in dem Editor schreiben.

2. Der Observer hat den gleichen Editor geöffnet. Das Getippte erscheint im Editor an der korrekten Stelle. Er kann nicht in den Editor schreiben.

Erweiterungen

- 2a) Der Observer hat den Editor nicht geöffnet. Die Textänderung wird direkt in die Datei geschrieben.

C.4. Übertragen des aktiven Editors

Hauptakteur

Driver

Vorraussetzungen

Beide Benutzer sind Teil einer gemeinsamen Sitzung

Beteiligte und Interessen

- Driver: Will einen neuen Editor öffnen.
- Observer: Will über die Anwesenheit des Drivers informiert werden.

Haupt-Erfolgsszenario

1. Der Driver wählt eine Ressource im Navigator und öffnet einen Editor.
2. Beim Observer erscheint bei der Ressource, die der entfernten Ressource entspricht, eine kleine Markierung.

D. Anleitungen

D.1. Benutzeranleitung

```
Saros Readme

-----------------------------------------------------------------
Content

1. Introduction
2. Installation
3. Connecting
4. The roster
5. Starting and joining a shared project
6. Working in a shared project session
7. Author and feedback

-----------------------------------------------------------------


1. Introduction

  Saros is a Eclipse plugin for distributed pair programming. It
  features code editing in real-time, view port and text
  selection awareness and full project support. Saros is
  is free open source software, licensed under the GPL.

2. Installation

  The update site can be found at http://riad.de/thesis/update

  Detailed instructions:
  - Open Eclipse
  - Open the software updates manager
    (Menu > Help > Software Updates > Find and Install...)
  - Select 'Search for new features to install' and click on next
  - Click on 'New Remote Site...'
  - Enter 'Saros' as name and 'http://riad.de/thesis/update' as
```

```
    URL. The new location should have been added to the list and
    be selected.
  - Click on 'Finish'
  - Check the plugin in the 'Search Results' and select 'Next'
  - Accept the license agreement and select 'Next'
  - In the next window select 'Finish' to start installation
  - Select 'Install' on the Feature Verification
  - Select 'Yes' when you're asked to restart

3. Connecting

  You need a Jabber account to use Saros. You can create a new
  account from Saros if you don't already have one.

  Creating a new account:
  - Open the Create Account wizard
    (Select it from Menu > Saros > Create Account...)
  - Fill the fields (you can use jabber.org as a Jabber server)
  - Check the 'Store the new configuration in your preferences'
  - Click 'Finish'

  Using an existing account:
  - Open the preferences window (Menu > Window > Preferences...).
  - Select the 'Saros' category.
  - Enter your account data.

4. The Roster

  The roster is where all your contacts are stored and which
  allows you to check their availability and to send them instant
  messages. To open the roster select the roster view from the
  available views (Menu > Window > Show View > Saros > Roster).

  Above the roster you will see plug-icon. Use it to connect or
  disconnect from the server. The button will always show you,
  your current connection status. You can also see the status
  line of the Roster View for more detailed information about
  your connection status.

  By default you will have to click the button to connect, but
  you can also automatically connect on startup. There is an
  option for this in the Saros preferences (Menu > Window >
  Preferences...). Once you've connected your contacts will be
  loaded from the server and shown in the roster viewer.

  To send an instant message to an contact, double-click on their
  name or use the action from the context menu. You can also add,
```

```
  delete and rename contacts.

5. Starting and joining shared project sessions

  You join sessions by either starting session yourself and
  inviting others or by being invited yourself.

  Either way when starting a distributed session, Saros will copy
  the shared project to the machine of the invited user. This
  allows responsiveness and full compiler support for both
  participants of the sessions. To shorten this process, Saros
  will look through the local projects of the invitee and compare
  them with the incoming project. The local project that is
  closest to the remote project, will be copied and used as the
  starting point for the following replication. Saros thus works
  best if you and your programming partner already have similar
  versions of the project installed on your computers.

  Starting a shared project session:
  - Right-Click on a project in your resource navigator (or
    package navigator) and select 'Share project...' from the
    context menu. Now you see the 'Shared Project Session' viewer
    and yourself as only participant.
  - To invite other users, select the 'Roster' viewer again.
  - Right-click on a user and select 'Invite user to shared
    project...'
  - If the other user accepts a progress bar will appear while
    the project is synchronized. If he actively rejects the
    invitation or an error occurs you will be prompted with an
    information dialog.

  Joining a running shared project session:
  - If you're invited a wizard dialog will appear on your screen,
    stating who the inviting user is and the name of the project.
    Select 'Next' if you want to accept the
    invitation. 'Cancel' if you want to reject the invitation.

    Before being able to work on the shared project it needs to
    be copied to your system. Saros automatically looks through
    all your local projects and compares them with the incoming
    project. Saros will show you the project that it has found
    and how much percentage it has in coming with the incoming
    project. This should give you an idea on how long the
    replication will take.
  - Enter the name of a new project. This will be the project
    that will represent the remotely shared project. Don't worry:
    None of your local projects will be modified.
```

```
  - Select 'Finish'.

6. Working in a shared project session

  There are two roles in an session: driver and observer. The
  driver is allowed to edit the resources of the shared project.
  This includes simple text changes on all files and also file
  system changes, like deleting files or copying new files into
  the project. The observer meanwhile is only allowed to - you
  guess it - observe. There can only be one driver at any time.
  To give the driver role to another person, simply select them
  and click 'Give driver role'. This can only be done by the
  current driver.

  There are multiple ways of staying aware of what the driver is
  currently doing:
  - A small yellow dot decorates all files in the resource
    navigator/package explorer that the driver has currently
    opened.
  - A small green dot decorate the file that the driver has
    currently focuses.
  - When the driver moves his cursor the cursor in your editor
    will also move to the corresponding position. Use this to see
    where the driver might write something next. Since text
    selections are also transmitted, you can also use this
    highlight some code for your observer.
  - On the left and right side of editors you can see green bars.
    These show at which part of the file, the driver is currently
    looking at.
  - If the driver writes something, his text will appear with
    some special background color, so that his text changes can
    be easily spotted.

  You can also use the Eclipse preferences to modify various of
  these settings. For example you can use the annotations area of
  the preferences to change the color and text style of the text
  changes by the driver. It also allows you to disable some
  features.

  There is also a 'Follow mode'. If you enable this as an
  observer, you will automatically follow the work of the current
  driver. You will automatically open any file that he opens and
  you will even follow his view port, so that you can exactly
  see, what he can see.

7. Author and feedback
```

```
This plugin was developed by Riad Djemili as part of a diploma
thesis for the Freie Universitaet Berlin.

The official sourceforge project is at:

  http://sourceforge.net/projects/dpp

You can provide feedback and ideas directly to saros@riad.de or
at the forum:

  http://sourceforge.net/forum/?group_id=167540

For bugs you should use the bug tracker:

  http://sourceforge.net/tracker/?atid=843359&group_id=167540
```

D.2. Entwickleranleitung

```
Saros Developer Readme

-------------------------------------------------------------------
Content

1. Introduction
2. The repository
3. Java packages

-------------------------------------------------------------------

1. Introduction

  Saros is a Eclipse plugin for distributed pair programming. It
  features code editing in real-time, view port and text
  selection awareness and full project support. Saros is
  is free open source software, licensed under the GPL.

  This document is for developing on the Saros project. If you're
  completly new to Saros it's better to read the README file
  first.

2. The repository

  The official SVN code repository for Saros can be found at

    https://svn.sourceforge.net/svnroot/dpp

  For more information on how to access it, please see the
  Sourceforge information at

    http://sourceforge.net/svn/?group_id=167540

  The repository is laid out in the following structure. All main
  development happens in the trunk.

  + root
    + trunk
      + de.fu_berlin.dpp

        Contains the code for the Saros plugin (including
        necessary libs and resources)
```

```
      + de.fu_berlin.dpp.test

        Contains unit tests for the Saros plugin.

      + de.fu_berlin.dpp.feature

        Contains the feature project for the Saros plugin

      + de.fu_berlin.dpp.ecg

        Contains the SarosECG plugin - an extension of the ECG-
        Eclipse sensor with new distributed-pair-programming
        micro activities.

      + de.fu_berlin.dpp.ecg.feature

        Contains the feature project for the SarosECG plugin

      + de.fu_berlin.dpp.update

        Contains the update site for the Saros and SarosECG
        plugins.

    + branches

      Contains stable releases and experimental feature branches.

    + tags

      Generally branches should be used over tags.

3. Java packages

  The Saros plugin (located under /trunk/de.fu_berlin.dpp) uses
  the following packages. Generally all main packages have a
  package named 'internal'. This shouldn't be accessed outside of
  the Saros plugin.

  + de.fu_berlin.dpp
    + de.fu_berlin.dpp.activities

      An activity is a action that can be captured and replicated
      on the remote system. It's the basic model for the Saros
      system. This package contains all activity classes.

    + de.fu_berlin.dpp.editor
    + de.fu_berlin.dpp.editor.internal
```

```
  Types that deal with editors and capturing and
  replicated text changes, view port changes, text selections
  and so on.

+ de.fu_berlin.dpp.editor.annotations

  Custom annotations that are used within the editors to add
  visual information (e.g. the current view port of the
  driver).

+ de.fu_berlin.dpp.editor.invitation
+ de.fu_berlin.dpp.editor.invitation.internal

  Types that deal with incoming and outgoing invitations to
  shared projects. They are used to initiate projects.

+ de.fu_berlin.dpp.editor.net
+ de.fu_berlin.dpp.editor.net.internal

  Types that deal with sending and receiving data over the
  network. Currently only a XMPP-implementation is supported.

+ de.fu_berlin.dpp.editor.project
+ de.fu_berlin.dpp.editor.project.internal

  Types that deal with general shared project organization,
  like handling the life cycle of a shared project or
  managing floor control.

+ de.fu_berlin.dpp.editor.ui
+ de.fu_berlin.dpp.editor.ui.actions
+ de.fu_berlin.dpp.editor.ui.wizards
+ de.fu_berlin.dpp.editor.ui.decorators

  Handles visual aspects, like showing wizards and adding
  actions and decorators.
```

E. Schemata der ECG-Mikroprozesse

msdt.dpppsession.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:annotation>
  <xs:documentation>
    This XML schema defines the structure of the "dppSession"
    MicroSensorDataType. The instance documents, the XML documents
    using this schema, are actual "dppSession" MicroActivities.
  </xs:documentation>
</xs:annotation>

<xs:include schemaLocation="msdt.common.xsd"/>

<xs:element name="microActivity">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="dppSession">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="activity">
          <xs:simpleType>
            <xs:restriction base="xs:string">

            <xs:annotation>
              <xs:documentation>
                The "activity" element of the "dppSession"
                MicroActivity can have one of the following values.
              </xs:documentation>
            </xs:annotation>

              <xs:enumeration value="started"/>
              <xs:enumeration value="ended"/>
              <xs:enumeration value="invited"/>
            </xs:restriction>
```

```
            </xs:simpleType>
          </xs:element>

        </xs:sequence>
        </xs:complexType>
      </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

msdt.dppparticipant.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:annotation>
  <xs:documentation>
    This XML schema defines the structure of the "dppParticipant"
    MicroSensorDataType. The instance documents, the XML documents
    using this schema, are actual "dppParticipant" MicroActivities.
  </xs:documentation>
</xs:annotation>

<xs:include schemaLocation="msdt.common.xsd"/>

<xs:element name="microActivity">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="dppParticipant">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="activity">
          <xs:simpleType>
          <xs:restriction base="xs:string">

          <xs:annotation>
            <xs:documentation>
            The "activity" element of the "dppParticipant"
            MicroActivity can have one of the following values.
            </xs:documentation>
          </xs:annotation>
```

```
            <xs:enumeration value="joined"/>
            <xs:enumeration value="left"/>
            <xs:enumeration value="driverChanged"/>
          </xs:restriction>
          </xs:simpleType>
        </xs:element>

        <xs:element name="user" type="xs:string"/>

      </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

msdt.dppeditor.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:annotation>
  <xs:documentation>
    This XML schema defines the structure of the "dppEditor"
    MicroSensorDataType. The instance documents, the XML documents
    using this schema, are actual "dppEditor" MicroActivities.
  </xs:documentation>
</xs:annotation>

<xs:include schemaLocation="msdt.common.xsd"/>

<xs:element name="microActivity">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="dppEditor">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="activity">
          <xs:simpleType>
          <xs:restriction base="xs:string">

          <xs:annotation>
```

```
            <xs:documentation>
            The "activity" element of the "dppEditor"
            MicroActivity can have one of the following values.
            </xs:documentation>
          </xs:annotation>

            <xs:enumeration value="activated"/>
            <xs:enumeration value="closed"/>
            <xs:enumeration value="followActivated"/>
            <xs:enumeration value="followDeactivated"/>
          </xs:restriction>
          </xs:simpleType>
        </xs:element>

        <xs:element name="path" type="xs:string"/>

      </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

msdt.dppchat.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:annotation>
  <xs:documentation>
    This XML schema defines the structure of the "ddpChat"
    MicroSensorDataType. The instance documents, the XML documents
    using this schema, are actual "ddpChat" MicroActivities.
  </xs:documentation>
</xs:annotation>

<xs:include schemaLocation="msdt.common.xsd"/>

<xs:element name="microActivity">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="dppChat">
```

```
      <xs:complexType>
      <xs:sequence>

        <xs:element name="user" type="xs:string"/>
        <xs:element name="message" type="xs:string"/>

      </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```