Freie Universität Berlin

Diplomarbeit zur Erlangung des akademischen Grades Diplom-Informatiker

Behandlung von Nebenläufigkeitsaspekten in einem Werkzeug zur Verteilten Paarprogrammierung

Sebastian Ziller
Matrikelnummer: 4223603
ziller@inf.fu-berlin.de

Betreuer: Christopher Oezbek und Stephan Salinger

Eingereicht bei: Prof. Dr. Lutz Prechelt

01. Oktober 2009

Selbstständigkeitserklärung

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

Berlin, den 1. Oktober 2009
Sebastian Ziller

Inhaltsverzeichnis

1 Einleitung **3**

2 Verteilte Paarprogrammierung mit Saros **4**
2.1 Paarprogrammierung 4
2.2 Verteilte Paarprogrammierung 5
2.3 Saros - Ein Werkzeug zur Verteilten Paarprogrammierung 5

3 Der Prozess des Projekts Saros **10**
3.1 Anforderungen 10
3.2 Projektplanung 10
3.3 Qualitätssicherung 11
3.4 Interne Kommunikation 11
3.5 Externe Kommunikation 12

4 Der Mehrschreibermodus und seine Nebenläufigkeitsprobleme **13**
4.1 Konsistenzerhaltung 13
4.1.1 Konvergenz 13
4.1.2 Kausalitätserhaltung 13
4.1.3 Intentionserhaltung 14
4.2 Rückgängigmachen fremder Änderungen 14
4.3 Synchronisierung komplexer Operationen 15

5 Behandlung der Nebenläufigkeitsprobleme im Mehrschreibermodus **16**
5.1 Echtzeit und Konsistenzerhaltung 16
5.1.1 Zentralisierte oder replizierte Architektur? 16
5.1.2 Jupiter-Architektur 17
5.1.3 Operationstransformation 18
5.1.4 zweidimensionaler Zustandsraum 20
5.1.5 Implementierung 21
5.2 Nebenläufiges Undo 21
5.2.1 Prinzip 21
5.2.2 Integration in Eclipse 22
5.2.3 Wiederherstellung der rückgängig gemachten Operation - Redo 24
5.2.4 Schwierigkeiten bei der Implementierung 25
5.2.5 Grenzen der Implementierung 28
5.3 StopManager zur Synchronisierung komplexer Operationen 29
5.3.1 Ablauf 29
5.3.2 Implementierung 29
5.3.3 Anwendung bei einem Rollenwechsel 34
5.3.4 Anwendung in der Konsistenzwiederherstellung 34

6 Empirische Untersuchung zu Unterschieden des Einzel- zum Mehrschreibermodus **36**
6.1 Aufbau 36
6.2 Ergebnisse und Diskussion 39
6.2.1 Ergebnisse 39
6.2.2 Diskussion 40

6.2.3 weitere Untersuchungen . . . 43
6.3 Einschränkungen der internen Gültigkeit . . . 44
6.3.1 Inkonsistenzen . . . 44
6.3.2 Qualitätsmessung . . . 45
6.3.3 Domänenwissen . . . 46
6.4 Einschränkungen der externen Gültigkeit . . . 47
6.5 Wiederholung der Studie . . . 47
6.6 Stärken von Saros . . . 50

7 Ausblick **51**

8 Zusammenfassung **52**

9 Literaturverzeichnis **53**

10 Abbildungsverzeichnis **55**

A Anhang **56**
A.1 Aufgaben . . . 56
A.1.1 Aufgabe 1 . . . 56
A.1.2 Aufgabe 2 . . . 58
A.2 Fragebögen . . . 59
A.2.1 Fragebogen 1 (vor der Sitzung) . . . 59
A.2.2 Fragebogen 2 (nach der ersten Sitzung) . . . 60
A.2.3 Fragebogen 3 (nach der zweiten Sitzung) . . . 61
A.3 Stichpunkte zur Einweisung in Saros . . . 63

1 Einleitung

Als Paarprogrammierung wird eine Praktik bezeichnet, bei der zwei Entwickler ihren Quelltext an einem gemeinsamen Rechner erarbeiten und dabei abwechslend die Eingabegeräte bedienen [Bec99]. Bei der Verteilten Paarprogrammierung arbeiten die beiden Entwickler auch synchron an einem Dokument, nur von verschiedenen Orten aus [BGS02]. Da nun jeder Entwickler eigene Eingabegeräte zur Verügung hat, ergibt sich als neues Anwendungsszenario der Fall, dass beide gleichzeitig schreiben wollen. Dies birgt für das sie unterstützende Werkzeug neue Herausforderungen. Welche technischen Probleme das mit sich bringt, und ob dieser Fall überhaupt Sinn macht, ist Gegenstand dieser Arbeit.

Im ersten Teil werde ich in die Begriffe “Paarprogrammierung” und “Verteilte Paarprogrammierung” einführen. Außerdem wird das Werkzeug Saros vorgestellt, das Verteilte Paarprogrammierung technisch ermöglicht. Danach werden einige Probleme herausgegriffen, die durch eine gleichzeitige Bearbeitung eines Artefakts hervorgerufen werden.

Anschließend möchte ich anhand von Saros zeigen, wie diese Probleme gelöst werden können. Im vorletzten Teil wird eine empirische Studie vorgestellt, die ich im Rahmen meiner Diplomarbeit durchgeführt habe. Sie soll aufzeigen, welche Unterschiede zwischen dem Fall, dass nur eine Person schreiben kann, zu dem Fall, dass beide Personen schreiben können, bestehen. Zuletzt werde ich einen Ausblick wagen, wie sich das hier behandelte Werkzeug Saros weiter entwickeln könnte und auf welche Ziele man sich bei der Entwicklung konzentrieren sollte.

2 Verteilte Paarprogrammierung mit Saros

2.1 Paarprogrammierung

Programmierung im Team bedeutet üblicherweise die Arbeit an einem komplexen System in Aufgaben aufzuteilen, die dann individuellen Entwicklern zugeordnet werden [Nos98]. Kent Beck empfiehlt in seinem Prozessmodell *Extreme Programming* statt individueller Entwicklung die Arbeit in Paaren [Bec99].

Das bedeutet, zwei Personen sitzen gemeinsam an einem Rechner und entwickeln zusammen Software, d.h. sie arbeiten z.B. gemeinsam am selben Entwurf, Algorithmus, Quelltext oder Test. Dabei bedient ein Partner, der sogenannte Driver, die Maus und die Tastatur und schreibt, während der andere Partner, der sogenannte Observer, die Aufgabe hat kontinuierlich die Arbeit aktiv zu überwachen, auf Defekte zu achten, über Alternativen nachzudenken und sich über strategische Auswirkungen Gedanken zu machen. Die beiden Rollen, Driver und Observer, können spontan und immer wieder getauscht werden. Diese, Paarprogrammierung genannte Praktik, hat einige Eigenschaften, die sich unterschiedlich auf den Entwicklungsprozess und das erarbeitete Artefakt auswirken können.

kontinuierliche Durchsicht

Es hat sich gezeigt, dass Durchsichten ein gutes Mittel sind, um frühzeitig Defekte im Code und Mängel am Entwurf festzustellen [Fag76, Fag86].

Durch den Umstand, dass in der Paarprogrammierung der Observer genannte Partner die Aufgabe hat ständig die Arbeit des Driver zu überwachen, findet eine kontinuierliche Durchsicht statt [WKCJ00], die Defekte und Entwurfsmängel bereits in der Entstehung vermeiden soll [CW01].

erhöhte Motivation und Disziplin

Experimentell hat sich gezeigt, dass Entwickler, die Paarprogrammierung betreiben, zufriedener mit ihrer Arbeit und dem Prozess sind [Nos98]. Darüber hinaus haben Williams et al. eine Art Gruppenzwang feststellen können, den die beiden Partner aufeinander ausüben. Dieses Phänomen, das sie *pair pressure* nennen, führt zu einer höheren Disziplin, die sich z.B. in der besseren Einhaltung von Terminen und Kodierstandards äußert [WKCJ00].

höhere Kosten

Die häufigste Frage, die Kritiker der Paarprogrammierung stellen, ist, warum man zwei Personen eine Arbeit geben soll, die auch eine Person erledigen könnte. Aus ihrer Sicht ist dies nur sinnvoll, wenn das Paar gegenüber einer Einzelperson doppelt so schnell arbeitet.

Es gibt unterschiedliche Ergebnisse bezüglich des Geschwindigkeitsvergleich [Nos98, WKCJ00, CW01]. In keiner dieser Studien konnte aber festgestellt werden, dass ein Paar in der gleichen

Zeit doppelt so viele Funktionalitäten implementiert wie eine Einzelperson. Daher entstehen personaltechnisch erst einmal höhere Kosten. Allerdings lassen sich diese Kosten durch weniger Defekte und bessere Entwürfe wieder einsparen [Bec99].

Anforderungen an die Arbeitsumgebung

Paarprogrammierung stellt an den Arbeitsplatz der Entwickler besondere Anforderungen. Schließlich müssen beide Entwickler bequem nebeneinandersitzen und dabei eine gute Sicht auf den Bildschirm haben können.

Auch sollte die Übergabe der Eingabegeräte mit möglichst geringem Aufwand möglich sein, da Rollenwechsel ein wesentliches Element der Paarprogrammierung darstellen. Dies erfordert teilweise größere Umstruktierungen der Büroräume, manchmal ist es aber auch unmöglich, weil z.B. die Entwickler an verschiedenen voneinander weit entfernten Standorten (z.B. verschiedene Städte) sitzen.

Die Verteilte Paarprogrammierung trägt diesem Umstand Rechnung.

2.2 Verteilte Paarprogrammierung

In der Verteilten Paarprogrammierung arbeiten ebenso zwei Entwickler synchron an dem gleichen Artefakt, nur an jeweils verschiedenen Rechnern.

In einem Experiment [BGS02] hat sich gezeigt, dass sich Verteilte Paarprogrammierung vergleichbar gut auf die Qualität der Arbeit auswirkt wie die nicht verteilte, wie sie in Abschnitt 2.1 beschrieben wurde.

Um Verteilte Paarprogrammierung zu nutzen, werden Werkzeuge benötigt, die diese Praktik technisch ermöglichen. Für solche Werkzeuge gibt es zwei Ansätze, *collaboration transparency*, meist in der Ausprägung *desktop sharing*, und *collaboration awareness* [Han04, LL90]. Beim *desktop sharing* wird lediglich der Inhalt der Arbeitsoberfläche des einen Partner auf den Rechner des anderen übertragen, d.h. zur eigentlichen Arbeit wird Software verwendet, die von nur einem Nutzer gleichzeitig verwendet werden kann. Werkzeuge auf der Basis von *collaboration awareness* dagegen wurden so entworfen, dass sie von mehreren Benutzern gleichzeitig verwendet werden können, sie unterstützen diese Zusammenarbeit z.B. durch eine entsprechende Benutzeroberfläche. Ein Beispiel dieser zweiten Kategorie sind Echtzeit-Gruppeneditoren.

2.3 Saros - Ein Werkzeug zur Verteilten Paarprogrammierung

In die Kategorie der Echtzeit-Gruppeneditoren lässt sich auch die Eclipse-Erweiterung Saros einordnen. Sie wurde 2006 von Riad Djemili im Rahmen seiner Diplomarbeit [Dje06] an der Freien Universität Berlin begonnen und seitdem durch weitere Studien- und Diplomarbeiten erweitert [Gus07, Rie08, Jac09, Rin09, Doh09].

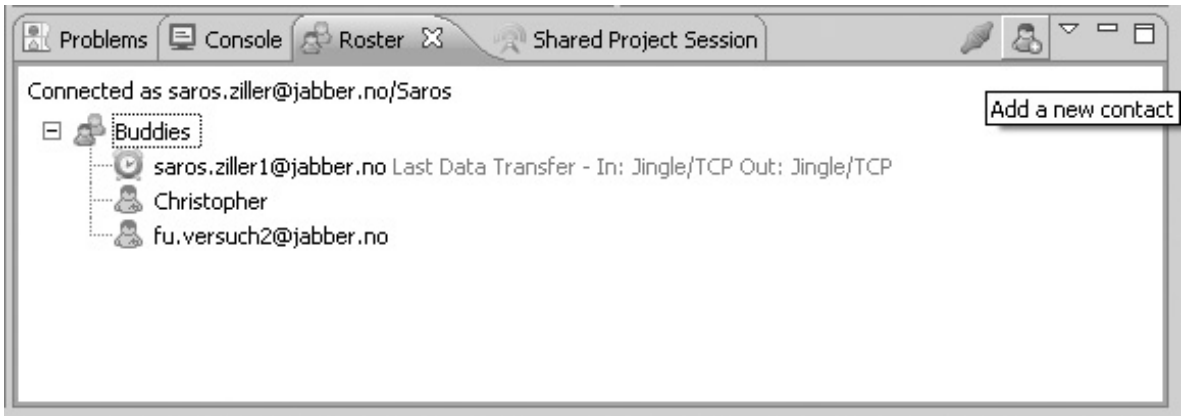

Abbildung 1: Roster View

Saros baut auf der Entwicklungsumgebung Eclipse auf und nutzt zur Kommunikation XMPP, ein offenes Netzwerkprotokoll, das früher Jabber genannt wurde. Eine Entwicklungssitzung mit Saros ist nicht auf zwei Personen festgelegt, sondern es können beliebig viele Entwickler teilnehmen.

Am Beispiel einer typischen Saros-Sitzung werden im Folgenden die wichtigsten Bedienelemente erklärt.

Nach Installation der Erweiterung stehen zwei weitere Bedienfenster (*Views*) mit den Namen *Roster* und *Shared Project Session* zur Verfügung.

Hat man im Konfigurationsdialog eine gültige Jabber-Adresse (nachfolgend JID genannt) angegeben, kann man sich durch Klick auf “Verbinden” im *Roster* verbinden (Abb. 1 oben, 5. Element von rechts). Durch einen Klick auf die Schaltfläche rechts daneben können unter Angabe der entsprechenden Jabber-Adressen neue Kontakte hinzugefügt werden.

Wenn alle für die Sitzung vorgesehenen Kontakte eingetragen sind, kann man nun ein Projekt teilen, indem man im Kontextmenü des zu teilenden Projekts “Share Project” anklickt. Nun kann der Gastgeber unter seinen Kontakten die gewünschten Teilnehmer der zu eröffnenenden Sitzung auswählen. Bestätigen diese die Einladung, erfolgt ein Synchronisierungsprozess, nach dem sie alle eine Kopie des Projekts des Gastgebers haben.

In der eröffneten Sitzung hat jeder Teilnehmer eine der aus der Paarprogrammierung (Abschnitt 2.1) bekannten Rollen, nämlich Driver oder Observer. Nur der Driver darf Änderungen am Projekt durchführen, d.h. Dateiinhalte verändern (z.B. durch Texteingabe) oder Dateioperationen ausführen, also Dateien löschen, hinzufügen oder verschieben. Der Observer sieht all diese Änderungen des Driver und kann sich auch Dateien ansehen, die der Schreiber gerade nicht bearbeitet. Er darf aber keine Änderungen am Projekt vornehmen.

Zu Beginn der Sitzung ist der Gastgeber alleiniger Driver. Aber er hat als einziger Teilnehmer die Möglichkeit Rollenwechsel zu veranlassen (Abb. 2). Er kann also einem anderen Teilnehmer die Rolle des Drivers übergeben und sich selbst zum Observer machen (“exclusive driver role”).

Im Mehrschreibermodus hat der Gastgeber sogar die Möglichkeit die Rolle des Driver an mehrere Teilnehmer zu vergeben, d.h. mehrere Teilnehmer haben die Möglichkeit das Projekt zu verändern. Man kann sich ausmalen, dass dieser Modus zu Konflikten führen kann, wenn zwei Driver gleichzeitig zwei unvereinbare Operationen ausführen möchten.

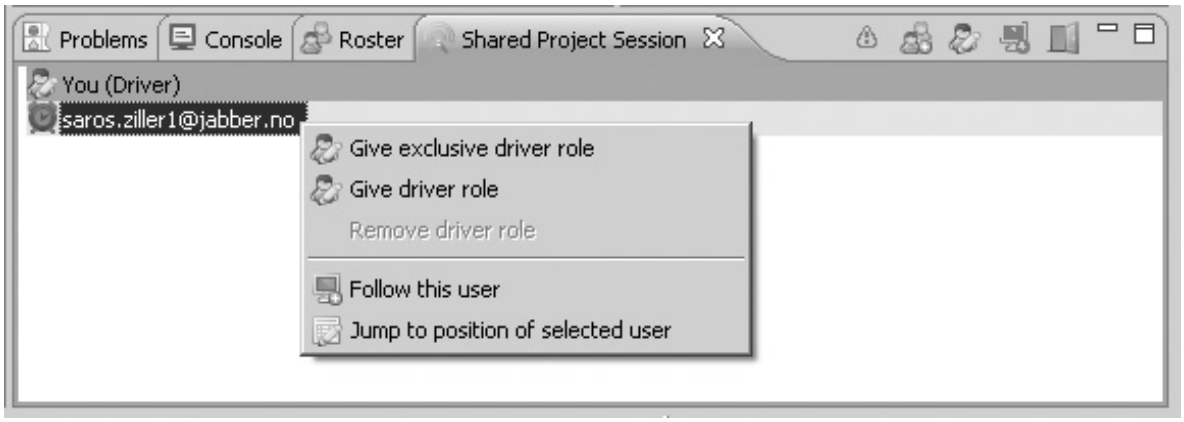

Abbildung 2: Shared Session View: Rollenwechsel

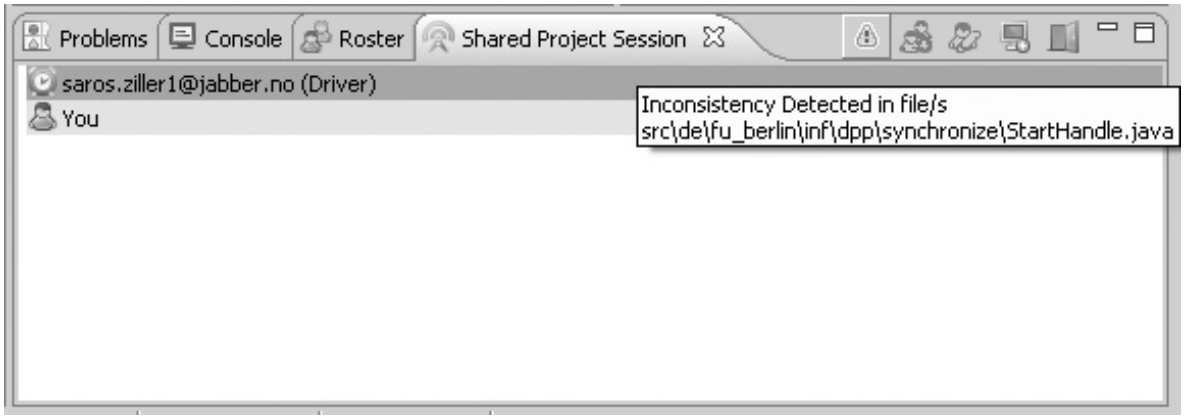

Abbildung 3: Warnung im inkonsistenten Zustand

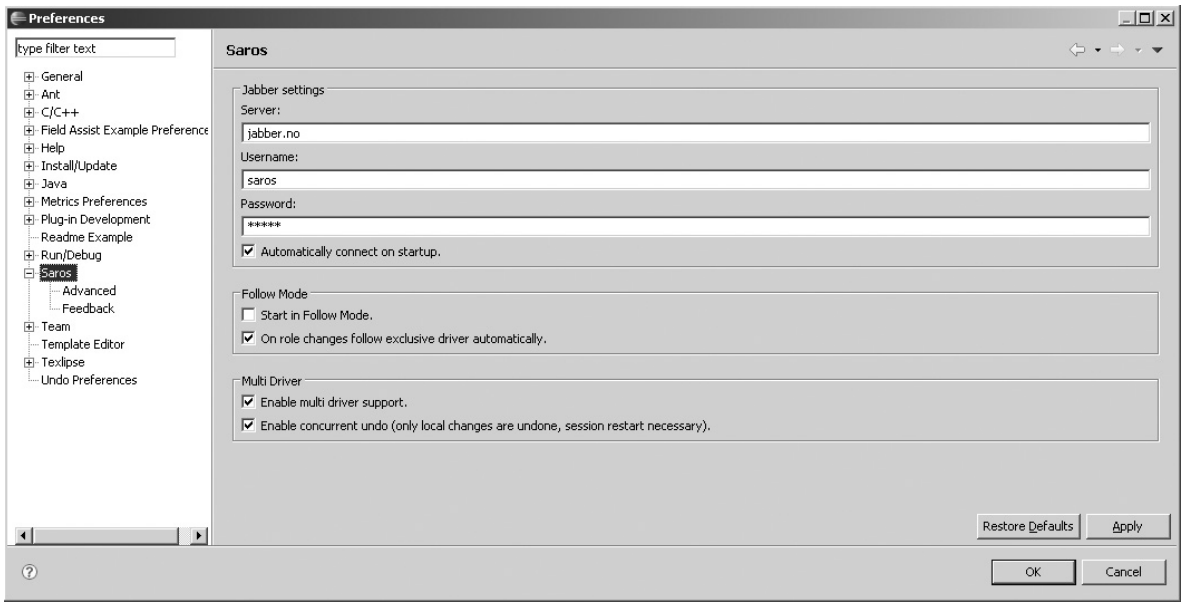

Abbildung 4: Konfiguration von Saros

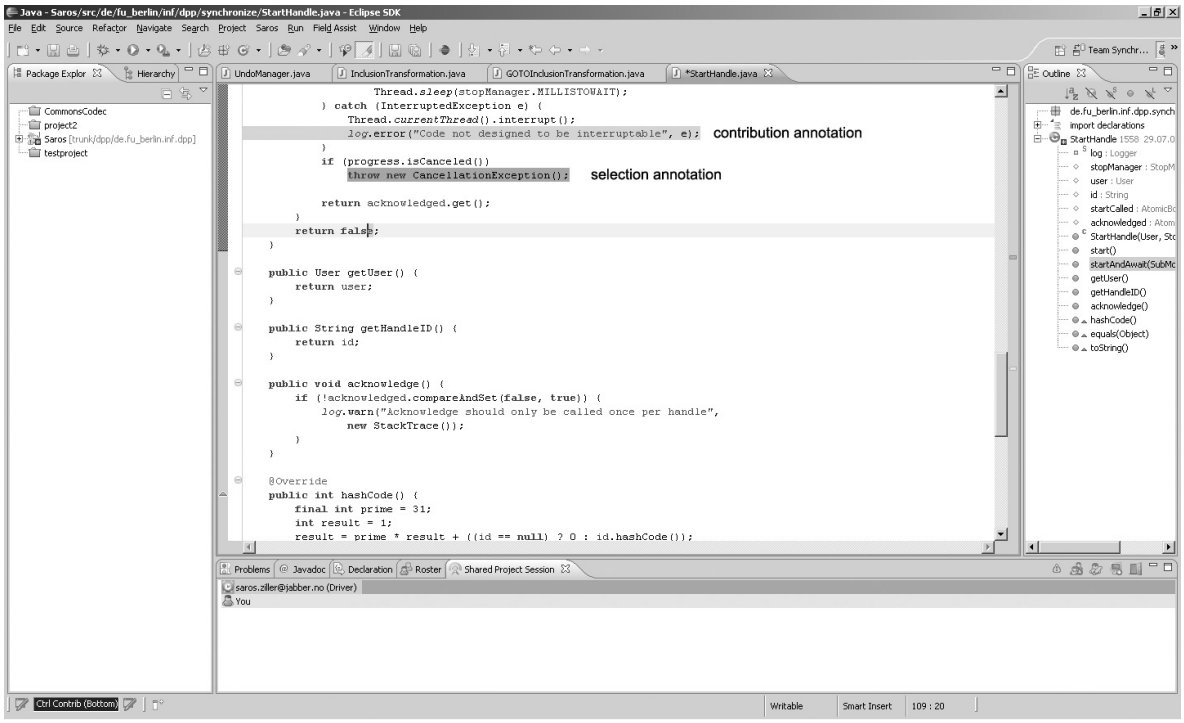

Abbildung 5: Annotationen in Saros

Die Auflösung solcher Konflikte, die im Mehrschreibermodus entstehen, ist Gegenstand dieser Arbeit (Abschnitte 4, 5). Falls ein solcher Konflikt einmal nicht aufgelöst werden kann, ist in Saros eine Notfallsicherung implementiert, die Konsistenzwiederherstellung.

Angenommen zwei Teilnehmer, die beide Driver sind, möchten zwei unvereinbare Operationen ausführen und die Konfliktauflösung versagt.

Dann wird auf der Kopie des einen Teilnehmers die eine Operation ausgeführt und auf der Kopie des anderen die andere Operation. Danach haben also nicht mehr alle Teilnehmer den gleichen Projektstand wie der Gastgeber. Sobald das System diese Inkonsistenz bemerkt, leuchtet ein Warndreieck bei dem betroffenen Teilnehmer auf (Abb. 3).

Klickt der Nutzer dieses an, wird eine Konsistenzwiederherstellung durchgeführt und die Kopie des betroffenen Teilnehmers wird auf den Stand des Gastgebers gebracht.

Um Nachteile auszugleichen, die durch die fehlende physische Präsenz der Teilnehmer entstehen, werden in Saros sog. Annotationen (Abb. 5) eingesetzt.

Das sind grafische Markierungen, welche es erleichtern sollen nachzuvollziehen, was die anderen Teilnehmer gerade machen.

So zeigt die “selection annotation” allen Teilnehmern an, ob und was ein (entfernter) Teilnehmer gerade markiert hat. Die “contribution annotation” dagegen teilt allen Teilnehmern mit, ob und was ein Teilnehmer gerade eingegeben hat. Diese und weitere Annotationen werden von Rintsch [Rin09] ausführlich erläutert.

Als weitere Funktionalität, welche die fehlende Präsenz kompensieren soll, gibt es den Verfolger-

modus. Ist er aktiviert, wird einem bestimmten Nutzer gefolgt, d.h. man sieht immer den Editor, den auch der Verfolgte gerade im Vordergrund hat. Bewegt sich der sichtbare Bereich des Verfolgten nach oben oder nach unten, so ändert sich entsprechend auch der Sichtbereich des Verfolgers. Allerdings funktioniert er nicht horizontal. Bewegt sich der Sichtbereich des Verfolgten nach rechts oder links, ändert sich der Sichtbereich des Verfolgers nicht.

Will man einmalig den Sichtbereich eines bestimmten Teilnehmers sehen, so kann man in der *Shared Project Session View* einen Doppelklick auf dem gesuchten Teilnehmer ausführen und man springt an die Position des Gesuchten ohne den Verfolgermodus zu aktivieren.

Möchte man alleine weiterarbeiten, kann man die Sitzung durch einen Klick auf das Tür-Symbol in der *Shared Project Session View* verlassen. Die übrigen Teilnehmer werden darüber benachrichtigt. Ist man der Gastgeber der Sitzung, beendet man durch das eigene Verlassen auch für die anderen die Sitzung.

Neben den genannten Funktionalitäten hat man natürlich außerdem noch die Möglichkeit die gewohnten Fähigkeiten von Eclipse zu nutzen, um die Entwicklungsarbeit zu unterstützen, wie z.B. Refaktorierungshilfen, Auto-Vervollständigung oder den Debugger.

3 Der Prozess des Projekts Saros

Zu Beginn der Entwicklung von Saros hatten die Entwickler — Studenten, die sich im Rahmen ihrer Abschluss- oder Studienarbeit damit beschäftigten — große Freiheiten hinsichtlich der Gestaltung ihrer Arbeit. Da die ersten drei Entwickler [Dje06, Gus07, Rie08] während ihrer Zeit im Projekt alleine daran arbeiteten, konnten sie den Quelltext komplett nach ihren Wünschen gestalten, ohne sich mit anderen Beteiligten absprechen zu müssen.

Das führte dazu, dass in dieser Zeit die Qualität der Software stetig abnahm, sodass sie Ende 2008 nicht mehr sinnvoll einzusetzen war [Jac09]. Um dem entgegenzuwirken wurde ein Prozess definiert, der die Qualität der Software so verbessern sollte, dass sie sinnvoll gewartet, erweitert und sogar im Unternehmenskontext eingesetzt werden konnte.

3.1 Anforderungen

Von Anfang an lagen im Projekt Anforderungen an die Software vor, welche hauptsächlich die Entwickler an sie gestellt haben [Dje06]. Allerdings änderten sich diese im Lauf der Zeit, während ihre Aktualisierung nur sporadisch erfolgte.

Durch einen Anstieg der Nutzer, einerseits im betrieblichen Umfeld durch den Einsatz in einem Unternehmen, andererseits durch Veröffentlichung der Software auf der Open Source-Plattform Sorceforge.net einhergehend mit steigender Bekanntheit, erwies es sich als notwendig, der Anforderungsverwaltung einen größeren Stellenwert einzuräumen.

Daher haben private Benutzer die Möglichkeit ihre Anforderungen im Internet an das Saros-Team zu übermitteln, entweder durch die Teilnahme an einer kurzen Umfrage [Doh09] oder durch Nutzung des Fallbearbeitungssystems “Feature Requests” von Sourceforge.net. Das Unternehmen, das die Software unter Betreuung des Saros-Teams einsetzt, kann zusätzlich mündlich oder schriftlich seine Wünsche direkt an ein Mitglied des Teams richten, das dann für die weitere Kommunikation und Verwaltung zuständig ist.

Eine Schwachstelle ergibt sich daraus, dass die bloße Existenz von Anforderungen nicht ihre Beachtung bei der Projektplanung erzwingt. Daher kann es natürlich passieren, dass manche Anforderungen lange unbeachtet bleiben. Eine Wiederholung des Wunsches durch andere Benutzer ruft die Anforderung aber wieder ins Gedächtnis und sorgt dafür, dass sie dann meist auch ihren Platz in der Planung findet.

3.2 Projektplanung

Die meiste Entwicklungsarbeit im Projekt wird von Studenten im Rahmen von Studien- und Abschlussarbeiten geleistet. Vor Beginn einer solchen Arbeit legt der Student im Gespräch mit zwei Projektverantwortlichen die Richtung seiner Arbeit fest. Damit entsteht eine mittelfristige Planung von drei bis sechs Monaten.

Konkretere Planungen finden im Abstand von drei Wochen im Gespräch mit den anderen Teammitgliedern statt. Dabei wird besprochen, welche Aufgaben in dieser Zeit zu erledigen sind. Für

die Ausarbeitung und Einhaltung seines Dreiwochenplans ist dabei jeder Entwickler selbst verantwortlich. Dieser sollte sowohl Elemente seiner Studien- oder Abschlussarbeit enthalten als auch bekannte Versagen, die es zu analysieren und zu beheben gilt. Der Plan bleibt über die Zeit seiner Gültigkeit für alle sichtbar im Büro, damit jeder informiert ist, woran die anderen gerade arbeiten.

Eine langfristige Planung wird durch die Wahl der Themen erreicht, die zur Studien- oder Abschlussarbeit ausgeschrieben werden.

3.3 Qualitätssicherung

Um zu vermeiden, dass Defekte in die gemeinsame Codebasis gelangen, wird jede Änderung unter Angabe einer Begründung und eventuell einer Erklärung an die anderen Teammitglieder zur Durchsicht übermittelt. Erst nach zwei bzw. nach 24 Stunden einer positiven Stimme darf die Änderung von der Versionsverwaltung erfasst werden.

Das verzögert natürlich die Überführung in die gemeinsame Codebasis, führt aber dazu, dass viele Defekte und Entwurfsmängel frühzeitig erkannt werden.

Da dennoch Defekte in die Software gelangen wird mittels manueller Tests Fehlverhalten der Software erkannt. Diese alle drei Wochen stattfindenden Tests werden von einem zyklisch wechselnden Testleiter vorbereitet. Dieser trägt alle Neuerungen an der Software zusammen, erstellt daraus einen Testplan, führt diesen mit anderen Teammitgliedern aus und wertet ihn anschließend aus.

Daneben gibt es noch automatisierte Modultests, die mit dem Framework JUnit erstellt werden. Leider lässt sich nicht jedes Modul unkompliziert testen, insbesondere wenn es mit anderen Modulen zu stark interagiert, da man in diesem Fall Stellvertreterobjekte, “Mocks”, bereitstellen müsste, was hohen Aufwand verursachen kann [Rin09]. Daher ist der Quelltext nur unzureichend durch JUnit-Tests abgedeckt.

Darüber hinaus existieren Konventionen zur Gestaltung des Quelltextes, deren Einhaltung in den Durchsichten geprüft wird. Diese sind nicht nur als Einschränkung persönlicher Gestaltungsfreiheiten zu sehen, sondern beinhalten auch Lösungsansätze zu immer wiederkehrenden Problemen, z.B. die korrekte Verwendung des Entwurfsmusters “Dependency Injection” [Fow04] zur Minimierung von Abhängigkeiten zwischen Objekten.

3.4 Interne Kommunikation

Die Kommunikation im Team läuft hauptsächlich über einen E-Mail-Verteiler. Darüber werden sowohl Änderungswünsche am Quelltext und ihre Durchsichten kommuniziert, als auch allgemeine Anfragen und Diskussionen.

Das hat den Vorteil, dass jeder die Möglichkeit hat, über alles informiert zu sein. Zudem werden Diskussionen und ihre Ergebnisse automatisch aufgezeichnet, was es bei schwierigen Entscheidungen ermöglicht, nochmal alle genannten Aspekte des Problems zu reflektieren.

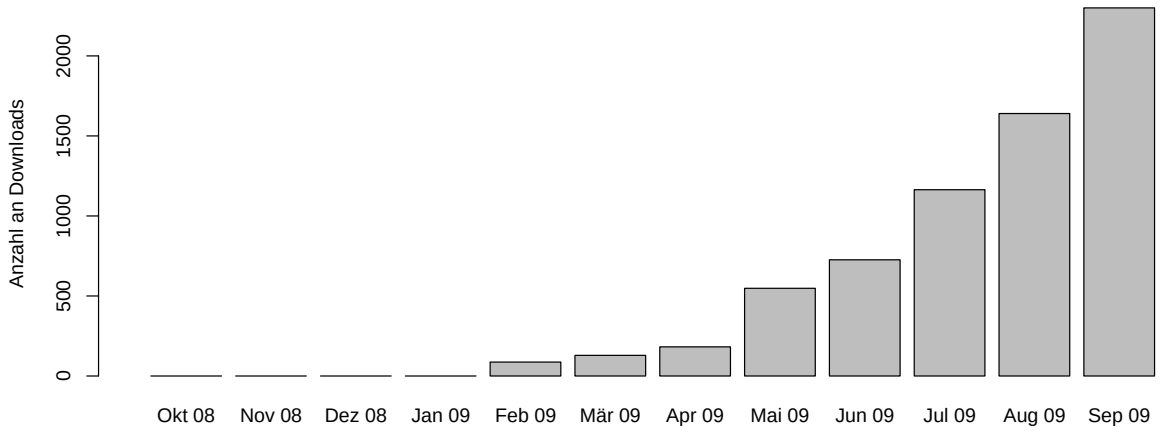

Abbildung 6: Entwicklung der Anzahl an monatlichen Downloads

Daneben wird natürlich auch die mündliche Kommunikation gepflegt, die kürzere Latenzzeiten mit sich bringt und den weiteren Vorteil hat, dass man es vermeiden kann, mit Informationen belästigt zu werden, die man für die eigene Arbeit als irrelevant empfindet.

Als drittes Kommunikationsmedium gibt es noch die Fallbearbeitungssysteme “Bugs” und “Feature Requests” von Sourceforge.net, falls die Kommunikation einen formaleren Rahmen benötigt, wie z.B. bei der Beschreibung eines Versagens der Software.

3.5 Externe Kommunikation

Um Rückmeldung durch Benutzer zu erhalten, haben diese die Möglichkeit an einer kurzen Umfrage teilzunehmen [Doh09], oben genannte Fallbearbeitungssysteme von Sourceforge.net zu nutzen oder dem Team eine E-Mail zu schreiben.

Damit es aber überhaupt Nutzer gibt, wurde der Bekanntheitsgrad der Software seit April 2009 gesteigert, indem an verschiedenen Plätzen im Internet Informationen über die Software hinterlassen wurden, z.B. auf eclipseplugincentral.com, DZone, Freshmeat und im Heise Softwareverzeichnis. Jedes Teammitglied zeichnet für einen oder mehrerer solcher Einträge verantwortlich und aktualisiert sie regelmäßig.[1]

Als Folge dieser Maßnahme wurde die Software zunehmend häufiger heruntergeladen (Abb. 6) und sorgte so für mehr Nutzer.

[1] Nicht immer waren Informationen über Saros erwünscht. So wurde z.B. ein Artikel über Saros von der deutschsprachigen Wikipedia mangels Relevanz abgelehnt.

4 Der Mehrschreibermodus und seine Nebenläufigkeitsprobleme

Der Mehrschreibermodus in Saros, in dem beliebig viele Teilnehmer gleichzeitig Änderungen am gemeinsamen Projekt durchführen können, bringt verschiedene Schwierigkeiten mit sich gegenüber einem Einzelschreibermodus, in dem nur ein Teilnehmer schreiben darf. Im Folgenden werden drei Probleme herausgegriffen und erläutert.

4.1 Konsistenzerhaltung

Die Konsistenzerhaltung setzt sich im Zusammenhang kollaborativer Editoren aus den drei Komponenten

- Konvergenz
- Kausalitätserhaltung
- Intentionserhaltung

zusammen [Sun02].

4.1.1 Konvergenz

Operationen können in unterschiedlichen Reihenfolgen die verschiedenen Teilnehmerseiten erreichen [SJZ+98].

Wie der Beispielabbildung 7 zu entnehmen ist, erreichen den Teilnehmer A die Operationen in der Reihenfolge O1, O2, O4, O3, während der Teilnehmer C die Operationen in der Reihenfolge O2, O4, O3, O1 empfängt. Kommutieren diese Operationen, spielt die Ausführungsreihenfolge keine Rolle.

Ansonsten muss durch ein Nebenläufigkeitskontrollsystem dafür gesorgt werden, dass nach Ausführung aller Operationen ein konvergenter Zustand erreicht wird, d.h. die Kopie des gemeinsamen Artefakts muss auf jeder Seite den gleichen Inhalt haben [Sun02].

4.1.2 Kausalitätserhaltung

Da die Latenzzeiten immer unterschiedliche lang sind, kann es passieren, dass Operationen außerhalb ihrer kausalen Ordnung eintreffen [SJZ+98].

In Abbildung 7 wird bei Teilnehmer B die Operation O1 ausgeführt und danach die Operation O3 erzeugt. Es ist also möglich, dass O3 von O1 abhängt. Auf Seite C hingegen trifft erst O3 ein und dann O1.

Im Falle einer Abhängigkeit der Operation O3 von O1 kann das zu Problemen führen. Als Beispiel könnte O3 Zeichen aus einem Wort löschen, das durch O1 erst eingefügt wird. Wird auf Seite C also erst O3, dann O1 ausgeführt, kommt es nicht zum erwarteten Ergebnis.

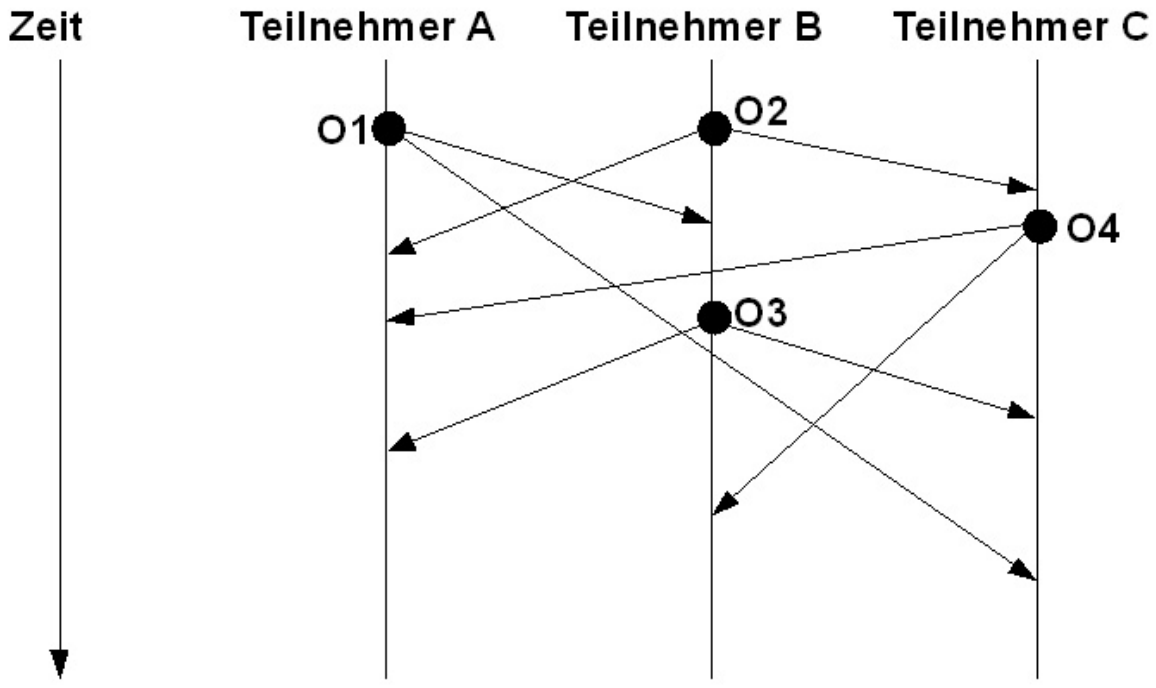

Abbildung 7: Operationstransfer [SJZ+98]

4.1.3 Intentionserhaltung

Da Operationen nebenläufig erzeugt werden, kann der Effekt einer Operation, der zur Erzeugungszeit gewünscht war, von dem, der zur Ausführungszeit tatsächlich bewirkt wird, abweichen [SJZ+98].

Wie in Abbildung 7 zu sehen ist, werden die Operationen O1 und O2 gleichzeitig erzeugt, sind also unabhängig voneinander. Es könnte also sein, dass sich O2 auf eine Position im gemeinsamen Dokument bezieht, die sich nach Ausführung von O1 verschoben hat.

Als Beispiel soll das gemeinsame Dokument den Inhalt “def” haben. Teilnehmer A möchte an 0. Stelle “abc” einfügen und Teilnehmer B möchte an 3. Position “ghi” einfügen. Ohne einen Intentionserhaltungsmechanismus ist der Inhalt des Dokuments nach Anwendung beider Operationen “abcghidef”. Gewünscht war aber “abcdefghi”.

4.2 Rückgängigmachen fremder Änderungen

Kaum jemand würde die Wichtigkeit einer “Undo”-Funktion, die Änderungen rückgängig macht, in Frage stellen [AD92]. Im Kontext eines kollaborativen Editors ist die Frage viel mehr, welche Änderungen rückgängig gemacht werden.

Von Systemen, bei denen ein Artefakt nur von höchstens einem Benutzer gleichzeitig verändert werden kann, ist man es gewohnt, dass die letzte Änderung zurückgenommen wird. Wird das Artefakt aber von mehreren Benutzern manipuliert, ist es möglich, dass die letzte Änderung aus

der Sicht desjenigen, der die Undo-Funktion ausführen möchte, nicht die eigene ist und er somit fremde Änderungen zurücknimmt.

Damit also nicht versehentlich fremde Arbeit zerstört wird, wurde an die Undo-Funktion von Saros die Anforderung gestellt, nur die letzte eigene Änderung zurückzunehmen.

Eclipse selbst hat aber von Haus aus nur einen linearen Undo-Mechanismus, d.h. eine Operation kann nur rückgängig gemacht werden, wenn es die letzte ausgeführte Operation war oder alle Operationen, die danach ausgeführt wurden, bereits rückgängig gemacht wurden.

Ein selektiver Undo-Mechanismus, der eine Operation rückgängig macht, die irgendwann ausgeführt wurde, ist von Eclipse nicht vorgesehen.

4.3 Synchronisierung komplexer Operationen

Bisher wurden in Beispielen nur einfache Operationen wie das Einfügen oder Entfernen von Zeichen betrachtet. Tatsächlich können Texteditoren auf der Basis dieser beiden Operationen aufgebaut werden [SJZ+98]. Saros bietet aber auch Funktionen, die nicht auf Basis dieser beiden Operationen aufgebaut sind und zudem potentiell mehr Zeit (als ein paar Milisekunden) in Anspruch nehmen wie z.B. die Konsistenzwiederherstellung oder Rollenwechsel.

Warum diese komplexen Operationen problematisch sein können, soll anhand des Beispiels Rollenwechsel verdeutlicht werden: Der Gastgeber G möchte einem Driver D das Schreibrecht entziehen. In einer naiven Implementierung schickt G daher D eine entsprechende Nachricht und setzt ihn in seiner Benutzerverwaltung auf Zuschauer. D empfängt die Nachricht und setzt sich in seiner eigenen Benutzerverwaltung auch auf Zuschauer, was D daran hindert, weitere Schreibaktivitäten zu unternehmen.

Kritisch wird es in der Zeit, in der die Nachricht unterwegs ist. D weiß noch nicht, dass er Zuschauer ist, kann also weiterhin Operationen erzeugen. G nimmt aber von D keine Änderungen mehr an, da G D bereits auf Zuschauer gesetzt hat und somit keine Änderungen mehr erwartet. In dieser kritischen Zeit können daher Inkonsistenzen entstehen.

5 Behandlung der Nebenläufigkeitsprobleme im Mehrschreibermodus

Im Folgenden werden Lösungen der im vorigen Kapitel erläuterteten Probleme präsentiert, wie sie auch in Saros implementiert wurden.

5.1 Echtzeit und Konsistenzerhaltung

5.1.1 Zentralisierte oder replizierte Architektur?

Zur Erreichung der Konvergenz gibt es zwei mögliche Ansätze: die zentralisierte oder die replizierte Architektur.

Bei ersterer gibt es einen zentralen Server, der das gemeinsame Artefakt verwaltet (Abb. 8). Operationen, welche die Klienten darauf ausführen möchten, werden an den Server geschickt, der sie auf das Dokument anwendet. Das bedeutet aber, dass derjenige, der die Operation erzeugt hat, darauf warten muss, bis die Änderung zum Server geschickt wurde und dieser eine Kopie des neuen Dokuments zurückgeschickt hat. Lokale Änderungen sind also nicht sofort sichtbar.

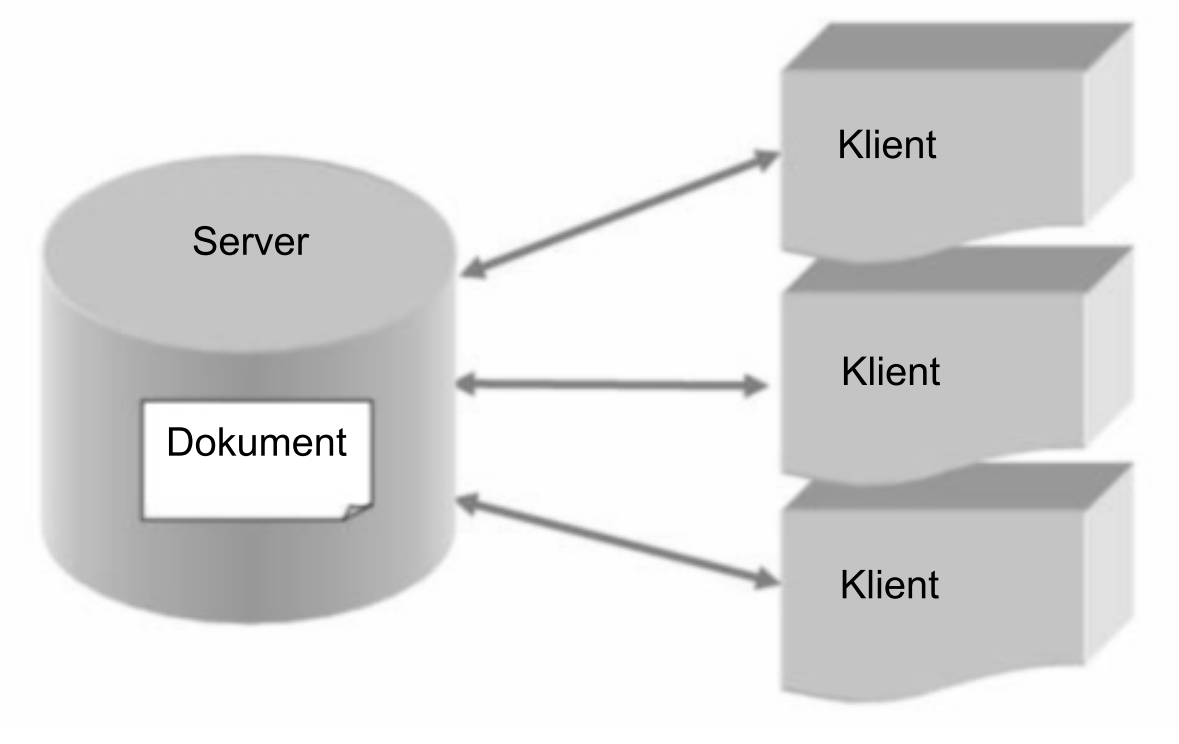

Abbildung 8: zentralisierte Architektur

Eine Anforderung an Saros ist aber eine schnelle Reaktion auf Textänderungen. Genauer, auf lokale Textänderungen soll sofort reagiert werden und auf entfernte Änderungen so schnell es eben netzwerklatenzbedingt möglich ist. Das heißt, gibt der lokale Benutzer Zeichen über seine Tas-

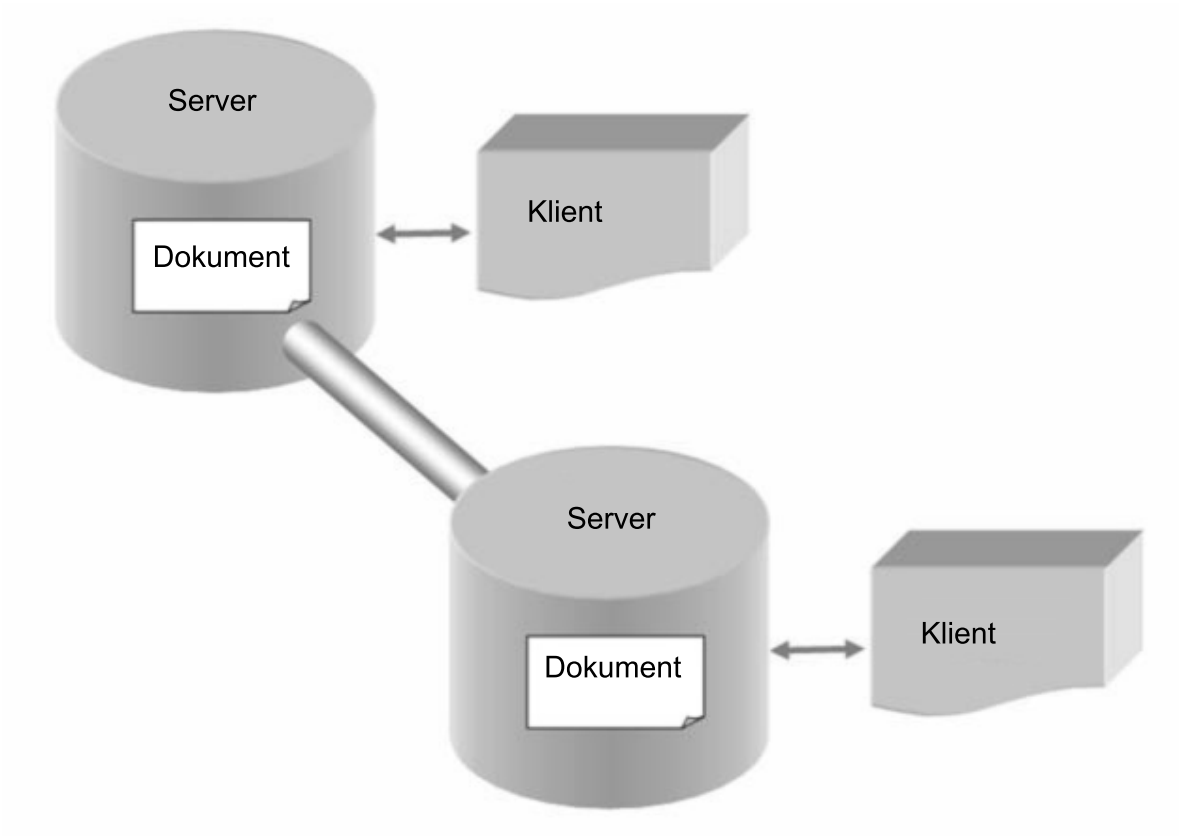

Abbildung 9: replizierte Architektur

tatur ein, sollen diese sofort auf seinem Bildschirm erscheinen. Daher scheidet der zentralisierte Ansatz für Saros aus.

Beim replizierten Ansatz (Abb. 9) hat jeder Klient seinen eigenen Server und somit sein eigenes Artefakt. Eine Operation wird an alle Teilnehmer kommuniziert, die daraufhin die Operation zur Konsistenzerhaltung in den Kontext eventueller anderer Operationen integriert, bevor sie sie auf ihrem Dokument anwenden können.

Hierbei muss also die eine Operation gegen N-1 mögliche andere Operationen transformiert werden, wobei N die Anzahl der Sitzungsteilnehmer ist. Die Umsetzung einer vollen N-Wege-Replikation ist aber sehr komplex [NCDL95], wodurch die Software schwer wartbar und sehr defektanfällig wird.

5.1.2 Jupiter-Architektur

Eine Mischung aus den beiden Ansätzen stellt die Jupiter-Architektur (Abb. 10) dar.

Jeder Klient verwaltet hierbei seine eigene Kopie des gemeinsamen Artefakts. Änderungen werden lokal sofort angewendet und dann an einen zentralen Server kommuniziert. Dieser serialisiert die ankommenden Operationen und kommuniziert sie an die anderen Klienten, welche die Operation gegebenenfalls transformieren und dann auf ihrer Kopie anwenden.

Dabei müssen sie also nur noch zwischen lokal und entfernt erzeugten Operationen unterscheiden. Von welchem Klienten eine Operation stammt, spielt somit keine Rolle.

Die entscheidenden Vorteile sind hierbei die lokale sofortige Ausführung der Operation und die 1:1-Transformation, deren vergleichsweise einfache Algorithmen die Wartbarkeit vereinfachen und die Defektanfälligkeit reduzieren.

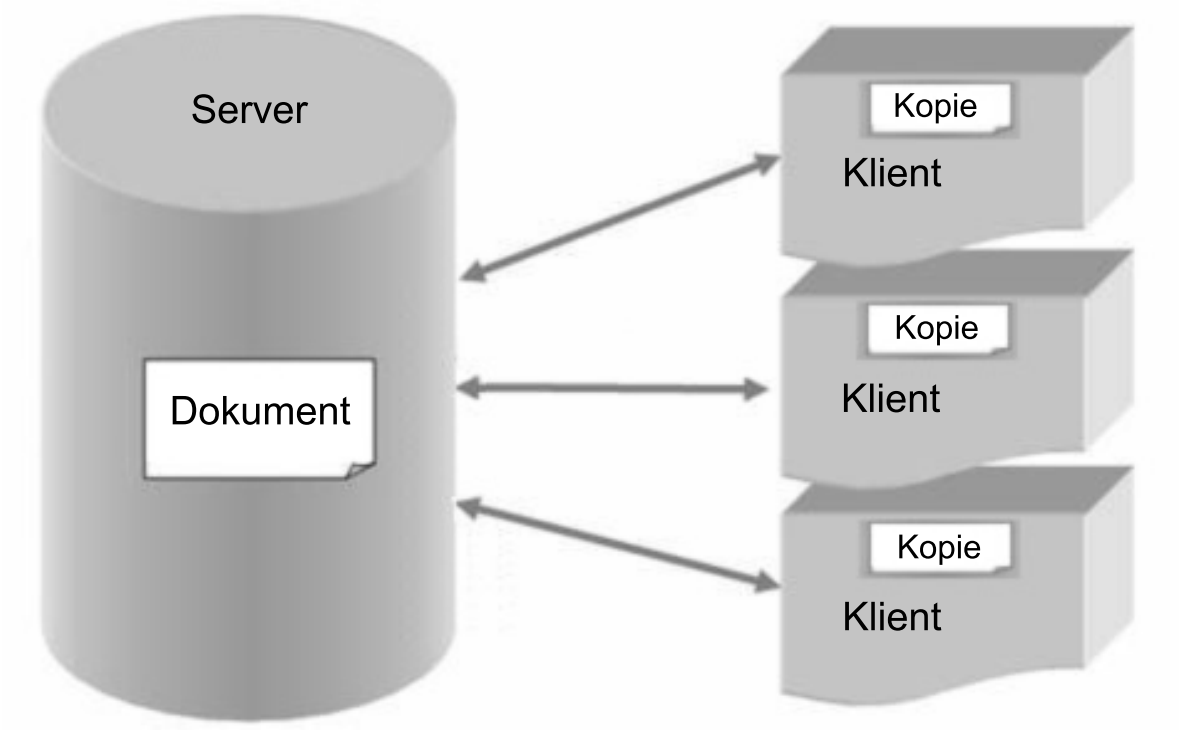

Abbildung 10: Jupiter-Architektur

5.1.3 Operationstransformation

Um die Erhaltung von Kausalität und Intention sicherzustellen, müssen beim Klienten ankommende, entfernt erzeugte Operationen eventuell transformiert werden.

Wie geht das vor sich?

Dazu kommen wir auf das Beispiel von oben zurück. Wir haben ein Dokument mit dem Inhalt “def” und zwei Operationen O1 und O2. O1 soll am Anfang des Dokuments, also an Position 0, “abc” einfügen, O2 soll am Ende des Dokuments, an Position 3, “ghi” einfügen.

Dazu führen wir folgende Notation ein:

Dokument D = “def”
O1 = Einfüge(0, “abc”)
O2 = Einfüge(3, “ghi”)

Wie bereits gesehen, führt ein simples Nacheinanderausführen der Operationen zu einem unerwünschten Ergebnis. Wir benötigen also eine Funktion t, die O2 so transformiert, dass das transformierte O2' nach O1 ausgeführt werden kann und dadurch das gewünschte Ergebnis entsteht.

Die Transformationsfunktion t soll folgendermaßen notiert werden:

t(O1, O2) = O1'
t(O2, O1) = O2'

Nach der Transformation kann man die Operationen entweder in der Reihenfolge O1, O2' ausführen oder O2, O1'.

Eine Lösung hierfür bietet der *GOTO-Algorithmus* [SE98], der auf dem dOPT-Algorithmus [EG89] basiert.

Angewandt auf die beiden Operationen Einfügen und Entfernen, ergeben sich folgende Transformationsvorschriften:

```
t(Einf(posA, wortA), Einf(posB, wortB)) {
  if (posA < posB) return Einf(posA, wortA);
  else return Einf(posA + laenge(wortB), wortA);
}

t(Einf(posA, wortA), Entf(posB, wortB)) {
  if (posA <= posB) return Einf(posA, wortA);
  else if (posA > posB + laenge(wortB)) return Einf(posA - laenge(wortB),
      wortA);
  else return Einf(posB, wortA)
}

t(Entf(posA, wortA), Einf(posB, wortB)) {
  if (posB >= posA + laenge(wortA)) return Entf(posA, wortA);
  else if (posA >= posB) return Entf(posA + laenge(wortB), wortA);
  else return Split(Entf(posA, wortA[0:(posB-posA)]), Entf(posB + laenge(B),
      wortA[(posB-posA+1):laenge(wortA)]));
}

t(Entf(posA, wortA), Entf(posB, wortB)) {
  if (posB >= posA + laenge(wortA)) return Entf(posA, wortA);
  else if posA >= posB + laenge(wortB) return Entf(posA - laenge(wortB),
      wortA);
  else {
    if posB <= posA && (posA + laenge(wortA) <= posB + laenge(wortB)) return
        null;
    else if posB <= posA && (posA + laenge(wortA) > posB + laenge(wortB))
        return Entf(posB, wortA[(posB+laenge(wortB)-posA):laenge(wortA)];
    else if posB > posA && (posB + laenge(wortB) >= posA + laenge(wortA))
        return Entf(posA, wortA[0:(posB-posA)]);
    else return Split(Entf(posA, wortA[0:(posB-posA)]), Entf(posB-posA, wortA
        [(posB+laenge(wortB)-posA):laenge(wortA));
  }
}
```

Wort[a:b] ist dabei die Folge der Zeichen aus wort ab Position a bis Position b.

In der dritten und vierten Transformationsfunktion ist die Operation Split zu beachten, die zwei andere Operationen beinhaltet. In diesen Fällen entstehen durch die Transformation also zwei Operationen, die nacheinander auszuführen sind.

Nun bleibt noch die Frage zu klären, wann welche Operation gegen welche transformiert wird?

5.1.4 zweidimensionaler Zustandsraum

Um eine sinnvolle Reihenfolge der Operationen zu ermöglichen, wird ein zweidimensionaler Zustandsraum (Abb. 11) eingeführt, den jeder Teilnehmer für sich verwaltet. Dieser hat die Dimensionen Klient und Server. Befindet sich der Teilnehmer also im Zustand (2,3) so hat er zwei lokale Operationen erzeugt und ausgeführt und drei Operationen vom Server empfangen und ausgeführt.

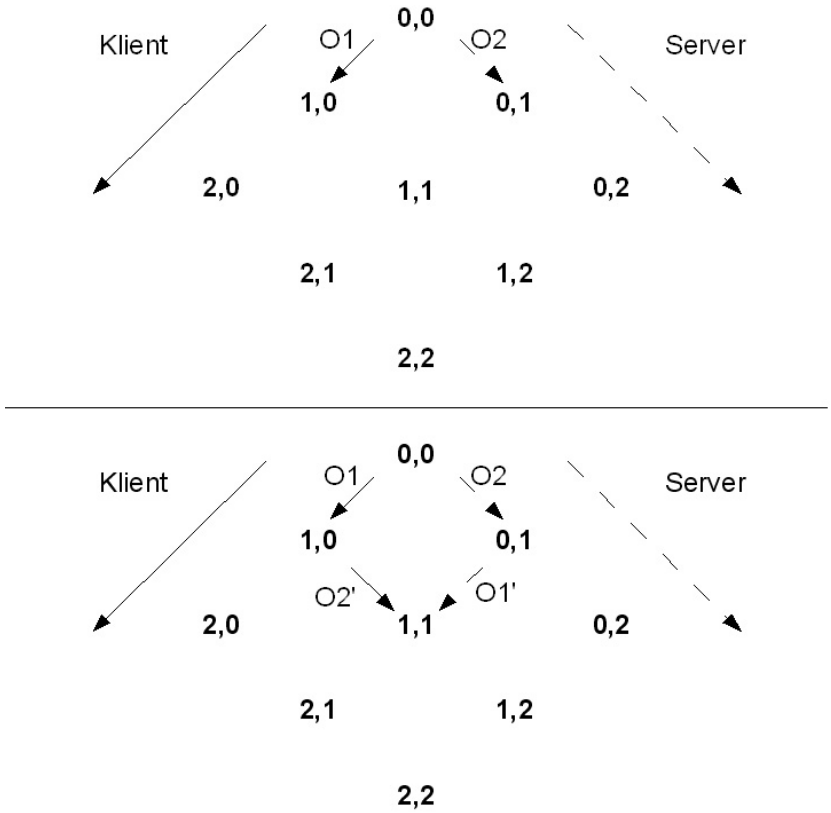

Abbildung 11: zweidimensionaler Zustandsraum

Mit jeder ausgeführten Operation wandert der Klient bzw. der Server durch den Zustandsraum. Führen sie Operationen in der gleichen Reihenfolge aus, nehmen sie den gleichen Pfad. Ansonsten divergieren sie.

Um diese Divergenz zu beheben und einen konvergenten Zustand herbeizuführen, muss ein Pfad vom Klient zum Server (oder umgekehrt) gefunden werden. Im Beispiel (Abb. 11) gibt es eine

Operation O1 vom Klienten und eine Operation O2 vom Server. Beide führen zuerst die jeweils eigene aus und somit entsteht eine Divergenz. Um wieder in einen gemeinsamen Zustand zu kommen, muss nun der Klient O2' ausführen, also die gegen O1 transformierte Operation O2. Der Server hingegen muss O1' ausführen, also die gegen O2 transformierte Operation O2. Dann befinden sich beide im selben Zustand und die Konvergenz ist wiederhergestellt.

5.1.5 Implementierung

Die Nebenläufigkeitskontrolle auf Jupiter-Basis war zur Zeit meines Eintritts in das Projekt bereits implementiert und wird von Rieger [Rie08] beschrieben.

Es waren auch JUnit-Tests vorhanden, von denen manche allerdings fehlschlugen. Zurückzuführen waren diese Versagen auf eine falsche Behandlung von Split-Operationen, die von Christopher Oezbek und mir korrigiert wurden.

5.2 Nebenläufiges Undo

Es ist zwar hilfreich, wenn ein kollaborativer Editor im Mehrbenutzerbetrieb die selbe Funktionalität mit der selben Semantik bietet, als arbeite man alleine damit [EGR91].

Doch im Falle der "Rückgängig"-Funktion (Undo) ist das nur eingeschränkt erwünscht, da man hier unterscheiden muss zwischen letzter Änderung und letzter eigener Änderung.

5.2.1 Prinzip

Um zu vermeiden, dass fremde Änderungen rückgängig gemacht werden, stellen sich zwei wichtige Fragen:

- Was ist die letzte eigene Änderung?
- Wie mache ich diese rückgängig?

Die Beantwortung der ersten Frage ist leicht. Man merkt sich einfach jede ausgeführte Operation in einer geeigneten Datenstruktur mit einem Vermerk, ob sie lokal oder entfernt erzeugt wurde. Die letzte eigene Änderung ist somit die letzte Operation, die in die Datenstruktur eingetragen wurde und den Vermerk lokal hat.

Die zweite Frage ist schon etwas schwieriger zu beantworten. Die beiden Basis-Editieroperationen Einfügen und Löschen sind Gegenoperationen zueinander, d.h. führt man sie direkt hintereinander mit den selben Argumenten aus, hebt die zweite Operation die Wirkung der ersten auf.

Als Beispiel: Führt man nach Einfüge(5, "abc") Lösche(5, "abc") aus, ist die Wirkung die, als hätte man beide Operationen nie ausgeführt. Die erste Operation wurde damit rückgängig gemacht.

Wurden also nach der letzten lokalen Operation keine weiteren Operationen ausgeführt, kann man sie zurücknehmen, indem man ihre Gegenoperation ausführt.

Falls aber danach weitere Operationen ausgeführt wurden, funktioniert das nicht in jedem Fall. Als Beispiel wurde erst die lokal erzeugte Operation Einfüge(5, “abc”) ausgeführt und danach die entfernt erzeugte Operation Einfüge(4, “xyz”). Will man nun die letzte eigene Änderung zurücknehmen, reicht es nicht einfach Lösche(5, “abc”) auszuführen, weil “abc” nicht mehr an Position 5, sondern an Position 8 steht.

Stattdessen muss man die Gegenoperation zur letzten lokalen Operation gegen alle nachfolgenden Operationen transformieren [Sun02], um sie dann auszuführen. Auf das Beispiel angewandt, erhielte man die passendere Operation Lösche(8, “abc”).

5.2.2 Integration in Eclipse

Um zu erklären, wie oben erklärter Mechanismus in Eclipse integriert wurde, ist es hilfreich zu wissen, wie Undo in Eclipse funktioniert [Fou].

Operationen, die zurückgenommen werden können, indem sie die Schnittstelle *IUndoableOperation* implementieren, werden nach erfolgreicher Ausführung in einer Datenstruktur namens *IOperationHistory* eingetragen.

Jede dort eingetragene Operation hat ein Etikett (Label), das die Kategorie der Operation enthält, z.B. *Typing*, und einen Kontext, den sog. *IUndoContext*, der z.B. den Editor angibt, in dem die Operation ausgeführt wurde.

Sobald der Benutzer Undo auslöst, z.B. durch Eingabe von strg+z, konsultiert die *IOperationHistory* alle registrierten *IOperationApprover*, ob die letzte Operation in dem aktuellen Kontext tatsächlich zurückgenommen werden soll. Falls hier kein Veto eingelegt wird, wird auf der Operation die Methode *undo()* aufgerufen und ist damit rückgängig gemacht.

Dies ist allerdings nur ausgelegt auf einen lineare Undo-Strategie, keine selektive, d.h. man kann immer nur die letzte Operation eines bestimmten Kontexts zurücknehmen, nicht eine beliebige. Für eine Undo-Semantik, die sich nur auf die letzte eigene Änderung bezieht, ist die Eclipse-eigene Implementierung also ungeeignet.

Stattdessen wurde eine Datenstruktur entwickelt, die ähnlich wie *IOperationHistory* den Operationsverlauf aufzeichnen soll. Einträge in diesem Verlauf erhalten die Operation selbst und ihren Typ (*local* / *remote* für lokal / entfernt erzeugte Operationen und *redoable* für Operationen, die rückgängig gemacht wurden).

```
1  /**
2   * @return operation that reverts the effect
3   *         of the latest local operation in the given editor
4   */
5  protected Operation calcUndoOperation(IPath editor) {
6
7      if (!undoHistory.canUndo(editor))
8          return new NoOperation(); // nothing to undo
9
10     Operation lastLocal = undoHistory.getLatestLocal(editor);
11
12     Operation undoOperation = lastLocal.invert();
13
```

```
14      for (EditorHistoryEntry entry : undoHistory.entriesToLatestLocal(editor))
15      {
16         undoOperation = transformation.transform(undoOperation, entry.
               getOperation(), Boolean.TRUE);
17      }
18
19      undoHistory.replaceType(editor, lastLocal, Type.LOCAL, Type.REMOTE);
20      // it is not relevant any more, so it is set remote
21
22      undoHistory.add(editor, Type.REDOABLE, undoOperation); // save for redo
23
24      return undoOperation;
25  }
```

Listing 1: Berechnung der Operation, welche die letzte eigene rückgängig macht

Verwendet wird diese Datenstruktur vom UndoManager, der das Verhalten des *nebenläufigen Undo* steuert. Er bietet eine Methode zur Berechnung der Operation, welche die letzte eigene Änderung rückgängig macht.

Dazu nimmt er die letzte lokale Operation, bildet die Gegenoperation und transformiert sie gegen alle nachfolgenden. Danach ersetzt er den Typ der ursprünglichen Operation durch *remote*, damit sie nicht mehr als letzte lokale Operation erkannt wird. Die neue Operation wird als *redoable* in den Operationsverlauf eingetragen.

```
1  /**
2   * IOperationApprover defines an interface for approving the undo
3   * or redo of a particular operation within an operation history.
4   */
5  public interface IOperationApprover {
6
7    /**
8     * Return a status indicating whether the specified operation
9     * should be redone. Any status that does not have severity
10    * IStatus.OK will not be approved. Implementers should not
11    * assume that the redo will be performed when the status is OK,
12    * since other operation approvers may veto the redo.
13    */
14   public IStatus proceedRedoing(IUndoableOperation operation,
         IOperationHistory history, IAdaptable info);
15
16   /**
17    * Return a status indicating whether the specified operation
18    * should be undone. Any status that does not have severity
19    * IStatus.OK will not be approved. Implementers should not
20    * assume that the undo will be performed when the status is OK,
21    * since other operation approvers can veto the undo.
22    */
23   public IStatus proceedUndoing(IUndoableOperation operation,
         IOperationHistory history, IAdaptable info);
```

Listing 2: IOperationApprover [Fou]

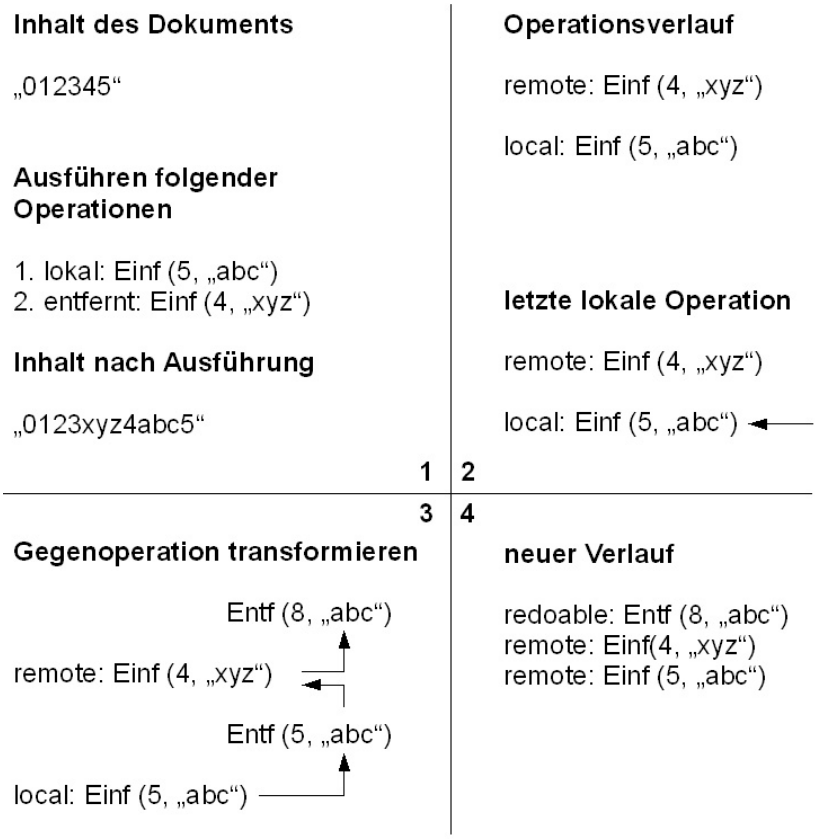

Abbildung 12: Undo-Mechanismus

Ist der UndoManager aktiviert (kann in den Eclipse-Einstellungen festgelegt werden, s. Abb. 4), wird nach Auslösung von Undo, der *IOperationApprover* des UndoManager konsultiert. Trägt die Operation das Etikett *Typing* (Texteingabe), wird ein Veto eingelegt und stattdessen im UndoManager durch oben beschriebene Methode die passende Operation berechnet. Anschließend wird sie direkt (durch *IDocument.replace()* [Fou]) auf das Dokument im aktuellen Editor angewendet.

5.2.3 Wiederherstellung der rückgängig gemachten Operation - Redo

Manchmal merkt man erst, nachdem man eine Operation rückgängig gemacht hat, dass sie eigentlich doch notwendig war. Man möchte also die Operation wiederherstellen. Diese Funktion nennt sich “Redo”.

Die Umsetzung ähnelt der Implementierung von Undo. Die letzte Operation, die lokal erzeugt wurde, um eine andere rückgängig zu machen, wird invertiert und gegen alle nachfolgenden Operationen transformiert.

Anschließend wird die ursprüngliche rückgängig machende Operation auf den Typ *remote* gesetzt und die berechnete Operation als lokal in den Operationsverlauf eingetragen.

```
/**
 * @return operation that reverts the effect of the latest undo in the given
 *         editor
 */
protected Operation calcRedoOperation(IPath editor) {

    if (!undoHistory.canRedo(editor))
        return new NoOperation(); // nothing to redo

    Operation lastUndo = undoHistory.getLatestRedoable(editor);

    Operation redoOperation = lastUndo.invert();

    for (EditorHistoryEntry entry : undoHistory.entriesToLatestRedoable(
        editor)) {

        redoOperation = transformation.transform(redoOperation, entry.
            getOperation(), Boolean.TRUE);
    }
    undoHistory.replaceType(editor, lastUndo, Type.REDOABLE, Type.REMOTE);
    // it is not relevant any more, so it is set remote

    undoHistory.add(editor, Type.LOCAL, redoOperation);

    return redoOperation;
}
```

Listing 3: Berechnung der Operation, welche die letzte lokal zurückgenommene wiederherstellt

5.2.4 Schwierigkeiten bei der Implementierung

Während der Implementierung sind einige Schwierigkeiten aufgetreten, mit denen unterschiedlich gut umgegangen werden konnte.

Aktivierung von Redo

Dadurch dass der von Eclipse vorgesehene Undo-Vorgang durch den *IOperationApprover* abgebrochen wird, wird niemals die Redo-Funktionalität aktiviert, d.h. die Schaltfläche ist ausgegraut (im Menü “Edit”) oder wird nicht angezeigt (im Kontextmenü). Auch die Eingabe des entsprechenden Tastaturkürzels hätte keinen Effekt.

Da Eclipse keine Möglichkeit bietet eine Aktivierung von Redo zu erzwingen, wird also ein Undo simuliert. Dazu wird eine NullOperation implementiert, die bei Ausführung, bei Zurücknahme und bei Wiederherstellung nur den Status OK zurückgibt und sonst nichts macht.

```
/**
 * Adds an undone NullOperation (i.e. has no effects on executing, undoing
 * or redoing) to the Eclipse IOperationHistory to be sure that redoing is
 * possible. This is necessary for the UI Redo possibility being activated.
 */
protected void simulateUndo() {
    IUndoableOperation auxOp = new NullOperation();
    try {
        auxOp.addContext(context);
        eclipseHistory.execute(auxOp, null, null);
        eclipseHistory.undo(context, null, null);
        if (!eclipseHistory.canRedo(context))
            log.error("Simulating Undo failed.");
    } catch (ExecutionException e) {
        log.error("Simulating Undo failed because of " + e);
    }
}
```

Listing 4: Simulation von Undo, um Redo zu aktivieren

Zur Simulation von Undo wird diese Operation ausgeführt und dann rückgängig gemacht. Der einzige Effekt dieser Aktion ist, dass Redo jetzt aktiviert ist. Somit ist garantiert, dass nach einem Undo ein Redo ausgeführt werden kann.

Vermeidung zeichenweiser Zurücknahme

Was bisher noch nicht geklärt wurde, ist diese Frage: Wenn Zeichen einzeln eingegeben werden, also z.B. 'W', 'o', 'r', 't', was wird dann rückgängig gemacht? Die Eingabe des letzten Zeichens, also 't'?

Das wäre nicht sehr praktikabel, denn in den meisten Fällen benutzt man Undo ja nicht, um Tippfehler zu korrigieren, sondern man möchte gleich ganze Zeilen oder Passagen entfernen oder wiederherstellen. Daher fasst Eclipse eine Operation mit den nachfolgenden zusammen, bis bestimmten Regeln folgend, die nächste Operation begonnen werden kann.

Diesen Mechanismus sollte die nebenläufige Undofunktionalität nachbilden. Dabei besteht der Unterschied zum herkömmlichen Mechanismus darin, dass unbedingt zwischen lokal und entfernt erzeugten Operationen unterschieden werden muss, um zu vermeiden, dass diese beiden zu einer Operation zusammengefasst werden.

Entfernt erzeugte Operationen sind hier unkompliziert. Sie müssen nicht zusammengefasst werden, da sie ja nicht rückgängig gemacht werden sollen. Bei lokalen dagegen unterscheide ich die aktuelle atomare Operation, die oft nur ein Zeichen betrifft, und die aktuelle zusammengefasste Operation.

Wenn also eine lokale Operation A erzeugt wurde, ist das die aktuelle atomare. Wird danach eine lokale Operation B ausgeführt, wird B zur aktuellen atomaren, während A die zusammengefasste Operation darstellt. Kommt eine lokale Operation C hinzu, ist C die aktuelle atomare und B wird mit A zu BA zusammengefasst. A, B und C wurden dabei noch nicht in den Operationsverlauf eingetragen.

Trifft jetzt eine entfernte Operation Z ein, wird sie sofort in den Operationsverlauf eingetragen. Damit BA und C die Konsistenzbedingungen einhalten, müssen sie jetzt allerdings gegen Z transformiert werden.

Wenn wir von Eclipse über einen *IOperationHistoryListener* benachrichtigt werden, dass eine neue Operation angefangen wurde, wird die zusammengefasste Operation in den Verlauf eingetragen. Auf die aktuelle atomare bezieht sich das nicht, weil diese ja die neu angefangene Operation ist, über die Eclipse uns benachrichtigt hat.

Auf diese Weise werden auch die uns unbekannten Regeln zur Bildung der zusammengefassten Operationen nachgebildet, mit dem einzigen Unterschied, dass bei uns nur lokale Operation zusammengefasst werden.

Aktiver Editor

Ein Projekt in Eclipse besteht üblicherweise aus mehreren Dateien. Jede dieser Dateien kann in einem eigenen Editor geöffnet werden, zwischen denen während einer Sitzung gewechselt werden kann. Da man nicht in einem Editor eine Operation rückgängig machen möchte, die man in einem anderen ausgeführt hatte, ist es also nötig sich zu jeder Operation zu merken, in welchem Editor sie ausgeführt wurde bzw. welche Datei betroffen ist. Dazu wird für jede Datei ein eigener Operationsverlauf angelegt.

Wenn nun ein Undo ausgelöst wird, kommt als erstes die Frage auf, mit welchem Operationsverlauf gearbeitet werden soll oder anders formuliert: Welches ist der aktuell aktive Editor?

Dazu wird ein *ISharedEditorListener* implementiert, der uns benachrichtigt, wenn der aktive Editor wechselt und wenn er geschlossen wird. Wenn er wechselt, wird die aktuelle atomare Operation mit der zusammengefassten im den Operationsverlauf gespeichert und die Variable, die auf den aktiven Editor verweist, aktualisiert. Wird er geschlossen, wird zusätzlich noch der Verlauf des Editors gelöscht, um das natürliche Verhalten von Eclipse nachzubilden.

```
1  /**
2   * Updates the local current active editor. If the editor changes the
3   * current local operation is stored in history.
4  */
5  protected ISharedEditorListener sharedEditorListener = new
       AbstractSharedEditorListener() {
6
7      @Override
8      public void activeEditorChanged(User user, IPath newActiveEditor) {
9
10         if (!user.isLocal() || currentActiveEditor == newActiveEditor)
11             return;
12
13                 //combine all current operations and move them to the history
14         updateCurrentLocalAtomicOperation(null);
15         storeCurrentLocalOperation();
16
17         currentActiveEditor = newActiveEditor;
18     }
19
```

```
20      @Override
21      public void editorRemoved(User user, IPath closedEditor) {
22          if (currentActiveEditor == closedEditor) {
23              updateCurrentLocalAtomicOperation(null);
24              storeCurrentLocalOperation();
25              currentActiveEditor = null;
26          }
27                  undoHistory.clearEditorHistory(closedEditor);
28      }
29  };
```

Listing 5: So bleibt man über den aktuell aktiven Editor auf dem Laufendem

5.2.5 Grenzen der Implementierung

Dadurch, dass Eclipse kein selektives Undo vorsieht, musste bei der Implementierung oft am System vorbeigearbeitet werden, wie im Fall der Redo-Aktivierung (Abschnitt 5.2.4).

Die aktuelle Implementierung stellt also nicht die sauberste Lösung dar und es bestehen zudem noch weitere Probleme, die gelöst werden müssen.

Weniger problematisch als möglicherweise etwas verwirrend für den Benutzer ist der Umstand, dass nach der ersten Texteingabe Undo für den betroffenen Editor immer aktiv ist. Sobald alle Operationen rückgängig gemacht wurden, erwartet man aber eigentlich, dass Undo wieder deaktiviert wird, d.h. die entsprechenden Schaltflächen sollten ausgegraut sein.

Dadurch, dass das ursprünglich vorgesehene Undo-Verfahren aber durch ein Veto des *IOperationApprover* abgebrochen wird, passiert das nie. Undo bleibt also immer aktiv. Auch die oben beschriebene Simulation von Undo bleibt wirkungslos, da die Null-Operation ja erst ausgeführt werden muss, bevor sie zurückgenommen wird.

Von der Schaltflächenaktivierung einmal abgesehen, bleibt das Verhalten aber erwartet. Versucht man nämlich mehr Operationen zurückzunehmen als lokal erzeugt wurden, wird darauf nicht reagiert.

Von der Wartbarkeit her gibt es ein anderes großes Problem. Es wurde hier versucht das Regelwerk von Eclipse im Umgang mit Undo/Redo nachzubilden. Diese Regeln, z.B. wann Redo möglich ist, sind aber in der übrigen Implementierung integriert. Sollten sich die Undo/Redo-Politik von Eclipse aber in der Zukunft ändern oder für Saros eine andere sinnvoller erscheinen, ist es sehr mühsam diese Änderungen im Quelltext durchzuführen.

Langfristig ist es daher angebracht die Regeln von der Implementierung zu trennen, damit Änderungen schnell und unkompliziert umgesetzt werden können.

5.3 StopManager zur Synchronisierung komplexer Operationen

5.3.1 Ablauf

In den bisherigen Beispielen wurden nur die beiden Basis-Editieroperationen Einfügen und Löschen betrachtet. Wie in Abschnitt 4.3 gezeigt, können komplexe Operationen Inkonsistenzen auslösen, wenn während ihrer Ausführung weitere Operationen erzeugt werden.

Um diese Gefahr zu bannen, wird ein Mechanismus gesucht, mit dem ein Zeitfenster gebildet werden kann, in dem ein oder mehrere gegebene Nutzer keine Operationen erzeugen können. Dieser Mechanismus wird nachfolgend StopManager genannt. Dazu soll es eine Methode “stop” geben, mit der man einen oder mehrere Teilnehmer anhalten kann und eine Methode “start”, mit der man die Nutzer wieder weiterarbeiten lässt.

Um den gewünschten Teilnehmer zum Anhalten zu bringen, wird ihm eine Nachricht geschickt, die ihm den Initiator und den Grund nennt. Daraufhin hält er an und bestätigt den Stop.

Wenn die Bestätigung beim Initiator ankommt, kann er sicher sein, dass bis zur Aufforderung zum Weitermachen keine weiteren Operationen von dem betroffenen Teilnehmer erzeugt werden, da Aktivitäten in der Reihenfolge der Erzeugung verarbeitet werden (Sequenzialisierung durch den *ActivitySequencer* [Rin09]) und nach dem Blockieren keine neuen Aktivitäten erzeugt werden. Somit kann er also jetzt seine komplexe Operation ausführen.

Anschließend verwendet er zum Starten des gestoppten Teilnehmers ein StartHandle, das er für diesen Vorgang und diesen Teilnehmer erhalten hat. Der angehaltene Teilnehmer erhält dadurch die Benachrichtigung, dass er wieder weiterarbeiten kann, falls er nicht noch aus anderen Gründen blockiert ist.

5.3.2 Implementierung

Der Kern der Implementierung ist das Verhindern der Erzeugung von Text- und Dateioperationen bei dem betroffenen Nutzer. Dazu implementiert jedes Modul, dass für die Erzeugung solcher Operationen zuständig ist, das Interface Blockable, das die Methoden block und unblock enthält.

```
1  /**
2   * Implementers of this interface can be blocked by the StopManager. Being
3   * blocked means that they don't generate any activities and don't generate
4   * local changes that can be realized by the user, e.g. text changes.
5   */
6  public interface Blockable {
7
8      public void block();
9
10     public void unblock();
11 }
```

Listing 6: Interface Blockable

Durch Aufruf von block sorgt das Modul dafür, dass es keine Operationen mehr erzeugt, bis unblock aufgerufen wird. Sind die Blockables der verschiedenen Module im StopManager regis-

triert, kann die Blockierung durchgeführt werden, ohne dass der StopManager etwas über das Innenleben der betroffenen Module wissen muss.

```
/**
 * The goal of this method is to ensure that the local user cannot cause any
 * editing activities (FileActivities and TextEditActivities).
 *
 * @param lock
 *            if true the project gets locked, else it gets unlocked
 */
protected void lockProject(boolean lock) {
    for (Blockable blockable : blockables) {
        if (lock)
            blockable.block();
        else
            blockable.unblock();
    }
}
```

Listing 7: Sperren des Projekts

Initiierung des Stop-Vorgangs

Doch der Reihe nach. Beginnen wir mit der Initiierung des Stop-Vorgangs. Dazu wird im StopManager die Methode stop unter Angabe des betroffenen Teilnehmers und des Grundes aufgerufen, welche zuerst ein StartHandle zur späteren Rückgabe generiert.

Dann wird eine Nachricht (*StopActivity*) formuliert und an den betroffenen Teilnehmer verschickt. Danach wird auf Antwort gewartet.

Empfängt der betroffene Teilnehmer die Anfrage, erzeugt er aus der Nachricht ein StartHandle, sperrt lokal das Projekt und sendet die erwartete Bestätigung.

Das Anlegen des StartHandles wird an dieser Stelle gemacht, damit im Falle mehrerer gleichzeitiger Stop-Vorgänge eine Verwaltung derselben möglich ist. Somit wird sichergestellt, dass erst gestartet wird, wenn alle Stop-Vorgänge, die sich auf den betroffenen Teilnehmer beziehen, abgeschlossen sind.

Erst wenn die Antwort beim Initiator eingetroffen ist, gibt die Methode “stop” das StartHandle zurück. Daher kann man nach Ausführung von “stop” sicher sein, dass der betroffene Teilnehmer keine Nachrichten mehr erzeugt.

Aufheben der Blockierung

Darf der blockierte Teilnehmer weiterarbeiten, ruft der Initiator auf dem für diesen Zweck zur Verfügung gestellten StartHandle die Methode “start” auf. Dadurch wird das StartHandle entfernt und dem betroffenen Teilnehmer eine entsprechende Nachricht geschickt (im Codeausschnitt mittelbar durch “initiateUnlock”).

```java
/**
 * Notifies the StopManager, that the operation for which this IStartHandle
 * was returned by a call to stop has finished.
 *
 * This method will return immediately and return true, if the stopped user
 * continued working or false if other StartHandles exist for this user,
 * which have not been called.
 *
 * @nonblocking
 *
 * @Throws IllegalStateException if start() is called twice on the same
 *         handle.
*/
public boolean start() {

    if (!startCalled.compareAndSet(false, true))
        throw new IllegalStateException("start can only be called once per
            StartHandle");

    stopManager.removeStartHandle(this);

    if (stopManager.getStartHandles(user).size() == 0) {
        stopManager.initiateUnlock(this);
        return true;
    }
    return false;
}
```

Listing 8: Entsperren eines Teilnehmers

Dieser hebt die lokale Blockierung des Projekts auf und schickt eine Bestätigung. Die Methode “start” wartet nicht auf die Bestätigung, sondern wird schon früher verlassen. Es gibt allerdings Fälle (wie in Abschnitt 5.3.4, in denen es wichtig ist, dass blockiert wird, bis die Bestätigung eingetroffen ist. Dafür gibt es die Methode “startAndAwait”, die erst verlassen wird, wenn das Entsperren vom Teilnehmer bestätigt wurde. Falls der Teilnehmer noch durch weitere Stop-Vorgänge blockiert ist, wird natürlich nicht auf die Bestätigung gewartet.

```java
/**
 * Notifies the StopManager, that the operation for which this IStartHandle
 * was returned by a call to stop has finished.
 *
 * This method returns true, if the stopped user continued working or false
 * if other IStartHandles exist for this user, which have not been called.
 *
 * @blocking waits until the blocked user acknowledged the start
 *
 * @Throws IllegalStateException if start() is called twice on the same
 *         handle.
 * @Throws CancellationException
 */
public boolean startAndAwait(final SubMonitor progress) {

```

```
16      if (!startCalled.compareAndSet(false, true))
17          throw new IllegalStateException("start can only be called once per
                StartHandle");
18
19      stopManager.removeStartHandle(this);
20
21      if (stopManager.getStartHandles(user).size() == 0) {
22
23          stopManager.initiateUnlock(this);
24
25          try {
26              while (!acknowledged.get() && !progress.isCanceled())
27                  Thread.sleep(stopManager.MILLISTOWAIT);
28           } catch (InterruptedException e) {
29              Thread.currentThread().interrupt();
30              log.error("Code not designed to be interruptable", e);
31          }
32          if (progress.isCanceled())
33              throw new CancellationException();
34
35          return acknowledged.get();
36      }
37      return false;
38  }
```

Listing 9: Entsperren eines Teilnehmers mit Warten auf Bestätigung

Blockieren mehrerer Teilnehmer

Möchte man mehrere Teilnehmer gleichzeitig blockieren, besteht dazu die Möglichkeit oben beschriebene Methode “stop” nacheinander für jeden Teilnehmer auszuführen. Das bedeutet aber, dass für jeden Teilnehmer erst auf die Bestätigung gewartet wird, bis der nächste blockiert wird. Da das sehr lange dauern kann, wird jeder Teilnehmer in einem eigenen Thread angehalten.

Die entsprechende Methode soll wie die oben beschriebene Methode “stop”, die nur für einen Teilnehmer verwendet wird, für den Benutzer abbrechbar sein, da es sich um eine Operation handelt, die möglicherweise mehrere Sekunden dauern kann. Macht der Nutzer von dieser Möglichkeit Gebrauch, kann aber ein Zustand entstehen, in dem einige Teilnehmer bereits blockiert sind und andere noch nicht. Um die Benutzung der Methode zu vereinfachen, werden im Falle eines Abbruchs alle bereits blockierten Teilnehmer wieder gestartet. Also ist im Falle eines Abbruchs keiner der Teilnehmer blockiert, während nach regulärem Ablauf alle blockiert sind.

```java
/**
 * Blocking method that asks the given users to halt all user-input and
 * returns a list of handles to be used when the users can start again.
 *
 * @param users
 *            the participants who has to stop
 * @param cause
 *            the cause for stopping as it is displayed in the progress
 *            monitor
 *
 * @param monitor
 *            The caller is expected to call beginTask and done on the given
 *            SubMonitor
 *
 * @noSWT This method mustn't be called from the SWT thread.
 *
 * @blocking returning after the given users acknowledged the stop
 *
 * @cancelable This method can be canceled by the user
 *
 * @throws CancellationException
 */
public List<StartHandle> stop(final Collection<User> users, final String
    cause, final SubMonitor monitor) throws CancellationException {

    final List<StartHandle> resultingHandles = Collections.synchronizedList(
        new LinkedList<StartHandle>());

    final CountDownLatch doneSignal = new CountDownLatch(users.size());

    for (final User user : users) {
        Util.runSafeAsync(log, new Runnable() {
            public void run() {
                try {
                    StartHandle startHandle = stop(user, cause, SubMonitor
                        .convert(new NullProgressMonitor()));
                    resultingHandles.add(startHandle);
                    doneSignal.countDown();
                } catch (CancellationException e) {
                    monitor.setCanceled(true);
                } catch (InterruptedException e) {
                    monitor.setCanceled(true);
                }
            }
        });
    }
    while (resultingHandles.size() != users.size() && !monitor.isCanceled())
        {
        try {
            // waiting for all startHandles
            doneSignal.await(MILLISTOWAIT, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
```

```
50                log.error("Stopping was interrupted. Not all users could
                    successfully be stopped.");
51            }
52        }
53        if (monitor.isCanceled()) {
54            // Restart the already stopped users
55            for (StartHandle startHandle : resultingHandles)
56                startHandle.start();
57            throw new CancellationException();
58        }
59        return resultingHandles;
60    }
```

Listing 10: Blockieren mehrerer Teilnehmer

5.3.3 Anwendung bei einem Rollenwechsel

Rollenwechsel dürfen nur vom Gastgeber der Sitzung, dem *Host*, durchgeführt werden. Dazu kann der berechtigte Benutzer einem Sitzungsteilnehmer das Schreibrecht geben, entziehen oder ihm das exklusive Schreibrecht (alle anderen dürfen nicht mehr schreiben) geben.

In den ersten beiden Fälle ist dazu jeweils ein Rollenwechsel durchzuführen, d.h. ein Teilnehmer erhält eine andere Rolle. Im dritten Fall hingegen sind möglicherweise mehrere Rollenwechsel erforderlich, da denjenigen, die noch ein Schreibrecht haben, außer dem zukünftigen exklusiven Schreiber, es erst noch entzogen werden muss.

Ein solcher Rollenwechsel findet folgendermaßen statt:

Falls der betroffene Teilnehmer nicht der Host ist, wird er angehalten durch Aufruf der Methode “stop” auf dem StopManager. Diese Methode blockiert, bis der Teilnehmer angehalten ist und sein Anhalten bestätigt hat. Anschließend wird eine Rollenaktivität verschickt, die allen Sitzungsteilnehmern die neue Rolle des betroffenen Teilnehmers mitteilt. Nachdem in der Nutzerverwaltung des Hosts die neue Rolle gesetzt wurde, wird die Sperre durch Aufruf von “start” auf dem StartHandle, das sich auf den Anhaltevorgang bezieht, aufgehoben.

```
1 StartHandle startHandle = stopManager.stop(user, "Performing role change",
      progress);
2 activityCreated(new RoleActivity(getLocalUser().getJID().toString(), user.
      getJID().toString(), newRole));
3 setUserRole(user, newRole);
4 startHandle.start();
```

Listing 11: Kernablauf eines Rollenwechsels

5.3.4 Anwendung in der Konsistenzwiederherstellung

Um nicht während der Behebung einer Inkonsistenz die nächste zu verursachen, indem während der Konsistenzwiederherstellung neue Operationen erzeugt werden, müssen während dieser Neusynchronisierung alle Teilnehmer angehalten werden. Erst dann kann die eigentliche Kon-

sistenzwiederherstellung gestartet werden, bei der der Gastgeber der Sitzung dem inkonsistenten Teilnehmer die Dateien zuschickt, in denen er sich vom Host unterscheidet.

Danach können die Teilnehmer wieder gestartet werden. Allerdings muss dafür Sorge getragen werden, dass der Teilnehmer, dessen Inkonsistenz, die Konsistenzwiederherstellung ausgelöst hat, diese verarbeitet hat, bevor neue Änderungen am Projekt erzeugt werden. Daher wird zuerst der inkonsistente Teilnehmer blockierend gestartet, d.h. es wird auf eine Bestätigung gewartet, dass die Erlaubnis zum Entsperren empfangen wurde.

Da die Operationen in der Reihenfolge ihrer Erzeugung verarbeitet werden, werden also erst die inkonsistenten Dateien und dann der Start verarbeitet, der zum Versand der Bestätigung führt. Somit können wir bei Eintreffen der Bestätigung sicher sein, dass die Konsistenzwiederherstellung abgeschlossen ist.

Anschließend können die übrigen Teilnehmer wieder gestartet werden und die Sitzung kann normal weitergehen.

```
1  startHandles = stopManager.stop(sessionManager.getSharedProject().
       getParticipants(), "Consistency recovery", progress);
2  performConsistencyRecovery(from, paths, progress);
3
4  if (startHandles != null) {
5    StartHandle inconsistentStartHandle = null;
6    for (StartHandle startHandle : startHandles) {
7      if (from.equals(startHandle.getUser().getJID())) {
8        inconsistentStartHandle = startHandle;
9        break;
10     }
11   }
12   if (inconsistentStartHandle == null)
13     log.error("Could not find the StartHandle of the inconsistent user");
14   else {
15     inconsistentStartHandle.startAndAwait(progress);
16     startHandles.remove(inconsistentStartHandle);
17   }
18
19   for (StartHandle startHandle : startHandles)
20     startHandle.start();
21 }
```

Listing 12: Rolle des StopManager in der Konsistenzwiederherstellung

6 Empirische Untersuchung zu Unterschieden des Einzel- zum Mehrschreibermodus

Seit 2008 ist es in Saros möglich, dass in einer Sitzung mehrere Teilnehmer gleichzeitig im geteilten Projekt schreiben können [Rie08]. Dieser sog. Mehrschreibermodus lässt sich über die Programmeinstellungen ein- und ausschalten (Abb. 4 in Abschnitt 2.3). Doch welche Vorteile ergeben sich daraus?

Dazu wurde eine empirische Untersuchung durchgeführt, welche die folgende Forschungsfrage untersucht: "Welche Unterschiede ergeben sich aus der technischen Einschränkung der maximalen Anzahl an gleichzeitigen Drivern auf einen Benutzer (Einzelschreibermodus) gegenüber der Möglichkeit beliebig vieler Driver (Mehrschreibermodus)?"

Insbesondere wurden dazu folgende Hypothesen untersucht:

H1: Quelltext wird im Mehrschreibermodus schneller als im Einzelschreibermodus entwickelt.

H2: Quelltext, der im Einzelschreibermodus entwickelt wird, ist qualitativ hochwertiger, als Quelltext, der im Mehrschreibermodus entsteht.

Grundlage der Hypothese H1 ist die Vermutung, dass durch die Möglichkeit der Teilnehmer im Mehrschreibermodus gleichzeitig an verschiedenen Teilen der Lösung arbeiten zu können, Zeit gespart wird und somit schneller entwickelt werden kann.

Die Hypothese H2 wurde aus der Beobachtung in der Paarprogrammierung entwickelt, dass eine kontinuierliche Durchsicht, wie sie im Einzelschreibermodus erwartet wird, die Qualität des Quelltexts positiv beeinflusst (Abschnitt 2.1).

Dadurch, dass diese Studie mit wenigen Teilnehmern und einer fragilen Version von Saros durchgeführt wurde, ist sie als Pilotstudie zu sehen, die aufzeigen soll, in welche Richtungen eine künftige Untersuchung gehen und auf welche Probleme man dabei besonders achten sollte.

6.1 Aufbau

Die 16 freiwilligen Teilnehmer, Studenten und Absolventen der Informatik (siehe Abb. 13, 14 und 15 bezüglich deren Demografie) wurden zu acht Paaren, nachfolgend Teams genannt, zusammengefasst. Diese Teams wurden zufällig in zwei Gruppen A (acht Teilnehmer) und B (acht Teilnehmer) verteilt. Die Partner von sechs der acht Teams kannten sich bereits vor der Studie und haben sich jeweils zur gemeinsamen Teilnahme angemeldet. Ein Team aus Gruppe A und eines aus Gruppe B bestand aus Teilnehmern, die sich vor der Studie nicht kannten.

Die Teilnehmer erhielten zwei Programmieraufgaben 1 und 2, die sie im Team in der Programmiersprache Java lösen sollten. In der ersten Aufgabe hatten sie vor dem Hintergrund eines Pokerspiels die drei vorgegebenen Schnittstellen Karte, Stapel und Blatt zu implementieren (Abschnitt A.1.1). In der zweiten Aufgabe galt es eine Funktion zu entwickeln, die eine quadratische Gleichung aus einer Zeichenkette einliest und die Lösungen dafür zurückgibt, die auch komplex sein können (Abschnitt A.1.2).

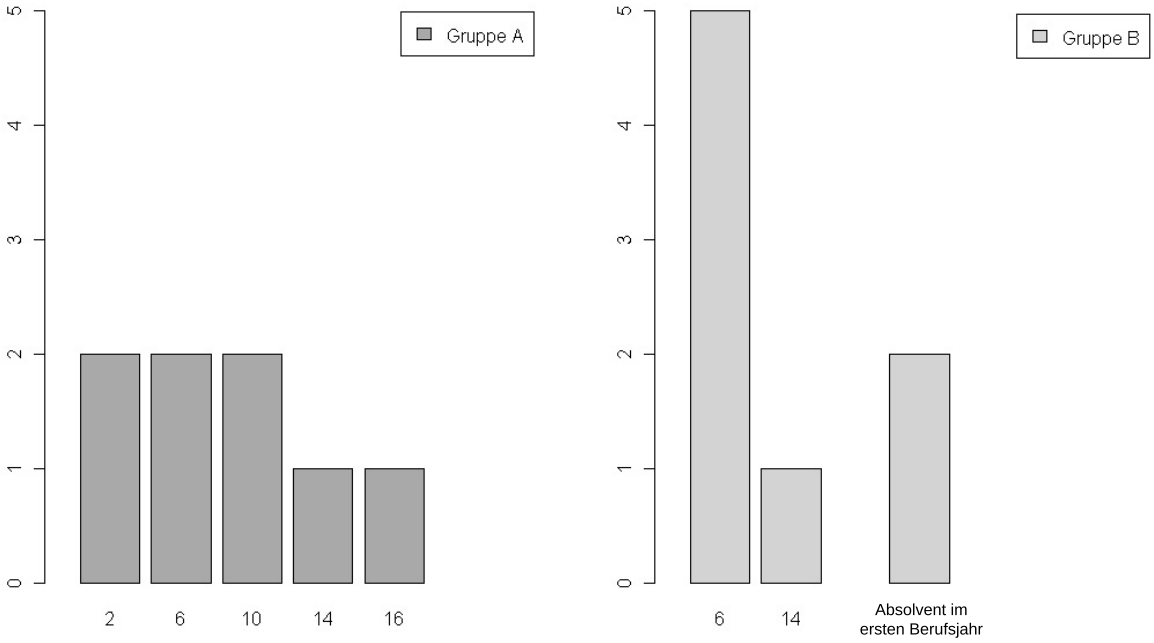

Abbildung 13: Informatik-Fachsemester der Teilnehmer

Die Paare der Gruppe A sollten dabei zuerst Aufgabe 1 im Einzel- und dann Aufgabe 2 im Mehrschreibermodus lösen, während die anderen Paare die Aufgabe 1 im Mehrschreibermodus lösen, bevor sie Aufgabe 2 im Einzelschreibermodus bearbeiteten. Dieses Überkreuzdesign sollte verhindern, dass ein Modus besser abschneidet, weil ein jeweiliges Paar schon eingespielt ist, indem es bereits eine Aufgabe gemeinsam gelöst hat. Zudem sollten qualitative Unterschiede der Teilnehmer hinsichtlich Kompetenz in der Softwareentwicklung dadurch ausgeglichen werden, da ja jeder Teilnehmer in beiden Modi arbeiten muss.

Während der Bearbeitung der Aufgaben saßen die beiden Teilnehmer eines Teams in getrennten Räumen an jeweils einem Rechner und waren über Skype (VoIP-Software) und Saros verbunden. Zur Recherche durften sie während der Bearbeitungszeit der Aufgaben auf das Internet zugreifen.

Vor der ersten, vor der zweiten und nach der zweiten Aufgabe erhielten die Teilnehmer einen Fragebogen (Abschnitt A.2), in dem sie Erfahrungen mit Saros und dem Entwicklungsprozess aufschreiben sollten.

Unmittelbar vor dem ersten Fragebogen erhielt jedes Paar eine kurze Einführung in Saros. Diese wurde vom Versuchsleiter anhand eines Stichpunktzettels (Abschnitt A.3) mit jeweils einem Team an einem Rechner durchgeführt.

Der Versuchsleiter maß die Bearbeitungszeit der Aufgaben, während er mit jeweils einem der Probanden im Raum saß. Kam es zu technischen Problemen mit Saros (meist bedingt durch ein Fehlverhalten der Software), sprach er Empfehlungen für ein angemessenes Vorgehen aus.

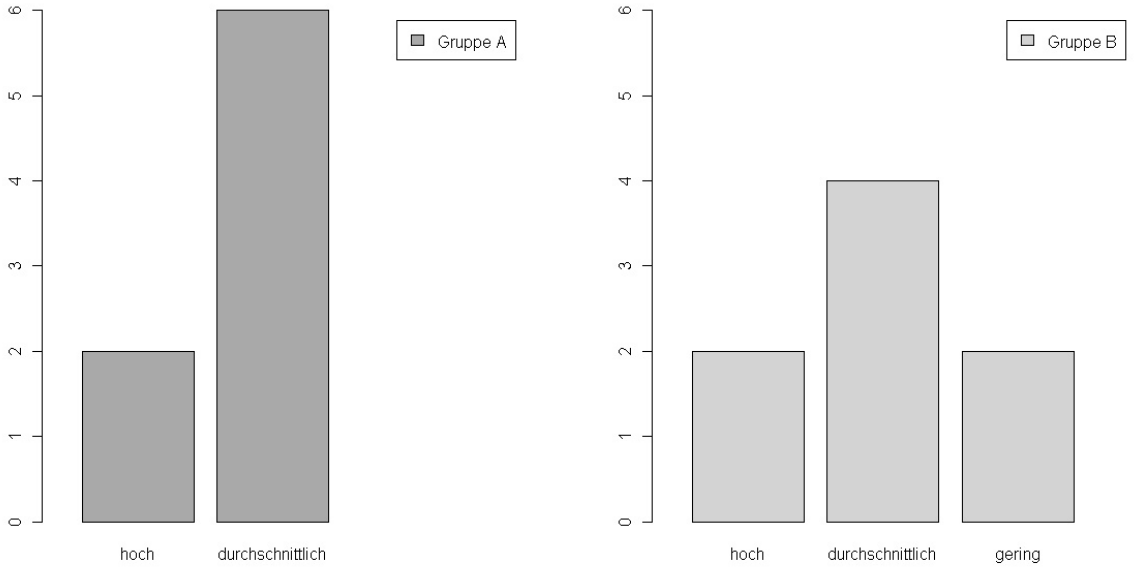

Abbildung 14: Java-Kenntnisse der Teilnehmer nach Selbsteinschätzung

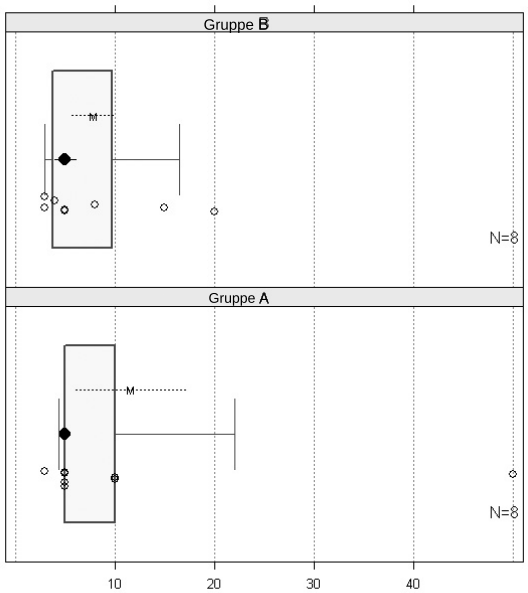

Abbildung 15: bislang absolvierte Paarprogrammierungssitzungen der Teilnehmer

Außerdem musste er bei einigen Teams Domänenwissen für die zweite Aufgabe nachreichen, da die betroffenen Teams sich nicht selbstständig hinreichend informieren konnten.

Um den Ablauf der Studie zu optimieren, wurde ein Vorversuch vorgenommen, bei dem ein Paar, welches an der Hauptstudie nicht mehr teilnahm, die vorgesehenen Aufgaben zu lösen und die Fragebögen zu beantworten hatte. Dies führte dazu, dass die erste Aufgabe stark gekürzt werden musste, weil die Bearbeitungszeit sonst zu lang gewesen wäre.

Zur Auswertung wurden von diesem ersten Paar nur die Fragebögen, von den übrigen Teams wurde zusätzlich noch der von den Probanden erzeugte Quelltext verwendet.

6.2 Ergebnisse und Diskussion

Zur Auswertung sollte vor allem die Bearbeitungszeit und die Qualität des Quelltexts festgestellt werden. Erstere wurde direkt und die Qualität in den Dimensionen Korrektheit und Komplexität gemessen.

Für die Korrektheit wurden JUnit-Tests verwendet. Für die erste Aufgabe gab es elf, für die zweite 13 Tests. Es wurden nur Eingaben getestet, die durch die Spezifikation der Aufgabe zulässig waren. Allerdings wurde darauf geachtet die Tests so zu wählen, dass ein Scheitern wahrscheinlich erschien.

Als Maß der Komplexität wurde die zyklomatische Komplexität nach McCabe [McC76] mit der Eclipse-Erweiterung Metrics 1.3.6 [RW05] ermittelt. Diese misst die Anzahl der möglichen Wege durch eine Methode. Für jede Verzweigung (*if, for, while, do, case, catch, ?:, &&* oder *||*) erhöht sich diese Größe um eins. Da die Messung nur methodenweise durchgeführt wurde, wird der Mittelwert über alle implementierten Methoden einer Aufgabe zur weiteren Verwendung betrachtet.

Ein Team aus Gruppe B hat Aufgabe 1 aus eigenem Wunsch nach 120 Minuten abgebrochen. Da dieses Paar zu dem Zeitpunkt bereits einen Großteil der Aufgabe gelöst hatte, wurde der erarbeitete Quelltext wie die übrigen, komplett gelösten ausgewertet und eine Bearbeitungszeit von 120 Minuten protokolliert.

6.2.1 Ergebnisse

Bezüglich der Zeitmessung konnte sich keiner der beiden Modi klar durchsetzen (Abb. 16).

Aufgabe 1 wurde im Einzelschreibermodus in durchschnittlich 113 Minuten (mit einer Standardabweichung σ von 22 Minuten) gelöst, während sie im Mehrschreibermodus in durchschnittlich 111 ($\sigma = 9$) Minuten gelöst wurde.

Auch in Aufgabe 2 war man im Einzelschreibermodus mit durchschnittlich 80 ($\sigma = 4$) Minuten etwas langsamer als im Mehrschreibermodus mit durchschnittlich 72 ($\sigma = 31$) Minuten.

Allerdings sind die Unterschiede, verglichen mit den Standardabweichungen sehr gering, weshalb sich hier höchstens die Tendenz erahnen lässt, dass man im Mehrschreibermodus schneller arbeiten kann.

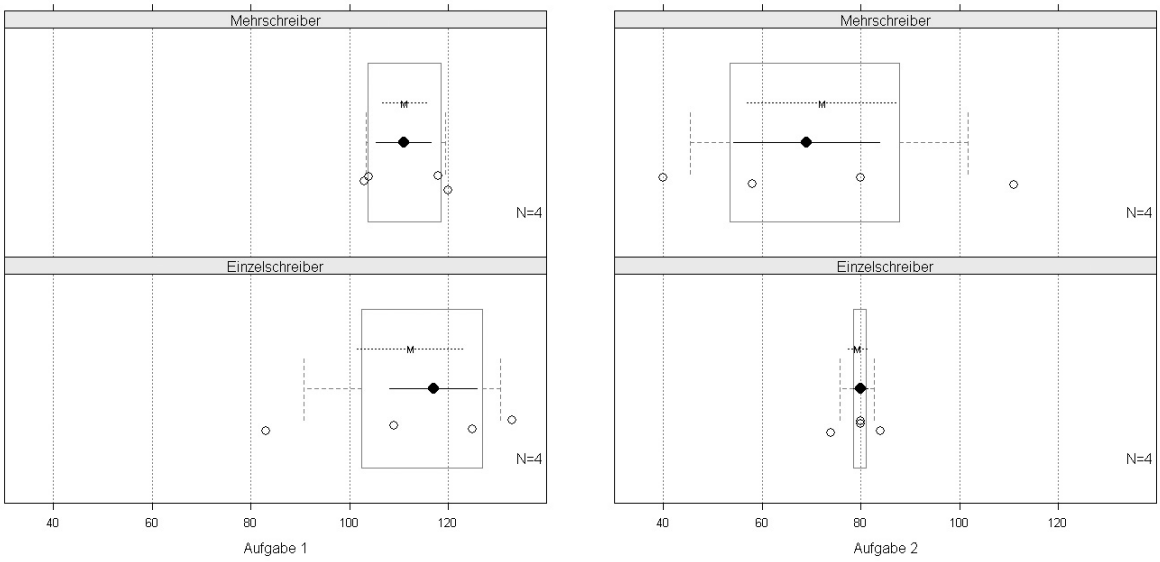

Abbildung 16: Zeitbedarf im Mehr- und im Einzelschreibermodus

Auch bezüglich der Korrektheit anhand von JUnit-Tests und der zyklomatische Komplexität nach McCabe kann man keine eindeutigen Aussagen treffen (Abb. 17, 18).

In Aufgabe 1 war die durchschnittliche Korrektheit im Einzelschreibermodus mit 53 ($\sigma = 35$) % höher als im Mehrschreibermodus mit 23 ($\sigma = 22$) %. Dagegen verhielt es sich in Aufgabe 2 genau andersherum mit 19 ($\sigma = 25$) % im Einzelschreibermodus gegen 58 ($\sigma = 15$) % im Mehrschreibermodus. Das deckt sich mit der zyklomatischen Komplexität, die in Aufgabe 1 im Einzelschreibermodus mit 1,7 ($\sigma = 0{,}4$) geringer war als im Mehrschreibermodus mit 3,3 ($\sigma = 1{,}1$), während in Aufgabe 2 der Einzelschreibermodus mit 2,1 ($\sigma = 1{,}0$) schlechter abschnitt als der Mehrschreibermodus mit 1,6 ($\sigma = 0{,}3$).

6.2.2 Diskussion

Dass kaum Unterschiede feststellbar sind, kann daran liegen, dass sich nicht jedes Paar im Mehrschreibermodus die Arbeit aufgeteilt hat. Drei der acht Paare arbeiteten auch im Mehrschreibermodus immer gemeinsam am gleichen Problem. Da dadurch in der Arbeitsweise kaum ein Unterschied zum Einzelschreibermodus entsteht, sollten sich auch die Messgrößen nicht stark unterscheiden. Aufgrund der geringen Fallzahl macht allerdings eine getrennte Untersuchung hier keinen Sinn.

Die oben genannten Hypothesen konnten also durch die Ergebnisse der Untersuchung weder eindeutig belegt noch widerlegt werden. Es ergeben sich zwei Vermutungen, die nachfolgend auch untersucht wurden:

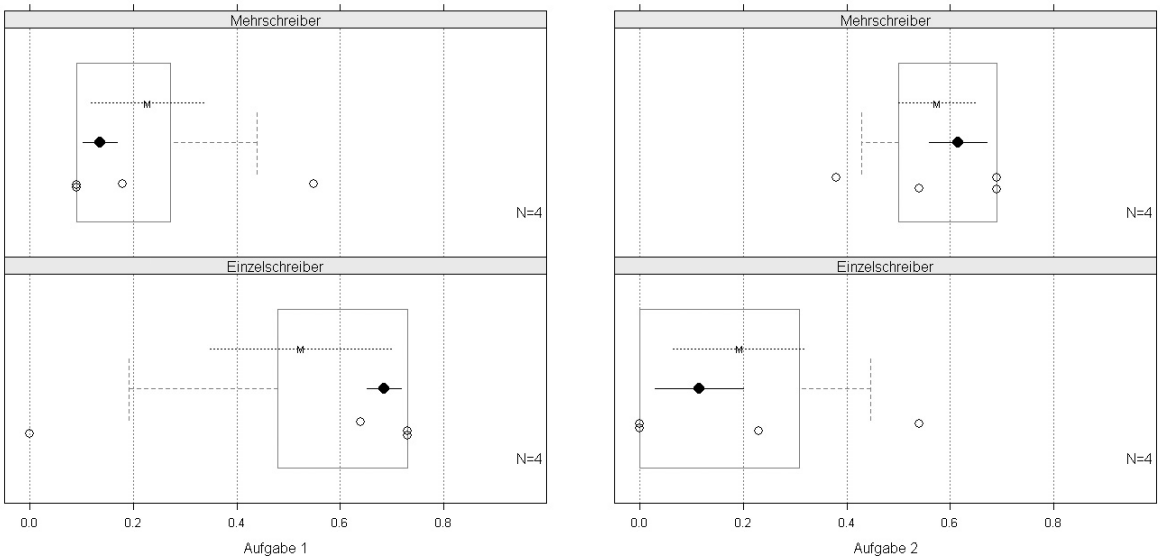

Abbildung 17: Korrektheit im Mehr- und im Einzelschreibermodus

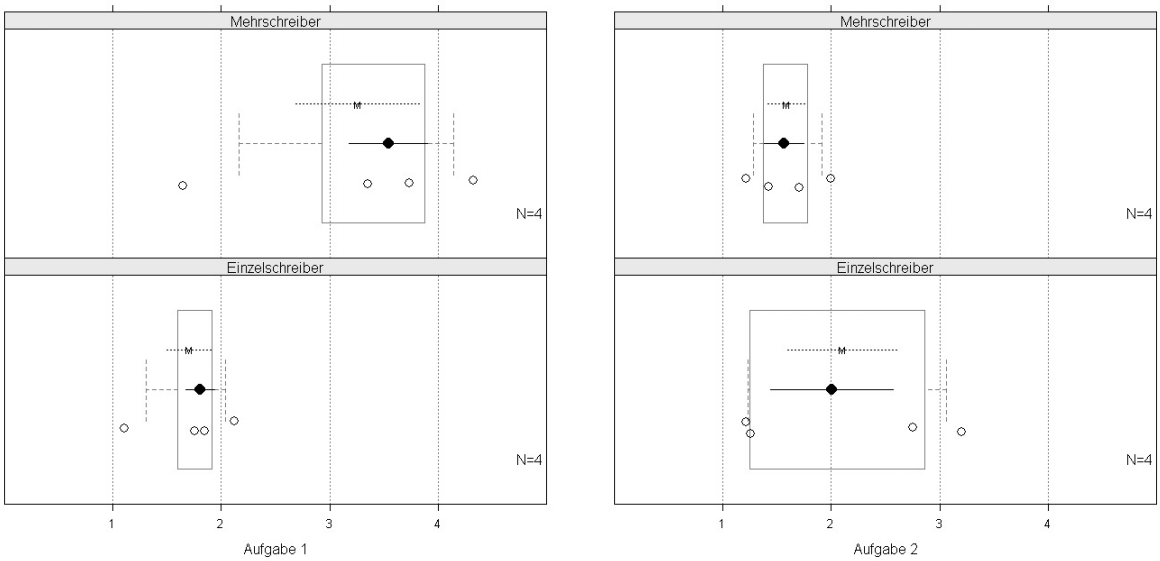

Abbildung 18: zyklomatische Komplexität nach McCabe im Mehr- und im Einzelschreibermodus

Vermutung 1: Ein schnelles Team bleibt schnell

Die Vermutung, die nach Widerlegung der Hypothese H1 entsteht, lautet: Ein Team, das die erste Aufgabe schnell löst, löst auch die zweite schnell, unabhängig vom verwendeten Modus.

Um das graphisch zu veranschaulichen, wird die Bearbeitungszeit der ersten Aufgabe gegen die der zweiten aufgetragen (Abb. 19). Um die Vermutung zu stützen, wird ein linearer steigender Zusammenhang erwartet.

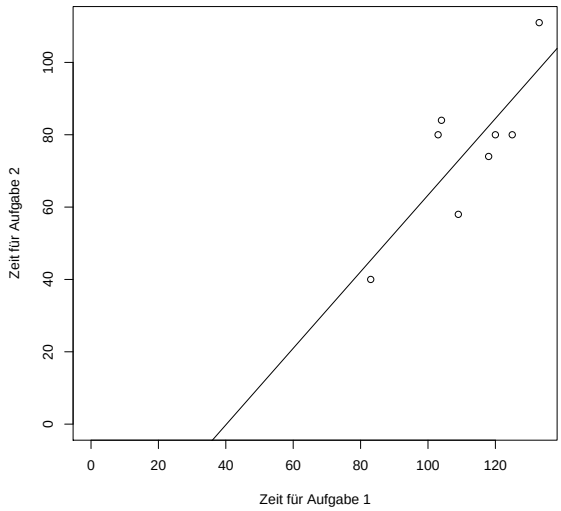

Abbildung 19: Bearbeitungszeit der ersten Aufgabe gegen die der zweiten

Tatsächlich kann dieser Zusammenhang beobachtet werden. Der Korrelationskoeffizient nach Pearson [BSMM01] beträgt 0,80, allerdings auf einem Signifikanzniveau von 0,16. Hier wäre also eine weitere Untersuchung mit einer größeren Anzahl an Teilnehmern angebracht, um den Zusammenhang näher zu untersuchen.

Vermutung 2: Die Qualität, die ein Team erreicht, hängt nicht von Aufgabe oder Modus ab

Analog entsteht die Vermutung, dass auch die Qualität der Lösung, die ein Team produziert, unabhängig von Aufgabe oder Modus ist, d.h. ein Team, das in der ersten Aufgabe gute Qualität geliefert hat, tut dies auch in der zweiten.

Hier lässt sich kein linearer Zusammenhang erkennen. Tatsächlich ergibt sich für die zyklomatische Komplexität ein Korrelationskoeffizient von 0,60 mit einer Signifikanz von 0,11 und für die Korrektheit ein Korrelationskoeffizient von sogar nur 0,24 mit einer Signifikanz von 0,57.

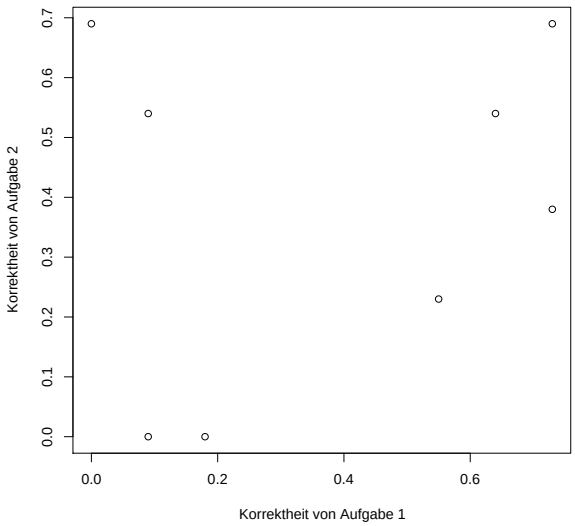

Abbildung 20: Korrektheit der ersten Aufgabe gegen die der zweiten

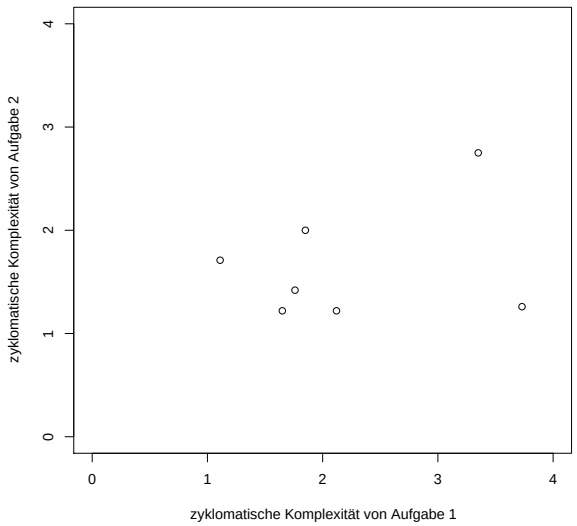

Abbildung 21: zyklomatische Komplexität der ersten Aufgabe gegen die der zweiten

6.2.3 weitere Untersuchungen

Anhand der Fragebögen wurden noch weitere Untersuchungen durchgeführt, die weder mit den Hypothesen H1 und H2, noch mit den beiden oben genannten Vermutungen in Zusammenhang stehen.

Vertrauen in die eigene Arbeit

Während der Studie äußerten Teilnehmer vereinzelt die Vermutung, dass die im Einzelschreibermodus entstandenen Lösungen korrekter seien als die übrigen. Diese Einschätzung teilten aber nicht alle. Bezüglich des Vertrauens in die eigene Arbeit konnte nämlich kein Unterschied zwischen dem Mehr- und dem Einzelschreibermodus festgestellt werden (Abb. 22). Auch ein Zusammenhang zwischen dem Vertrauen und der tatsächlichen Korrektheit war nicht auszumachen.

Rolle des Gastgebers

Wie in Abschnitt 2.3 beschrieben, hat der Gastgeber einer Saros-Sitzung zwei Privilegien. Nur er kann Rollenwechsel veranlassen und im Fall einer Konsistenzwiederherstellung gilt sein Projektstand als Referenz.

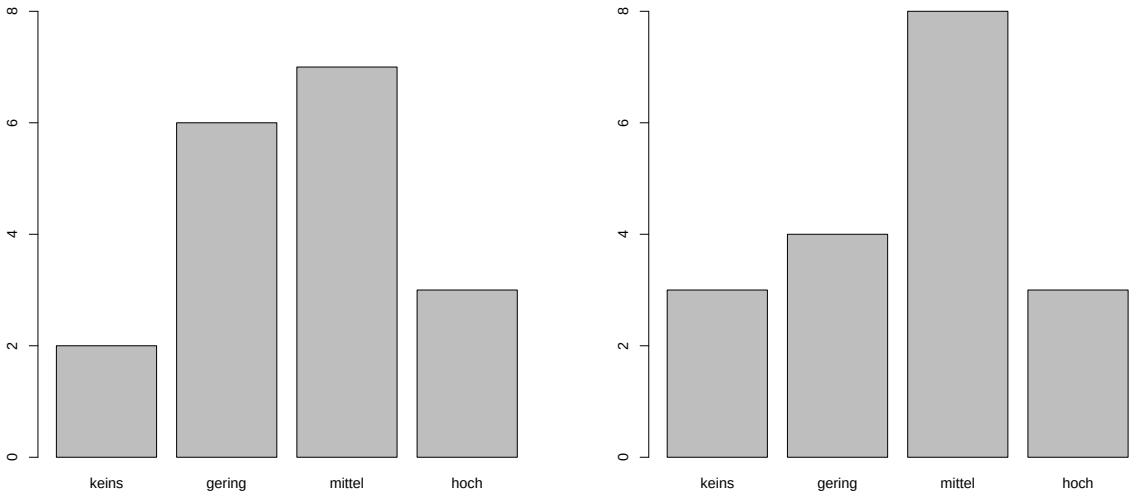

Abbildung 22: Fragebogen: Wie viel Vertrauen haben Sie in die Korrektheit ihrer Lösung? Die Antworten sind in der Abbildung (links im Einzel-, rechts im Mehrschreibermodus) gekürzt wiedergegeben. Der volle Wortlaut ist in Abschnitt A.2 zu finden.

Daher wurden die Teilnehmer befragt, wie der Gastgeber im Einzelschreibermodus [2] bestimmt wurde. 11 % bestimmten den Host aufgrund seiner höheren Rechnerleistung, 22 % aufgrund höherer Kompetenz in Java oder im Umgang mit der Software und 67 % sprachen von einer zufälligen Wahl.

Für eine zufällige Wahl gibt es zwei mögliche Ursachen. Entweder war den Teilnehmern die besondere Eigenschaft des Gastgebers während der Entscheidungsfindung nicht bewusst oder es war ihnen einfach egal. Welches der zutreffende Grund war, kann nicht gesagt werden, da die entsprechenden Daten nicht erhoben wurden.

6.3 Einschränkungen der internen Gültigkeit

6.3.1 Inkonsistenzen

Während der Untersuchung traten immer wieder Inkonsistenzen auf, d.h. die beiden Partner hatten verschiedene Projektzustände, weil Änderungen des einen nicht korrekt bei dem anderen ausgeführt wurden. Dabei handelt es sich nicht um einen Bedienfehler der Benutzer, sondern um ein Versagen der Software.

Durch Betätigen der Konsistenzwiederherstellung kann der Benutzer wieder einen konsistenten Zustand herbeiführen, indem der Gastgeber der Sitzung den anderen Teilnehmer auf seinen Projektzustand bringt. Dabei können Änderungen dieses Teilnehmers verlorengehen. Die meisten

[2] Im Mehrschreibermodus entfielen die Rollenwechsel, da in allen Sitzung gleich zu Beginn beide Teilnehmer Driver wurden. Daher hat der Gastgeber, der als einziger Teilnehmer Rollenwechsel durchführen kann, im Einzelschreibermodus eine höhere Bedeutung.

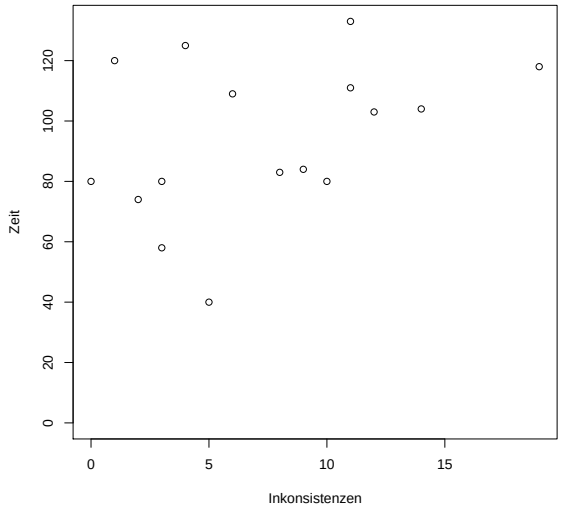

Abbildung 23: Bearbeitungszeit gegen die aufgetretenen Inkonsistenzen

Teams reagierten so darauf, dass sie die potenziell gefährdeten Änderungen vor Auslösung der Konsistenzwiederherstellung in der Zwischenablage gesichert haben, um sie in konsistentem Zustand wieder einzufügen.

In wenigen Fällen versagte sogar die Konsistenzwiederherstellung und die Sitzung musste neu gestartet werden.

Die hohe Anzahl an Inkonsistenzen soll daher genauer betrachtet werden, um gültige Aussagen treffen zu können. In Gruppe A traten 51 und in Gruppe B 67 Inkonsistenzen auf. Da nicht bekannt ist, wie stark der Arbeitsfluss durch diese Störungen beeinflusst wurde, und nicht davon auszugehen ist, dass die Störung bei jedem Team die gleiche Wirkung hatte (Abb. 23), enthält also die Zeitmessung einen systematischen Messfehler.

Es bleibt die Frage, ob sich die Inkonsistenzen auch auf die Qualität ausgewirkt hatten. Bei der Korrektheit lässt sich kein Zusammenhang zu den Inkonsistenzen beobachten (Abb. 24). Hingegen lässt sich bezüglich der zyklomatischen Komplexität ein linearer Zusammenhang erahnen (Abb 25), welcher sich durch Rechnung allerdings nicht bestätigen lässt (Korrelationskoeffizient von 0,32 auf einem Signifikanzniveau von 0,23). Ob die Inkonsistenzen Einfluss auf die Qualität der entstandenen Lösung hatten, kann daher nicht beantwortet werden.

Eine weitere Studie wäre daher nur mit einer stabileren Saros-Version sinnvoll.

6.3.2 Qualitätsmessung

Die Qualität wurde in dieser Untersuchung in den beiden Kategorien Korrektheit und Wartbarkeit untersucht. Als Maß für die Korrektheit wurden automatische JUnit-Tests verwendet, die aber unmöglich alle Fälle abdecken können. Bei dem Entwurf der Tests wurden besonders solche konstruiert, bei denen ein Scheitern wahrscheinlich erschien. Das bedeutet, dass die oben ange-

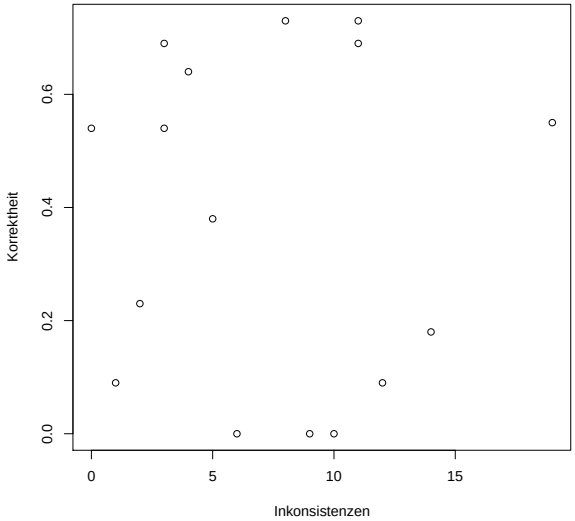

Abbildung 24: Korrektheit gegen die aufgetretenen Inkonsistenzen

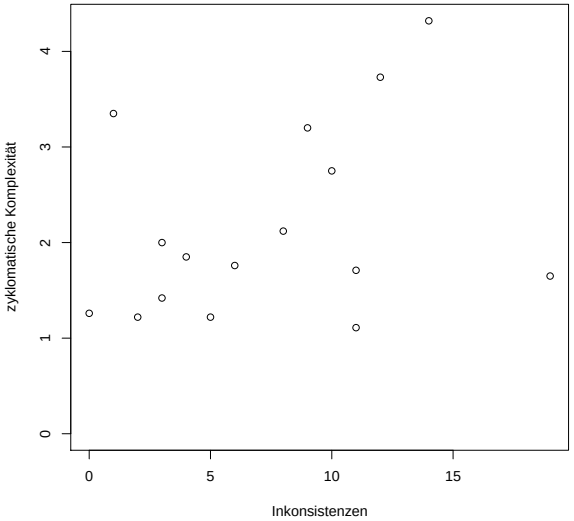

Abbildung 25: zyklomatische Komplexität gegen die aufgetretenen Inkonsistenzen

gebene Korrektheit als Zahl nicht absolut zu sehen ist. Da aber für jede Lösung die selben Test angewendet wurden, ist sie ein guter Vergleichswert.

Für die Wartbarkeit wurde nur die zyklomatische Komplexität nach McCabe gemessen. Natürlich hat Wartbarkeit noch viele weitere Facetten als Komplexität, wie z.B. eine geeignete Modularisierung oder das Einhalten von Kodierkonventionen.

Leider sind Kodierstandards, welche die Teilnehmer in den nur vier Stunden ihrer Teilnahme einhalten sollen, nicht durchsetzbar. So wäre es sinnlos gewesen ihre Einhaltung zu messen. Auch die Messung einer guten Modularisierung ist durch den geringen Umfang der Aufgabe nicht sinnvoll.

Die zyklomatische Komplexität stellt hingegen eine sinnvolle und vor allem objektive und automatisiert messbare Größe dar. Allerdings ist auch sie nur als Vergleichswert, nicht als absolutes Maß für Wartbarkeit zu sehen.

6.3.3 Domänenwissen

Beide Aufgaben benötigten Domänenwissen. In der ersten Aufgabe handelte es sich dabei um die Rangfolge der verschiedenen Pokerblätter, die in der Aufgabenstellung beschrieben war, während es in der zweiten Aufgabe um komplexe Zahlen ging.

Das benötigte Wissen wurde vom Versuchsleiter bei den Teilnehmern fälschlicherweise vorausgesetzt und war nicht in der Aufgabenstellung beschrieben. Die Teams kamen unterschiedlich mit dieser Hürde klar. Einige hatten das nötige Wissen, andere konnten es selbstständig im Internet

erwerben und wieder andere haben 15 Minuten erfolglos recherchiert, bis der Versuchsleiter dem Paar eine kurze Einführung in komplexe Zahlen gab.

Das hat natürlich die Zeitmessung verfälscht, kann aber auch dazu geführt haben, dass die Korrektheit dadurch beeinflusst wurde. Im Falle einer Wiederholung der Studie sollte vorher also intensiv über eventuell benötigtes Domänenwissen nachgedacht werden.

6.4 Einschränkungen der externen Gültigkeit

Obwohl an der Studie nur Studenten und Berufseinsteiger teilgenommen haben, wird dadurch nicht die externe Gültigkeit eingeschränkt, wie empirische Studien gezeigt haben [SWN+03, DB98]. Problematisch ist aber hier die große Qualitätsbandbreite der Teilnehmer. Schließlich waren Studenten vom zweiten bis zum 16. Semester vertreten und zusätzlich noch zwei Absolventen. Entsprechend breit gefächert waren auch die Programmier- und speziell Java-Kenntnisse.

Stellt die Art der Aufgabenstellung auch ein Problem dar?

In beiden Aufgaben konnten die Teilnehmer ihre Lösung “auf der grünen Wiese” entwickeln. Sie hatten keine äußeren Bedingungen zu beachten, wie es in großen Softwareprojekten der Fall ist. So war es z.B. den Teilnehmern überlassen, ob sie ihre Lösung testen. Sie mussten auch nicht an andere Teile eines größeren Projektes anknüpfen, sondern ihre Aufgabe stand völlig isoliert ohne einen größeren Zusammenhang.

Die Alternative wäre gewesen, die Teilnehmer an ein größeres Projekt zu setzen, in dem sie eine gegebene Funktionalität einzubauen hätten. Ich habe mich gegen diese Möglichkeit entschieden, da ein großer Teil der Bearbeitungszeit darin bestanden hätte, sich in dem gegebenen Projekt zurechtzufinden und somit nur Aufgaben mit sehr geringem algorithmischen Anspruch zeitlich zumutbar gewesen wären.

6.5 Wiederholung der Studie

Während der Studie ergaben sich einige Schwierigkeiten, die vorher nicht bedacht wurden. Das ist sicherlich einer der Gründe für die mangelnde Aussagekraft der Untersuchung. Daher wird eine Wiederholung mit mehr Teilnehmern empfohlen. Zusammenfassend möchte ich an dieser Stelle die Dinge nennen, die beachtet werden sollten, damit eine Wiederholung mehr Erfolg zeigt:

Zuverlässigkeit der Software

In den Abbildungen 26 und 27 sind die Punkte zu sehen, die von den Teilnehmern am häufigsten als störend bezeichnet wurden. Wie man sieht, wurden die immer wieder aufgetretenen Inkonsistenzen von den Teilnehmern als extrem lästig empfunden. Zudem verfälschen sie die Zeitmessung. Bevor die Studie wiederholt werden kann, muss Saros daher eine Qualität erreichen, in der Inkonsistenzen nur ganz selten oder idealerweise gar nicht mehr auftreten.

Leider sind die Ursachen der Entstehung von Inkonsistenzen noch nicht vollständig bekannt, weswegen zur Zeit noch keine hinreichende Zuverlässigkeit für eine Wiederholung vorhanden ist.

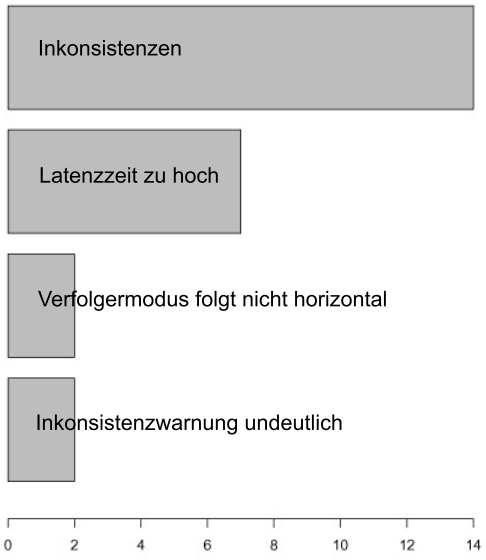

Abbildung 26: Fragebogen: Was hat Sie an Saros gestört?

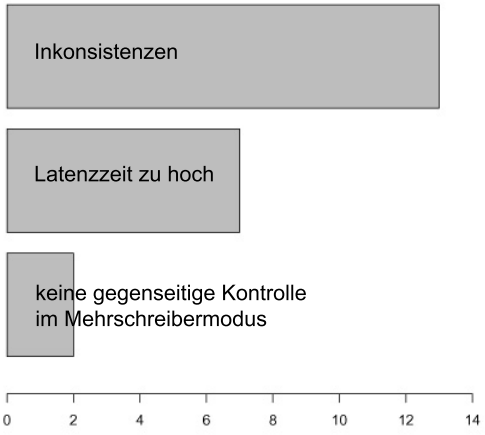

Abbildung 27: Fragebogen: Was hat Sie an der Sitzung gestört?

Hohe Latenz

Am zweithäufigsten unter den Dingen, welche die Teilnehmer als störend empfanden, wurde die hohe Latenzzeit genannt. Das bedeutet, es hat zu lange gedauert, bis eine Operation, die ein Teilnehmer erzeugt hat, bei dem anderen ausgeführt wurde.

Der Hauptgrund ist vermutlich nicht in der Software oder in der Java-Laufzeitumgebung zu finden, sondern in dem Ort des Jabber-Servers, über den alle Operationen laufen. Im Vorfeld erwies sich jabber.no, ein norwegischer öffentlicher Server als der zuverlässigste im Hinblick auf den Synchronisierungsprozess zu Beginn der Sitzung.

Dadurch, dass aber nun alle Aktivitäten erst nach Norwegen übermittelt wurden, um sie dann wieder im Labor zu empfangen, entstand eine für die Teilnehmer spürbare Latenz.

Eine Alternative, die man im Fall einer Wiederholung nutzen sollte, ist ein lokaler, eigener Jabber-Server, von dem kürzere Latenzzeiten erwartet werden.

Aufgabenstellung: Umfang und Domänenwissen

Fünf der acht Teams hatten Wissenslücken im Umgang mit komplexen Zahlen, die sie selbstständig nicht schließen konnten. Dieses Domänenwissen wurde aber vom Versuchsleiter vorausgesetzt, weswegen in der Aufgabenstellung keine Hilfe vorhanden war.

In einem neuen Entwurf der Aufgabenstellung sollte daher genau geprüft werden, welche spezifischen Vorkenntnisse benötigt werden und den aufgabenrelevanten Teil den Teilnehmern schriftlich mitteilen. Nur so ist gewährleistet, dass die Zeitmessung nicht durch langwierige Recherche und die Korrektheit nicht durch mangelndes Domänenwissen verfälscht wird.

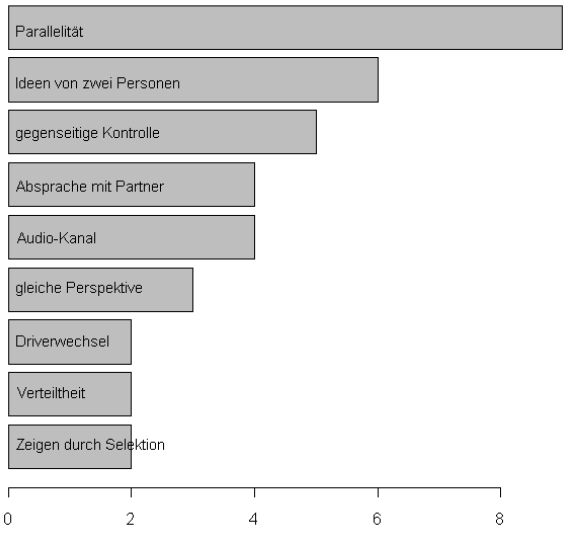

Abbildung 28: Fragebogen: Was fanden Sie gut in der Sitzung?

Es ist nicht leicht eine Aufgabe zu finden, die weder zu einfach noch zu umfangreich ist. Für die erste Aufgabe wurden hier durchschnittlich 112 Minuten und für die zweite 76 Minuten benötigt, zusammen also 188 Minuten. Mit Fragebögen, Einweisung in Saros und einer eventuellen Pause zwischen den Aufgaben, lag der Zeitaufwand bei fast vier Stunden.

Es ist schwierig Teilnehmer zu finden, die freiwillig bereit sind, so viel Zeit zu investieren. Daher wird empfohlen beide Aufgaben so zu kürzen, dass sie jeweils in einer Stunde lösbar sind. Mit einer leichten Kürzung der Fragebögen sollte man dann auf unter drei Stunden kommen.

Falls man die Studie mit einer gut besuchten Lehrveranstaltung verknüpft, wäre es somit ein angemessener Tausch ein Übungsblatt durch die Teilnahme an der Studie zu ersetzen.

Parallelität der Sitzungen

Durch eine Verknüpfung mit einer Lehrveranstaltung hätte der Versuchsleiter die Möglichkeit mit geringem Aufwand viele Teilnehmer zu rekrutieren. Dadurch wird es aber schwierig mit diesem Ansturm zeitlich zurecht zu kommen. In der vorliegenden Studie hat der Versuchsleiter nicht mehr als zwei Sitzungen am Tag geschafft und auch nach Kürzung der Aufgabenstellung wird man nicht mehr als drei schaffen.

Es sollte also ein Weg gefunden werden, wie man mehrere Sitzungen parallel realisieren kann. Das Problem dabei ist, dass für jeden Teilnehmer ein eigener Raum gefunden werden muss. Schließlich ist es sonst möglich, dass sich zwei Teilnehmer aus verschiedenen Teams gegenseitig beeinflussen, indem z.B. ein Team den Lösungsvorschlag eines anderen Teams aufgreift, den es gerade aufgeschnappt hat.

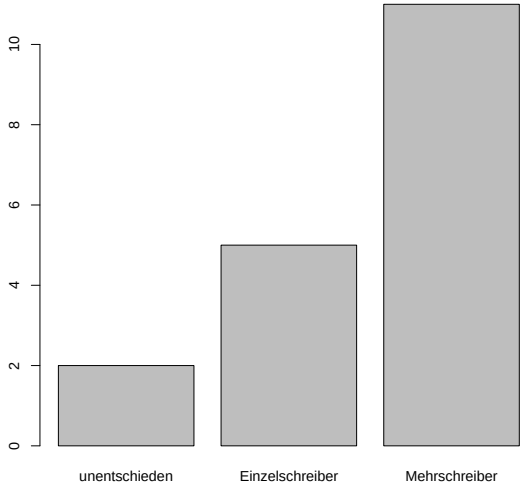

Abbildung 29: Fragebogen: Hat Ihnen die heutige Sitzung im Mehr- oder im Einzelschreiberbetrieb mehr Spaß gemacht?

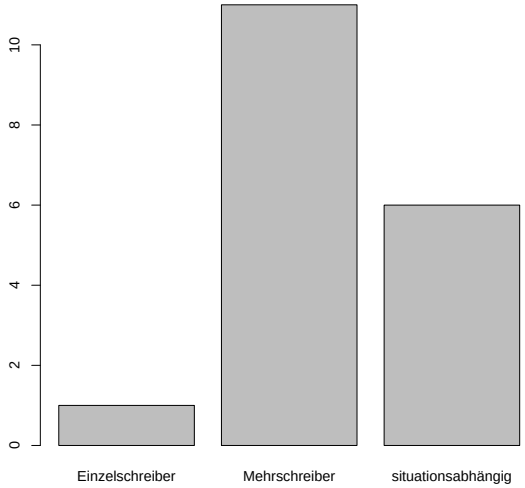

Abbildung 30: Fragebogen: Würden Sie bei einer zukünftigen Verwendung den Einzel- oder den Mehrschreiberbetrieb vorziehen?

6.6 Stärken von Saros

Bei allen Schwierigkeiten, die im Umgang mit der Software auftraten, hatten dennoch 67 % der Teilnehmer laut eigenen Angaben Spaß an den Sitzungen. Alle Teilnehmer gaben an, dass sie zufrieden mit der Zusammenarbeit mit ihrem jeweiligen Partner waren (die Teilnehmer waren während der Befragung voneinander getrennt).

Weitere Punkte, die den Teilnehmern positiv aufgefallen sind, kann man Abb. 28 entnehmen.

Wie in den Abb. 29 und 30 zu sehen ist, wird der Mehrschreibermodus bevorzugt verwendet.

Das deckt sich mit den Aussagen der Teilnehmer nach den Aufgaben. Die Sitzungen im Mehrschreibermodus hat 78 % der Teilnehmer Spaß gemacht, während im Einzelschreibermodus nur 56 % diese Aussage trafen.

7 Ausblick

Wie in Abb. 31 zu sehen, wächst der Bekanntheitsgrad von Saros aufgrund der in Abschnitt 3.5 genannten Maßnahmen stetig. Daraus kann man auf eine wachsende Benutzergemeinde schließen, von der man sich Beteiligung am Projekt erhofft, z.B. in Form von Informationen über Fehlverhalten der Software oder sogar eigenen Entwicklungsbeiträgen.

Große Erfolge sind dabei aber bisher ausgeblieben. Zwar gab es Rückmeldungen von Benutzern, indem sie an einer Umfrage [Doh09] teilnahmen, allerdings waren die gemeldeten Versagen fast immer bekannt. Entwicklungsbeteiligung von externen Entwicklern gab es bisher auch nicht, aber vielleicht ist dafür die Nutzergemeinde noch zu klein.

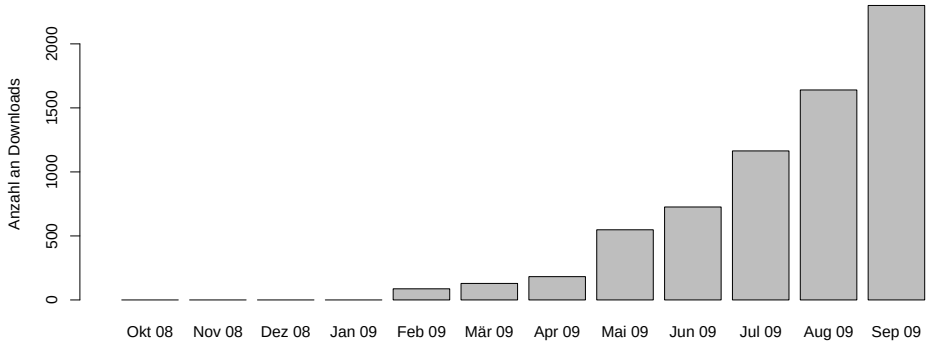

Abbildung 31: Entwicklung der Anzahl an monatlichen Downloads

Um die größten Probleme, die in naher Zukunft gelöst werden sollten, zu identifizieren, bedarf es jedoch keiner weiteren Rückmeldung. In Abschnitt 6.5 wurde gezeigt, dass es sich dabei um die hohe Anzahl an Inkonsistenzen und um die lange Latenzzeit handelt.

Bezüglich der Inkonsistenzen sind die Ursachen immer noch nicht vollständig bekannt. Es sollte nach Möglichkeiten gesucht werden, diese systematisch zu ergründen. Ein möglicher Ansatz könnte dabei der Ausbau automatisierter Tests sein. Das Problem dabei ist, dass es bisher nicht gelungen ist, einen automatisierten Test unter realistischen Bedingungen durchzuführen, d.h. die Software sollte auf mindestens zwei verschiedenen Rechnern laufen, die miteinander kommunizieren.

Die lange Latenzzeit kann man mit einem lokalen Jabber-Server wie OpenFire reduzieren. Da man aber dem Nutzer den Aufwand der Einrichtung ersparen möchte, sollten die Nachrichten, welche die Teilnehmer zur Kommunikation verwenden, nicht erst über einen Jabber-Server gehen, sondern auf direktem Wege den Empfänger erreichen. Diese Technik wird bereits im Einladungsprozess zu Beginn der Sitzung zur Synchronisation verwendet, danach aber nicht mehr.

Sind diese beiden Probleme gelöst, hat man mit Saros ein nützliches Werkzeug, mit dem es Spaß macht Verteilte Paarprogrammierung zu praktizieren.

8 Zusammenfassung

In dieser Arbeit werden die Begriffe “Paarprogrammierung” und “Verteilte Paarprogrammierung” erklärt. Da “Verteilte Paarprogrammierung” immer technische Unterstützung voraussetzt, wird die Software Saros vorgestellt, eine Erweiterung der Entwicklungsumgebung Eclipse.

Dieses Werkzeug zur Verteilten Paarprogrammierung wird an der Freien Universität Berlin von Studenten und wissenschaftlichen Mitarbeitern entwickelt. Um der anfänglich stetig sinkenden Qualität der Software entgegenzuwirken, wurde ein Prozess definiert, der Verhaltensregeln zur Anforderungsverwaltung, Projektplanung, Qualitätssicherung und interner, sowie externer Kommunikation enthält, in den in Abschnitt 3 eingeführt wird.

Da man mit Saros im Mehrschreibermodus die Möglichkeit hat das klassische Driver-Observer-Muster der Paarprogrammierung aufzubrechen, und stattdessen mehrere Teilnehmer zumindest technisch zu Drivern zu machen, ergeben sich einige Nebenläufigkeitsprobleme, die in Abschnitt 4 erläutert werden. Dazu gehört das Problem, dass parallel erzeugte, nicht vertauschbare Operationen so bei allen Teilnehmern ausgeführt werden, dass danach ein konsistenter Zustand erhalten bleibt. Weitere Komplikationen können bei der parallelen Ausführung von langläufigen Operationen auftreten. Außerdem muss die Semantik der Undo-Funktion angepasst werden, da man in Zusammenarbeit mit anderen Personen meist nur die letzte eigene Änderung rückgängig machen möchte statt der global letzten Änderung.

In Abschnitt 5 werden Lösungen oben genannter Probleme präsentiert. Für das Nebenläufigkeitskontrollsystem zur Erhaltung eines konsistenten Zustandes wird die Jupiter-Architektur in Verbindung mit dem GOTO-Algorithmus verwendet. Zur Synchronisierung langläufiger Operationen wird ein Sperrmechanismus entwickelt, der Teilnehmer für eine bestimmte Zeit an der Erzeugung von Operationen hindert. Das Undo-Problem wird mit einer Aufzeichnung des Operationsverlaufs und dem GOTO-Algorithmus gelöst. Als schwierig erwies sich hierbei die Einbindung in Eclipse.

Um den Nutzen des Mehrschreibermodus, der in dieser Arbeit verbessert werden sollte, einschätzen zu können, wird in Abschnitt 6 eine empirische Studie vorgestellt, die zum Ziel hatte, Unterschiede in der Benutzung zwischen dem Mehr- und dem Einzelschreibermodus auszumachen. Durch die geringe Teilnehmerzahl können jedoch keine quantitativ belegbaren Aussagen gemacht werden. Daher wird empfohlen die Studie in größerem Umfang und unter Einbeziehung der Erkenntnisse der vorliegenden Untersuchung zu wiederholen.

Zuletzt wird in einem Ausblick erklärt, welche Probleme in Saros als nächstes gelöst werden sollten, um die Software entscheidend voranzubringen.

9 Literaturverzeichnis

Literatur

[AD92] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interact. Comput.*, 4(3):317–342, 1992.

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

[BGS02] Prashant Baheti, Edward F. Gehringer, and P. David Stotts. Exploring the efficacy of distributed pair programming. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 208–220, London, UK, 2002. Springer-Verlag.

[BSMM01] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 5th edition, 2001.

[CW01] Alistair Cockburn and Laurie Williams. *Extreme Programming examined*, chapter The costs and benefits of pair programming, pages 223–243. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[DB98] Allen H. Dutoit and Bernd Bruegge. Communication metrics for software development. *IEEE Transactions on Software Engineering*, 24(8):615–628, 1998.

[Dje06] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung Verteilter Paarprogrammierung. Diplomarbeit, Freie Universität Berlin, 2006.

[Doh09] Lisa Dohrmann. Erhebung von Benutzerfeedback aus der Nutzung eines Werkzeugs zur Verteilten Paarprogrammierung. Bachelorarbeit, Freie Universität Berlin, 2009.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, New York, NY, USA, 1989. ACM.

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

[Fag76] Michael Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fag86] Michael Fagan. Advances in software inspections. *IEEE Trans. Softw. Eng.*, 12(7):744–751, 1986.

[Fou] Eclipse Foundation. Entwicklerdokumentation der Eclipse API. `http://help.eclipse.org/ganymede/index.jsps`. (Online; abgerufen am 27. August 2009).

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, Jan 2004. (Online; abgerufen am 23. September 2009).

[Gus07] Björn Gustavs. Weiterentwicklung eines Eclipse-Plug-Ins zur Verteilten Paarprogrammierung. Studienarbeit, Freie Universität Berlin, 2007.

[Han04] Brian Hanks. Distributed pair programming: An empirical study. In *Extreme Programming and Agile Methods - XP/Agile Universe 2004*, pages 81–91, 2004.

[Jac09] Christoph Jacob. Weiterentwicklung eines Werkzeuges zur verteilten, kollaborativen Softwareentwicklung. Diplomarbeit, Freie Universität Berlin, 2009.

[LL90] J. Chris Lauwers and Keith A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 303–311, New York, NY, USA, 1990. ACM.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 4:308–320, December 1976.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, New York, NY, USA, 1995. ACM.

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

[Rie08] Oliver Rieger. Weiterentwicklung einer Eclipse Erweiterung zur Realisierung und Protokollierung Verteilter Paarprogrammierung im Hinblick auf Kollaboration und Kommunikation. Diplomarbeit, Freie Universität Berlin, 2008.

[Rin09] Marc Rintsch. Agile Weiterentwicklung eines Software-Werkzeuges zur verteilten, kollaborativen Programmierung in Echtzeit. Studienarbeit, Freie Universität Berlin, 2009.

[RW05] Jörg Rech and Sebastian Weber. Werkzeuge zur Ermittlung von Software-Produktmetriken und Qualitätsdefekten. Technical report, Fraunhofer IESE, 11 2005.

[SE98] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68, New York, NY, USA, 1998. ACM.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[Sun02] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, 2002.

[SWN+03] David Stotts, Laurie Williams, Nachiappan Nagappan, Prashant Baheti, Dennis Jen, and Anne Jackson. Virtual teaming: Experiments and experiences with distributed pair programming. Technical report, University of North Carolina, 2003.

[WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, July:19–25, 2000.

10 Abbildungsverzeichnis

Abbildungsverzeichnis

1 Roster View 6
2 Shared Session View 7
3 Warnung im inkonsistenten Zustand 7
4 Konfiguration von Saros 7
5 Annotationen in Saros 8
6 Entwicklung der Anzahl an monatlichen Downloads 12
7 Operationstransfer [SJZ+98] 14
8 zentralisierte Architektur 16
9 replizierte Architektur 17
10 Jupiter-Architektur 18
11 zweidimensionaler Zustandsraum 20
12 Undo-Mechanismus 24
13 Informatik-Fachsemester der Teilnehmer 37
14 Java-Kenntnisse der Teilnehmer 38
15 Erfahrungen der Teilnehmer in der Paarprogrammierung 38
16 Zeitbedarf im Mehr- und im Einzelschreibermodus 40
17 Korrektheit im Mehr- und im Einzelschreibermodus 41
18 zyklomatische Komplexität nach McCabe im Mehr- und im Einzelschreibermodus 41
19 Bearbeitungszeit der ersten Aufgabe gegen die der zweiten 42
20 Korrektheit der ersten Aufgabe gegen die der zweiten 43
21 zyklomatische Komplexität der ersten Aufgabe gegen die der zweiten 43
22 Vertrauen in die eigene Arbeit 44
23 Bearbeitungszeit gegen die aufgetretenen Inkonsistenzen 45
24 Korrektheit gegen die aufgetretenen Inkonsistenzen 46
25 zyklomatische Komplexität gegen die aufgetretenen Inkonsistenzen 46
26 Mängel an Saros 48
27 Mängel an der Sitzung 48
28 Vorteile der Sitzung 49
29 Neigung zum Mehrschreibermodus 50
30 Vorzug des Mehrschreibermodus 50
31 Entwicklung der Anzahl an monatlichen Downloads 51

A Anhang

A.1 Aufgaben

A.1.1 Aufgabe 1

Implementieren Sie für ein Pokerspiel folgende Interfaces:

```
/**
 * Eine Karte hat eine Farbe (z.B. Pik) und einen Wert (z.B. Dame).
 */
public interface Karte extends Comparable<Karte> {

  public enum Farbe {
        KREUZ, PIK, HERZ, KARO
  }

  public enum Wert {
        Ass, Koenig, Dame, Bube, Zehn, Neun, Acht, Sieben, Sechs, Fuenf,
            Vier, Drei, Zwei
  }

  public Farbe getFarbe();

  public Wert getWert();

  /**
   * @return eine negative Zahl, falls diese Karte einen kleineren Wert als
   *         die andere Karte hat, 0 bei Gleichheit und eine positive Zahl
   *         sonst. Beachten Sie, dass Sie die Farbe der Karten in dieser
   *         Methode komplett ignorieren können.
   */
  public int compareTo(Karte andereKarte);
}

/**
 * Ein Stapel besteht anfangs aus 52 Karten. Es kann gemischt werden und
    Karten
 * können an die Mitspieler verteilt werden, wodurch sich die Zahl der Karten
     im
 * Stapel entsprechend reduziert.
 */
public interface Stapel {

  /**
   * Gibt jedem Spieler eine bestimmte Anzahl an Karten. Benutzen Sie dazu
   * die Methode Spieler.addKarte(Karte karte). Achten Sie unbedingt
   * darauf, dass Sie richtig ausgeben, d.h. die Karten werden nur von oben
   * ausgegeben (heißt: ziehen Sie keine Karten aus der Mitte des Stapels)
   * und die Karten werden einzeln ausgegeben (heißt: erst wenn jeder
   * Spieler die erste Karte bekommen hat, wird die zweite ausgegeben).
   *
```

```
   * @param anzahl
   *            Zahl der auszugebenden Karten
   * @param spieler
   *            Liste der teilnehmenden Spieler
   */
  public void gibAus(int anzahl, List<Spieler> spieler);

  /**
   * Nach dem Mischen ist die Reihenfolge der Karten im Stapel verändert.
   * Achten Sie darauf, dass Sie gründlich mischen und insbesondere darauf,
   * dass Sie nicht nach einer vorhersagbaren Anzahl an Mischvorgängen
   * wieder die ursprüngliche Reihenfolge haben.
   */
  public void mische();
}

/**
 * Ein Blatt besteht aus fünf Karten. Die Zusammenstellung der Karten
 * entscheidet über den Wert des Blatts. Die enthaltenen Methoden sind Tester
    ,
 * die zurückgeben ob ein Blatt eine bestimmte Kombination darstellt.
 * Für jedes Blatt gibt es höchstens eine Methode, die true zurückgibt. So
 * soll z.B. im Fall von zwei Paaren nur isTwoPairs() und nicht auch noch
 * isOnePair() true zurückgeben.
 */
public interface Blatt {

        public boolean isOnePair();

        public boolean isTwoPairs();

        public boolean isThreeOfAKind();

        public boolean isStraight();

        public boolean isFlush();

        public boolean isFullHouse();

        public boolean isPoker();

        public boolean isStraigthFlush();

        public boolean isRoyalFlush();
}
```

Pokerblätter:

- **One Pair:** Das Blatt enthält ein Kartenpaar gleichen Wertes.
- **Two Pairs:** Zwei Paare gleichen Wertes.
- **3 Of A Kind:** Drei Karten gleichen Wertes.

- **Straight:** Ein Straight ist ein Blatt, bei dem alle Karten aufeinander folgen (Straße), jedoch nicht alle Karten von einer Farbe sind. Dabei gibt es keine Fortsetzung der Folge über das Ass hinaus.
  Bei einer Straße darf das Ass entweder als Ass oder als 1 gelten. Daraus ergibt sich die von ihrer Wertigkeit niedrigste Straße (A, 2, 3, 4, 5) und die höchste mit (10, J, Q, K, A)

- **Flush:** Fünf Karten einer Farbe.

- **Full House:** Ein Full House ist eine Kombination aus drei gleichen Karten und einem Paar, also einem Drilling und einem Paar.

- **Poker:** Vier Karten gleichen Wertes.

- **Straight Flush:** Folge von 5 Karten mit der gleichen Farbe. Es ist eine Kombination aus Flush und Straight. Somit sind alle Karten von derselben Farbe und aufeinander folgend.

- **Royal Flush:** Ein Royal Flush ist ein Straight Flush mit einem Ass als höchstem Kartenwert.

A.1.2 Aufgabe 2

Eine quadratische Gleichung ist eine Gleichung der Form:

a x^2 + b x + c = 0

Sie hat die Lösungen:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Beachten Sie, dass dabei auch komplexe Zahlen entstehen können.

Implementieren Sie eine Methode
```
public Collection<Complex> solve(String equation)
```
die eine als String gegebene quadratische Gleichung in eine geeignete Datenstruktur parst und sie dann anschließend löst. Entwickeln Sie dazu auch den Datentyp `Complex`, der eine komplexe Zahl repräsentiert und die Methoden
```
public double getReal()
public double getImag()
```
die den Realteil bzw. Imaginärteil der Zahl zurückgeben sollen.

Beachten Sie, dass die Gleichung auch folgendermaßen vorliegen darf:

b x + a x^2 + c = 0 oder auch c + a x^2 + bx = 0

Zur Vereinfachung dürfen Sie aber jede Gleichung zurückweisen, die rechts vom "=" keine 0 enthält. Leerzeichen dürfen Sie ignorieren.

A.2 Fragebögen

A.2.1 Fragebogen 1 (vor der Sitzung)

Welches Fach, in welchem Studiengang und in welchem Semester studieren Sie?

In wie vielen Semestern (dieses nicht mitgerechnet) schließen Sie voraussichtlich Ihr Studium ab?

Wie hoch schätzen Sie Ihre Programmierfähigkeiten in Java ein?

- o überragend
- o hoch
- o durchschnittlich
- o gering

Haben Sie Erfahrung mit Poker, d.h. kennen Sie die verschiedenen Pokerblätter (One Pair, Two Pair...)?

- o ja
- o nein
- o kenne sie, aber weiß sie nicht auswendig

Wie oft haben Sie bisher (ohne die heutige Sitzung) mit jemandem paarweise programmiert?

1. im letzten halben Jahr
2. insgesamt

Haben Sie auch schon Verteilte Paarprogrammierung praktiziert, d.h. Sie und Ihr Partner befanden sich an verschiedenen Orten? Falls ja, wie oft und welches Werkzeug wurde dazu benutzt?

Haben Sie mit Ihrem heutigen Partner schon paarweise programmiert? Falls ja, wie oft?

A.2.2 Fragebogen 2 (nach der ersten Sitzung)

Falls die Aufgabe im Einzelschreiberbetrieb gelöst wurde:
Welche Gründe haben dazu geführt, dass Sie der Host der Sitzung waren bzw. nicht waren?

Wie empfanden Sie die Sitzung? Was hat Sie gestört, was fanden Sie gut? Beurteilen Sie vor allem den Prozess und nicht nur das verwendete Werkzeug.

Gut fand ich:

1.

2.

3.

Gestört hat mich:

1.

2.

3.

Hat Ihnen die Sitzung Spaß gemacht?
Falls nein, woran lag es?

Wie viel Vertrauen haben Sie in die Korrektheit Ihrer Lösung?

- o Alles ist richtig.
- o Kann sein, dass uns der ein oder andere Fehler passiert ist.
- o Gut möglich, dass uns der ein oder andere Fehler passiert ist.
- o Das ist natürlich nur ein erster Versuch und wird so noch nicht funktionieren.

A.2.3 Fragebogen 3 (nach der zweiten Sitzung)

Falls die Aufgabe im Einzelschreiberbetrieb gelöst wurde:
Welche Gründe haben dazu geführt, dass Sie der Host der Sitzung waren bzw. nicht waren?

Wie empfanden Sie die Sitzung? Was hat Sie gestört, was fanden Sie gut? Beurteilen Sie vor allem den Prozess und nicht nur das verwendete Werkzeug.

Gut fand ich:

1.

2.

3.

Gestört hat mich:

1.

2.

3.

Hat Ihnen die Sitzung Spaß gemacht?
Falls nein, woran lag es?

Wie viel Vertrauen haben Sie in die Korrektheit Ihrer Lösung?

- o Alles ist richtig.
- o Kann sein, dass uns der ein oder andere Fehler passiert ist.
- o Gut möglich, dass uns der ein oder andere Fehler passiert ist.
- o Das ist natürlich nur ein erster Versuch und wird so noch nicht funktionieren.

Was hat Sie an Saros gestört, was hat Ihnen gefehlt?

Hat Ihnen die heutige Sitzung im Mehrschreiber- oder im Einzelschreiberbetrieb mehr Spaß gemacht? Warum?

Könnten Sie sich vorstellen auch zukünftig Saros zu benutzen? Falls ja, würden Sie den Einzelschreiber- oder den Mehrschreiberbetrieb vorziehen? Warum?

Wie empfanden Sie die Zusammenarbeit mit Ihrem Partner?

- o angenehm
- o anstrengend, weil ich lieber alleine arbeite
- o anstrengend, weil ich mit anderen Partnern besser zusammenarbeiten kann

A.3 Stichpunkte zur Einweisung in Saros

- Zweck von Saros: kollaborative Programmierung, Unterschied zu VCS hervorheben (Synchronität)
- Installation (nicht vorführen, nur kurze Erklärung)
- JID eingeben und verbinden (über Wizard, dann über Preferences, neue JID über Menü registrieren)
- Share Project (über Kontextmenü vom Projekt, nachträgliche Einladung über Kontextmenü des Nutzers)
- erste Texteingabe (Text wird bei Host eingegeben und ist bei Client zu sehen)
- Verfolgermodus, Doppelklick auf User
- Driver-Rolle setzen (Unterschied zwischen Einzelschreiber- und Mehrschreibermodus hervorheben)
- Selection Annotation, Contribution Annotation, Viewport (vorführen anhand erneuter Texteingabe)
- Inkonsistenzen
- Sitzung verlassen