Freie Universität Berlin

Diplomarbeit zur Erlangung des akademischen Grades Diplom-Informatiker

Weiterentwicklung eines Werkzeuges zur verteilten, kollaborativen Softwareentwicklung

Christoph Jacob
Matrikelnummer: 3450250
chjacob@inf.fu-berlin.de

Betreuer: Christopher Oezbek und Stephan Salinger

3. April 2009

Danksagung Ich möchte mich bei einigen Leuten bedanken, die mich im besonderen Maße bei meiner Diplomarbeit unterstützt haben.

Da sind zunächst meine Betreuer Stephan Salinger und Christopher Oezbek, die mir zu jeder Zeit mit Rat und Tat zur Seite standen.

Die Arbeit wäre aber auch nicht ohne die große Anzahl von Freiwilligen möglich, die sich als Tester und Teilnehmer am empirischen Experiment zur Verfügung gestellt haben. An dieser Stelle möchte ich Moritz Blöcker, Max Höflich und Andreas Rebenstorf nennen, die bei all diesen Terminen Zeit für mich gefunden haben.

Ein ganz besonderer Dank geht an Inés Raurich-León, die mich in den letzten Wochen unbeschreiblich unterstützt hat.

Zum Schluss möchte ich mich bei meinen Eltern bedanken, die mich während des Studiums immer unterstützt haben.

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

Berlin, den 3. April 2009
Christoph Jacob

Inhaltsverzeichnis

1 Einleitung

Saros ist ein Werkzeug zur verteilten, kollaborativen Softwareentwicklung. Es ermöglicht, dass mehrere Entwickler eines standortverteilten (virtuellen) Teams gemeinsam den Quelltext einer Software bearbeiten. Hierbei wird nicht nur die Praktik der verteilten Paarprogrammierung unterstützt, sondern durch die Bereitstellung eines kollaborativen Echtzeit-Gruppeneditors ein gleichzeitiges Schreiben von mehr als zwei Entwicklern ermöglicht.

In der vorliegenden Diplomarbeit wurde die Funktionalität von Saros nicht erweitert, sondern lediglich einer Qualitätssicherung unterzogen. Dies war notwendig geworden, da die neu hinzugekommene Mehrschreiberfunktionalität teilweise fehlerhaft implementiert wurde und notwendig gewordene Veränderungen am bestehenden Quelltext nicht durchgeführt worden sind.

Als Basis für die Qualitätssicherung wurde zunächst eine erneute Anforderungsbestimmung durchgeführt und anschließend analysiert, ob die in Saros verwendeten Technologien den Anforderungen genügen. Durch Funktionstests, Strukturtests und Code-Durchsichten wurden das Versagen der Software und die zugrunde liegenden Defekte der Software bestimmt. In der Implementierungsphase meiner Arbeit wurden die schwerwiegendsten Defekte behoben. Durch eine anschließend durchgeführte Mini-Fallstudie wurde die Software einer empirischen Bewertung unterzogen und dadurch erste Beiträge für die Validierung der Software erbracht.

Nachdem ich in Kapitel 2 auf die Grundlagen der verteilten sowie der nicht verteilten, kollaborativen Softwareentwicklung eingehe, stelle ich in Kapitel 3 die Entstehungsgeschichte von Saros dar. In Kapitel 4 werde ich die in Saros verwendeten Technologien kurz vorstellen.

Ein nicht unwesentlicher Teil meiner Arbeit beschäftigt sich mit den an Saros gestellten Anforderungen. In Kapitel 5 werde ich durch die bestimmten Anforderungen eine Produktdefinition liefern und anschließend eine Anforderungsanalyse durchführen. Des Weiteren gehe ich auf die durch eine Kooperation mit einem Wirtschaftsunternehmen entstandenen Änderungen an den Anforderungen ein.

Die in der Implementierungsphase durchgeführten Arbeiten werde ich in Kapitel 7 dokumentieren. Hierzu gehört neben der Behebung von Defekten, die Neuimplementierung von Direktverbindungen zwischen den Teilnehmern sowie der Realisierung einer erweiterten Konsistenzsicherung. Ich werde in Kapitel 8 ebenfalls auf die noch offenen Probleme eingehen und erste Lösungsansätze angeben.

In Kapitel 9 werde ich die gefundenen Defekte nach Fehlern der Entwickler klassifizieren, indem ich für die häufigsten Fehler Kategorien aufstelle. Außerdem werde ich einige Probleme im Softwareprozess bei der Entwicklung an Saros nennen.

Die Ergebnisse der von mir durchgeführte Mini-Fallstudie stelle ich in Kapitel 10 dar und gebe in Kapitel 11 abschließend einen Ausblick, in welche Richtung sich das Projekt in Zukunft entwicklen könnte.

2 Grundlagen

Bei der *Paarprogrammierung* (Pair Programming oder Collaborative Programming) arbeiten zwei Programmierer zusammen an einem Computer, um gemeinsam eine Software zu entwicklen. Beide sind zu jeder Zeit gleichermaßen am Prozess beteiligt. Im Idealfall agieren sie wie ein kohärent intelligenter Organismus, der eine bestimmte Absicht verfolgt und für jeden Aspekt des Artefakts verantwortlich ist [1].

Dabei hat einer der Softwareentwickler (*Driver*) die Kontrolle über die Eingabegeräte und schreibt den Entwurf oder Quelltext. Der andere Entwickler (*Observer*) beobachtet kontinuierlich und aktiv die Arbeit des Drivers, schaut nach Defekten aus, denkt über Alternativen nach und ermittelt strategische Implikationen der aktuellen Lösung. Die Rollen des Drivers und Observers können dabei jederzeit gewechselt werden.

Eines der Hauptmerkmale der Paarprogrammierung ist eine Art von kontinuierlichen Review [2]. Es existieren einige empirische Untersuchungen, die der Paarprogrammierung Vorteile gegenüber der individuellen Programmierung bescheinigen [1] [2] [3]. Zu den möglichen Vorteilen gehören neben einer höheren Qualität der Lösungen, einer Zeitersparnis und einen höheren Grad von Zufriedenheit der Entwickler, [3] auch ein erhöhter Wissenstransfer und verbesserte Kommunikation innerhalb eines Teams [2].

Oft ist die Praktik der Paarprogrammierung aufgrund von räumlicher Distanz nicht möglich. So sind in Zeiten von global agierenden Konzernen und Unternehmen Teams nicht selten auf der ganzen Welt verteilt. Man nennt eine solche Gruppe von Personen, die zusammen an einem gemeinsamen Ziel über Grenzen von Zeit, Entfernung, Kulturen und Organisationen hinweg arbeiten, auch oft *virtuelle Teams* [4].

In solchen Fällen von räumlicher Distanz kann die Praktik der *verteilten Paarprogrammierung* angewendet werden, bei der zwei Mitglieder eines Teams von verschieden Arbeitsplätzen aus zeitgleich und gemeinsam an einem Entwurf oder Quelltext arbeiten [5].

Einige empirische Untersuchungen [5] [6] [7] zeigen, dass die Praktik der verteilten Paarprogrammierung durchführbar und die resultierende Software bezüglich Produktivität und Qualität vergleichbar mit der aus nicht verteilter Paarprogrammierung ist [5]. Gegenüber von individuellen Entwicklern in verteilten Teams, die ihre Arbeit anschließend integrieren müssen, bleibt bei der verteilten Paarprogrammierung einige der oben genannten Vorteile der Paarprogrammierung erhalten [6].

Die Schwierigkeit bei der verteilten Paarprogrammierung ist, dass die Kommunikation aufgrund der räumlichen Trennung erschwert wird. Die Herausforderung bei der Konzeption und Umsetzung von Werkzeugen für verteile Paarprogrammierung ist es deshalb, ein verteiltes Arbeiten an den gleichen Artefakten zu ermöglichen, ohne die Vorteile, die durch die direkte Interaktion zwischen den Entwicklern entstehen, zu verlieren. Dies versuchen die Werkzeuge durch Übertragung von Audio und Video, Textnachrichten sowie Informationen über die Aktivitäten des Drivers (*Awareness-Informationen*) innerhalb des Editors zu kompensieren.

Es existieren zwei verschiedene Ansätze, um eine zeitgleiche Zusammenarbeit von unterschiedlichen Arbeitsplätzen zu realisieren. Bei dem Ansatz des *Desktop Sharings* wird die

Arbeitsfläche (Desktop) des einen Computers auf dem Bildschirm eines entfernten Computers übertragen. Der entfernte Benutzer kann die Kontrolle des anderen Computers übernehmen, indem er die Eingabegeräte seines Computers verwendet.

Bei dem Ansatz der *Collaboration Awareness* ist die Mehrbenutzerunterstützung bereits in die Werkzeuge integriert. Beispiele für solche Werkzeuge sind *Echtzeit-Gruppeneditoren*, die ein gleichzeitiges Arbeiten an Dokumenten ermöglichen. *Saros* ist ebenfalls ein solches Werkzeug, dass die integrierte Entwicklungsumgebung *Eclipse* um die Möglichkeit der verteilen Paarprogrammierung erweitert.

In den vorangegangenen Arbeiten [8] an Saros unterschieden wir zwischen verschiedenen Ebenen der Parallelität. Die *Parallelität auf Programmebene* bietet dem Observer lediglich die Möglichkeit, neben dem Werkzeug für die verteilte Paarprogrammierung, andere Programme auszuführen und zu diesen zu wechseln. Von *Parallelität auf Sichtebene* sprechen wir, wenn der Observer vom Aufmerksamkeitsbereich des Drivers abweichen und eigenständig andere Dateien des Projektes anschauen kann. Die *Parallelität auf Schreibebene* bezeichnet die Möglichkeit, dass mehrere Driver gleichzeitig an den gemeinsamen Dateien schreiben können.

Saros ist ein Werkzeug für die verteilte Paarprogrammierung, welches die Parallelität auf Schreibebene ermöglicht. Da mehrere Driver zur gleichen Zeit existieren können, handelt es sich hierbei nicht mehr um verteile Paarprogrammierung im engeren Sinne, weshalb wir stattdessen von *verteilter, kollaborativer Softwareentwicklung* als Anwendungsgebiet reden.

3 Entstehungsgeschichte von Saros

In diesem Kapitel gebe ich kurz die Entstehungsgeschichte von Saros wieder. Der Schwerpunkt dieser Ausführungen liegt auf den Veränderungen, die es hinsichtlich des Verwendungszweckes und damit einhergehend der Anforderungen gab.

Die Geburt von Saros liegt in der Diplomarbeit „Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung“ (DPPI) [9] von Riad Djemili. Das Ziel war es, ein Werkzeug für die verteilte Paarprogrammierung zu entwickeln. Saros sollte zwei Softwareentwicklern ermöglichen im Paar zu programmieren ohne am gleichen Ort sein zu müssen.

Saros konnte in dieser ersten Version nur verwendet werden, um Paarprogrammierung im klassischen Sinne durchzuführen, das heißt, einer von beiden Programmierern nahm die Rolle des Drivers, der andere die des Observers ein. Die Rollen konnten zu jeder Zeit getauscht werden. Der Entwickler, der die Rolle des Observers einnahm, konnte keine Veränderungen an dem Quelltext vornehmen, auch nicht um kleinere Fehler des anderen zu korrigieren.

In seiner Diplomarbeit [9] identifizierte Djemili zwei verschiedene Ansätze um ein Werkzeug zur verteilten Paarprogrammierung zu realisieren. Bei dem Ansatz des *virtuellen Zugriffs* liegen nur bei einem Entwickler die Dateien tatsächlich vor. Greift der andere Entwickler auf eine gemeinsame Datei zu, so muss vorher der Inhalt der Datei übertragen werden. Bei dem *Replikationsansatz* wird stattdessen vor dem Starten der Programmiersitzung sichergestellt, dass alle auf den gleichen Datenbestand arbeiten und es werden nur die Änderungsinformationen übertragen.

Als zentrale Entwurfsentscheidung entschied Djemili sich für den Replikationsansatz, denn dieser ermöglicht die Übersetzbarkeit des Quelltextes sowie die Ausführung bei jedem Teilnehmer der Programmiersitzung.

In seiner Arbeit führte er ebenfalls den Begriff *Parallelität* ein und unterschied zwischen Parallelität auf Programm-, Sicht- und Schreib-Ebene (siehe Kapitel 2).

In der ersten Version von Saros wurde sich auf die Parallelität auf Sichtebene beschränkt, das heißt, es gab zu jeder Zeit nur einen Driver, der schreibend auf die Dateien zugreifen konnte. Des Weiteren wurde aus Zeitgründen zunächst auf die Implementierung der Möglichkeit von mehr als einem Observer verzichtet.

Die Möglichkeit von mehreren Observern in einer Programmiersitzung wurde von Björn Gustavs in seiner Studienarbeit „Weiterentwicklung eines Eclipse-Plugins zur Verteilten Paarprogrammierung“ (DPPII) [10] nachträglich implementiert.

In der darauf aufbauenden Diplomarbeit „Weiterentwicklung einer Eclipse Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung im Hinblick auf Kollaboration und Kommunikation“ (DPPIII) [11] wurde Saros von Oliver Rieger hinsichtlich der Parallelität auf Schreibebene erweitert. Neben dieser Unterstützung von mehreren Drivern konnte die Dateiübertragung nun auch direkt zwischen den Teilnehmern geschehen. Da die Übertragung nun nicht mehr über das langsamere Kommunikationsnetzwerk geschah, brachte dies einen deutlichen Geschwindigkeitsgewinn.

4 Verwendete Technologien

In diesem Kapitel werde ich auf die wichtigsten in Saros verwendeten Technologien eingehen. Der Schwerpunkt liegt hier auf der Ablaufplattform *Eclipse*, dem Nachrichtenprotokoll *XMPP* sowie der Nebenläufigkeitskontrolle *Jupiter*.

4.1 Eclipse als Ablaufplattform

Den meisten ist Eclipse als mächtige integrierte Entwicklungsumgebung (IDE) für Java ein Begriff. Sie bietet den Entwicklern sehr weit gehende Unterstützung bei deren Programmiertätigkeiten von einer automatischen Codevervollständigung, einem intelligenten Korrektur-Assistenten bis hin zu weitreichenden Refaktorisierungen. Was aber viele nicht wissen ist, dass Eclipse viel mehr als nur eine Entwicklungsumgebung für Java ist. Durch die konsequente Plugin-Architektur kann Eclipse nicht nur durch eine Vielzahl von existierenden Plugins für die Softwareentwicklung in anderen Programmiersprachen verwendet werden, es können sogar auf Basis der in der Version 3 eingeführten *Rich Client Platform (RCP)* generische Anwendungen für andere Kategorien von Anwendungen implementiert werden.

4.1.1 Plugin-Architektur

Seit der Version 3 verwendet Eclipse für den Ablauf der Plugins die offene Ablaufplattform *OSGi*, dessen Komponentenmodell ein dynamisches Hinzufügen und Entfernen von Plugins sowie ein konfliktfreies Ausführen von unterschiedlichen Versionen des gleichen Plugins erlaubt [12]. OSGi-konforme Plugins, auch *Bundles* genannt, können nicht nur in Eclipse ablaufen, sondern auf jeder anderen OSGi-Implementierung. Der den OSGi-Standard implementierende Ablaufkern von Eclipse heißt *Equinox* und erweiterte den Standard durch einige zusätzliche Techniken wie das Nachladen eines Plugins, wenn dieses das erste Mal benötigt wird (*LazyStart*-Technik), was einen schnelleren Start der Ablaufplattform ermöglicht. Die LazyStart-Technik wurde in der Version 4.1 [13] in den OSGi-Standard aufgenommen.

Ein Kernkonzept der Plugin-Architektur von Eclipse sind Erweiterungspunkte, sogenannte *Extensions Points*. Plugins stellen nicht nur Funktionalität bereit indem sie sich für bestimmte Erweiterungspunkte anmelden, sie können auch neue Erweiterungspunkte zur Verfügung stellen, die wiederum andere Plugins zur Erweiterung verwenden können. Eclipse schaut zur Laufzeit welche Plugins einen bestimmten Erweiterungspunkt zur Erweiterung nutzen und welche zusätzlichen Erweiterungspunkte die Plugins zur Verfügung stellen.

Es besteht die Möglichkeit mehrere Plugins zu einer Auslieferungseinheit zusammenzufassen, sogenannten *Features*. Das Ausliefern eines Eclipse-Plugins, beziehungsweise eines Features, an den Kunden kann auf unterschiedliche Weisen geschehen. Die komfortabelste Möglichkeit ist die Bereitstellung einer *Update-Site*, von der sich der Kunde das Plugin über das Internet herunterladen und installieren kann. Dem Kunden wird lediglich eine URL der Update-Site bekannt gegeben. Diese Art der Auslieferung hat ebenfalls den Vorteil, dass von Eclipse die Abhängigkeiten zwischen den Plugins überprüft und gegebenenfalls benötigte Plugins nachgeladen werden. Des Weiteren kann Eclipse zu jeder Zeit überprüfen, ob eine neuere Version des Plugins existiert, und das Plugin gegebenenfalls aktualisieren.

4.1.2 Eclipse als IDE erweitern

In diesem Kapitel stelle ich kurz die zentralen Komponenten und Erweiterungspunkte vor, die ein Plugin nutzen kann um Eclipse zu erweitern.

Ablaufsteuerung In dem Paket *org.eclipse.core.runtime* befindet sich der Kern von Eclipse, der nur die Funktionalität besitzt, Plugins zum Laufen zu bringen. Die zentralen Klassen dieser Ablaufsteuerung sind *Platform* und *Plugin*. Von der Klasse *Platform* kann kein Exemplar erzeugt werden, sie beinhaltet ausschließlich statische Methoden um Informationen über die Ablaufumgebung zu erhalten. Die abstrakte Klasse *Plugin* wird von dem eigenen Plugin erweitert. Möchte man ein Plugin mit einer Benutzeroberfläche entwickeln, sollte man die abstrakte Klasse *AbstractUIPlugin* erweitern, welche wiederum eine Unterklasse von *Plugin* ist. Von dieser Klasse kann es zu jeder Zeit nur ein Exemplar geben, das im Konstruktor erstellt wird und auf welches man mittels der Methode *getDefault* zugreifen kann. Dies stellt das Singleton Entwurfsmuster dar [14].

Workbench Das Wurzelobjekt der gesamten Benutzeroberfläche von Eclipse ist das Exemplar vom Typ *IWorkbench*, das man mittels der Methode *getWorkbench* auf der Klasse *PlatformUI* erhält. Die Workbench besteht aus einem oder mehreren Workbench-Fenstern (vom Typ *IWorkbenchWindow*). Normalerweise wird Eclipse mit nur einem Workbench-Fenster gestartet, man kann aber zum Beispiel für jede Perspektive (eine bestimmte Anordung von Workbench-Komponenten) ein zusätzliches Fenster öffnen, so dass eine Workbench aus mehreren Fenstern besteht. Auf das aktive Fenster der Workbench kann man mittels der Methode *getActiveWorkbenchWindow* auf einem Exemplar von *IWorbench* zugreifen. Jedes Fenster wiederum kann aus mehreren *WorkbenchPages* bestehen, diese Möglichkeit wird aber derzeit von Eclipse nicht verwendet.

Editoren Die zentralen Komponenten der Workbench sind die Editoren. Es kann zu jeder Zeit immer nur ein Editor sichtbar (aktiviert) sein. Jeder Workbench Editor hat die abstrakte Klasse *EditorPart* als Basisklasse und eine Klasse vom Typ *IEditorInput* als Datenquelle. Die wichtigste Art von Editoren sind die vom Typ *AbstractTextEditor*, denn diese Klasse ist die Basis für alle textbasierten Editoren. Auf die Editoren kann man mittels der aktiven *WorkbenchPage* zugreifen, auf die man wiederum mittels der Methode *getActivePage* auf dem aktiven Workbench-Fenster zugreifen kann.

Views Im Gegensatz zu den Editoren kann es zu jedem Zeitpunkt mehrere sichtbare Views geben, die den Zustand des aktiven Editors oder der Workbench anzeigen. Sie werden ausschließlich von der Workbench instantiiert. Eclipse bringt bereits sehr viele Views mit, die verwendet oder erweitert werden können.

Workspace Die Elemente des Workspaces sind alle vom Basistyp *IResource*. Eine Ressource ist entweder eine Datei (*IFile*) oder ein Container (*IContainer*). Es gibt drei verschiedene Arten von Container: Projekte (*IProject*), Verzeichnisse (*IFolder*) und das Wurzelverzeichnis des

Workspaces (*IWorkspaceRoot*). Dies stellt das Kompositum Entwurfsmuster mit speziellen Komposita für Eclipse Projekte und dem Wurzelverzeichnis dar [14].

Eine der wichtigsten Methoden, die jede Ressource bereitstellen muss, ist die *accept* Methode. Diese bekommt eine Besucherklasse (*IVisitor*) übergeben, auf der die *visit* Methode aufgerufen wird. Liefert diese den Wert *true* zurück, werden alle Kinder der Komponente weiter traversiert, indem auf diesen wiederum die *accept* Methode aufgerufen wird. Dies stellt das Besucher Entwurfsmuster dar [14].

4.1.3 Saros als Eclipse-Plugin

Saros ist als Erweiterung von Eclipse als IDE konzipiert, die weitestgehend unabhängig von der verwendeten Programmiersprache funktioniert. Dies wird erreicht, indem die Verwendung von Plugins für spezielle Programmiersprachen wie das Plugin *Java Developer Toolkit (JDT)* oder das Plugin *C/C++ Developer Tooling (CDT)* so weit wie möglich vermieden wird und anstelle dessen Funktionalitäten von allgemeineren Plugins genutzt wird. An den Stellen, an denen es sich partout nicht vermeiden lässt, wird über eine Fassade auf die benötigte Funktionalität zugegriffen und es wird zur Laufzeit überprüft, ob das entsprechende Plugin, beziehungsweise das OSGi-Bundle, zur Verfügung steht. Damit nicht für alle unterstützen Programmiersprachen die Plugins durch die Überprüfung während der Installation von Saros nachinstalliert werden, wurden die unterstützen Plugins als optionale Abhängigkeit deklariert.

Neben Einstellungsmöglichkeiten in den Preferences und Einträgen in Menüs werden folgende Erweiterungspunkte von Saros verwendet, um die Funktionalität von Eclipse zu erweitern:

Views Durch Saros werden drei Views bereitgestellt. Im sogenannten *Roster* werden die Kontakte des Benutzers angezeigt. Zu jedem Kontakt wird angezeigt, ob dieser gerade zur Verfügung steht oder nicht. Des Weiteren wird während einer Programmiersitzung der Status der Direktverbindung für die Dateiübertragung zu diesem Benutzer angezeigt. Siehe Kapitel 7.3 für nähere Informationen über die Dateiübertragung per Direktverbindung.

Im View *Shared Project Session* werden alle Teilnehmer der Programmiersitzung angezeigt und welche Rolle (Driver oder Observer) sie besitzen. Der einladende Teilnehmer hat hier zusätzlich die Möglichkeit den Teilnehmern die Driver Rolle, also technisch gesehen die Schreibrechte, zu geben und wieder zu entziehen.

In dem View *Chat* kann man den anderen Teilnehmer der Programmiersitzung Textnachrichten zukommen lassen. Der Chat funktioniert wie ein Gesprächskanal (Channel) im Internet Relay Chat, das heißt eine Nachricht, die der Benutzer sendet, wird an alle anderen Teilnehmer der Sitzung geschickt. Dies wird durch die Erweiterung XEP-0045 (Multi User Chat) [15] des Nachrichtenprotokolls XMPP realisiert. In Kapitel 4.2 wird weiter auf die Verwendung von XMPP als Nachrichtenprotokoll eingegangen.

Dekorationen Der Begriff *Dekorationen* bezeichnet in Eclipse die Möglichkeit, Elemente der grafischen Benutzerschnittstelle mit zusätzlichen Auszeichnungen zu versehen. Hiervon wird in Saros an vielen Stellen Gebrauch gemacht. So wird diese Möglichkeit zum Beispiel genutzt, um dem Benutzer anzuzeigen, welches Projekt gerade innerhalb der laufenden Programmiersitzung bearbeitet wird. Des Weiteren wird den Observers angezeigt, welche Dateien der beobachtete Driver gerade geöffnet hat und in welcher er sich gerade befindet. Diese Dekorationen werden in allen Views verwendet, in denen Eclipse Projekte und Dateien anzeigt, also hauptsächlich in dem *Package Explorer* sowie in dem *Navigator* von Eclipse.

Annotationen In Saros werden einige *Annotationen* im Editor von Eclipse verwendet, um dem Benutzer Awareness-Informationen über die Aktivitäten der anderen Teilnehmer der Programmiersitzung zu geben. Die von den Drivern durchgeführten Veränderungen werden mit der global eindeutigen Farbe für den Driver hinterlegt und die Position deren Cursor in dieser Farbe dargestellt. Des Weiteren wird vom beobachteten Driver der Ausschnitt des Dokumentes hervor gehoben, der für diesen gerade sichtbar ist. Alle Teilnehmer der Sitzung besitzen die Möglichkeit, durch Markieren von Text innerhalb des Editors, Textpassagen für die anderen Teilnehmern hervorzuheben. Dies wird realisiert, in dem die Informationen über die Selektierung zu den anderen Teilnehmer übermittelt und die Selektion als Annotation im entsprechenden Editor dargestellt wird.

ModelProvider für Ressorcen Zu guter Letzt wird der Erweiterungspunkt für eigene ModelProvider für Ressourcen innerhalb des Workspaces verwendet, um Observer einer Programmiersitzung daran zu hindern, Veränderungen am Dateibaum vorzunehmen.

Bemerkung über Dekorationen und Annotationen Die Darstellung von Awareness-Informationen ist noch nicht für die Möglichkeit von mehreren Drivern zur gleichen Zeit angepasst worden. So besteht weder die Möglichkeit, den zu beobachtenden Driver auszuwählen, noch ist sichergestellt, dass die angezeigten Informationen nur von einem Driver stammen. Für die Art der Darstellung müssen zum Teil neue Konzepte erarbeitet werden. Der Grund, warum diese notwendigen Veränderungen von mir nicht durchgeführt worden sind, ist zeitlicher Natur und liegt in den veränderten Anforderungen beziehungsweise in deren Priorisierung begründet. In Kapitel 5.3 gehe ich näher auf diesen Umstand ein.

4.2 XMPP als Nachrichtenprotokoll

XMPP steht für *Extensible Messaging and Presence Protocol* und ist eine Internetprotokollfamilie, die hauptsächlich im Bereich von Instant Messaging eingesetzt wird. Die in XMPP verwendeten Protokolle gingen hauptsächlich aus dem Jabber-Protokoll hervor, dass 1998 von Jeremie Miller ins Leben gerufen worden ist. Das Ziel des Jabber Projektes war von Anfang an die Unterstützung von offenen Standards und der Interoperabilität im Bereich von Echtzeitkommunikation [16]. Im Jahre 2004 wurde XMPP von der Internet Engineering Task Force (IETF) als Standard veröffentlicht [17].

Riad Djemili wählte im Rahmen seiner Diplomarbeit [9] XMPP als Nachrichtenprotokoll aus. Dies begründete er unter anderem mit der einfachen Erweiterbarkeit und der Tatsache, dass XMPP, beziehungsweise die Softwarebibliotheken, die diesen Standard implementieren, bereits einige von Saros benötigte Funktionen bieten. Hierzu gehören, neben der Übertragung von Nachrichten, die Anwesenheitsvermittlung und die Möglichkeit der Dateiübertragung. Die Verwendung eines etablierten Standards erhöht des Weiteren die Beständigkeit der Software. Die Existenz eines ausgereiften Standardisierungsprozess bietet einem zudem in Zukunft die Möglichkeit, die in Saros verwendeten Erweiterungen von XMPP als Standard für verteilte Paarprogrammierung vorzuschlagen.

Weiterhin gibt es einige für Saros interessante bestehende Erweiterungen von XMPP, wie zum Beispiel die Möglichkeit des Multi-User Chats (XEP-0045) oder die Möglichkeit, eine Direktverbindung zwischen zwei Kommunikationsteilnehmern mithilfe des Jabber-Serververbundes herzustellen (XEP-0166, auch *Jingle* genannt). Eine vollständige Liste der Erweiterungen von XMPP findet man auf den Seiten der *XMPP Standards Foundation* [15], die für den Standardisierungsprozess der Erweiterungen verantwortlich sind. Die Erweiterungen sind teilweise noch in einem Entwurfsstadium oder es existieren für diese noch keine Implementierungen.

Die Begriffe XMPP und *Jabber* werden oft synonym verwendet. Ich verwende im Folgenden XMPP als Begriff für das Nachrichtenprotokoll und Jabber für die Anwendung von XMPP für die Echtzeitkommunikation.

Die Architektur von Jabber beziehungsweise XMPP besteht aus einem dezentralen Serververbund. Es gibt keine zentrale Verwaltung der Jabber-Server, womit es jedem frei steht einen eigenen Jabber-Server in den Serververbund einzubringen. Nachrichten werden grundsätzlich über die Server verschickt, eine direkte Übertragung der Nachrichten zwischen zwei Teilnehmern wurde nicht vorgesehen. Das Nachrichtenprotokoll XMPP regelt das Vermitteln von XML-Nachrichten (Routing). Das Protokoll befindet sich im TCP/IP Protokollstapel in der Anwendungsschicht und setzt auf TCP als Transportprotokoll auf.

Der eindeutige Bezeichner, mit denen Benutzer identifiziert werden, setzt sich aus einem Benutzernamen und dem Domänennamen des Servers zusammen[1]. Den XML-Nachrichten können selbst definierte XML-Elemente angefügt werden, womit eine sehr einfache Erweiterbarkeit der Nachrichten gegeben ist. Von dieser Möglichkeit wird in Saros oft Gebrauch gemacht, um eigene strukturierte Nachrichten zwischen den Teilnehmern einer Programmiersitzung zu verschicken.

Saros verwendet die Java-Bibliothek *Smack* [18], die von der Firma Jive Software als Open Source entwickelt wird. Während meiner Arbeiten führte ich die Migration auf die Version 3.1.0 durch. Ein wesentlicher Grund für diesen Schritt war, dass ich nicht eindeutig erkennen konnte, welche genaue Version der Bibliothek in Saros vorher eingesetzt wurde. Es ergab sich der Eindruck, dass nicht ein offizielles Release, sondern eine Zwischenversion, die aus dem Repository des Projektes selbst erstellt worden ist, verwendet wurde. Die Stabilität der Zwischenversion konnte ich schlecht einschätzen. Des Weiteren ist es für das Erstellen von Fehlerberichten unabdingbar zu wissen, auf welche genaue Version der Software sich die Fehlerberichte beziehen sollen.

[1] zum Beispiel christoph.jacob@jabber.org

Smack implementiert die weiter oben schon erwähnten Erweiterungen von XMPP für die Verwendung eines Multi-User Chats und die Möglichkeit, eine Direktverbindung zwischen zwei Kommunikationsteilnehmern aufzubauen. Saros verwendet den Multi-User Chat um eine Gruppenkommunikation zu realisieren und die Direktverbindungen um Dateien zwischen den Teilnehmern zu übertragen.

Ob das Nachrichtenprotokoll XMPP und die Softwarebibliothek Smack den Anforderungen von Saros genügen, wird in Kapitel 5.2.2 diskutiert.

4.3 Jupiter Algorithmus als Nebenläufigkeitskontrolle

Im Rahmen der Diplomarbeit von Herrn Rieger [9] wurde Saros um die Möglichkeit des parallelen Schreibens von mehreren Drivern innerhalb einer Programmiersitzung erweitert. Um dabei weiterhin die Konsistenz der Kopien bei allen Teilnehmern zu gewährleisten, wurde die Verwendung einer Nebenläufigkeitskontrolle notwendig.

Unter einer Nebenläufigkeitskontrolle versteht man die Koordination der Aktionen verschiedener Prozesse, die gleichzeitig auf gemeinsame Daten zugreifen und sich damit potenziell behindern [19].

Pessimistische Sperrverfahren, wie sie typischerweise in Datenbanken verwendet werden, kamen für den Einsatz in Saros nicht infrage, da die durch die Berechtigungsvergabe entstehende Verzögerung für einen Echtzeit-Gruppeneditor nicht akzeptabel ist.

Auch optimistische Verfahren, wie sie zum Beispiel in Versionsmanagementsystemen zum Einsatz kommen, sind für den Einsatz in Echtzeit-Gruppeneditoren ungeeignet, da die Erkennung von Konflikten zu aufwendig und die automatisierte Konfliktauflösung nicht in allen Fällen gegeben ist.

In Echtzeit-Gruppeneditoren kommt am meisten ein Verfahren namens *Operational Transformation (OT)* zum Einsatz. Dieses ermöglicht ein verzögerungsfreies Editieren von allen Teilen eines Dokumentes und kommt ohne jede Art von Sperren aus. Dadurch ist die Zeit, in der lokale Veränderungen durchgeführt werden können, nicht abhängig von Latenzzeiten der Netzwerkkommunikation, wodurch dieses Verfahren sich ideal für verteilte Systeme eignet.

Operational Transformation wurde erstmals 1989 von C.A. Ellis und S.J. Gibbs als Verfahren für eine Nebenläufigkeitskontrolle von Groupware Systemen vorgeschlagen [20]. Sie lieferten mit dem *dOPT* Algorithmus, der in dem *GROVE* (Group Outline Viewing Editor) System implementiert worden ist, den ersten Algorithmus, der nach diesem Verfahren arbeitete. Zwar konnte der dOPT Algorithmus nicht für alle möglichen Szenarien die Konsistenzerhaltung sicherstellen [20] (dOPT-Puzzle), dennoch diente dieser als Basis für viele folgende OT-Algorithmen.

Die Funktionsweise des Operational Transformation Verfahrens sieht im Allgemeinen wie folgt aus:

1. sofortiges Ausführen der Operation auf der lokalen Seite
2. Senden der Operation an alle anderen Seiten (Broadcast)

3. Empfangen der Operation von den anderen Seiten
4. Ausführen der gegebenenfalls transformierten Operation von den anderen Seiten

Man kann jeden Operational Transformation Algorithmus in zwei verschiedene Komponenten unterteilen [21]:

1. Der *Integrationsalgorithmus* ist für das Empfangen, Versenden und Ausführen von Operationen verantwortlich. Falls nötig ruft dieser die Transformationsfunktion auf. Dieser Teil des Algorithmus ist unabhängig von der Art der Daten, die kontrolliert werden.
2. Die *Transformationsfunktion* ist für das Zusammenführen von zwei in Konflikt stehenden Operationen verantwortlich. Diese ist sehr stark von der Art der kontrollierten Daten abhängig.

Eine gute Übersicht über verschiedene Ansätze und Algorithmen, die nach diesem Verfahren arbeiten, geben Ellis und Sun [22].

Im Rahmen seiner Diplomarbeit [11] hat sich Rieger für den Jupiter Algorithmus [23] als Nebenläufigkeitskontrolle entschieden, der nach dem Operational Transformation Verfahren arbeitet. Als Entscheidungskriterien nannte er die einfache Umsetzung des Algorithmus sowie ein geringen Kommunikationsaufwand. Jupiter basiert auf den dOPT Algorithmus für 2 Seiten und erweitert diesen auf n mögliche Seiten.

Das Besondere an dem Jupiter Algorithmus ist, dass es neben den n Seiten (Clients) einen zentralen Server gibt und die Kommunikation ausschließlich zwischen den Clients und dem Server stattfindet. Für die Synchronisierung zwischen jedem Client und dem Server wird unabhängig voneinander das 2-Wege Protokoll vom dOPT Algorithmus verwendet. Diese Architektur von einem zentralen Server und mehreren Clients bildet eine sternförmige Topologie, siehe Abbildung 1.

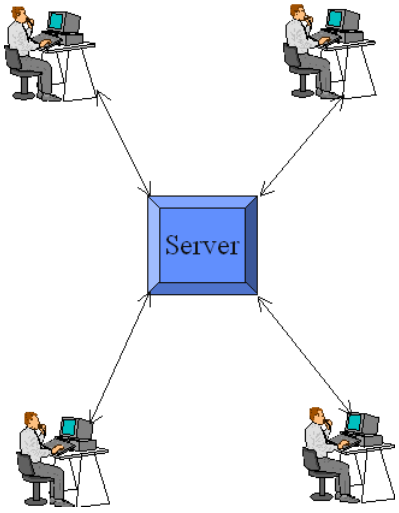

Abbildung 1: Sternförmige Topologie des Jupiter Algorithmus [24]

Der Jupiter Algorithmus löst das unter dOPT-Puzzle bekannte Problem vom dOPT Algorithmus unter der Bedingung, dass nur 2-Wege Kommunikation zugelassen ist [22]. Der Jupiter Algorithmus setzt eine zuverlässige, reihenfolgeerhaltende Kommunikation zwischen den

Listing 1: Algorithmus für die 2-Wegekommunikation [23]

```
int myMsgs                = 0; /* number of messages generated */
int otherMsgs             = 0; /* number of messages received */
queue outgoing  = {}
Generate(op) {
  apply op locally;
  send(op, my Msgs, otherMsgs);
  add (op, myMsgs) to outgoing;
  myMsgs = myMsgs + 1;
}
Receive(msg) {
  /* Discard acknowledged messages. */
  for m in (outgoing){
    if (m.myMsgs < msg.otherMsgs)
      remove m from outgoing
  }
  /* ASSERT msg.myMsgs == otherMsgs. */
  for i in [1..length(outgoing)] {
    /* Transform new message and the ones in the queue. */
    {msg, outgoing[i]} = xform(msg, outgoing[i]);
  }
  apply msg.op locally;
  otherMsgs = otherMsgs + 1;
}
```

Clients und dem Server voraus, wodurch sich dieser gegenüber dem dOPT Algorithmus erheblich vereinfacht [23]. Für den 2-Wege Algorithmus zwischen dem Server und den Clients wird das gleiche Protokoll wie bei dem dOPT Algorithmus verwendet. Das Protokoll basiert auf einen zwei-dimensionalen Zustandsraum.

Wenn ein Client eine Veränderungsoperation durchführt, wird diese unverzögert auf der lokalen Kopie des gemeinsam bearbeiteten Dokumentes ausgeführt und wird zu dem zentralen Server propagiert. Der Server transformiert diese falls notwendig, wendet sie auf seiner eigenen Kopie an und schickt die Transformationsoperation an alle Clients. Diese empfangen die Operation, transformieren diese wiederum gegebenenfalls und wenden sie auf der lokalen Kopie an. Den Algorithmus für die 2-Wegekommunikation zeigt das Listing 1.

Die hier verwendeten Datenstrukturen sind zwei Sequenznummern *myMsgs* und *otherMsgs* für die Anzahl von generierten und empfangenen Operationen sowie einer Nachrichtenwarteschlange *outgoing*. Die beim Empfangen einer Operation angewendete Transformationsfunktion trägt den Namen *xform*.

Die Funktionsweise des 2-Wege Algorithmus kann man sich am besten durch einen gerichteten azyklischen Graphen (DAG) verdeutlichen. Die Knoten dieses Graphen sind die Elemente des zweidimensionalen Zustandsraumes, wobei die Tupel gleichbedeutend mit der Anzahl der generierten und empfangenen Operationen sind. Die Kanten stellen die auf den

Daten angewendeten Veränderungsoperationen dar. In der Abbildung 2 wird der zweidimensionale Zustandsraum dargestellt.

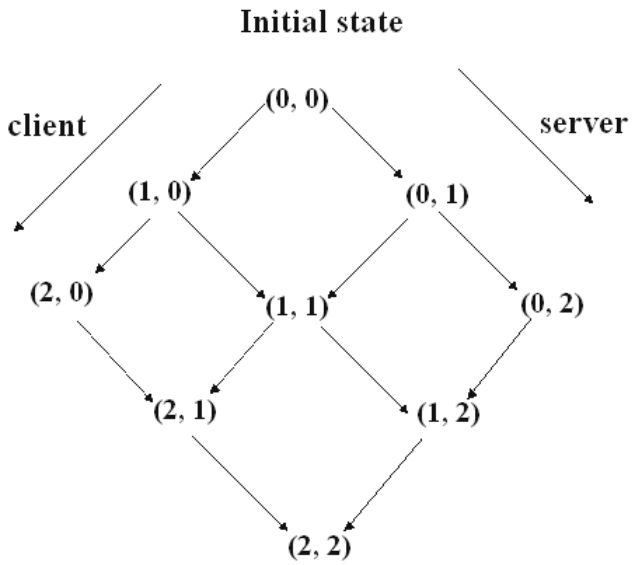

Abbildung 2: Der zweidimensionale Zustandsraum lässt sich als gerichteter azyklischer Graph darstellen, wobei die Knoten die Zustände und die Kanten die Veränderungsoperationen repräsentieren [23].

Im Folgenden verwende ich den Begriff Konflikt zwischen zwei Operationen *a* und *b* von unterschiedlichen Seiten, wenn die Ausführungsreihenfolge bei den Seiten nicht dieselbe ist. Anders ausgedrückt überlappen sich die Zeitintervalle von der lokalen Ausführung bis zur entfernten Ausführung der Operationen. Die Abbildung 3 veranschaulicht diesen Begriff.

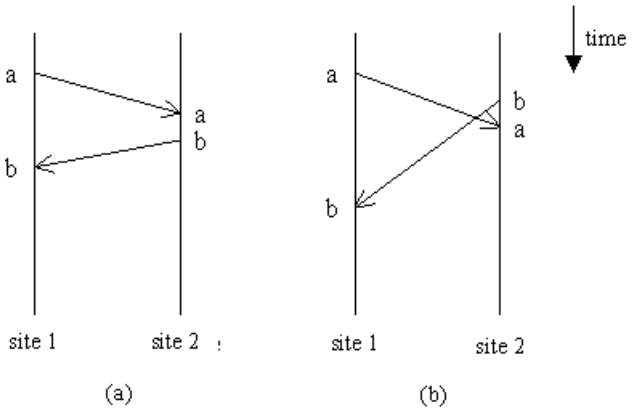

Abbildung 3: (a) überlappende Operationen *a* und *b* die im Konflikt stehen (b) nicht überlappende Operationen *a* und *b* ohne Konflikt

Existiert in einem Ablauf von Veränderungsoperationen kein Konflikt, so nehmen Client und Server den gleichen Pfad in dem dazugehörigen Graphen. Existiert jedoch ein Konflikt, so sind die Pfade von Server und Client verschieden. Der Algorithmus muss nun gewährleisten, dass beide Pfade am Ende in dem gleichen Knoten enden, was genau die Konvergenz

darstellt.

Vorbedingung für die Korrektheit des Algorithmus ist, dass der Client und der Server vom gleichen Zustand aus starten. Divergieren die Pfade nur um die Länge 1, so kann die Transformationsfunktion *xform* direkt verwendet werden um die Transformationsoperationen zu berechnen und damit sicherzustellen, dass der Server und der Client sich im gleichen Zustand wieder treffen.

Nehmen wir zum Beispiel an, dass sowohl der Server als auch der Client sich im Zustand *(0,0)* befinden. Der Client führt lokal die Operation c aus während der Server die zu c im Konflikt stehende Operation s ausführt. Beim Empfangen der jeweils vom anderen erzeugten Operation berechnet die Transformationsfunktion *xform* die Transformationsoperationen c' und s' mit deren Anwendung die beiden Kopien der Dokumente in den Zustand *(1,1)* konvergieren. Dieser einfache Fall wird in Abbildung 4 dargestellt.

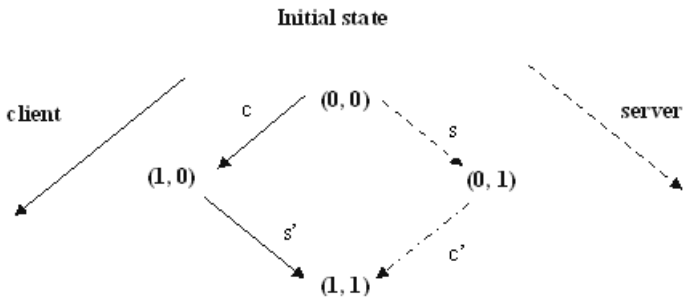

Abbildung 4: Diese Abbildung zeigt den einfachsten Fall eines Konfliktes. Der Server und der Client berechnen mittels der Transformationsfunktion die transformierten Operationen s' und c' und gelangen somit in den gemeinsamen Zustand $(1, 1)$ [23].

Schwieriger wird es, wenn die Divergenz der Pfade größer als ein Schritt ist. Dieser etwas schwierigere Fall wird mit folgendem Beispiel [23] erläutert und ist in der Abbildung 5 veranschaulicht.

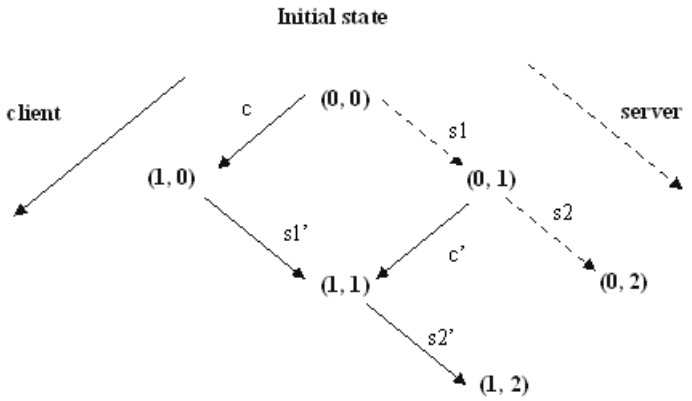

Abbildung 5: Diese Abbildung zeigt den beschriebenen Fall, dass die Pfade von Client und Server um mehr als einen Schritt divergieren. Der Client muss sich zu der Transformationsoperation *s1'* die hypothetische Operation c' merken, obwohl er diese niemals ausführen wird [23].

Nehmen wir an der Client führt eine Operation *c* aus, während der Server die zu *c* im Konflikt stehenden Operationen *s1* und *s2* ausführt. Nachdem der Client *c* ausgeführt hat und sich im Zustand *(1,0)* befindet, empfängt dieser die Operation *s1*. Er berechnet mit der Transformationsfunktion *s1′* und gelangt so in den Zustand *(1,1)*. Der Server führt aber nun die Operation *s2* aus ohne die Nachricht von *c* bearbeitet zu haben und schickt dem Client die Nachricht für *s2*. Der Client empfängt diese Operation, kann nun aber nicht die Transformationsoperation mit den Parametern *c* und *s2* ausführen, da diese Operationen von unterschiedlichen Zuständen als Ausgangslage ausgehen.

Die Lösung dieses Problems besteht darin, dass der Client sich bei der Berechnung von *s1′* den zweiten von der Transformationsfunktion zurückgelieferten Wert *c′* merkt, der die hypothetische Veränderungsoperation darstellt, um vom Zustand *(0,1)* in den Zustand *(1,1)* zu gelangen. Beim Empfangen von *s2* wendet er nun also die Transformationsoperation mit *c′* statt *c* an um *s2′* zu berechnen und gelangt mit der Ausführung dieser Transformationsoperation in den Zustand *(1,2)*. Bearbeitet der Server nun die Nachricht für die Operation *c*, so gelangt er ebenfalls in den Zustand *(1,2)* und die Konvergenz ist sichergestellt. Führt dieser aber wiederum zunächst eine Operation *s3* aus, so verwendet der Client *c″* um die dazugehörige Transformationsoperation zu berechnen.

Herr Rieger verwendete eine bereits bestehende Implementierung des Jupiter Algorithmus aus dem Open Source Projekt ACE [25] und integrierte diese in Saros. ACE ist ein einfacher in Java geschriebener Echtzeit-Gruppeneditor. Leider ist dieses Projekt anscheinend schon seit 2007 nicht mehr aktiv und der angegebene Entwicklungsstand wurde von den drei bei Sourceforge [25] registrierten Entwicklern als Alpha angegeben. Das letzte Release in Binärform ist aus dem Jahre 2006, die letzte Veränderung am Quelltext wurde 2007 vorgenommen.

In Kapitel 5.2.1 diskutiere ich inwieweit der Jupiter Algorithmus im Allgemeinen und die verwendete Implementierung aus dem ACE Projekt im Speziellen den für Saros aufgestellten Anforderungen genügen.

5 Anforderungen

In der ersten Phase meiner Diplomarbeit wurden die Anforderungen an die Software Saros neu bestimmt und analysiert. Hierfür wurden die in den bereits vorhandenen Dokumenten des Projektes aufgestellten Anforderungen zunächst zusammengetragen und neu bewertet. Diese Neubewertung wurde notwendig, da die Anforderungen aus den Arbeiten DPPI [9] und DPPII [10] teilweise aufgrund von veränderten Anforderungen nicht mehr gültig waren, sie im Rahmen von DPPIII [11] aber nicht aktualisiert wurden. Zum Teil wurden Anforderungen auch zu ungenau beschrieben oder fehlten komplett.

In Kapitel 5.2 analysiere ich die Anforderungen, um festzustellen, ob die verwendeten Technologien und Softwarebibliotheken den Anforderungen genügen.

Abschließend stelle ich in Kapitel 5.3 die durch die geplante Kooperation mit einem Wirtschaftsunternehmen vorgenommenen Änderungen an den Anforderungen beziehungsweise deren Priorisierung dar.

5.1 Anforderungsbestimmung

Für die Erhebung von neuen und der Überprüfung der von mir zusammengetragenen existierenden Anforderungen gab es eine Besprechung zwischen meinen Betreuern Christopher Oezbek und Stephan Salinger, die in diesem Gespräch die Rolle des Kunden einnahmen. Auf Basis dieses Feedbacks wurden die Anforderungen erneut überarbeitet und nach einer weiteren Besprechung dann freigegeben.

Die Anforderungen wurden von mir durch zwei sich ergänzende Notationen beschrieben, zum einem in Form eines Anforderungskataloges, zum anderen als Anwendungsfälle. Die Anwendungsfälle wurden in Textform unter Verwendung einer mir als geeignet vorkommenden Schablone angefertigt, die sich an einer im Buch „Applying UML and Patterns“ [26] von Craig Larman verwendeten Schablone anlehnt.

5.1.1 Produktdefinition

Im Folgenden werde ich die wichtigsten Anforderungen kurz erläutern, im Anhang A und B befinden sich der gesamte Anforderungskatalog sowie die Anwendungsfälle.

Das Werkzeug muss mehreren, auch mehr als zwei, räumlich voneinander getrennten Entwicklern ein gemeinsames Arbeiten am Quelltext ermöglichen. Alle Entwickler müssen zu jeder Zeit sowohl als Driver als auch Observer an der Programmiersitzung teilnehmen können.

Als Mindestanforderung für die Kommunikation zwischen den Teilnehmern der Programmiersitzung muss diese über Textnachrichten geschehen können. Dabei soll es möglich sein, eine Nachricht an alle oder nur an einen bestimmten Teilnehmer zu schicken. Eine Kommunikation über Sprache und/oder Video muss nicht durch das Werkzeug selber geleistet werden, sondern kann durch externe Programme realisiert werden.

Den Observern muss es nicht nur möglich sein, andere Programme neben der Entwicklungsumgebung auszuführen (*Parallelität auf Programmebene*), sondern auch innerhalb der Entwicklungsumgebung von dem Aufmerksamkeitsbereich des/der Driver abzuweichen und eigenständig Dateien eines Projektes anzuschauen (*Parallelität auf Sichtebene*).

Das Werkzeug muss es ermöglichen, dass zur gleichen Zeit mehrere Driver, die gleichzeitig schreibend auf die gemeinsamen Dateien zugreifen, existieren können (*Parallelität auf Schreibebene*). Die Driver sollen einfache Editieroperationen (Einfügen/Löschen) blockierungsfrei durchführen können. Für Dateioperationen und komplexere Editieroperationen (zum Beispiel Refaktorisierungen) können temporär einzelne Dateien gesperrt werden (*exklusive Schreibsperre*). Die Undo-Funktion von Eclipse zum Rückgängigmachen der eigenen, und zwar nur der eigenen, Änderungen muss weiterhin funktionieren.

Es muss zu jeder Zeit visualisiert werden, wer gerade an welcher Datei arbeitet, zum Beispiel durch Markierung in der Dateiliste. Des Weiteren müssen die aktuellen Cursorpositionen der Driver übertragen werden. Die Änderungen der Driver müssen durch eine geeignete Annotation den jeweiligen Drivern zugeordnet werden können, zum Beispiel durch global eindeutige Hintergrundfarben der Änderungen.

Den Observern muss es möglich sein, dem Sichtbereich eines ausgewählten Drivers automatisch zu folgen, falls dieser sich nicht mehr in dem eigenen Sichtbereich befindet. Der Sichtbereich von diesem Driver muss zusätzlich durch eine Annotation im Editor erkennbar sein. Dieser Verfolgermodus (*Follow Mode*) soll vom Observer selbstständig aktiviert und deaktiviert werden können. Bei Deaktivierung kann man sich für kurze Zeit vom Sichtbereich des Drivers entfernen um andere Stellen im Quelltext zu lesen. Durch erneutes aktivieren des Verfolgermodus gelangt der Observer wieder zum aktuellen Sichtbereich des Drivers. Dieses entspricht genau der oben genannten Parallelität auf Sichtebene. Die Observer müssen außerdem die Möglichkeit besitzen, durch selektieren von Text den anderen Teilnehmern bestimmte Textpassagen zu zeigen.

Bei allen Teilnehmern der Programmiersitzung muss der Quelltext genau so übersetzt werden können als wäre es eine normale, nicht verteilte Programmiersitzung. Dies ist ein Alleinstellungsmerkmal zu anderen Echtzeit-Gruppeneditoren, die meistens den Ansatz des virtuellen Zugriffs (siehe Kapitel 3) verfolgen.

Das Werkzeug muss in allen Netzwerktopologien eingesetzt werden können, sofern eine Verbindung zu einem Jabber-Server aufgebaut werden kann. Für den Austausch von Dateien muss eine Direktverbindung genutzt werden, sofern diese durch die letzte veröffentlichte, stabile Version der Bibliothek Smack [18] möglich ist. Sollte eine Direktverbindung nicht möglich sein, so muss es eine Rückfallstrategie geben, die trotzdem eine erfolgreiche Dateiübertragung sicherstellt. Für eine Analyse dieser Anforderungen bezüglich der Einsatzfähigkeit in allen Netzwerktopologien siehe Kapitel 5.2.2.

Die lokalen Änderungen müssen verzögerungsfrei wie bei einem Single-Editor durchgeführt werden. Änderungen von entfernten Entwicklern können mit einer Verzögerung von ein paar wenigen Sekunden nachvollzogen werden.

Jeder Teilnehmer der Programmiersitzung muss ein konsistentes Abbild aller Dateien des gemeinsamen Projektes besitzen. Die Nebenläufigkeitskontrolle für den Gruppeneditor muss dabei sowohl Konvergenz und Kausalitätserhaltung, als auch die Intentionserhaltung ge-

währleisten. Für eine nähere Diskussion dieser Anforderungen siehe Kapitel 5.2.1. Sollte es, aus welchen Gründen auch immer, zu einen nicht konsistenten Zustand kommen, zum Beispiel durch das Ersetzen von Dateien außerhalb von Eclipse, müssen diese Inkonsistenzen innerhalb von 15 Sekunden erkannt und ein konsistenter Zustand wiederhergestellt werden.

5.2 Anforderungsanalyse

Nachdem die Anforderungen bestimmt worden sind, folgte eine Anforderungsanalyse, um zu klären, welche der Anforderungen am schwierigsten oder sogar gar nicht zu erfüllen sind. Außerdem wurde analysiert, ob die im Projekt verwendeten Technologien und Softwarebibliotheken hierfür geeignet sind.

Der Schwerpunkt dieser Analyse lag bei der Nebenläufigkeitskontrolle und dem Aufbauen von Direktverbindungen zwischen den Teilnehmern in verschiedenen Netzwerktopologien.

5.2.1 Nebenläuffigkeitskontrolle

Die bestimmten Anforderungen (siehe Kapitel 5.1.1) für die Nebenläufigkeitskontrolle kann man wie folgt zusammenfassen:

- Unterstützung von replizierten Daten
- blockierungsfreie Veränderungen an der lokalen Kopie
- Konvergenz
- Kausalitätserhaltung
- Intentionserhaltung

Die ersten beiden Anforderungen werden durch die Verwendung eines auf Operation Transformation basierenden Algorithmus (siehe Kapitel 4.3) trivialerweise erfüllt. Die letzten drei, also die Konvergenz sowie die Erhaltung der Kausalität und der Intention, müssen einer genaueren Betrachtung unterzogen werden.

Das während der Arbeiten an dem *dOPT* Algorithmus [20] entstandene Konsistenzmodell umfasste die Forderung der Konvergenz und der Kausalitätserhaltung. Dieses Basismodell für die Konsistenz wird auch gelegentlich *CC-Modell* genannt. Im Folgenden verwende ich den Begriff *Konsistenz* im Sinne dieses Modells.

Der in Saros verwendete Operational Transformation Algorithmus Jupiter basiert auf dem dOPT Algorithmus. Dieser garantiert einem die Konvergenz und die Erhaltung der Kausalität im Falle von zwei Teilnehmern [20]. Bei mehreren Teilnehmern konnte in dem Fall, dass die Kopien um mehr als einen Schritt im dazugehörigen Zustandsraum divergieren, die Konsistenz nicht für alle Szenarien sichergestellt werden (dOPT-Puzzle) [23]. Dieses Problem umgeht der Jupiter Algorithmus, indem er zwischen jedem Client und dem Server unabhängig voneinander den Algorithmus für zwei Seiten verwendet. Dadurch ist die Konsistenz beim Jupiter Algorithmus in allen möglichen Fällen gegeben.

Bleibt die Forderung nach der Intentionserhaltung. Das oben genannte Modell für die Konsistenz wurde von C. Sun um die Forderung nach der Intentionserhaltung erweitert [27]. Dieses erweiterte Modell wird manchmal auch *CCI-Modell* genannt. Hierbei muss zunächst geklärt werden, was unter einer Intentionserhaltung zu verstehen ist. Unter *Intentionserhaltung* versteht Sun, dass für alle Operationen O der Effekt der Ausführung von O auf allen Seiten gleich der Intention von O sein muss, sowie dass der Effekt der Ausführung von O nicht die Effekte von anderen unabhängigen Operationen verändert [22].

Diese Anforderung ist nicht besonders formal und schwierig zu verifizieren. Aber auch wenn man sich auf den Begriff einlässt, ist es für einen Algorithmus schwierig, dieser Forderung Genüge zu leisten.

Nehmen wir zum Beispiel an, eine Textpassage wird von zwei Schreibern zur gleichen Zeit bearbeitet. Der eine markiert die ganze Textpassage und löscht diese. Währenddessen fügt der andere Schreiber mitten in dieser Textpassage ein neues Wort ein. Wie muss jetzt die Transformationsfunktion des Operational Transformation Algorithmus die Intentionen beider Operationen erhalten?

Die Transformationsfunktion von Jupiter fügt das Wort trotzdem ein, um die Benutzereingabe zu erhalten. Die transformierte Operation muss nun zwei Regionen löschen, die Region vor dem Wort und die nach dem Wort. Für diesen Fall wurde vorgeschlagen, die Operation in zwei Nachrichten aufzuteilen und Zwischenzustände einzuführen [23]. In der Jupiter-Implementierung vom ACE Projekt [25] werden sogenannte *Splitoperationen* eingeführt, die diese beiden Löschoperationen zu einer Operation zusammenfassen.

Wie man sieht, ist der Begriff der Intention schwierig zu definieren und die dazugehörige Anforderung schwierig zu erfüllen. Auf der anderen Seite sollte man schon fordern, dass die Transformationsfunktion etwas „sinnvolles“ berechnet. So würde zum Beispiel die Transformationsoperation, die immer alles löscht, auch die Konvergenz sicherstellen und die Kausalität erhalten, ist aber sicherlich nicht im Sinne der Benutzer.

Zusammenfassend kann man sagen, dass der Jupiter Algorithmus die Konsistenz nach dem CC-Modell, also die Konvergenz und die Kausalitätserhaltung, gewährleistet. Die Forderung der Intentionserhaltung des erweiterten CCI-Modells zu überprüfen, ist aufgrund der schwierigen Semantik des Begriffes nicht erschöpfend möglich.

Zum Schluss der Analyse möchte ich kurz auf die in Saros verwendete Jupiter-Implementierung eingehen. Das potenzielle Problem mit dieser besteht darin, dass das ACE Projekt [25], von dem die Implementierung ursprünglich stammte, nicht mehr aktiv zu sein scheint und der Status der Software von den Entwicklern als Alpha eingestuft worden ist (siehe Kapitel 4.3). Daher war nicht ganz ersichtlich, inwieweit die Implementierung des Jupiter Algorithmus vollständig und korrekt ist. Es existieren im Saros Projekt einige JUnit-Tests, welche die Korrektheit des Algorithmus für einige Fälle überprüfen, daher bin ich von einer korrekten Implementierung ausgegangen.

5.2.2 Analyse der Netzwerkanforderungen

In den für Saros aufgestellten Anforderungen (siehe Kapitel 5.1.1) wird gefordert, dass das Werkzeug in allen Netzwerktopologien eingesetzt werden können muss. Es wird lediglich

vorausgesetzt, dass eine Verbindung zu einem Jabber-Server aufgenommen werden kann.

Für den Austausch von Dateien muss eine Direktverbindung genutzt werden, sofern eine solche aufgebaut werden kann. Sollte eine Direktverbindung nicht möglich sein, so muss es eine Rückfallstrategie geben, die trotzdem eine erfolgreiche Dateiübertragung sicherstellt.

Netzwerktopologien

Die interessante Frage ist nun, in welchen Fällen eine Direktverbindung zustande kommen kann und in welchen Fällen dies nicht möglich ist. Wir untersuchen deshalb die Kommunikation zwischen zwei Teilnehmern mit dem Internet Protokoll IP in der Version 4. Hierfür teilen wir zunächst alle möglichen Szenarien in drei verschiedene Fälle ein:

Fall 1: Gleiches Netz - beide Teilnehmer sind im gleichen Adressraum Beide Teilnehmer sind in einem Netz und in einem Adressraum (egal ob privat oder öffentlich) und können direkt miteinander kommunizieren.

Fall 2: Höheres Netz Ein Teilnehmer befindet sich in einem untergeordneten Subnetz des anderen, so dass nur derjenige aus dem untergeordneten Subnetz denjenigen im höher liegenden Subnetz erreichen kann. Zum Beispiel kann ein Teilnehmer in Subnetz-Klasse C einen Teilnehmer mit öffentlicher IP-Adresse erreichen. Ohne eine dritte Partei kann aber umgekehrt der Teilnehmer im höher liegenden Netz demjenigen im unterliegenden Netz nichts signalisieren.

Fall 3: Unterschiedliche Netze Beide Teilnehmer befinden sich in unterschiedlichen Subnetzen, so dass sie füreinander jeweils keine öffentliche IP-Adresse haben. Ohne eine dritte Partei können beide Teilnehmer sich nichts gegenseitig signalisieren.

Allgemeine Lösungen für die unterschiedlichen Netzwerktopologien

Im Fall 2 und 3 der oben identifizierten Netzwerktopologien findet eine *Network Address Translation (NAT)* statt, wodurch beim Aufbau einer Direktverbindung Probleme auftreten können [28].

Wenn nur ein Teilnehmer hinter einem NAT sitzt, kann ohne Hilfe von Dritten nur in eine Richtung die Direktverbindung aufgenommen werden. Soll in die andere Richtung eine Direktverbindung aufgenommen werden, kann ein Verfahren namens *Connection Reversal* [28] zum Einsatz kommen. Dieses Verfahren verwendet einen sogenannten *Rendezvous-Server*, mit dem beide Teilnehmer bereits eine Verbindung aufgenommen haben. Im Fall von Saros ist dies ein Jabber-Server beziehungsweise der Jabber-Serververbund. Der Teilnehmer, der keine Verbindung zum anderen aufnehmen kann, bittet den Rendezvous-Server den anderen Teilnehmer Bescheid zu sagen, dass er versucht zu diesem eine Direktverbindung aufzunehmen. Dieser nimmt dann eine anders gerichtete Verbindung zum anderen auf.

Wenn beide Teilnehmer hinter einem NAT sitzen, kann das *Hole Punching* Verfahren [28] angewendet werden. Dieses Verfahren arbeitet wieder mit einem Rendezvous-Server, mit dem

bereits beide Teilnehmer verbunden sind. Dieser Server kennt von beiden Teilnehmern eine öffentliche IP-Adresse, unter der sie erreichbar sind. Die Teilnehmer haben dem Server ihre öffentliche IP-Adresse zuvor mitgeteilt. Wenn diese keine öffentliche IP-Adressen kennen, unter der sie erreichbar sind, können sie die IP-Adresse durch spezielle Techniken, wie dem *STUN* Verfahren (Simple traversal of UDP over NATs) [29], in Erfahrung bringen. Durch das STUN Protokoll, das aus mehreren aufeinander folgenden Anfragen an einen öffentlichen STUN-Server besteht, kann man nicht nur eine öffentliche IP-Adresse sondern auch zusätzliche Informationen über die NATs in seiner Umgebung herausfinden. Das STUN Verfahren funktioniert nicht mit allen Typen von NATs (zum Beispiel nicht mit symmetrischem NAT) und nicht mit allen Typen von Routern [28]. Außerdem wird nur UDP als Transportprotokoll unterstützt[2].

Wenn nun eine Direktverbindung mittels des Hole Punching Verfahrens aufgenommen werden soll, teilt der Server den beiden Teilnehmern die öffentlichen IP-Adressen des jeweils anderen mit. Danach initiieren beide „gleichzeitig“ eine Verbindung zum anderen. Der eine Versuch scheitert, da die Firewall des anderen das Paket blockiert (siehe Abbildung 6), der anders gerichtete Versuch klappt aber, da zuvor ein Verbindungsendpunkt eingerichtet worden ist. Es wurde sozusagen ein Loch in die NAT/Firewall geschlagen (siehe Abbildung 7).

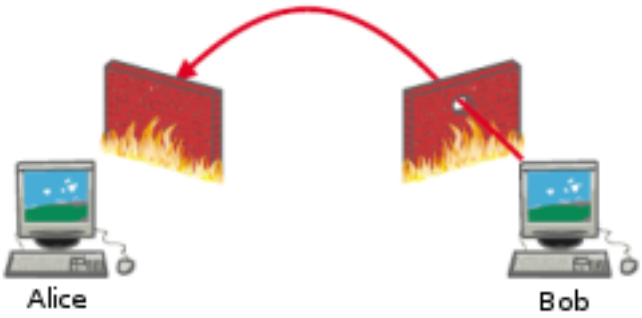

Abbildung 6: Der Versuch in die eine Richtung beim Hole Punching scheitert zunächst [30].

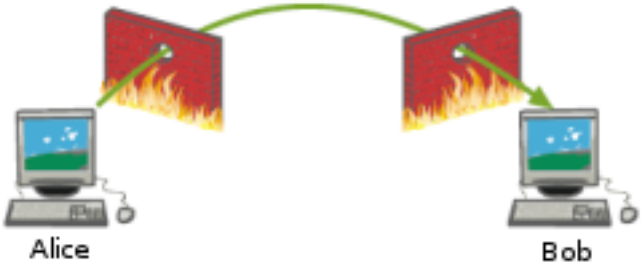

Abbildung 7: Der nachfolgende, anders gerichtete Versuch hat Erfolg, da zuvor ein Loch in die Firewall geschlagen wurde [30].

[2]STUN wird zurzeit von der IETF überarbeitet und soll in Zukunft auch TCP unterstützen, dann würde das Akronym für „Session Traversal Utilities for NAT“ stehen.

Dieses Hole Punching Verfahren ist nicht mit allen NAT-Typen möglich, insbesondere bei symmetrischen versagt es [28]. In diesem Fall, genauso wie wenn als Transportprotokoll TCP verwendet werden soll, muss die Verbindung über einen Relay-Server (zum Beispiel TURN [31]) geschehen. Es gibt zwar auch Bestrebungen um ein Hole Punching mit TCP zu ermöglichen, dies ist aber weniger Erfolg versprechend [28].

Lösungen in Saros

Kontaktaufnahme In Saros findet die Kontaktaufnahme zwischen den Teilnehmern über den Jabber-Serververbund mittels der Java API Smack [18] statt. Es wird daher vorausgesetzt, dass alle Teilnehmer eine Verbindung zu einem Server im Jabber-Serververbund aufnehmen können. Da die Jabber-Server üblicherweise mit einer öffentlich erreichbaren IP-Adresse betrieben werden, sollte das Vorhandensein einer Verbindung ins Internet eine hinreichende Bedingung für die Kontaktaufnahme in Saros sein. Alternativ kann auch ein isolierter Jabber-Server im Intranet verwendet werden, zu dem sich die Teilnehmer verbinden.

Smack als API für Direktverbindungen Bei der Initiierung der Direktverbindung durch die Smack-Implementierung der XMPP-Erweiterung Jingle (XEP-0166) wird das Hole Punching Verfahren innerhalb des *ICE* Verfahrens (*Interactive Connectivity Establishment*) [32] verwendet, das wiederum das STUN Verfahren benutzt um eine NAT-Traversierung zu realisieren. Damit sollte Smack eine NAT-Traversierung durch nahe zu allen NAT-Typen ermöglichen. Eine NAT-Traversierung ist aber nur bei der Verwendung von UDP als Transportprotokoll möglich. Zwar existieren schon Bestrebungen ICE zu erweitern, um auch TCP zu ermöglichen [33], dies wird in Smack aber noch nicht unterstützt und es gibt hierfür auch noch keinen konkreten Entwurf bei der XMPP Standards Foundation.

Aus diesem Grund muss für eine erfolgreiche NAT-Traversierung UDP als Transportprotokoll verwendet werden und für die Zuverlässigkeit der Verbindung innerhalb der Anwendungsschicht selber gesorgt werden, siehe Kapitel 7.3.

Sollte es trotzdem nicht möglich sein, mittels Smack/Jingle eine Direktverbindung aufzubauen, wird als Rückfallstrategie XMPP für den Datenaustausch verwendet. Für die Dateiübertragung wird dann die XMPP-Erweiterung XEP-0096 (File Transfer über XMPP) [15] benutzt. Aufgrund der geringen Bandbreite sollten die Dateien vor der Übertragung gepackt und ein komprimiertes Archiv übertragen werden.

Anmerkung: Es gibt zwar schon Bestrebungen für eine Standardisierung der Dateiübertragung mittels Jingle (XEP-0234) [15], diese sind aber noch nicht weit fortgeschritten und wurden in Smack noch nicht implementiert.

5.3 Anforderungsänderungen

Mitten in der dritten Iteration (siehe Kapitel 7.1) wurde an mich herangetragen, dass schon bald eine Kooperation mit einem in Berlin ansässigen Wirtschaftsunternehmen geplant sei, das Saros für Schulungszwecke einsetzen möchte. Dies hat die Priorisierung der Anforderungen und der noch ausstehenden Arbeitspakete teilweise massiv verändert.

Das Szenario, in dem die Software möglichst bald zum Einsatz kommen sollte, kann man wie folgt beschreiben:

- keine Unterstützung von mehreren Drivern zur gleichen Zeit
- gebündelter Rollenwechsel, das heißt einem Teilnehmer in nur einem Schritt die exklusive Driver Rolle geben
- Unterstützung der Programmiersprachen C und C++
- keine NAT-Traversierung notwendig, da sich alle Teilnehmer im gleichen Netzwerk befinden

Um den Einsatz in diesem gegebenen Einsatzszenario möglichst schell zu ermöglichen, wurde der bestehende Plan dahingehend modifiziert, dass ich mich in der laufenden Iteration zunächst um die Unterstützung von den Programmiersprachen C/C++ kümmerte. Zwar konnte man davon ausgehen, dass die Unterstützung gegeben, und wenn überhaupt nur kleinere Veränderungen an der Software notwendig sein sollten, aber dies wurde noch nie zuvor getestet. Es stellte sich dann auch heraus, dass ein paar kleinere Veränderungen notwendig waren, damit Saros mit dem Eclipse-Plugin für C/C++ Entwicklung (CDT) zusammenarbeitete.

Des Weiteren wurde in der laufenden Iteration die neue Funktionalität des gebündelten Rollenwechsels realisiert. Dies wurde durch die Tatsache erschwert, dass zu diesem Zeitpunkt in der Software der Host immer die Driver Rolle inne hatte und diese nicht abgeben konnte. Dies wurde nach meinem Kenntnisstand im Rahmen von DPPIII [11] eingeführt, da dies aufgrund der zentralisierten Architektur des Jupiter Algorithmus (siehe Kapitel 4.3) notwendig geworden war. Dieses Problem ließ sich lösen, indem es innerhalb der Nebenläufigkeitskontrolle eine Spezialbehandlung für den Host gab. Aus dem Blickwinkel der Nebenläufigkeitskontrolle ist der Host dadurch auch dann ein Schreiber wenn er die Observer-Rolle einnimmt (er schreibt halt nur nicht). Um zu verhindern, dass der Host nichts schreiben kann, wird ihm einfach die Schreibrechte am Editor entzogen.

Eine neue Priorisierung der noch ausstehenden Arbeitspakte führte dazu, dass die weiterhin bestehenden Probleme bei dem Aufbau von Direktverbindungen erst einmal nicht weiter untersucht wurden. Des Weiteren wurde das noch nicht bearbeitete Arbeitspaket der Awareness-Informationen als nächstes angegangen. Dieses Arbeitspaket habe ich komplett an den neu in das Projekt gekommenen Kommilitonen Marc Rintsch abgegeben, der dies im Rahmen seiner Studienarbeit erledigte. Allerdings wurden die Awareness-Informationen zunächst nur für den Einsatz des obigen Szenarios implementiert, was dazu führte, dass die Unterstützung von mehreren Drivern bei der Darstellung der Awareness-Informationen in der aktuellen Version von Saros noch nicht gegeben ist.

6 Bestimmung von Versagen und Defekte

In diesem Kapitel werde ich die am meisten angewendeten Verfahren zur Bestimmung von Versagen und der dazugehörigen Defekte der Software beschreiben. Neben den Testexperimenten, in denen Funktionstests von externen Personen durchgeführt worden sind, gehörten hierzu auch von mir alleine durchgeführte Defekttests und Code-Durchsichten.

6.1 Testexperimente

Das Testen von verteilten Systemen gestaltet sich oft als nicht ganz so einfach, denn die durchzuführenden Tests lassen sich zum Teil nur schwer, und wenn überhaupt dann nur mit erheblichem Aufwand, automatisieren. Auch wenn man die Tests manuell durchführt, so braucht man häufig mehrere Personen, welche die einzelnen Seiten des verteilten Systems bedienen.

Aus diesem Grund sowie dem Phänomen, dass externe Personen mehr Versagen der Software entdecken, da diese noch unbedarft an die Sache herangehen (keine „Betriebsblindheit“), wurden insgesamt zwei Testexperimente durchgeführt.

6.1.1 Erstes Testexperiment

Nachdem in den ersten beiden Wochen meiner Arbeit der Fokus auf das Erstellen eines Anforderungskataloges und von Anwendungsfällen für Saros lag (siehe Kapitel 5), wurde dieser nun auf das Finden von Versagen der Software gelegt. Zu diesem Zweck war ein erstes Testexperiment für die dritte Woche angesetzt, bei dem externe Entwickler die unbearbeitete Software auf Versagen hin testen sollten. Dieser erste Test musste um zwei Wochen nach hinten verschoben werden, da die Software sich in einem Zustand befand, indem es keinen Sinn gemacht hätte ein Testexperiment durchzuführen.

Die Software war derart unstabil, dass nach nur einigen wenigen Editieroperationen sie nicht mehr zu benutzen war. Spätestens bei der Eingabe eines Newline Zeichens wurden endlos neue Zeilen eingefügt. Neben dem nicht funktionierenden Chat gab es weitere schwerwiegende Probleme, wie zum Beispiel den Programmabstürzen bei dem Versuch, einen Kontakt mit einem nicht existierenden Jabber Identifier hinzuzufügen.

Um einen sinnvollen ersten Test der Software durchführen zu können, musste ich also innerhalb kurzer Zeit die schwerwiegendsten Defekte in der Software finden und korrigieren. Dies gelang mir insoweit, dass in der vierten Woche das Testexperiment zumindest ansatzweise durchgeführt werden konnte. In Kapitel 7.2.2 gehe ich näher darauf ein, wie der obige Defekt mit den Newline Zeichen behoben werden konnte.

Das Ziel dieses ersten Tests war es nicht eine empirische Untersuchung durchzuführen, sondern lediglich Versagen der Software aufzudecken. Es wurde weder das Verhalten der Tester beobachtet noch die Artefakte der Programmiersitzung untersucht.

Technische Umsetzung An dem Test nahmen drei Softwareentwickler teil, die ich aus meinem Freundeskreis akquirieren konnte. Einer davon ist ein ehemaliger Student, die anderen beiden waren zu diesem Zeitpunkt noch Studenten des Instituts Informatik der Freien Universität Berlin.

Die Entwickler saßen im gleichen Raum und konnten somit direkt kommunizieren. Da es bei diesem Test nicht um die Validierung der Software ging, war es nicht nötig eine möglichst realistische Einsatzumgebung des Produktes zu schaffen. Vielmehr wurde versucht eine kontrollierte Umgebung zu realisieren, um somit Fehlerquellen außerhalb von Saros zu minimieren.

Um möglichst Störungen von außen zu vermeiden, wurde ein eigenes lokales Netzwerk ohne Verbindung zum Internet aufgebaut. Dieses Netzwerk bestand aus einem Router (der nur als Switch verwendet wurde) und vier Computer, die mittels Ethernet-Kabel verbunden waren. Die drei Tester brachten ihre eigenen Laptops für die Programmiersitzung mit (zwei mit Windows und einer mit Linux Betriebssystem). Auf dem vierten Computer installierte und konfigurierte ich einen eigenen Jabber-Server (Openfire[3]), der unter Windows lief.

Damit die drei Arbeitsrechner den Jabber-Server finden konnten, wurde der Host-Name des Servers manuell in die Host-Datei der Rechner eingetragen. Die Personal Firewalls wurden auf allen Rechnern abgeschaltet, um möglichen Problemen mit diesen aus dem Weg zu gehen. Auf den Laptops wurden zusätzlich die WLAN-Netzwerkkarten deaktiviert. Die Topologie des im Test verwendeten Netzwerkes ist in Abbildung 8 dargestellt.

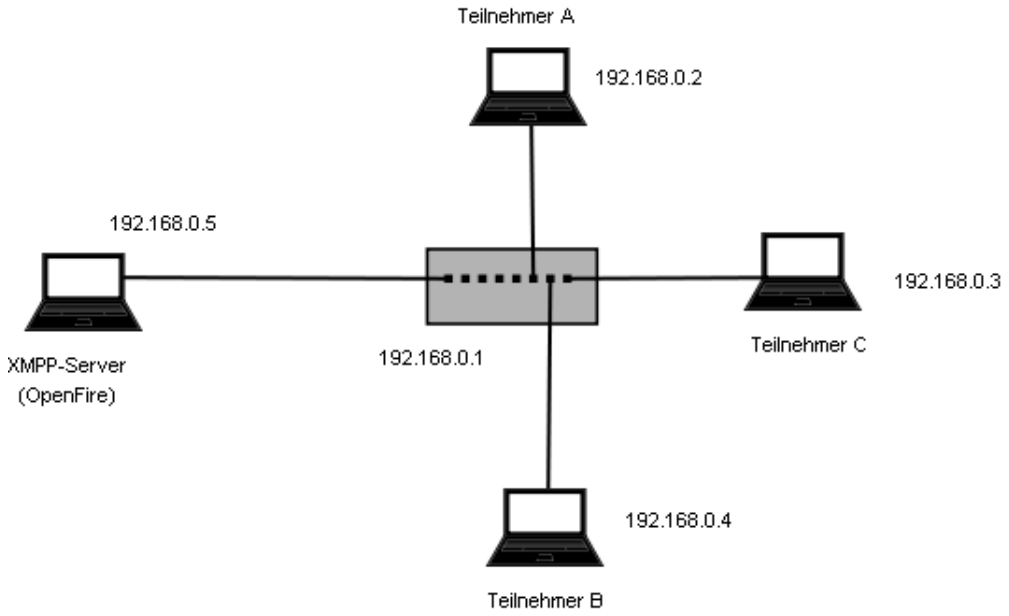

Abbildung 8: Topologie des im ersten Test verwendeten Netzwerkes

Um mögliche Probleme mit bestehenden Eclipse-Installationen zu vermeiden, wurde auf den Arbeitsrechnern ein neues Eclipse 4.3 (in der Classic Variante) entpackt. Saros wurde als lokale Update-Site vom Dateisystem aus installiert, da die Rechner keine Verbindung mit

[3] http://www.igniterealtime.org/projects/openfire

dem Internet besaßen.

Aufgabenstellung Den Testern wurde eine Schritt-für-Schritt-Anleitung zur Installation und Nutzung von Saros ausgehändigt. Dies hatte den Zweck die im Projekt bereits bestehende Benutzerdokumentation einer Evaluierung zu unterziehen. Des Weiteren bekamen sie eine schriftliche Aufgabenstellung für die Programmiersitzung. Offene Fragen seitens der Programmierer wurden im Vorfeld geklärt. Das erfolgreiche Bearbeiten der Aufgaben war nicht von zu beobachtendem Interesse.

Um auch den von mir neu implementierten Chat zu testen, wurden die Tester gebeten, diesen während der Programmiersitzung zu verwenden, obwohl dies aufgrund der räumlichen Nähe nicht nötig gewesen wäre. Die neue Implementierung war notwendig geworden, da die alte Implementierung nicht als Eclipse View realisiert war, und somit bei eintreffenden Textnachrichten der sich öffnende Chat dem Editor den Fokus entriss.

Es gab insgesamt drei Aufgaben, die unterschiedliche Szenarien testen sollten. Zusätzlich zu den Lösungen sollten die Programmierer kleine Tests entwickeln. Hierfür wurden keine JUnit Tests verlangt, sondern es reichte wenn in der *main* Methode einige wenige Fälle getestet wurden.

Die erste Aufgabe wurde von nur zwei Entwicklern bearbeitet, von denen einer Driver und der andere Observer war. Die Rollen wurden nicht getauscht. Es sollte eine Methode entwickelt werden, die überprüft, ob eine gegebene Zeichenkette richtig geklammert ist. Hierfür muss die Methode nicht nur überprüfen ob die Anzahl der schließenden gleich der Anzahl der öffnenden Klammern ist, sondern dass keine schließende Klammer in der Zeichenkette ohne vorangegangener öffnende Klammer vorkommt.

In der zweiten Aufgabe sollten von drei Entwicklern, von denen zwei jeweils die Rolle eines Drivers einnahmen, zwei Methoden entwickelt werden. Die eine sollte zählen, wie viele Palindrome, mit Mindestlänge von zwei Zeichen, eine gegebene Zeichenkette als Teilzeichenkette besitzt. Die andere sollte das längste Palindrom berechnen, das eine gegebene Zeichenkette als Teilzeichenkette besitzt. Die Rollen sollten bei dieser Aufgabe wieder nicht gewechselt werden.

Die dritte Aufgabe war eine Aufgabe aus einem ACM Programming Contest der vergangenen Jahre. In der Aufgabe mit dem Namen „Why Johnny Can't Count“ [34] musste eine gegebene positive Zahl unter einer Million in eine Zeichenkette umgewandelt werden, die in englischer Sprache diese Zahl in Worten darstellt. Als Vereinfachung konnte das Wort „and“ weggelassen und nötige Pluralformen der Wörter nicht berücksichtigt werden. Diese letzte Aufgabe wurde von drei Programmierern bearbeitet, die in unregelmäßigen Abständen zwischen den Rollen Driver und Observer wechselten.

Durchgeführte Aufzeichnungen Während der Programmiersitzung schaute ich den Testern über die Schulter und machte mir Notizen über den Verlauf der Sitzung. Diese Notizen umfassten von den Programmierern gemeldete sowie das von mir beobachtete Versagen der Software. Des Weiteren wurden von den Bildschirmen der Arbeitsrechner Aufzeichnungen gemacht, um im Nachhinein genau das Zustandekommen des Versagens analysieren zu

können. Für diesen Zweck wurde auf den Windows Rechnern die Software *Camtasia*[4] und unter Linux die Software *RecordMyDesktop*[5] verwendet.

Direkt nach dem Experiment mussten die Tester einen Fragebogen ausfüllen, in dem sie unter anderem nach der Schwere des von ihnen gefundenen Versagens befragt wurden. In einer abschließenden offenen Diskussionsrunde wurden die Tester nochmals über den Verlauf des Experimentes befragt und konnten ihre Eindrücke schildern. Als Diskussionsleiter achtete ich darauf, dass innerhalb kurzer Zeit möglichst viele Themen angesprochen wurden. Die Diskussion dauerte etwa 15 Minuten und wurde bis auf die ersten Minuten aufgezeichnet (am Anfang gab es Aufnahmeprobleme).

Ergebnisse des ersten Tests Neben einigen missverständlichen Formulierungen und nicht mehr zu der aktuellen Version passenden Anleitungen in der Benutzerdokumentation wurde folgendes Versagen der Software festgestellt:

Die schwerwiegendsten Probleme bereitete die Tatsache, dass nach etwa 10 Minuten die Teilnehmer der Programmiersitzung keine konsistenten Kopien des bearbeiteten Quelltextes mehr besaßen. Zwar konnte nach einem erneuten Einladungsprozess wieder für kurze Zeit weiter programmiert werden (mit teilweise erheblichen Verlusten von Code), von einem vernünftigen Bearbeiten der Aufgaben konnte aber nicht die Rede sein. Die beiden ersten Aufgaben konnten dennoch mit einigen Zwangsunterbrechungen gelöst werden. Die dritte Aufgabe musste dann aus Zeitgründen frühzeitig abgebrochen werden.

Die Analyse der aufgezeichneten Videos brachte erste Hinweise, in welchen Situationen es zu inkonsistenten Zuständen kam. So konnte man erkennen, dass oft Speicheroperationen einem inkonsistenten Zustand vorhergegangen sind. Des Weiteren wurde deutlich, dass es Probleme mit unterschiedlichen Zeichensätzen gab. Insbesondere nachdem die Software bei Auftreten von Inkonsistenzen versucht hatte die Konsistenz wiederherzustellen (dies gelang ihr jedoch nie), wurden Umlaute nicht mehr richtig dargestellt. Diese Problematik wurde dadurch verstärkt, dass auf den Laptops unterschiedliche Betriebssysteme installiert waren (Windows Vista, Windows XP und Ubuntu Linux) und Eclipse je nach Betriebssystem unterschiedliche Zeichensätze als Voreinstellung verwendet. Aber auch das reine Editieren ohne spezielle Operationen führte zu inkonsistenten Zuständen, so dass man den Eindruck gewinnen musste, dass gar keine Nebenläufigkeitskontrolle in der Software existiert.

Bei der Darstellung von Awareness-Informationen wurden zahlreiche Unzulänglichkeiten entdeckt. So waren die für die einzelnen Teilnehmer der Programmiersitzung vorgesehenen Farben nicht global eindeutig. Dies führte zu einer erschwerten Kommunikation, wenn man über die einzelnen Selektionen im Text reden wollte. Die im Quelltext vorgenommenen Änderungen waren immer mit der gleichen neutralen Farbe hinterlegt, egal von welchem Schreiber die Veränderungen durchgeführt worden sind. Die Tester sprachen sich einhellig dafür aus, dass diese Veränderungen in der Farbe des Schreibenden hinterlegt werden sollten. Die Anzeige des Sichtbarkeitsbereiches der Driver wurde oft nicht angezeigt, vor allem am Anfang einer Sitzung, und war auf den Host, also dem einladenden Teilnehmer, beschränkt. Des Weiteren wurden die jeweiligen Cursorpositionen der Driver nicht übermittelt, obwohl dies laut Benutzerdokumentation so geschehen sollte.

[4] http://www.techsmith.de/camtasia.asp
[5] http://recordmydesktop.sourceforge.net

Die Tester gaben in der späteren Diskussion an, dass sie sich gewünscht hätten, dass der Host seine Schreibrechte abgeben kann. Als störend wurde die teilweise starke Einschränkung von Funktionen der Entwicklungsumgebung empfunden. So wurde zum Beispiel die Auswahlliste, die Eclipse dem Benutzer für die Autovervollständigung anzeigt, sofort wieder geschlossen, sobald ein anderer Driver etwas geschrieben hat. Des Weiteren wurde bei der Verwendung der Undo-Funktion nicht nur die eigenen, sondern die Veränderungen von allen Schreibern rückgängig gemacht. Dies führte im schlechtesten Fall dazu, dass Veränderungen nur pro Buchstabe rückgängig gemacht werden konnten.

Es gab einige Versagen Dateien betreffend. So kam einmal am Anfang einer Programmiersitzung die Meldung, dass eine bestimmte Datei nicht gefunden werden konnte. Ein anderes Mal kam innerhalb der Sitzung die Meldung, dass sich der Inhalt der Datei auf dem lokalen Dateisystem verändert habe und es erschien die Frage, ob die Datei neu geladen werden soll oder nicht. Nach dem Beenden einer Sitzung konnten die Teilnehmer, die zum Schluss der Sitzung die Rolle Observer hatten, die Dateien des Projektes nicht mehr löschen.

Nach erfolgreichem Einladungsprozess konnte man keine weiteren Teilnehmer zu der Programmiersitzung einladen, da der entsprechende Knopf im Einladungsdialog deaktiviert war. Die Tester wünschten sich außerdem die Möglichkeit, mehrere Kontakte auf ein Mal zu einer Sitzung einladen zu können.

In dem *View Shared Project* Session waren die Rollenangaben teilweise falsch und wurden bei den einzelnen Teilnehmern unterschiedlich angezeigt. Wenn man während der Sitzung vom Observer zu einem Driver wechselte, wurde der Verfolgermodus nicht automatisch verlassen. Die Tester äußerten zudem, dass sie sich einen Fortschrittsbalken oder eine andere Art des Feedbacks wünschten, wenn der Benutzer die Verbindung zum Jabber-Server herstellt. Es wurde des Weiteren bemerkt, dass die in allen Dialogen vorhandenen Hilfe-Knöpfe funktionslos waren.

Die Tester haben weiter bemängelt, dass beim Empfangen einer neuen Textnachricht der View *Chat* nicht automatisch aktiviert wurde und dass im Chat die einzelnen Nachrichten nicht mit Zeitangaben versehen waren. Während des Einladungsprozesses konnte der Chat nicht verwendet werden, was ein Teilnehmer sich aber gewünscht hätte.

Schlussfolgerungen aus dem ersten Test Das Problem, dass nach kürzester Zeit die Kopien der Dokumente bei den einzelnen Teilnehmern einer Programmiersitzung nicht mehr konsistent waren, wurde als schwerwiegendstes Versagen der Software identifiziert. Dies wurde für die anstehende Phase des Lokalisierens und Ausbesserns der verantwortlichen Defekte mit der höchsten Priorität versehen.

Im Bereich Risikomanagement hatte ich mit den Betreuern vereinbart, dass ich eine Warnmeldung abgeben würde, wenn innerhalb des nächsten Monats nicht erkennbar wäre, dass dieses Problem in Griff zu bekommen sei. In diesem Fall hätte man durch die Früherkennung dieses Risikos geeignete Gegenmaßnahmen ergreifen können, um den Erfolg des „Projektes“ Diplomarbeit nicht zu gefährden.

6.1.2 Zweites Testexperiment

Der Schwerpunkt in der ersten Implementierungsphase (Iteration 4-5, siehe Kapitel 7.1) war das Beheben von Defekten, welche die Konsistenz der Kopien von den gemeinsam bearbeiteten Dokumenten bedrohten. Zusätzlich wurde die Übertragung von Dateien über eine Direktverbindung zwischen den Teilnehmern neu implementiert (siehe Kapitel 7.3). Als Abschluss dieser sechs Wochen dauernden Iteration stand ein zweites Testexperiment mit externen Programmierern, die den aktuellen Stand der Software überprüfen sollten.

Technische Umsetzung Als Tester kamen die gleichen Personen wie bei dem ersten Test zum Einsatz. Bei der technischen Umsetzung wurde diesmal statt einem eigenen lokalen Netzwerk die Infrastruktur des Informatik-Instituts der Freien Universität genutzt. Es wurden sowohl Computer des Instituts als auch Laptops der Tester verwendet. Die Laptops waren mittels WLAN und einer VPN-Verbindung mit den anderen Rechnern verbunden. Als Jabber-Server kam diesmal ein im Internet befindlicher, öffentlicher Server zum Einsatz. Die Abbildung 9 stellt den technischen Aufbau des Experimentes dar.

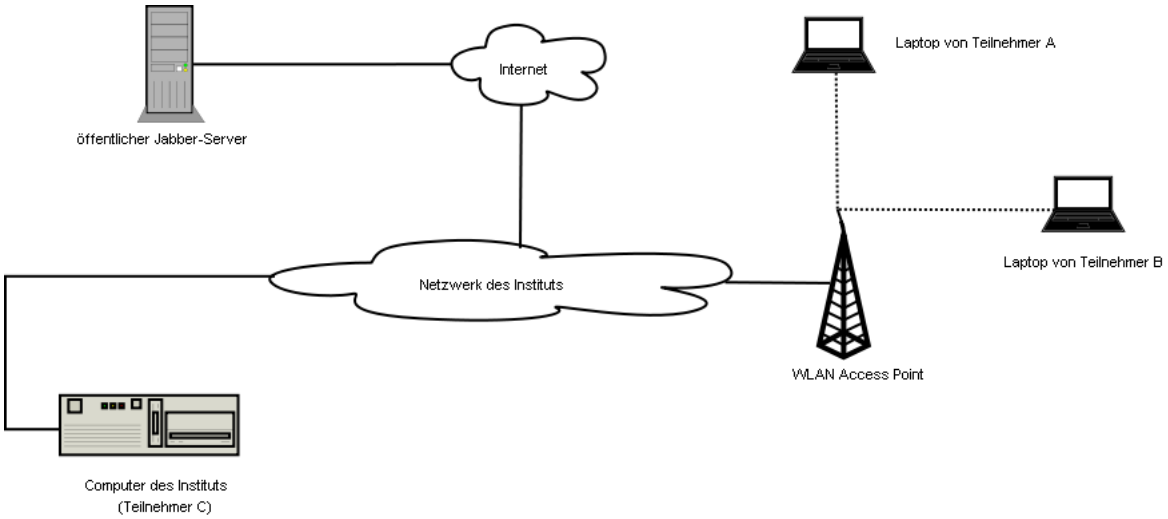

Abbildung 9: Topologie des im zweiten Test verwendeten Netzwerkes

Von der technischen Seite aus gesehen, wurde zwar ein realistischeres Szenario geschaffen, dennoch befanden sich die Teilnehmer wieder alle in einem Raum. Somit konnte das Experiment besser beobachtet werden. Da es auch bei diesem zweiten Test hauptsächlich um das Testen der Software und weniger um eine empirische Untersuchung ging, war diese unrealistische Durchführung im Sinne der Zielsetzung.

Aufgabenstellung Da die letzte Aufgabe des ersten Tests („Why Johnny Can't Count") [34] aufgrund der schlechten Stabilität nicht komplett gelöst werden konnte (siehe Kapitel 6.1.1), wurde diese Aufgabe im zweiten Test wiederholt. Ziel dieser Wiederholung war es zu testen, ob die Stabilität nun aufgrund der Tätigkeiten der letzten Wochen aus dem Bereich der analytischen Qualitätssicherung gegeben ist. Die Aufgabe wurde gleichzeitig von zwei Drivern mit einem zusätzlichen Observer bearbeitet.

Mit der zweiten und dritten Aufgabe sollte getestet werden, ob die Unterstützung von Dateioperationen und Refaktorisierungen in Saros ausreichend gegeben ist. Die Unterstützung solcher Operationen war noch kein Bestandteil vorangegangener Tests und somit dienten diese Aufgaben der Entdeckung von weiterem Versagen der Software. Für diese Aufgaben wurde den Programmierern der Quelltext von Saros gegeben, damit diese verschiedene vorgegebene Veränderungen daran durchführen. Diese Veränderungen waren unter anderem:

- Verschieben einer Klasse
- Umbenennen eines Interfaces
- Umbenennen einer Methode
- Extrahieren eines Interfaces für alle Methoden einer Klasse
- Löschen einer Klasse
- Löschen eines Paketes (mit und ohne Unterpaketen)
- Löschen des gesamten Projektes

In der zweiten Aufgabe wurden diese Veränderungen hintereinander durchgeführt. Bei der dritten Aufgabe führten die beiden Driver die Veränderungen, die teilweise die gleichen Dateien betrafen, gleichzeitig durch.

In einem gesonderten Teil des Tests wurde versucht, ob ein weiterer Tester von Zuhause aus eine Programmiersitzung mit einem anderen, im Institut sitzenden, Teilnehmer initiieren konnte.

Ergebnisse des zweiten Test Das Ergebnis der ersten Aufgabe war erfreulich, da in einer 30-minütigen Programmiersitzung ohne jegliche Komplikationen gearbeitet werden konnte. Die gleichzeitige Bearbeitung von zwei Drivern führte zu keinen Inkonsistenzen mehr. Es wurde lediglich von den Testern, wie schon im ersten Test, bemängelt, dass die Undo-Funktion nicht sinnvoll benutzbar war, sowie dass die Autovervollständigung von Eclipse durch den anderen Driver gestört wurde.

In der ersten Aufgabe wurden aber keine Dateioperationen verwendet. Die Bearbeitung der zweiten und dritten Aufgabe brachte zutage, dass diese von Saros noch sehr schlecht unterstützt wurden. So konnten die Dateioperationen zum Beispiel auch von Observern durchgeführt werden. Das meiste Versagen passierte, wenn diese nebenläufig ausgeführt worden sind. Auch die von mir neu implementierte erweiterte Konsistenzsicherung (siehe Kapitel 7.4) führte jedoch zu Programmabstürzen beim Verschieben und Löschen von Dateien.

Der Versuch in dem gesonderten Teil des Experimentes, zwischen zwei Teilnehmern in völlig unterschiedlichen Netzwerken Saros zu verwenden, schlug fehl, da keine Dateiverschickung möglich war.

Schlussfolgerungen aus dem zweiten Test Das zweite Testexperiment zeigte zum einem, dass die Bemühungen die Inkonsistenzen verursachenden Defekte zu beseitigen von Erfolg gekrönt waren. Dadurch war die Verwendung der Mehrschreiberfunktionalität von Saros, zumindest wenn keine Dateioperationen benutzt wurden, nun möglich.

Zum anderen zeigte der Test aber auch, dass die Neuimplementierung der Dateiverschickung noch fehlerbehaftet war. So gelang das Versenden von Dateien teilweise gar nicht, wenn keine Direktverbindung aufgebaut werden konnte. In diesen Fällen müsste aber zumindest das Zurückfallen auf das Versenden über den Jabber-Serververbund funktionieren, was aber im Test nicht glückte.

Des Weiteren zeigte der Test, dass die Verwendung von Dateioperationen zu erheblichen Problemen führte, die zwar teilweise auch durch die von mir nicht ganz korrekt implementierte erweiterte Konsistenzsicherung hervorgerufen, aber hauptsächlich durch das Fehlen von exklusiven Schreibsperren verursacht wurden.

Bemerkung: Das Arbeitspaket der Awareness-Informationen (insbesondere eine geeignete Darstellung bei mehreren Drivern) war zu diesem Zeitpunkt noch nicht bearbeitet worden, da dieses eine niedrigere Priorität besaß, und blieb in diesem Test somit unberücksichtigt.

6.2 Defekttests

Die allein durchgeführten Defekttests bestanden aus einer Mischung aus Funktionstests (*Black-Box-Tests*) und Strukturtests (*White-Box-Tests*). Zusätzlich wurden einige Stress- und Lasttests durchgeführt, um zu untersuchen, wie schnell das System an die Grenzen seiner Leistungsfähigkeit gelangt. Im Folgenden gehe ich nur auf die Funktionstests und Strukturtests ein.

6.2.1 Funktionstests

Die Funktionstests wurden aus den zuvor aufgestellten funktionalen Anforderungen (siehe Kapitel 5.1.1) und Anwendungsfällen abgeleitet. Teilweise wurden auch im Projekt bereits bestehende Testpläne verwendet.

Die ersten Testfälle ergaben sich aus den in den ersten Wochen erarbeiteten Anwendungsfällen. Ich erläutere mein Vorgehen beispielhaft an zwei Anwendungsfällen, für weitere Anwendungsfälle siehe Anhang B.

Schauen wir uns nun zwei Anwendungsfälle genauer an. Zunächst den Anwendungsfall *Projekt veröffentlichen*.

Use Case 1

Name: Projekt veröffentlichen

Primärakteur: Host

Stakeholder und Interessen:

- Host: möchte eine verteilte Programmiersitzung erstellen

Vorbedingungen:

- Saros muss korrekt installiert sein
- Host muss mit Kommunikationsnetz verbunden sein (Jabber)

Nachbedingungen:

- verteilte Programmiersitzung ist erstellt
- Host hat zunächst die Rolle Driver und wird im View *Shared Project Session* als solcher angezeigt
- die Möglichkeit ein weiteres Projekt zu veröffentlichen im ist Kontextmenü deaktiviert (ausgegraut)

Standardablauf:

1. Host wählt im Kontextmenü eines Projektes *Share Project* aus
2. Einladungsdialog wird angezeigt
3. Teilnehmer werden eingeladen (siehe Use Case *Teilnehmer einladen*)

Erweiterungen / alternative Abläufe: keine

spezielle Anforderungen: keine

Häufigkeit des Auftretens: mittel

Aus diesem Anwendungsfall kann nun leicht ein Testfall abgeleitet werden. Zunächst muss für die Vorbereitung des Testfalls Saros installiert werden und zum Jabber-Serververbund eine Verbindung aufgestellt werden. Dadurch wird implizit als Vorbedingung gefordert, dass bereits ein Account auf einem Jabber-Server existieren muss.

Nach den Vorbereitungen für den Testfall wird nun der Standardablauf des Anwendungsfalls durchgeführt. Der Tester klickt mit der rechten Maustaste auf ein existierendes Projekt im View *Package Explorer* oder *Navigator* oder in einem anderen View, in dem die im Workspace befindlichen Projekte dargestellt werden. In dem sich öffnenden Kontextmenü wählt dieser nun den Punkt *Share Project* aus.

Danach muss sich ein Einladungsdialog öffnen, in dem die verfügbaren Kontakte dargestellt werden. Nun wählt der Tester einen Teilnehmer aus und initiiert einen Einladungsprozess. Dieser Prozess wird im folgenden Anwendungsfall *Teilnehmer einladen* beschrieben.

Use Case 2

Name: Teilnehmer einladen

Primärakteur: Host

Stakeholder und Interessen:

- Host: möchte Teilnehmer zu einer verteilten Programmiersitzung einladen
- Teilnehmer: möchten an der Programmiersitzung teilnehmen

Vorbedingungen:

- Projekt muss bereits veröffentlicht sein (siehe Use Case *Projekt veröffentlichen*)

Nachbedingungen (bei Erfolg):

- neue Teilnehmer nehmen an der Programmiersitzung teil und werden im View *Shared Project Session* angezeigt
- neue Teilnehmer haben zunächst die Rolle Observer

Nachbedingungen (bei Misserfolg):

- falls es bei der Synchronisation Probleme gab, muss der Originalzustand (vor der Einladung) hergestellt werden

Standardablauf:

1. Host öffnet den Einladungsdialog (Button im View *Shared Project Session*)
2. Liste von verfügbaren Teilnehmern wird angezeigt
3. Host wählt den einzuladenden Teilnehmer aus und klickt auf den Button *Invite*
4. dem einzuladenden Teilnehmer wird eine Einladung geschickt und in der Liste wird der Status (warte auf Bestätigung) angezeigt
5. eingeladene Teilnehmer akzeptiert die Einladung
6. Dateiliste aller Dateien im Projekt werden zum eingeladenen Teilnehmer verschickt (Status in der Liste wird aktualisiert)
7. eingeladener Teilnehmer wählt aus, ob in seinem Workspace ein neues Projekt angelegt oder ob ein bestehendes Projekt als Ausgangsbasis genommen werden soll
8. Synchronisation der Dateien findet statt, so dass auf beiden Seiten der gleiche Datenbestand vorliegt (Status in der Liste wird aktualisiert)
9. nach erfolgreicher Synchronisation ist der Einladungsvorgang abgeschlossen und in der Liste wird als Status der Erfolg der Einladung angezeigt
10. Host wählt einen weiteren Teilnehmer aus (springe zu Schritt 3) oder schließt den Einladungsdialog

Erweiterungen / alternative Abläufe:

- Alternativer Anfang 1: gerade zuvor wurde ein Projekt veröffentlicht (Use Case 1) und der Einladungsdialog ist dadurch bereits geöffnet (Schritt 1 entfällt)
- Alternativer Anfang 2: im Kontextmenü des Teilnehmers (View *Roster*) wurde *Invite user to shared project* ausgewählt (Schritte 1-3 entfallen)
- 7a: Der eingeladene Teilnehmer wählt ein bestehendes Projekt als Ausgangsbasis und lässt für die verteilte Programmiersitzung eine Kopie des Projektes erstellen.
- 7b: Der eingeladene Teilnehmer lässt vom Werkzeug ein bestehendes Projekt finden, dass vom Datenbestand am ähnlichsten ist. Der Grad der Übereinstimmung wird angezeigt. Danach hat er alle Möglichkeiten wie in 7/7a beschrieben.
- 5a: Der eingeladene Teilnehmer lehnt die Einladung ab, dies wird als Status in der Liste angezeigt und zu Schritt 10 gesprungen.

- 5b: Der Einzuladende ist bereits in einer anderen Programmiersitzung. Er wird über die Einladung informiert und kann entscheiden, ob er die Sitzung wechseln möchte.

spezielle Anforderungen:

- Für eine schnelle Synchronisation ist eine Peer-to-Peer Verbindung zwischen den Teilnehmern nötig.

Häufigkeit des Auftretens: mittel

Der Tester durchläuft nun in mehreren Tests den im Anwendungsfall beschriebenen Standardablauf sowie die alternativen Abläufe und überprüft die angegebenen Nachbedingungen. Bei den Nachbedingungen wird zwischen einem Erfolgsfall und einem Misserfolg unterschieden. Sind die Nachbedingen erfüllt, so gilt der Testfall als bestanden, andernfalls als durchgefallen.

Anmerkung: Die hier abgedruckten Anwendungsfälle sind in einer Version, wie sie in den ersten beiden Wochen erhoben wurden, das heißt vor der Durchführung der Testexperimente. Während des ersten Testexperimentes (siehe Kapitel 6.1.1) kam von den Benutzern der Wunsch auf, dass die Teilnehmer gleichzeitig eingeladen werden können sollen. Dieser alternative Ablauf wurde dann im Laufe der Implementierungsphase realisiert. Dagegen wurde der alternative Ablauf 5b noch nicht realisiert, da diese Funktionalität eine geringe Wichtigkeit besitzt.

6.2.2 Strukturtests

Für die Bestimmung von einigen Tests war es erforderlich, die Implementierung der Funktionalität zu kennen, um so geeignete Testfälle zu identifizieren.

Nur mit dem Wissen, wie der Jupiter Algorithmus für die Nebenläufigkeitskontrolle im Einzelnen funktioniert, konnte ich verschiedene Testfälle aufstellen, wie zum Beispiel:

- Editieren von zwei Teilnehmern in einer Zeile
- Editieren von zwei Teilnehmern an exakt der gleichen Position
- der eine Teilnehmer fügt Text in einer Textpassage ein, die ein anderer gerade löscht (siehe Splitoperation im Kapitel 5.2.1)
- usw.

Sind solche Testfälle von mir identifiziert worden, wurden diese von mir durchgeführt und bei einem Versagen die genauen Stellen im Quelltext gesucht und inspiziert. In einigen Fällen wurde ein Debugger verwendet um die Defekte zu finden, die das Versagen verursachten.

6.3 Code-Durchsichten

Bei einigen kritischen Modulen und Codeabschnitte wurde von mir eine Code-Durchsicht durchgeführt. Je nach Kontext wählte ich eine andere Perspektive bei der durchgeführten Code-Durchsicht. Einige Codeabschnitte, bei denen ich zwar eine Code-Durchsicht für nötig hielt, diese aber aus Zeitgründen nicht sofort durchführen konnte, wurden mit einem entsprechenden Kommentar (tag) versehen um diese später leicht wiederfinden zu können.

Zum Beispiel untersuchte ich die Anbindung an die Nebenläufigkeitskontrolle. Das durch das erste Testexperiment (siehe Kapitel 6.1.1) gefundene schwerwiegende Versagen bestand darin, dass die Nebenläufigkeitskontrolle nicht korrekt funktionierte und es ziemlich leicht zu nicht konsistenten Zuständen kam. Ich vermutete, dass die Ursache des Problems etwas mit einer Race Condition zu tun hatte. Daher wendete ich die Praktik der Code-Durchsicht bei den Codeabschnitten an, die den Jupiter Algorithmus in Saros integrierten, und verwendete als Perspektive eine Sicht auf Synchronisierungsaspekte. Wie sich herausstellte, war wirklich eine Race Condition die Ursache des Problems. In Kapitel 7.2.4 gehe ich darauf ein, welcher Defekt dieses Versagen verursachte und wie dieser behoben werden konnte.

7 Dokumentation der Softwareentwicklung

7.1 Iterationen

Im Folgenden werde ich meine Arbeit im Saros Projekt skizzieren. Dies hat zum einen das Ziel eine Art Erfahrungsbericht zu liefern, zum anderen wird dadurch ersichtlich, welche Teile von Saros von mir bearbeitet worden sind und welche Tätigkeiten ich ausgeführt habe.

Hierfür werde ich die einzelnen Iterationen kurz beschreiben, indem ich die Dauer, Schwerpunkte, die wichtigsten ausgeführten Tätigkeiten und die dadurch entstandenen Artefakte aufzähle.

Iteration 1:

Dauer: 2 Wochen
Schwerpunkt: Anforderungsbestimmung (siehe Kapitel 5.1)
Tätigkeiten:

- bestehende Anforderungen zusammengetragen
- Anforderungserhebung durch „Kundengespräch“ mit Betreuern
- Einigung über Anforderungen

Artefakte: Anforderungskatalog (siehe Anhang A), Anwendungsfälle (Use Cases, siehe Anhang B)

Iteration 2:

Dauer: 2 Wochen
Schwerpunkt: erstes Testexperiment (siehe Kapitel 6.1.1)
Tätigkeiten:

- Beseitigen der schwerwiegendsten Defekte, damit erstes Testexperiment durchgeführt werden kann
- Planung
- Durchführung
- Analyse der Aufzeichnungen

Artefakte: Testplan, Testbericht, Bug-Reports

Iteration 3:

Dauer: 2 Wochen
Schwerpunkt: Anforderungsanalyse (siehe Kapitel 5.2)
Tätigkeiten:

- Analysieren, wie die Netzwerkanforderungen in allen möglichen Netzwerktopologien sichergestellt werden können
- Analysieren, ob der Jupiter Algorithmus als Nebenläufigkeitskontrolle geeignet ist

Artefakte: Analysedokumente

Iteration 4:

Dauer: 2 Wochen
Schwerpunkt: Defekttests (siehe Kapitel 6.2), Defektbehebung (siehe Kapitel 7.2)
Tätigkeiten:

- Funktionstests (Black-Box-Tests)
- Strukturtests (White-Box-Tests)
- Defektlokalisierung (zum Teil mit Debugger)
- Defektbehebungen

Artefakte: Code

Iteration 5:

Dauer: 3 Wochen
Schwerpunkt: Implementierung, zweites Testexperiment (siehe Kapitel 6.1.2)
Tätigkeiten:

- Neuimplementierung der Direktverbindungen
- Implementieren der erweiterten Konsistenzsicherung (zunächst nur Erkennung von nicht konsistenten Zuständen), siehe Kapitel 7.4

Artefakte: Code, Bug-Reports

Iteration 6:

Dauer: 3 Wochen
Schwerpunkt: Implementierung, Netzwerktests
Tätigkeiten:

- Migration auf neuste stabile Version der Softwarebibliothek Smack [18]
- Implementieren einer manuellen Erlaubnisvergabe für das Hinzufügen von Kontakten
- Netzwerktest mit einem NAT-Router zum Fachbereichs-VPN

Artefakte: Code

Iteration 7:

Dauer: 2 Wochen
Schwerpunkt: Anforderungsänderungen (siehe Kapitel 5.3), Implementierung
Tätigkeiten:

- Einarbeitung von neuem Mitarbeiter Herrn Rintsch
- Implementierung einer Unterstützung von C/C++ in Saros
- Realisierung eines gebündelten Rollenwechsels

Artefakte: Code

Iteration 8:

Dauer: 2 Wochen
Schwerpunkt: Peer Review und Paarprogrammierung mit Betreuer Oezbek
Tätigkeiten:

- Struktur-Refaktorisierungen der Jingle-Komponente zusammen mit Herrn Oezbek
- Weiterentwicklung der erweiterten Konsistenzsicherung (Behebung der Inkonsistenzen durch erneute Dateiübertragung vom Host), siehe Kapitel 7.4

Artefakte: Code

Iteration 9:

Dauer: 2 Wochen
Schwerpunkt: Testsitzungen für Einsatz bei Unternehmen, Präsentation
Tätigkeiten:

- von Betreuer Salinger organisierte Testsitzungen durchgeführt
- schwerwiegendste Defekte zusammen mit Herrn Rintsch und Herrn Oezbek beseitigt
- Produktpräsentation bei einem Wirtschaftsunternehmen

Artefakte: Code, Bug-Reports

Iteration 10:

Dauer: 2 Wochen
Schwerpunkt: Mini-Fallstudie (siehe Kapitel 10)
Tätigkeiten:

- Aufstellen der Ziele und Fragen des empirischen Experimentes
- Planung (Aufbau, Aufgabenstellung, Fragebögen)
- Durchführung (2 Tage)
- Auswertung

Artefakte: Dokument über Ziele und Fragen, Fragebögen, Ergebnisse der Mini-Fallstudie (siehe Kapitel 10.2.5)

Ab der Iteration 7 hat sich der Charakter der Arbeit im Saros Projekt grundlegend geändert. Bis dahin arbeitete ich hauptsächlich alleine und führte so ziemlich alle in einem Projekt anfallenden Arbeiten selbstständig durch. Meine Betreuer standen mir dabei beratend zur Seite, Stephan Salinger bei den organisatorischen und Christopher Oezbek bei den technischen Fragen. Ab Iteration 7 wurde in einem etwas größeren Team im Projekt gearbeitet. Das Team bestand aus Herrn Salinger, der als Projektleiter fungierte, Herrn Oezbek als technischer Leiter, Herrn Rintsch als neu hinzugekommener Softwareentwickler, Frau Rosen als Ansprechpartnerin für den Kunden und Prof. Prechelt als Chef der Arbeitsgruppe. Alle haben ihren Teil für den erfolgreichen Verlauf des Projektes geleistet.

Die angewendeten Praktiken bei der Softwareentwicklung haben sich stark verändert, hin zu mehr Kommunikation und kollaborativer Softwareentwicklung. Es kamen dabei viele Praktiken der agilen Softwareentwicklung zum Einsatz, wie Paarprogrammierung, kollektives Eigentum, permanente Integration, Refaktorisierungen und Codierrichtlinien. Ich arbeitete ab diesem Zeitpunkt zusammen mit Marc Rintsch an der Entwicklung der Software. Diese Zusammenarbeit war sehr gut und er hat sich schnell zurechtgefunden. Er übernahm zunächst das Arbeitspaket der Awareness-Informationen, da dieses von mir noch nicht bearbeitet worden war. Zum Ende meiner Arbeit hin übernahm er immer mehr Verantwortung, so dass eine fließende und reibungslose Übergabe der Entwicklungsarbeiten an ihn stattfand.

Ab der Iteration 8 hat sich Herr Oezbek stark an den Entwicklungsarbeiten beteiligt. Wir haben in vielen und langen Paarprogrammier-Sitzungen zusammengearbeitet. Ich empfand die Zusammenarbeit mit ihm als sehr angenehm und ich muss sagen, dass ich eine Menge von ihm gelernt habe. Dies hat mir zum erneuten Male gezeigt, dass die Arbeit in einem Paar mir mehr Spaß macht, motivierender ist, es einen Wissenstransfer gibt und ich fest der Überzeugung bin, dass sich dies auch positiv auf die Qualität des Codes/Entwurfs auswirkt. Wir benutzten Saros ein paar Mal für die verteilte Paarprogrammierung. Nicht nur an dem Werkzeug zu programmieren, sondern es auch für die eigene Entwicklung einzusetzen, war von nicht zu unterschätzendem Wert, da man so zusätzlich die Rolle des Benutzers einnahm.

7.2 Defektbehebungen

In diesem Kapitel werde ich auf einige der gefunden Defekte und deren Behebung eingehen. In Kapitel 9 versuche ich dann die gefundenen Defekte zu klassifizieren, indem ich Kategorien für die am häufigsten gemachten Fehler aufstelle.

7.2.1 Unterschiedliche Zeilentrenner

Mit Saros sollen auch Teilnehmer miteinander programmieren können, die unterschiedliche Betriebssysteme verwenden, lediglich ein Eclipse wird vorausgesetzt. Das Problem dabei ist, dass unterschiedliche Betriebssysteme standardmäßig unterschiedliche Zeichen für ein Zeilenende verwenden. Die Betriebssysteme benutzen dafür entweder das Steuerzeichen für einen Wagenrücklauf (*Carriage Return*), das Steuerzeichen für einen Zeilenvorschub (*Line Feed*) oder beide aufeinander folgend. Dass für ein Zeilenende zwei verschiedene Steuerzeichen existieren, ist historisch bedingt und stammt aus einer Zeit, als ein Zeilenumbruch noch bei einigen Ausgabegeräten in zwei getrennten Schritten ausgeführt worden ist.

Das damit einhergehende Problem in Saros ist nun, dass die Änderungsinformationen, die an die anderen Teilnehmer verschickt werden, sich immer auf den Index innerhalb des Dokumentes beziehen. Wenn jetzt ein Teilnehmer ein Betriebssystem benutzt, das für einen Zeilenumbruch zwei Steuerzeichen verwendet (zum Beispiel Windows), ein anderer aber ein Betriebssystem, welches dafür nur ein Steuerzeichen verwendet (zum Beispiel Linux), dann führt dies durch das Anwenden einer übertragenen entfernten Veränderung am Dokument zu einem nicht konsistenten Zustand.

Da die Teilnehmer des ersten Testexperimentes (siehe Kapitel 6.1.1) unterschiedliche Betriebssysteme (Windows und Linux) auf ihren selbst mitgebrachten Laptops verwendeten, führte dieser Defekt in diesem Experiment zu erheblichen Versagen.

Die Lösung des Problems bestand nun darin, immer grundsätzlich nur das Steuerzeichen für einen Zeilenvorschub (Line Feed) für einen Zeilenumbruch zu verwenden und vor der Verwendung eines jeden Dokumentes zu überprüfen, ob als Zeilenumbruch andere Zeichen verwendet wurden und gegebenenfalls das komplette Dokument zu konvertieren. Aus Gründen der Performanz werden nicht am Anfang einer Programmiersitzung alle auf der Seite vorliegenden Dokumente konvertiert, sondern erst wenn ein Dokument das erste Mal verwendet wird.

7.2.2 Das Problem mit dem Endlos-Zyklus

Ich sollte am Anfang meiner Arbeit ein erstes Testexperiment der von mir nicht bearbeiteten Version durchführen. Der Zustand der Software ließ dies allerdings nicht zu, in Kapitel 6.1.1 gehe ich auf diesen Umstand etwas mehr ein. Das Versagen der Software, welches die Durchführung eines ersten Testexperimentes verhinderte, war das Problem, dass das Betätigen der Eingabetaste für einen Zeilenumbruch einen Endlos-Zyklus verursachte und bei allen Teilnehmern unendlich viele Zeilen eingefügt worden sind.

Die Ursache des Versagens ließ sich ziemlich schnell auf die Klasse *EditorManager* zurückführen, die für die Verwaltung und Interaktion mit den Editoren in der Eclipse Workbench verantwortlich ist. Diese Klasse ist Beobachter (Observer) von den Dokumenten, für die Editoren geöffnet sind. Falls ein Dokument verändert wurde, wird der *EditorManager* darüber informiert und dieser erstellt für die Veränderung ein entsprechendes Objekt für die beobachtete Aktivität. Dieses Objekt wird dann, eventuell durch den Umweg über die Nebenläufigkeitskontrolle, zu den anderen Teilnehmern verschickt.

Wird eine Aktivität von einem anderen Teilnehmer empfangen, so gelangt diese ebenfalls zum *EditorManager*, der diese von einem entfernten Teilnehmer durchgeführte Veränderung am lokalen Editor anwendet. Das grundsätzliche Problem besteht nun darin, dass durch diese Veränderung wiederum der *EditorManager* in seiner Rolle als Observer des Dokumentes über diese Veränderung informiert wird, der daraus wieder ein Objekt für die Aktivität erstellen möchte, um dieses an die anderen Teilnehmer verschicken zu lassen. Dadurch haben wir einen Zyklus, der ohne einen zusätzlichen Mechanismus, der dieses Problem verhindert, endlos so weiter geht.

Hierfür wird in Saros sich die von einem anderen Teilnehmer stammende Aktivität gemerkt und bei der nächsten Veränderung mittels einer Vergleichsmethode *sameLike* überprüft, ob diese Aktivität gleich der zuvor ausgeführten Aktivität ist. Dieser Mechanismus war fehlerbehaftet und führte gleich zu mehreren Problemen.

Vor der Ausführung einer entfernten Aktivität wurde versucht, das Problem mit den unterschiedlichen Steuerzeichen für den Zeilenumbruch in Griff zu bekommen, indem einfach alle Vorkommnisse eines Zeilentrenners durch den im Dokument verwendeten Zeilentrenner ersetzt worden sind. Leider aber nur in dem Text, der in das Dokument eingefügt wird, nicht in der gemerkten Aktivität. Somit ergab die Überprüfung mittels der *sameLike* Metho-

de, dass es sich bei der beobachteten Veränderung um eine neue vom Benutzer stammende Veränderung handele und veranlasste, dass diese zu den anderen Teilnehmern übertragen wird. Damit entstand der beobachtete Endlos-Zyklus, der immer wieder neue Zeilen hinzufügte.

Eine schnelle Lösung bestand darin, diese textuelle Ersetzung nicht nur in dem einzufügenden Text, sondern auch in der gemerkten Aktivität durchzuführen. Dies löste zumindest das Problem des Endlos-Zyklus. Mir der Tatsache bewusst, dass es sich hierbei nur um einen „Hot-Fix“ handelte, führte ich nun das erste Testexperiment durch. Mir war zu diesem Zeitpunkt aber nicht bewusst, wie viel weitreichender das Problem mit den unterschiedlichen Zeilentrennern war (siehe Kapitel 7.2.1).

Dieser Mechanismus hatte einen weiteren Defekt. Dieser Defekt äußerte sich darin, dass wenn zwei verschiedene Driver den gleichen Buchstaben an derselben Stelle direkt hintereinander einfügten, bei einem der Teilnehmer der Buchstabe nur genau ein Mal eingefügt worden ist. Dieses Versagen wurde ebenfalls durch den oben dargestellten Mechanismus verursacht, da die gemerkte Aktivität bei der nachfolgend beobachteten Veränderung nicht wieder gelöscht und somit die darauf folgende lokale Veränderung fälschlicherweise auf die zuvor entfernte Ativität zurückgeführt worden ist.

Historisch betrachtet ist dieser Defekt ein Beispiel dafür, dass bestehender Code bei der Einführung von der Mehrschreiberfunktionalität nicht korrekt modifiziert wurde. Dieser Fehler hat sich an etlichen anderen Stellen wiedergefunden, ich gehe in Kapitel 9.5 etwas mehr auf dieses Thema ein.

7.2.3 Anwendung der Aktivitäten in falschen Editoren

Wenn es mehr als einen Driver in einer Programmiersitzung gab, wurden die Aktivitäten von den entfernten Teilnehmern teilweise nicht in dem richtigen Editor angewendet. Dieser Defekt hatte als Ursache, dass die Information, auf welchen Editor beziehungsweise auf welches Dokument sich die Aktivität bezieht, einfach nicht mit übertragen worden ist. Dies war vor der Einführung der Mehrschreiberfunktionalität auch nicht nötig, da die Observer zu jedem Zeitpunkt wussten, in welchem Editor sich der Driver befand. Dieser Defekt ist also wieder ein Beispiel dafür, dass mit der Einführung der Mehrschreiberfunktionalität notwendige Veränderungen am Code nicht vorgenommen worden sind. Zusätzlich kann man sich aber auch die Frage stellen, warum diese Information in Hinblick auf mögliche spätere Erweiterungen der Software nicht schon vorher übertragen worden ist. Dies kann man sich eventuell mit der beschränkten zur Verfügung stehenden Bandbreite des Jabber-Serververbundes erklären, obwohl diese paar zusätzlichen Bytes sicherlich nicht besonders geschadet hätten.

7.2.4 Falsche Synchronisation bei der Integration des Jupiter Algorithmus

Das erste Testexperiment (siehe Kapitel 6.1.1) brachte so viele und so schnell Versagen bezüglich der Konsistenzsicherung zutage, dass man sich in der Tat fragen musste, ob die Nebenläufigkeitskontrolle überhaupt ihre Aufgabe erfüllt. Da es einige JUnit-Tests im Projekt

gab, die den Eindruck erweckten, dass die Implementierung des Jupiter Algorithmus funktioniere, vermutete ich Race Conditions als Ursache des Problems. Daher unterzog ich den Quelltext, der für die Integration des Jupiter Algorithmus verantwortlich war, einer Code-Durchsicht mit dem Fokus auf Synchronisationsaspekte. Wie sich herausstellte, war in der Tat ein Synchronisationsproblem die Ursache für das Versagen der Nebenläufigkeitskontrolle.

Es musste sichergestellt werden, dass zwischen dem Berechnen einer Transformationsoperation für eine entfernte Aktivität und deren Anwendung keine Veränderungen am Dokument von Seiten des Benutzers stattfindet. Das Problem wurde durch einen wechselseitigen Ausschluss (mutual exclusion) gelöst, indem der entsprechende Codeabschnitt in dem (einzigen) GUI-Thread ausgeführt worden ist.

Dieser Defekt ist ein Beispiel von falscher Synchronisation. In Kapitel 9.2 gehe ich auf häufig vorkommende Defekte bezüglich der Synchronisation der in Saros verwendeten Threads ein.

7.3 Dateiübertragung über Direktverbindungen

Für den Aufbau von Direktverbindungen zwischen den Teilnehmern wird in Saros die Java-Bibliothek Smack [18] verwendet, welche die Jingle-Erweiterung (XEP-0166) [15] von XMPP implementiert. Für nähere Informationen über das Nachrichtenprotokoll XMPP und den verwendeten Erweiterungen siehe Kapitel 4.2.

Der in der Diplomarbeit DPPIII [11] begangene Fehler war, dass die Dateiübertragung nur mittels Java-Sockets auf TCP Basis realisiert wurde. Wie in Kapitel 5.2.2 bereits dargestellt, kann aber nur bei der Verwendung von UDP als Transportprotokoll eine NAT-Traversierung sichergestellt werden, da Smack die Verfahren ICE/STUN dafür verwendet.

Die Aufgabe bestand daher darin, eine Implementierung der Dateiübertragung mittels UDP als Transportprotokoll zu realisieren. Das Problem dabei ist, dass UDP nur einen verbindungslosen, nicht zuverlässigen Übertragungsdienst bereitstellt. Daher muss auf Anwendungsschicht dafür gesorgt werden, dass Pakete nicht verloren gehen, doppelt ankommen oder sich überholen.

Ich suchte nach bereits bestehenden Lösungen, die ich wiederverwenden könnte, um diese Korrekturmaßnahmen nicht selbst implementieren zu müssen. Ich fand eine Implementierung des LimeWire Projektes[6]. In dem Projekt wird eine in Java geschriebene Software für Filesharing entwickelt, das ein Peer-To-Peer-Netz verwendet. Bei diesem Projekt handelt es sich um ein Open Source Projekt und die Software steht unter der GNU General Public License (GPL). In dieser Software existiert eine Komponente namens *rudp* (reliable UDP), die einen zuverlässigen Kommunikationskanal auf Basis von UDP als Transportprotokoll zur Verfügung stellt. Durch Beiträge in Internetforen erfuhr ich, dass die Architektur von LimeWire gut modularisiert ist und vor einiger Zeit eine Refaktorisierung stattgefunden hat, um anderen Projekten eine einfachere Wiederverwendung dieser Komponente zu ermöglichen.

[6] http://www.limewire.com

In der sehr guten Entwicklerdokumentation des Projektes werden die Abhängigkeiten zwischen den Komponenten der Software erläutert. Dies ermöglichte es mir, nur die benötigten Teile der Software zu übersetzen und diese in Saros einzubinden. Die Abbildung 10 stammt aus der Entwicklerdokumentation [35] des LimeWire Projektes und zeigt die Abhängigkeiten zwischen den einzelnen Komponenten.

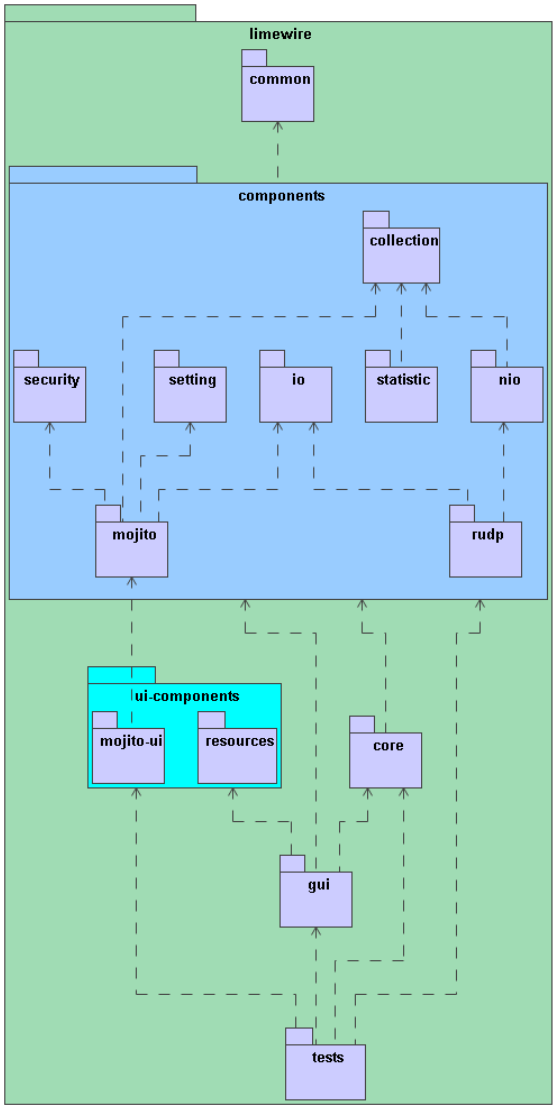

Abbildung 10: Abhängigkeiten zwischen den einzelnen Komponenten von LimeWire

Aus diesem Abhängigkeitsgraphen ist ersichtlich, dass man für die Verwendung der *rudp*-Komponente die anderen Module *io*, *nio*, *collection* und *common* der Software benötigt. Neben diesen Modulen habe ich außerdem die JUnit-Tests für die *rudp*-Komponente in die Test-Suite von Saros übernommen.

Im Folgenden gehe ich auf besonders relevante Codestellen für den Aufbau einer Direktverbindung zwischen zwei Teilnehmern ein. Diese stammen aus der Klasse *JingleFileTransferSession*, die für das Zustandekommen einer Direktverbindung und dem Übertragen von

Dateien über diese verantwortlich ist.

In diesem Kontext von besonderem Interesse ist die Initialisierungsmethode, die im Listing 2 zu sehen ist:

Listing 2: Initialisierungsmethode der Session für die Direktverbindung

```
/**
 * Initialization of the session. It tries to create sockets for both, TCP
 * and UDP. The UDP Socket is a reliable implementation from the Limewire
 * project. Documentation can be found at http://wiki.limewire.org.
 */
@Override
public void initialize() {

    [...]

    // create RUDP service
    RudpMessageDispatcher dispatcher = new RudpMessageDispatcher();
    DefaultUDPService service = new DefaultUDPService(dispatcher);
    RUDPMessageFactory factory = new DefaultMessageFactory();
    udpSelectorProvider = new UDPSelectorProvider(new DefaultRUDPContext(
        factory, NIODispatcher.instance().getTransportListener(), service,
        new DefaultRUDPSettings()));
    UDPMultiplexor udpMultiplexor = udpSelectorProvider.openSelector();
    dispatcher.setUDPMultiplexor(udpMultiplexor);
    NIODispatcher.instance().registerSelector(udpMultiplexor,
        udpSelectorProvider.getUDPSocketChannelClass());

    try {
        service.start(localPort);
    } catch (IOException e) {
        logger.error("Jingle [" + connectTo.getName()
            + "] Failed to create RUDP service");
    }


    if (getJingleSession().getInitiator().equals(
        getJingleSession().getConnection().getUser())) {

        initializeAsServer();
    } else {
        initializeAsClient();
    }
}
```

Zunächst wird bei beiden Seiten der RUDP Service erstellt, der einem den Dienst für einen zuverlässigen Kommunikationskanal mit UDP ermöglicht. Danach wird unterschieden, welche der beiden Seiten die Direktverbindung initiiert hat. Die initiierende Seite übernimmt die Rolle des Servers und die andere Seite (Client) baut eine Verbindung zu diesem auf. Die folgenden Listings zeigen die weiteren Schritte der Initialisierung, die entweder in der Methode *initializeAsClient* als Client oder in der Methode *initializeAsServer* als Server stattfinden.

Listing 3: Initialisierung des Servers

```
protected void initializeAsServer() {

    ArrayList<SocketCreator> creators = new ArrayList<SocketCreator>(2);

    creators.add(new SocketCreator(NetTransferMode.JINGLETCP) {

        public Socket call() throws Exception {

            ServerSocket serverSocket = new ServerSocket(localPort);
            serverSocket.setSoTimeout(30000);

            return serverSocket.accept();
        }
    });

    creators.add(new SocketCreator(NetTransferMode.JINGLEUDP) {

        public Socket call() throws Exception {
            Socket usock = udpSelectorProvider.openAcceptorSocketChannel()
                .socket();
            usock.setSoTimeout(0);
            usock.connect(new InetSocketAddress(InetAddress
                .getByName(remoteIp), remotePort));
            usock.setKeepAlive(true);

            return usock;
        }
    });

    connect(creators);
}
```

Das Listing 3 zeigt die Initialisierung des Servers, bei der zwei ausführbare Exemplare vom Typ *SocketCreator* erstellt werden. Diese beiden Exemplare sind für das Erstellen von zwei verschiedenen Sockets, einmal für TCP und einmal für UDP, verantwortlich. Das Erstellen des Sockets für TCP erfolgt, wie in Java üblich, durch das Aufrufen der *accept* Methode auf einem *ServerSocket*, während das Socket für UDP mittels des Aufrufs von *udpSelectorProvider.openSocketChannel().socket()* erstellt wird. Die beiden Exemplare *creators* werden der *connect* Methode übergeben, die für die eigentliche Ausführung dieser beiden verantwortlich ist.

Listing 4: Initialisierung des Clients

```
protected void initializeAsClient() {

    ArrayList<SocketCreator> creators = new ArrayList<SocketCreator>(2);

    creators.add(SocketCreator.getWrapped(NetTransferMode.JINGLETCP, Util
        .retryEvery500ms(new Callable<Socket>() {
            public Socket call() throws Exception {
                return new Socket(remoteIp, remotePort);
```

```
                }
            }));

        creators.add(SocketCreator.getWrapped(NetTransferMode.JINGLEUDP, Util
            .delay(7500, Util.retryEvery500ms(new Callable<Socket>() {
                public Socket call() throws Exception {

                    Socket usock = udpSelectorProvider.openSocketChannel()
                        .socket();
                    usock.setSoTimeout(0);
                    usock.setKeepAlive(true);
                    usock.connect(new InetSocketAddress(InetAddress
                        .getByName(remoteIp), remotePort));
                    return usock;
                }
            }))));

        connect(creators);
    }
```

Die Initialisierung des Clients ist in Listing 4 zu sehen. Der Client versucht alle 500 ms eine Verbindung zum Server mit TCP aufzunehmen. Mit einer Verzögerung von 7,5 s versucht der Client es ebenfalls mit UDP. Dies ist nötig, da die Versuche mit TCP aufgrund der NAT-Problematik fehlschlagen können, siehe Kapitel 5.2.2.

Listing 5: Klasse *SocketCreator*

```
    abstract static class SocketCreator implements Callable<Socket> {

        SocketCreator(NetTransferMode type) {
            this.type = type;
        }

        NetTransferMode type;

        public NetTransferMode getType() {
            return this.type;
        }

        public static SocketCreator getWrapped(NetTransferMode type,
            final Callable<Socket> callable) {
            return new SocketCreator(type) {
                public Socket call() throws Exception {
                    return callable.call();
                }
            };
        }
    }
```

Die Klasse SocketCreator (siehe Listing 5) kapselt die Information des verwendeten Transportprotokolltyps und stellt eine statische Methode *getWrapped* für das Erstellen eines *SocketCreator*s zur Verfügung. Diese erhält als Argument ein ausführbares *Callable* Objekt, das für die Implementierung des Erstellens des Sockets zuständig ist.

Listing 6: Klasse *SocketCreator*

```
private void connect(Collection<SocketCreator> connects) {

    ExecutorCompletionService<Socket> completionService =
      new ExecutorCompletionService<Socket>(
        Executors.newFixedThreadPool(connects.size(),
            new NamedThreadFactory("Jingle-Connect-" + connectTo.getName()
                + "-")));

    Map<Future<Socket>, SocketCreator> futures =
      new HashMap<Future<Socket>, SocketCreator>();

    for (SocketCreator creator : connects) {
        futures.put(completionService.submit(creator), creator);
    }

    for (int i = 0; i < connects.size(); i++) {

        Future<Socket> socketFuture = null;
        try {
            socketFuture = completionService.poll(TIMEOUTSECONDS,
                TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            JingleFileTransferSession.logger.error(
                "Unexpected interrupted exception in startTrasmit", e);
        }

        if (socketFuture == null) {
            logger.debug("Jingle [" + connectTo.getName()
                + "] Could not connect with either TCP or UDP.");
            break;
        }

        this.socket = socketFuture.get();

        this.objectOutputStream = new ObjectOutputStream(socket
            .getOutputStream());
        this.objectInputStream = new ObjectInputStream(socket
            .getInputStream());

        this.receiveThread = new ReceiverThread(objectInputStream);
        this.receiveThread.start();

        this.connectionType = futures.get(socketFuture).getType();

        // Make sure the other connect-futures are canceled
        for (Future<Socket> future : futures.keySet()) {
            future.cancel(true);
        }

        return;
    }
```

```
        // Timeout, so cancel all
        for (Future<Socket> future : futures.keySet()) {
            future.cancel(true);
        }

        assert objectOutputStream == null && objectInputStream == null;
    }
```

Das Listing 6 zeigt die *connect* Methode, die für das Aufbauen der Direktverbindung zuständig ist. In dieser wird zunächst ein *ExecutorCompletionService* (Klasse der Java API) mit einem Thread-Pool fester Länge, und zwar die Anzahl der übergebenen *SocketCreator*s, erstellt. Mit der *poll* Methode auf diesem *ExecutorCompletionService* wird das *Future* Objekt von dem zuerst fertig ausgeführten *SocketCreator* geholt. Auf dem *Future* Objekt bekommen wir mittels der *get* Methode das erstellte Socket, von dem wir uns für die Übertragung der zu übermittelnden Objekte entsprechende *ObjectStreams*, sowohl für die Ausgabe als auch für die Eingabe, holen.

Falls die Erstellung des Sockets weder mit TCP noch mit UDP erfolgreich war und damit *socketFuture* gleich *null* ist (mit einem Timeout von *TIMEOUTSECONDS* Sekunden), wird mittels *break* die *for*-Schleife und damit auch die *connect* Methode verlassen, das heißt der Aufbau einer Direktverbindung war nicht erfolgreich. Bei einer späteren Dateiübertragung kommt die Rückfallstrategie zum Tragen und die Datei wird über den Jabber-Serververbund verschickt.

Anmerkungen:

1. Die Ausnahmebehandlungen (*try/catch* Blöcke) wurden für die bessere Lesbarkeit des Quelltextes weggelassen. Falls es zu einer Ausnahme bei der Erstellung eines Sockets kommt, wird mittels *continue* zur nächsten Iteration der *for*-Schleife gesprungen.

2. Der hier abgedruckte Quelltext entstand durch mehrere Refaktorisierungen, welche die Struktur des Quelltextes verbesserten. Diese Refaktorisierungen wurden zum Teil in gemeinschaftlicher Arbeit innerhalb einer Paarprogrammier-Sitzung mit meinem Betreuer Christopher Oezbek durchgeführt.

7.4 Erweiterte Konsistenzsicherung

Neben der in Kapitel 4.3 beschriebenen Nebenläufigkeitskontrolle existiert in Saros eine sogenannte erweiterte Konsistenzsicherung. Diese hat die Aufgabe in regelmäßigen Abständen zu überprüfen, ob sich die Kopien bei allen Drivern in einem konsistenten Zustand befinden. Dies ist notwendig, da nicht verhindert werden kann, dass der Benutzer eigenmächtig an Saros vorbei Änderungen an den gemeinsamen Dokumenten vornimmt. Dies kann er zum Beispiel dadurch erreichen, indem er außerhalb von Eclipse Änderungen an den Dateien vornimmt oder diese sogar komplett ersetzt.

Um trotzdem einen konsistenten Zustand sicherzustellen und damit ein weiteres Arbeiten mit Saros zu ermöglichen wird ein Watchdog (*ConsistencyWatchdog*) verwendet, der in regelmäßigen Abständen die Konsistenz der gemeinsam verwendeten Dokumenten überwacht

und gegebenenfalls die Konsistenz wiederherstellt. Zwar existierte schon vorher ein Mechanismus der erweiterten Konsistenzsicherung, diese wurde aber zu selten durchgeführt (nur beim Speichern eines Dokumentes) und war stark fehlerbehaftet. Dieser Umstand machte eine Neuimplementierung notwendig.

Der *ConsistencyWatchdog* ist als Eclipse *Job* realisiert, der beim Host, also dem einladenden Teilnehmer, läuft. Dieser erstellt für alle von Jupiter kontrollierten Dokumente ein *Document-Checksum* Objekt, das aus der Länge und dem Hash-Wert des Dokumentes besteht. Diese Informationen verschickt er alle 10 Sekunden an alle Clients im System. Das Listing 7 zeigt die Implementierung des ConsistencyWatchdogs.

Listing 7: Implementierung der Klasse *ConsistencyWatchdog*

```
/**
 * This class is an eclipse job run on the host side ONLY.
 *
 * The job computes checksums for all files currently managed by Jupiter
 * (the ConcurrentDocumentManager) and sends them to all guests.
 *
 * These will call their ConcurrentDocumentManager.check(...) method, to
 * verify that their version is correct.
 *
 * Once started with schedule() the job is scheduled to rerun every 10
 * seconds.
 *
 * @author chjacob
 */
public class ConsistencyWatchdog extends Job {

    public ConsistencyWatchdog(String name) {
        super(name);
    }

    // this map holds for all jupiter controlled documents the checksums
    private final HashMap<IPath, DocumentChecksum> docsChecksums
        = new HashMap<IPath, DocumentChecksum>();

    @Override
    protected IStatus run(IProgressMonitor monitor) {

        assert isHostSide() : "This job is intended to be run on host side!";

        [..]

        Set<IDocument> missingDocuments = new HashSet<IDocument>(
            registeredDocuments);

        // Update Checksums for all documents controlled by jupiter
        for (IPath docPath : clientDocs.keySet()) {

            // Get document
            IDocument doc = EditorManager.getDefault().getDocument(docPath);
```

```
                if (doc == null) {
                    logger.error("Can't get Document");
                    docsChecksums.remove(docPath);
                    continue;
                }

                // Update listener management
                missingDocuments.remove(doc);
                if (!registeredDocuments.contains(doc)) {
                    registeredDocuments.add(doc);
                    doc.addDocumentListener(dirtyListener);
                    dirtyDocument.add(doc);
                }

                // If document not changed, skip
                if (!dirtyDocument.contains(doc))
                    continue;

                // If no entry for this document exists create a new one
                if (docsChecksums.get(docPath) == null) {
                    DocumentChecksum c = new DocumentChecksum(docPath, doc
                        .getLength(), doc.get().hashCode());
                    docsChecksums.put(docPath, c);
                } else {
                    // else set new length and hash
                    DocumentChecksum c = docsChecksums.get(docPath);
                    c.setLength(doc.getLength());
                    c.setHash(doc.get().hashCode());
                }
            }

            // Reset dirty states
            dirtyDocument.clear();

            // Unregister all documents that are no longer there
            for (IDocument missing : missingDocuments) {
                registeredDocuments.remove(missing);
                missing.removeDocumentListener(dirtyListener);
            }

            // Send to all Clients
            if (docsChecksums.values().size() > 0) {
                Saros.getDefault().getSessionManager().getTransmitter()
                    .sendDocChecksumsToClients(docsChecksums.values());
            }

            // Reschedule the next run in 10 seconds
            schedule(10000);
            return Status.OK_STATUS;
        }
    }
```

Beim Empfangen der *DocumentChecksum*s bei den Clients, wird die Methode *performCheck* aufgerufen, die im Listing 8 abgedruckt ist. Diese überprüft für alle empfangenen *DocumentChecksum*s, ob die Länge und der Hash-Wert mit dem lokal vorliegenden Dokument übereinstimmen. Wenn dies nicht der Fall sein sollte, werden alle an dem Ereignis interessierten Klassen darüber informiert. Dies geschieht über das Objekt *inconsistencyToResolve*, das bei Änderungen des gekapselten Wertes mittels der *setValue* Methode alle registrierten Beobachter hiervon in Kenntnis setzt. Dieser sehr allgemein verwendbare Mechanismus wird in der Hilfsklasse *VariableProxy* realisiert, die von Christopher Oezbek in das Projekt eingebracht wurde.

Listing 8: Implementierung der *performCheck* Methode

```
/**
 * Checks the local documents against the given checksums.
 *
 * Use the VariableProxy getConsistenciesToResolve() to be notified if
 * inconsistencies are found or resolved.
 *
 * @param checksums
 *            the checksums to check the documents against
 *
 * @nonReentrant This method cannot be called twice at the same time.
 */
public void performCheck(List<DocumentChecksum> checksums) {

    if (checksums == null) {
        logger
            .warn("Consistency␣Check␣triggered␣without␣preceeding␣call"
                + "␣of␣setChecksums()");
        return;
    }

    ConcurrentDocumentManager.this.pathsWithWrongChecksums.clear();

    for (DocumentChecksum checksum : checksums) {
        if (isInconsistent(checksum)) {

            ConcurrentDocumentManager.this.pathsWithWrongChecksums
                .add(checksum.getPath());
        }
    }

    if (pathsWithWrongChecksums.isEmpty()) {
        if (inconsistencyToResolve.getValue()) {
            logger.debug("All␣Inconsistencies␣are␣resolved");
            inconsistencyToResolve.setValue(false);
        }
    } else {
        inconsistencyToResolve.setValue(true);
    }
}
```

Falls ein nicht konsistenter Zustand entdeckt worden ist, erhält der Benutzer eine Information darüber. Auf Wunsch des Benutzers hin wird die Behandlung dieser Ausnahme angestoßen und es wird durch eine erneute Dateiübertragung vom Host versucht, einen konsistenten Zustand wiederherzustellen. Beim Empfangen der Datei (siehe Listing 9) wird die alte Datei ersetzt und die entsprechenden Verwaltungsinformationen des Jupiter Algorithmus zurückgesetzt. Zusätzlich wird der Inhalt der neuen mit dem Inhalt der alten Datei verglichen und der Unterschied als Debug-Informationen ausgegeben. Dies hilft den Entwicklern beim Lokalisieren des Problems, falls der inkonsistente Zustand unerwartet nicht durch eine unsachgemäße Verwendung der Software, sondern durch einen Defekt in der Software entstanden ist.

Listing 9: Empfangen einer Datei im Rahmen einer Konsistenzwiederherstellung

```
public boolean receivedResource(JID from, Path path, InputStream input,
    int time) {

    log.debug("Received␣consistency␣file␣[" + from.getName() + "]␣"
        + path.toString());

    ISharedProject project = sessionManager.getSharedProject();

    // Project might have ended in between
    if (project == null)
        return false;

    final IFile file = project.getProject().getFile(path);

    if (log.isDebugEnabled()) {
        input = logDiff(log, from, path, input, file);
    }

    final InputStream toWrite = input;
    Util.runSafeSWTSync(log, new Runnable() {
        public void run() {
            FileUtil.writeFile(toWrite, file);
        }
    });

    ConcurrentDocumentManager concurrentManager = project
        .getConcurrentDocumentManager();

    // The file contents has been replaced, now reset Jupiter
    if (concurrentManager.isManagedByJupiter(path))
        concurrentManager.resetJupiterClient(path);

    // Trigger a new consistency check, so we don't have to wait for new
    // checksums from the host
    concurrentManager.checkConsistency();

    return true;
}
```

8 Offene Probleme

In diesem Kapitel werde ich die noch in der Software Saros bestehenden Probleme erläutern. Neben einer kurzen Beschreibung der einzelnen Probleme stelle ich mögliche Lösungsansätze für diese Probleme dar. Ob diese möglichen Lösungsansätze geeignet sind, die bestehenden Probleme zu lösen, sollte von der nachfolgenden Generation von Mitarbeitern im Projekt untersucht werden.

8.1 Undo/Redo-Funktionalität

Die von mir aufgestellte funktionale Anforderung FR5 (siehe Anforderungskatalog im Anhang A) fordert, dass die Undo-Funktion bei der Verwendung von Saros weiterhin funktionieren muss. Diese soll nur die eigenen und nicht die der anderen Drivern durchgeführten Veränderungen berücksichtigen. Das erste Testexperiment (siehe Kapitel 6.1.1) zeigte, dass zurzeit in Saros aber nicht nur die eigenen, sondern das Ergebnis der nebenläufigen Schreibzugriffe von der Undo-Funktion berücksichtigt werden.

Die Implementierung einer Undo-Funktion, die diesen Anforderungen genügt, ist in Gruppen-Editoren eine technische Herausforderung [36]. Ein von mir durchgeführter naiver Versuch, eine solche Funktion zu implementieren, brachte nach und nach die Schwierigkeiten eines solchen Unterfangens zutage. Das Hauptproblem liegt darin, dass die Standardimplementierung in Eclipse nur linear ist, das heißt, es können die Veränderungen nur in chronologischer Reihenfolge rückgängig gemacht werden. Außerdem ist die Implementierung indexbasiert, was bei Veränderungen von nicht lokalen Drivern eine Modifizierung von allen aufgezeichneten Undo-Operationen nötig macht.

Ich sehe zwei mögliche Ansätze, solch eine Funktion zu realisieren:

1. einen eigenen Eclipse *UndoManager* vom Typ *IDocumentUndoManager* zu implementieren
2. die Nebenläufigkeitskontrolle um die Möglichkeit von Undo als inverse Operation zu erweitern

Ich halte die Realisierung einer Undo-Funktion in Gruppen-Editoren mit mehr als einen Schreiber ohne eine Unterstützung seitens der Nebenläufigkeitskontrolle für nur schwer zu realisieren und tendiere eher zum zweiten Lösungsansatz. In der von Saros verwendeten Implementierung des Jupiter Algorithmus aus dem ACE Projekt [25] existieren sogar entsprechende Methoden *redo* und *undo*, diese werfen bei Verwendung aber nur eine *CannotUndoException* beziehungsweise eine *CannotRedoException*. In den Kommentaren zu diesen Methoden haben die Entwickler aus dem ACE Projekt vermerkt, dass Undo/Redo nicht von dieser Implementierung unterstützt wird.

Ich empfehle eine weitere Recherche darüber, wie Operational Transformation Verfahren im Allgemeinen und der Jupiter Algorithmus im Speziellen um eine Unterstützung einer Undo-Funktion erweitert werden können. Als Ausgangspunkt einer solchen Recherche könnte zum Beispiel der Artikel „Undo as Concurrent Inverse in Group Editors“ [36] von C. Sun sein.

8.2 Einschränkungen bei der Autovervollständigung

Die Funktion der Autovervollständigung in Eclipse ist grundsätzlich auch möglich, wenn Saros für eine verteile, kollaborative Softwareentwicklung genutzt wird. Allerdings kommt es zu Einschränkungen, wenn mehrere Driver an dem gleichen Dokument arbeiten. Wenn einer der Driver die Funktion der Autovervollständigung verwendet und es mehr als eine Möglichkeit der Vervollständigung existiert, dann zeigt Eclipse eine Box mit den Möglichkeiten an. Der Benutzer kann dann in dieser Box die gewünschte Vervollständigung auswählen.

Das Problem bei mehreren gleichzeitigen Schreibern ist nun, dass sobald eine von einem anderen Driver angewendete Veränderung an dem Dokument vorgenommen wird, diese Box wieder verschwindet. Dieses Problem wurde während der Testexperimente sowie der Mini-Fallstudie von Seiten der Benutzer als schwerwiegend angesehen.

Für dieses Problem habe ich noch keine konkrete Lösung gefunden. Ein möglicher Ansatz wäre, die Veränderungen am Dokument auf eine andere Art und Weise vorzunehmen, so dass Eclipse die Veränderung nicht als Benutzereingabe deutet. Der am vielversprechendste Punkt um hierfür anzusetzen, ist die Ebene der unter den Dokumenten liegenden Buffer. Ein Ansetzten unterhalb von Eclipse, also auf Betriebssystemebene, halte ich dagegen für nicht Erfolg versprechend, Eclipse würde den Benutzer dann darüber informieren, dass die Datei außerhalb von Eclipse verändert worden ist und nachfragen, wie damit umgegangen werden soll.

8.3 NAT-Traversierung

Das Aufbauen von Direktverbindungen in komplexeren Netzwerktopologien ist und bleibt ein Problem in Saros. Zwar konnte durch die Verwendung von UDP als Transportprotokoll (siehe Kapitel 7.3) eine gewisse Verbesserung erreicht werden (siehe zum Vergleich das Testprotokoll in [11], in der praktischen Umsetzung gibt es aber noch einige ungeklärte Probleme.

In der Iteration 6 habe ich einige Netzwerktests mit einem eigenen NAT-Router durchgeführt, die erste Hinweise dafür lieferten, was in der Praxis zu beachten ist. Durch die Anforderungsänderungen (siehe Kapitel 5.3) in der Iteration 7 wurden die Untersuchungen bezüglich dieser Problematik aber nicht weitergeführt. Ich gebe diese ersten Erkenntnisse hier wieder, mit dem Hinweis, dass es sich hierbei nur um erste, noch nicht zu Ende untersuchte, vorläufige Ergebnisse handelt.

- Ich hatte unterschiedliche Ergebnisse in Abhängigkeit vom verwendeten STUN-Server. Deshalb wurde von mir die Möglichkeit in Saros integriert, den zu verwendenden STUN-Server in den Einstellungen von Eclipse zu konfigurieren.
- Die Firewalls in dem verwendeten Netzwerk dürfen das STUN-Protokoll nicht blockieren (Port 3478).
- Es gibt in der Softwarebibliothek Smack [18] neben der Möglichkeit ICE als Verfahren auch die Möglichkeit direkt das STUN-Verfahren für die IP-Auflösung zu verwenden. Dies brachte teilweise bessere Ergebnisse.

Leider brachte eine Anfrage im Entwickler-Forum für die Softwarebibliothek Smack[7] keine verwertbare Unterstützung.

Als weitere Vorgehensweise empfehle ich eine genauere Analyse des Problems auf Netzwerkebene, zum Beispiel durch die Verwendung eines Sniffers. Des Weiteren könnte es für weitergehende Tests sinnvoll sein, einen eigenen STUN-Server aufzusetzen.

8.4 Sperrmechanismen bei Dateioperationen

Wenn von einem Driver eine Datei- oder Ordneroperation durchgeführt wird, während ein anderer Driver eine von dieser Operation betroffenen Datei editiert oder selbst gerade eine in Konflikt stehende Dateioperation vornimmt, kann dies zu einem Versagen der Software führen. Im Moment wird dieses Problem dadurch vermieden, dass vor einer solchen Operation überprüft wird, ob der Teilnehmer, der diese Operation durchführen möchte, gerade der einzige Driver in der Programmiersitzung ist. Wenn dies nicht der Fall sein sollte, so wird dem Benutzer eine entsprechende Warnmeldung angezeigt, der den Benutzer darauf hinweist, dass er für eine solche Operation exklusiver Driver sein sollte.

In den von mir erhobenen Anforderungen wird aber ein anderes Verhalten verlangt, siehe hierfür Use Case 9 im Anhang B. Dieser Anwendungsfall sieht vor, dass alle anderen Driver, die dieses Dokument geöffnet haben, darüber informiert werden und ihre Zustimmung abgeben müssen. Haben alle anderen Driver ihre Zustimmung gegeben (man sollte auch einen Mechanismus integrieren, der das Fortfahren auch ohne eine solche Zustimmung ermöglicht), wird diesen die Schreibberechtigung für diese Datei entzogen und erst nach der Ausführung der Operation wieder zurück gegeben.

Neben dieser im oben genanten Anwendungsfall beschriebenen Lösung mittels eines exklusiven Schreibzugriffs kann auch darüber nachgedacht werden, ob solche Konflikte bei Dateioperationen durch die Nebenläufigkeitskontrolle aufgelöst werden können. Hierzu müsste untersucht werden, ob das Verfahren der Operation Transformation hierfür geeignet ist und wie entsprechende Transformationsoperationen auszusehen hätten.

[7] http://www.igniterealtime.org/community/community/developers/smack

9 Kategorisierung der Defekte

In diesem Kapitel werde ich die gefundenen Defekte hinsichtlich der von den Entwicklern gemachten Fehlern klassifizieren und Kategorien für die am häufigsten vorkommenden Fehler aufstellen.

9.1 Ungeeignete Datenstrukturen

In dem Quelltext von Saros fanden sich etliche Stellen, an denen ungeeignete Datenstrukturen verwendet worden sind. Dabei ist nicht die fehlende Verwendung von fortgeschrittenen Datenstrukturen gemeint, sondern die falsche Verwendung der in der Java API bereits vorhandenen Datenstrukturen.

Am häufigsten wurde der Fehler begangen, keine Abbildungen als Datenstruktur zu benutzen, an Stellen wo der Einsatz von solchen, zum Beispiel in Form einer Hashmap, angebracht wäre. So wurden oft Listen verwendet, die dann komplett iteriert werden mussten, um das richtige Element zu finden. Dies führte nicht nur zu schlecht lesbarem Quelltext, sondern hatte eine teilweise erheblich verschlechterte Laufzeit zur Folge.

Fast genauso häufig, aber mit weniger schlimmen Auswirkungen, wurde statt einer Menge von Objekten eine Liste verwendet. Die Überprüfung, ob ein Objekt schon in der Liste vorhanden ist, wurde entweder gar nicht oder wieder durch eine Traversierung der gesamten Liste vorgenommen.

An einigen Stellen wurde die Verwendung von Abbildungen dadurch erschwert, dass die Definitionsmenge der Abbildung nicht aus einzelnen Objekten sondern aus Tupeln von Objekten bestand. Die Komponenten der Tupel besaßen dabei oft einen unterschiedlichen Typ. Wir verwendeten eine eigene generische Klasse für solche Paare, um damit Abbildungen mit Tupeln in der Definitionsmenge realisieren zu können. Mir ist es ein Rätsel, warum in der Java API noch keine entsprechende Klasse existiert.

9.2 Nicht korrekte oder fehlende Synchronisation

Das Versagen der Software konnte in einigen Fällen auf das Fehlen oder der nicht korrekten Verwendung von Synchronisationsmaßnahmen zurückgeführt werden. Saros besteht aus einer beachtlichen Anzahl von Threads, welche die Verwendung von solchen Maßnahmen notwendig machen.

Oft wurde zu wenig von der Möglichkeit des wechselseitigen Ausschlusses (mutual exclusion) Gebrauch gemacht oder die Threadsicherheit der verwendeten Datenstrukturen war nicht gegeben. So musste tendenziell mehr Code in kritischen Abschnitten ausgeführt und bei den verwendeten Datenstrukturen die Threadsicherheit sichergestellt werden. Mit Verklemmungen (Deadlocks) gab es dagegen wenig bis gar keine Probleme.

9.3 Falsche Verwendung von asynchronen Methoden

Ein beliebter Fehler war der falsche Umgang mit asynchronen Methoden. Viele der Methoden mit einer potenziell längeren Laufzeit sind im Eclipse Framework nicht blockierend realisiert. Als Argument wird diesen Methoden eine Klasse (meist anonym) übergeben auf der eine Callback Methode aufgerufen wird, sobald die Ausführung erfolgreich beendet worden ist. Der Fehler, der bei dem Umgang mit diesen nicht blockierenden Methoden gemacht worden ist, war nun, dass als Argument eine leere Implementierung dieser geforderten Klasse übergeben worden ist. Danach wurde fortgefahren, als wäre die Methode bereits erfolgreich beendet worden. So wurde zum Beispiel beim Erstellen einer Datei aus einem gegebenen Eingabestream nicht darauf gewartet, dass die Datei tatsächlich geschrieben worden ist, sondern sofort mit dieser einfach weiter gearbeitet. Dies geht bei kleineren Dateien auch erstaunlich oft gut, bei größeren Dateien führte dies allerdings zu Problemen.

9.4 Zu häufige Verwendung von Interfaces

In dem Quelltext von Saros wurde von der Möglichkeit Interfaces zu verwenden inflationär Gebrauch gemacht. Beim Hinzufügen einer neuen Klasse wurde grundsätzlich ebenso ein weiteres Interface eingeführt, das diese Klasse dann implementiert. Dies wurde auch dann getan, wenn es ausgeschlossen oder sehr unwahrscheinlich ist, dass es jemals eine andere Klasse geben wird, die dieses Interface implementieren wird. Dies führt nicht nur zu einer schlechten Wartbarkeit der Software, sondern erschwert auch ein Navigieren im Quelltext bei einem Code-Walkthrough.

9.5 Altlasten im Quelltext

Kommen wir nun zu dem wahrscheinlich schwerwiegendsten Problem, dass der Quelltext von Saros hatte und auch an einigen Stellen immer noch hat. Dieses Problem besteht aus der Tatsache, dass bei der Implementierung einer neuen Funktionalität oder einer Architekturänderung nicht alle im Quelltext notwendigen Veränderungen vorgenommen worden sind. Ich benutze hierfür den Begriff *Altlasten*.

Die Gründe für solche Altlasten sind höchstwahrscheinlich mannigfaltig. Im Folgenden versuche ich einige mögliche Gründe dafür zu finden.

Die meisten Gründe haben wahrscheinlich mit den häufig stattgefundenen Änderungen an den Anforderungen in der Geschichte von Saros (siehe Kapitel 3) zu tun. Für diese zum Teil massiven Anforderungsänderungen haben sich sowohl der anfängliche Entwurf als auch der verwendete Softwareprozess als nicht ganz tauglich erwiesen.

Eines der konkreten Probleme ist die hohe Fluktuation bei den Mitarbeitern und die zeitlich beschränkte Arbeit von diesen im Projekt. Dies führt dazu, dass einige notwendigen Veränderungen an dem bestehenden Code aus Zeitgründen nicht erfolgte und die Informationen darüber, welche Änderungen noch ausstehen nicht an die Nachfolger weiter gereicht worden sind. In Kapitel 9.6 gehe ich weiter auf dieses Problem im Softwareprozess ein.

Es gab teilweise erhebliche Änderungen an der Architektur, die viele notwendige Veränderungen nach sich gezogen haben. So wurde die Art der Netzwerkkommunikation durch die Einführung einer Nebenläufigkeitskontrolle für die Mehrschreiberfunktionalität stark verändert. Vor der Einführung der Nebenläufigkeitskontrolle wurden alle Aktivitäten sofort an alle anderen Teilnehmer übertragen. Durch die Verwendung des Jupiter Algorithmus (siehe Kapitel 4.3) fand die Kommunikation, zumindest für die Übertragung der Textveränderungen, nun sternförmig über den zentralen Server statt. Dies hatte zur Folge, dass viele Teile der Software nicht mehr so ohne Weiteres korrekt funktionierten und stark verändert werden mussten.

Teilweise war der Entwurf nicht für die häufigen Anforderungsänderungen flexibel genug. Zum Beispiel waren die von Saros definierten Erweiterungen des Nachrichtenprotokolls XMPP (siehe Kapitel 4.2) für die verteilte Paarprogrammierung nicht so gestaltet, dass eine Mehrschreiberfunktionalität ohne erneute Veränderungen an den XMPP-Extensions möglich war. So besaß zum Beispiel die Erweiterung für die Aktivitäten keinerlei Information darüber, auf welches Dokument sich die Aktivitäten überhaupt beziehen (siehe Kapitel 7.2.3). Diese nicht weit schauende Gestaltung des Protokolls machte einige Änderungen an vielen Stellen im Quelltext notwendig.

9.6 Probleme im Softwareprozess

Der Charakter des Arbeitens im Projekt ist maßgeblich davon beeinflusst, dass die Mitarbeiter zu meist Studenten sind, die im Rahmen einer Diplom- oder Studienarbeit an der Software arbeiten. Hierbei geht es meistens um das Erweitern der Software bezüglich eines oder mehreren Funktionalitäten. Da diese Arbeiten zeitlich beschränkt sind und es bisher nicht gelungen ist, die Studenten zu motivieren sich über die Verpflichtungen im Studium hinaus sich im Projekt zu engagieren, existiert eine hohe Fluktuation bei den Mitarbeitern und die Qualität der Software kommt nicht über einen Prototypcharakter (Proof of Concept) hinaus.

Im Folgenden werde ich einige Empfehlungen geben, wie dieses Problem angegangen werden kann. Ich tue dies, da ich der Meinung bin, dass ich als direkt Betroffener im besonderen Maße weiß, worüber ich rede:

Die Zeitintervalle, in denen die Studenten im Projekt arbeiten, sollten überlappend sein, um eine reibungslose und fließende Übergabe auf den Nachfolger zu gewährleisten. Dadurch ist eine gute Einarbeitung von neuen Mitarbeitern möglich und das Wissen über einzelne Teile des Quelltextes wird weitergegeben. Für diese Einarbeitung ist eventuell die Praktik der Paarprogrammierung sinnvoll [37].

Grundsätzlich sollte immer mehr als ein Mitarbeiter im Projekt arbeiten. Dies ermöglicht nicht nur den Austausch von Ideen um bestehende Probleme zu lösen, sondern bietet einem erst die Möglichkeit, bestimmte agile Praktiken wie die Paarprogrammierung bei der Entwicklung an Saros zu verwenden. Eventuell können ja auch externe Leute dafür gewonnen werden sich im Projekt zu engagieren. Mit diesen externen Mitarbeitern könnte verteilte Paarprogrammierung praktiziert werden. Ein passendes Werkzeug dafür würde sich bestimmt auch schell finden lassen.

Der Schwerpunkt bei den Arbeiten an Saros sollte weniger auf das Hinzufügen einer neuen Funktionalität oder der Implementierung eines neuen Moduls, sondern auf die korrekte Anwendung von softwaretechnischen Grundlagen liegen. Es sollte mehr darauf geachtet werden, dass das theoretische Wissen über Softwaretechnik korrekt in die Praxis überführt wird. Die Diplomarbeit oder Studienarbeit würde dann eine Art Dokumentation der geleisteten Projektarbeit darstellen.

Anmerkungen:

1. Der Inhalt dieses Kapitels sollte von meiner Arbeitsgruppe weniger als Kritik aufgefasst, sondern viel mehr als Denkanstöße für eine kontinuierliche Prozessverbesserung angesehen werden.
2. Einige der oben angerissenen Probleme konnte durch die gute Betreuung von Herrn Oezbek kompensiert werden. So diente zum Beispiel oft ein gemeinsamer Besuch der hiesigen Mensa für einen intensiven Ideenaustausch.
3. Teilweise wurden die hier genannten Vorschläge schon in die Realität umgesetzt. So arbeitete ich zum Ende meiner Arbeit hin nicht mehr alleine, sondern wir waren zeitweise sogar drei Entwickler (mit Herrn Oezbek, der sich zeitweise beteiligte). Auch eine Einarbeitung des neuen Studenten Herrn Rintsch wurde von mir und den Betreuern geleistet, so dass meiner Meinung nach eine reibungslose und fließende Übergabe erfolgte.

10 Empirische Bewertung

Am Ende der Entwicklungstätigkeiten wurde eine Mini-Fallstudie durchgeführt. Die Ergebnisse des empirischen Experimentes besitzen aufgrund von zu wenigen Datenpunkten keinen Anspruch verallgemeinerbar zu sein. Das Ziel des Experimentes war lediglich, erste Erkenntnisse für eine Validierung der Software zu gewinnen.

10.1 Fragestellung

Bei der verteilten, kollaborativen Softwareentwicklung ist die Interaktion zwischen den Entwicklern aufgrund der räumlichen Distanz erschwert. Die Herausforderung bei der Konzeption und Umsetzung von solchen Werkzeugen für die verteile Paarprogrammierung ist deshalb, ein verteiltes Arbeiten an den gleichen Artefakten zu ermöglichen, ohne die Vorteile, die durch die direkte Interaktion zwischen den Entwicklern entstehen, zu verlieren. Werkzeuge für die verteilte Paarprogrammierung versuchen diesem Problem durch die Übertragung von Audio und/oder Video, Textnachrichten sowie Awareness-Informationen innerhalb des Editors zu begegnen. In der Mini-Fallstudie sollte untersucht werden, ob die Interaktionsmöglichkeiten von Saros mit einer zusätzlichen Audioübertragung ausreichend für ein gemeinsames Arbeiten am Quelltext sind.

Saros ermöglicht es, dass mehrere Entwickler gleichzeitig Quelltext editieren können. Diese Fähigkeit des Werkzeuges ermöglicht es einen, viele neue Fragestellungen zu erforschen. Zum Beispiel könnte es von Interesse empirischer Untersuchungen sein, ob, und wenn ja wie viele zusätzliche Driver sinnvoll sind und wie viele Observer benötigt werden, um diese effektiv zu beobachten. Diese Mini-Fallstudie sollte lediglich erste Hinweise dafür liefern, ob die Funktionalität des gleichzeitigen Schreibens von den Entwicklern grundsätzlich begrüßt wird oder nicht. Andere Auswirkungen, zum Beispiel auf die Qualität des Quelltextes, wurden nicht untersucht.

Es sollten außerdem mögliche Einsatzszenarien des Produktes Saros evaluiert werden. So könnte zum Beispiel das Werkzeug von den Entwicklern für Code-Durchsichten oder Schulungen als besonders nützlich empfunden werden, für den Gebrauch in reinen Programmiersitzungen aber eher weniger.

Zusammenfassend kann man die zu untersuchenden Fragen wie folgt formulieren:

1. Wird durch Saros eine hinreichende Interaktion zwischen den Entwicklern ermöglicht?
2. Wird die Möglichkeit des gleichzeitigen Schreibens in einer Programmiersitzung von den Entwicklern verwendet und wird diese als positiv bewertet?
3. Für welche Entwicklungstätigkeiten wird die Verwendung von verteilter Paarprogrammierung mit Saros von den Entwicklern als nützlich und für welche als weniger nützlich empfunden?

Von diesen Fragen ausgehend wurde ein empirisches Experiment entworfen, um erste Antworten zu finden.

10.2 Empirisches Experiment

Das empirische Experiment wurde im März 2009 am Institut Informatik der Freien Universität Berlin durchgeführt.

10.2.1 Teilnehmer

Insgesamt haben zehn Softwareentwickler in vier Gruppen an dem Experiment teilgenommen. Die vier Gruppen waren auf zwei Tage zu je zwei Gruppen aufgeteilt. Die Gruppen am ersten Tag bestanden aus je drei, die am zweiten Tag aus je zwei Entwicklern.

Die Teilnehmer wurden aus vier verschiedenen Quellen akquiriert:

1. Diplomanden des Instituts für Informatik der Freien Universität Berlins (2)
2. Freundeskreis (aktive oder ehemalige Studenten des Diplomstudienganges Informatik am oben genannten Instituts) (4)
3. Teilnehmer des Seminars „Open Source Software Engineering“, das von Herrn Oezbek durchgeführt worden ist (ebenfalls Studenten) (2)
4. an Diplomarbeiten im Saros Projekt interessierte Studenten (2)

Diese Gruppen sind weder als Benutzer von einem Werkzeug für verteilte, kollaborative Softwareentwicklung repräsentativ noch gegenüber dem Projekt Saros und meiner Person völlig unbefangen. Dennoch erhoffe ich einige Hinweise für die in Kapitel 10.1 genannten Fragestellungen zu gewinnen, da ich alle Beteiligten um eine möglichst objektive Bewertung des Produktes Saros gebeten habe.

Die folgende Tabelle zeigt aus welchen der vier Quellen sich die vier Gruppen zusammensetzten:

	Quelle 1	Quelle 2	Quelle 3	Quelle 4
Gruppe A	2	1	0	0
Gruppe B	0	3	0	0
Gruppe C	0	0	1	1
Gruppe D	0	0	1	1

Tabelle 1: Zusammenstellung der einzelnen Gruppen

Nicht für alle Teilnehmer war das Werkzeug unbekannt. Die zwei Diplomanden hatten schon Vorwissen bezüglich Saros, hatten die Software aber noch nie selber benutzt. Drei von den vier Teilnehmern aus meinem Freundeskreis nahmen bereits an mindestens einem der beiden Testexperimente teil und hatten somit bereits Erfahrung in der Verwendung des Werkzeuges.

10.2.2 Technische Umsetzung

Die Topologie des verwendeten Netzwerkes war die gleiche wie in dem zweiten Testexperiment (siehe Kapitel 6.1.2). Als Rechner wurden wieder sowohl Computer des Instituts als auch von den Teilnehmern selbst mitgebrachte Laptops verwendet, die mittels WLAN und einer VPN-Verbindung mit dem Netzwerk des Fachbereichs verbunden waren. Als Jabber-Server wurde ein öffentlicher, sich im Internet befindlicher Server[8]verwendet.

Bis auf zwei Laptops mit MacOS (in der Gruppe D) lief auf allen Computern ein Windows Betriebssystem (Windows XP oder Windows Vista).

Anders als bei den Testexperimenten saßen die Teilnehmer diesmal nicht im gleichen Raum, sondern waren in unterschiedlichen Räumen untergebracht. Dieses Szenario entspricht eher dem Einsatz eines Werkzeuges für die verteilte, kollaborative Softwareentwicklung in der Realität.

Für die Kommunikation zwischen den Teilnehmern kam eine Sprachübertragung mittels Voice over IP zum Einsatz. Hierzu wurde das Programm Skype[9] verwendet, da dies für die am weit verbreitetesten Betriebssysteme (Windows, Linux und MacOS) zur Verfügung steht. Alle Teilnehmer verwendeten Headsets, die ihnen zur Verfügung gestellt worden sind.

10.2.3 Programmieraufgaben

Bevor die Teilnehmer die Aufgaben bearbeiteten, wurde ihnen ein gewisses Grundwissen über die Praktiken Paarprogrammierung und verteilte Paarprogrammierung vermittelt. Außerdem wurde ihnen kurz die Benutzung von Saros demonstriert.

In dem Experiment wurden drei Aufgaben bearbeitet, die alle aus unterschiedlichen Gründen ausgewählt worden sind.

In der ersten, sehr leichten Aufgabe ging es lediglich darum das Werkzeug kennen zulernen. Die Aufgabe war es aus einer Textdatei die Anzahl der Zeilen zu zählen. Die Teilnehmer wurden während der Bearbeitung dieser Aufgabe von mir bei der Benutzung von Saros unterstützt. Diese Art von kurzem Training stellte sicher, dass die Teilnehmer mit dem Umgang des Werkzeuges ein wenig vertraut waren. Bei der Gruppe B wurde dieses Training nicht durchgeführt, da aus dieser Gruppe alle Teilnehmer das Werkzeug bereits aus einem vorherigen Testexperiment kannten.

Die zweite Aufgabe wurde aus einem ACM Programming Contest entnommen [38]. Diese Aufgabe war algorithmischer Natur, mit der das gemeinsame Arbeiten mit Saros an einem algorithmischen Problem untersucht werden sollte. Für die genaue Aufgabenstellung siehe Anhang C.

Mit der dritten Aufgabe sollte untersucht werden, wie gut sich das Werkzeug für eine gemeinsame Code-Durchsicht eignet. Diese basierte auf einer Aufgabe aus den Übungen zur

[8]jabber.cc
[9]www.skype.de

Vorlesung „Spezielle Themen der Softwaretechnik“, bei der Defekte in einem Quelltext gefunden werden müssen. Hierfür wurde in den Quelltext vom JUnit-Framework [10] absichtlich Defekte hinzugefügt.

10.2.4 Qualitative Untersuchungen

Die Untersuchungen in diesem Experiment sind qualitativer Natur. Diese Untersuchungen bestehen aus folgenden Komponenten, die ich hier in absteigender Wichtigkeit angebe:

- Fragebögen vor und nach der Programmiersitzung
- Befragungen in einer anschließenden Diskussionsrunde
- meinen Beobachtungen während des Experimentes

Den Kern der Untersuchungen lieferten die Fragebögen, die vor und nach dem Experiment von den Teilnehmern ausgefüllt wurden. Der Fragebogen vor der Sitzung diente dazu, die Vorkenntnisse und bereits vor dem Experiment gebildeten Meinungen zu den Themen Paarprogrammierung und verteilter Paarprogrammierung zu erfragen. Der Fragebogen nach der Sitzung diente zum einen dazu, den Verlauf der Sitzung zu erfragen und zum anderen, das Werkzeug bezüglich der Benutzbarkeit und Brauchbarkeit von den Teilnehmern bewerten zu lassen.

Die abschließende Diskussionsrunde hatte nur ergänzenden Charakter. Die Teilnehmer wurden noch einmal zu den Themenkomplexen Interaktionsmöglichkeiten, Mehrschreiberfunktionalität und mögliche Einsatzszenarien von Saros befragt. Zusätzlich wurde ihnen die Möglichkeit geboten, über nicht durch die Fragebögen abgedeckten Themen zu reden.

Die Beobachtungen während der Programmiersitzung spielten, wenn überhaupt, nur eine untergeordnete Rolle. Ich notierte mir die noch auftretenden Defekte und ein paar Stichpunkte über den Verlauf der Sitzung, welche lediglich dazu dienten, die Angaben der Teilnehmer in den Fragebögen ein wenig besser einordnen zu können.

Bei den meisten Antwortmöglichkeiten sollten die Teilnehmer auf einer Ordinalskala etwas einstufen und dies zusätzlich begründen. Für die genauen Fragestellungen siehe Anhang D. Das Interesse dieser Untersuchung lag mehr bei den Begründungen als bei der reinen Einstufung. Deshalb ermunterte ich die Teilnehmer, möglichst viel Freitext zu schreiben.

10.2.5 Ergebnisse

Vorbemerkung: Während der Durchführung des empirischen Experimentes kam es bei unterschiedlichen Aktionen zu einem Versagen der Software. Diese Probleme konnten aber durch ein erneutes Initiieren einer neuen Programmiersitzung beseitigt werden, so dass mit einigen Zwangsunterbrechungen das Experiment durchgeführt werden konnte. Ich denke daher, dass der Einfluss dieser Tatsache auf die Ergebnisse sich in Grenzen hält. An einigen wenigen Stellen, an denen ich denke, dass dies eine gewisse Rolle gespielt haben könnte,

[10] www.junit.org

gehe ich auf diesen Umstand ein.

Ergebnisse bezüglich der Interaktionsmöglichkeiten

Für das gemeinsame Arbeiten am Quelltext wurden Interaktionsmöglichkeiten von Saros mit einer zusätzlichen Sprachübertragung von allen Teilnehmern als ausreichend oder besser empfunden. Auf die Frage, wie sie die Interaktionsmöglichkeit einstufen würden (bei den Antwortmöglichkeiten *gut*, *ausreichend*, *nicht ausreichend* und *viel zu wenig*) haben fünf von den zehn Teilnehmern mit *gut* geantwortet und fünf mit *ausreichend*. Die Verteilung auf die oben genannten Gruppen war hierbei ungefähr gleich.

Die überwiegende Mehrheit der Teilnehmer (80 Prozent) wünschte sich zudem eine Möglichkeit der Übertragung von Skizzen, ohne dass die Teilnehmer explizit danach befragt worden sind. Dies nannten sie im Freitextfeld der Frage über die Einstufung der Interaktionsmöglichkeiten oder der Frage, ob sie sich eine Videoübertragung wünschten. Die Teilnehmer verwendeten für eine solche Funktion die Begriffe „Zeichentafel“, „Fenster für Skizzen“, „Skizzenblock“, „Whiteboard“, „verteilte Zeichnung“, „Mal-Tool“ oder „handschriftliche Skizzen“. Das Bedürfnis nach einer solchen Funktion lag höchstwahrscheinlich darin begründet, dass die zweite Aufgabe algorithmischer Natur war. So fanden sich auf fast allen der ausgehändigten Aufgabenzettel nach der Programmiersitzung handschriftliche Skizzen wieder. Bei der Beobachtung der Kommunikation zwischen den Teilnehmern fiel auf, dass es Probleme bei der Diskussion über ihre Zeichnungen gab, da sie die Zeichnungen der anderen nicht sehen konnten.

Eine Videoübertragung wünschten sich nur zwei Teilnehmer. Einer der beiden nannte in der Begründung, dass er sich diese Videoübertragung ausschließlich für die Übertragung von Skizzen wünsche und nicht um sein Gegenüber zu sehen. Der andere gab an, sein Partner sehen zu wollen um so eine, für ihn wichtige, direkte Kommunikation zu realisieren. In der Diskussion wurde von einem Teilnehmer geäußert, dass er nur für das „Kennenlernen“ des Programmierpartners eine Videoübertragung als nützlich empfinden würde.

Durch die Frage, welche Arten der Interaktion die Teilnehmer wie oft genutzt hätten (Auswahlmöglichkeiten *viel*, *wenig* und *gar nicht*), wurde deutlich, wie wichtig den Teilnehmern die Möglichkeit der Sprachübertragung ist. Alle gaben an, Sprache *viel* und Textnachrichten *gar nicht* verwendet zu haben. In der Diskussion äußerte einer der Teilnehmer ergänzend, dass die Möglichkeit der Übertragung von Textnachrichten nur als Fallback wichtig sei, wenn aus irgendwelchen Gründen keine Audioübertragung möglich ist.

Wie oft die Teilnehmer Textselektionen verwendeten, um mit den anderen Teilnehmern zu interagieren, hing davon ab, welche Rolle der Teilnehmer hauptsächlich während der Programmiersitzung einnahm. Bei der entsprechenden Frage wurde beim Driver zusätzlich unterschieden, wie oft derjenige auch wirklich in der Programmiersitzung geschrieben hat. Dies war notwendig, denn auch wenn ein Teilnehmer die Driver Rolle in Saros besitzt, und damit Schreibrechte hat, impliziert das nicht, dass er davon auch wirklich Gebrauch macht. Die Driver, die angaben auch viel geschrieben zu haben, fanden die Möglichkeit der Textselektionen für die Interaktion unwichtiger als die Driver, die wenig geschrieben hatten. Der einzige Teilnehmer, der angab nie Textselektionen für die Interaktion verwendet zu haben, war in der Programmiersitzung hauptsächlich Driver und hatte viel geschrieben.

Ergebnisse bezüglich der Mehrschreiberfunktionalität

Ziel der Untersuchung war es unter anderem, erste Hinweise dafür zu bekommen, ob Benutzer eines Werkzeuges für verteilte, kollaborative Softwareentwicklung die Funktionalität des gleichzeitigen Schreibens verwenden würden und ob sie eine solche Funktionalität grundsätzlich als nützlich empfinden. Hierfür wurden den Teilnehmern freigestellt, ob sie von dieser Möglichkeit während des Programmierens Gebrauch machen oder nicht.

Die Ergebnisse sind sehr inhomogen in Abhängigkeit der oben genannten Gruppen. Man muss für weitergehende Untersuchungen die Gruppen genauer charakterisieren:

Als unabhängige Variablen lassen sich die Erfahrung mit Paarprogrammierung und verteilter Paarprogrammierung, sowie die Größe der Teams identifizieren. Die Unterschiede der Gruppen bezüglich der unabhängigen Variablen stellt die Tabelle 2 dar.

	Teamgröße	*Erfahrung bezüglich PP*	*Erfahrung bezüglich DPP*
Gruppe A	3	wenig Erfahrung, diese aber eher positiv	keine Erfahrung, nur einer hatte zuvor etwas über DPP gehört
Gruppe B	3	2 von den 3 Teilnehmern hatten viel Erfahrung, diese war positiv	beschränkte sich auf die Erfahrungen aus den Testexperimenten
Gruppe C	2	der eine hatte selten mit positiver Erfahrung, der andere niemals mit PP entwickelt	keine Erfahrung, nur einer hatte zuvor etwas über DPP gehört
Gruppe D	2	der eine hatte viel (eher positive) der andere wenig (negative) Erfahrung	der mit der mehr Erfahrung in PP hatte schon einige Male DPP (eher positive Erfahrung), der andere noch nie DPP verwendet

Tabelle 2: Mögliche Faktoren bei der Bewertung der Mehrschreiberfunktionalität

Genauere Charakterisierung der einzelnen Gruppen bezüglich der unabhängigen Variablen

Alle Teilnehmer der **Gruppe A** haben zunächst die Gemeinsamkeit, dass sie Saros noch nie verwendet haben. Somit haben sie sich höchstwahrscheinlich noch keine zu fest stehende Meinung zu diesem konkreten Werkzeug gebildet. Sie wussten lediglich, dass es sich um ein Werkzeug für verteilte Paarprogrammierung handelt. Die Teilnehmer hatten nur wenig Erfahrung mit Paarprogrammierung, zum Teil nur im Rahmen von den Übungen während des Studiums, ohne dass sie sich überhaupt bewusst waren diese Praktik zu verwenden. Diese Erfahrungen waren dann aber eher positiv, wobei ein stattgefundener Wissensaustausch und ein schnelleres Arbeitstempo als Gründe hierfür angegeben wurden. Erfahrungen mit verteilter Paarprogrammierung besaß kein Teilnehmer der Gruppe, nur einer hatte etwas über diese Praktik in einer Vorlesung gehört.

Die Teilnehmer der **Gruppe B** haben alle zuvor an mindestens einem der Testexperimente teilgenommen, waren also mit dem Werkzeug bereits vertraut und hatten sich somit schon eine Meinung gebildet. Zwei der drei Teilnehmer hatten im Beruf viel positive Erfahrung mit Paarprogrammierung gesammelt. Der eine hat bereits zwei Jahre Berufserfahrung und gab

an ungefähr 40 Mal mit dieser Praktik gearbeitet zu haben. Der andere arbeitet neben dem Studium und gab an, über 100 Mal bereits Paarprogrammierung verwendet zu haben. Die Erfahrung mit verteilter Paarprogrammierung beschränkte sich auf die Verwendung von Saros in den vorherigen Testexperimenten.

Die **Gruppe C** hatte wenig bis keine Erfahrung mit Paarprogrammierung und hatte niemals zuvor verteilte Paarprogrammierung als Praktik verwendet.

In der **Gruppe D** waren die Erfahrungen mit den Praktiken Paarprogrammierung und verteilter Paarprogrammierung gemischt. Der eine hatte schon oft Paarprogrammierung verwendet (ca. 40 Mal) und einige wenige Male verteilte Paarprogrammierung (5-20 Mal) und war von beiden Praktiken angetan. Der andere Teilnehmer hatte nur wenig (weniger als 5 Mal), dafür negative Erfahrung mit Paarprogrammierung gesammelt und hatte niemals zuvor verteilte Paarprogrammierung verwendet. Als Grund für die negative Erfahrung mit Paarprogrammierung gab er die zu unterschiedlichen Fähigkeiten zwischen ihm und seinen Partnern an.

Ergebnisse der einzelnen Gruppen

In der **Gruppe A** wurde nur wenig von der Möglichkeit des gleichzeitigen Schreibens Gebrauch gemacht. Auf die Frage, wie ihnen diese Funktionalität gefallen hat, antworteten zwei von den drei Teilnehmern mit *weiß nicht*. Beide gaben als Begründung an, dass diese Funktionalität wenig verwendet worden ist, und äußerten die Befürchtung, dass das gleichzeitige Schreiben verwirren und man den Überblick verlieren kann. Als positiv wurde von einem der beiden die Möglichkeit empfunden, kleinere Fehler eines Programmierpartners schnell korrigieren zu können.

Der dritte Teilnehmer der Gruppe empfand die Funktionalität des gleichzeitigen Schreibens als *eher positiv*, aber auch dieser hatte Vorbehalte und empfahl Richtlinien für das gleichzeitige Editieren. So wünschte er sich Absprachen darüber, inwiefern man den Code der anderen verändern darf. Als möglichen Nachteil nannte dieser, dass man Veränderungen von anderen Drivern nicht gut mitbekommen könne, und empfahl daher eine verstärkte Kommunikation zwischen den Teilnehmern.

Die **Gruppe B** verwendete das Werkzeug grundlegend anders als die erste Gruppe. Sie verwendeten das gleichzeitige Schreiben sehr häufig. Dies ist eventuell darauf zurückzuführen, dass sie bereits mit dem Werkzeug und dessen Funktionalität vertraut waren.

Die beiden Teilnehmer mit der größeren Erfahrung in der Paarprogrammierung nahmen dabei den aktiven Teil in der Programmiersitzung ein. Der Dritte, mit weniger Erfahrung, nahm meistens die Rolle des Observers ein, obwohl dieser in dem Werkzeug Schreibrechte besaß, und somit im Programm auch als Driver geführt wurde. Die beiden aktiven Teilnehmer programmierten an verschiedenen Klassen und verwendeten die Sprachübertragung um die Arbeit an den verschiedenen Klassen zu koordinieren. So wurde darüber geredet, welche Parameter und welche Rückgabewerte die einzelnen Methoden besitzen sollten. Es sind dabei Sätze wie „Schreib mir mal eine Methode auf deiner Klasse, die als Parameter X, Y bekommt und Z zurückliefert.“ sind dabei gefallen.

Die Funktionalität des gleichzeitigen Schreibens wurde von allen Teilnehmern der Gruppe als positiv empfunden (zwei Mal *positiv* und ein Mal *eher positiv*). Der Teilnehmer, der nur

eher positiv nannte, war einer der beiden aktiven Drivern und gab zu bedenken, dass das Arbeiten an einem Problem gut zu parallelisieren sein muss, damit das gleichzeitige Schreiben Vorteile bringt. Außerdem forderte er Absprachen, an die man sich zu halten hat, damit es nicht chaotisch wird. Er empfand es für nötig, schon vorher die Signaturen der Methoden vereinbart zu haben. Der andere aktive Driver äußerte alle diese Bedenken nicht und stufte die Funktionalität sogar als zwingend notwenig ein. Der ständige Observer empfand die Möglichkeit auch grundsätzlich als positiv und beschwerte sich nur darüber, dass die Awareness-Informationen und der Verfolgermodus noch nicht besonders gut mit mehreren Drivern funktionierten.

Ob die Bewertungen der Funktionalität dadurch beeinflusst worden sind, dass es sich bei den Personen um Freunde von mir handelt, kann nicht geklärt werden.

Die **Gruppe C** hat niemals gleichzeitig geschrieben. Auf die Frage, wie ihnen diese Funktionalität gefallen würde, nannten beide *weiß nicht*. Einer der beiden Teilnehmern gab an, diese Möglichkeit für unnötig zu halten. Bei diesem Ergebnis spielen eventuell die wenigen Erfahrungen der Teilnehmer mit den Praktiken Paarprogrammierung und verteilter Paarprogrammierung eine gewisse Rolle.

Die Teilnehmer in der **Gruppe D** gaben an, viel gleichzeitig geschrieben zu haben. Dies deckt sich aber nicht mit meinen Beobachtungen, durch die ich den Eindruck gewann, dass diese Möglichkeit eher weniger genutzt worden ist. Die Funktionalität wurde von beiden Teilnehmern als *positiv* empfunden, Begründungen dafür wurden aber leider nicht angegeben.

Die Ergebnisse der einzelnen Gruppen sind sehr inhomogen. Die Identifizierung möglicher Faktoren, die Einfluss auf die Häufigkeit der Verwendung des gleichzeitigen Schreibens, sowie der Bewertung dieser Möglichkeit haben könnten, konnte im Rahmen dieses Experimentes nicht geleistet werden. Dafür wurde dieses Experiment zu wenig kontrolliert. Es können lediglich Kandidaten für solche möglichen Faktoren aufgestellt werden:

- Erfahrung mit Paarprogrammierung und verteilter Paarprogrammierung
- Vertrautheit mit dem eingesetzten Werkzeug

Die Größe der Teams (Tupel oder Tripel) hatte zumindest in diesem Experiment keinen Einfluss.

Erwähnenswert ist noch, dass sich viele Teilnehmer für Absprachen oder Richtlinien für das gleichzeitige Schreiben ausgesprochen haben, damit dieses nicht zu unkoordiniert abläuft. Von einigen wurde die Befürchtung geäußert, dass diese Funktionalität verwirren und man den Überblick über die Veränderungen am Quelltext verlieren könnte.

Ergebnisse bezüglich der Einsatzmöglichkeiten des Werkzeuges

Die Teilnehmer wurden in den Fragebögen danach gefragt, für wie geeignet sie Saros für den Einsatz für bestimmte Entwicklungstätigkeiten halten. Dabei wurden nach drei möglichen Entwicklungstätigkeiten gefragt:

1. gemeinsame Programmiersitzung
2. Code-Durchsichten
3. Schulungen

Die Antwortmöglichkeiten waren *sehr geeignet*, *geeignet*, *weniger geeignet* und *gar nicht geeignet*.

Für die Verwendung in einer **gemeinsamen Programmiersitzung** hielten vier Teilnehmer das Werkzeug für *sehr geeignet*, fünf Teilnehmer für *geeignet* und nur einer hielt es für *weniger geeignet* (siehe Abbildung 11). Der zuletzt genannte gab als Begründung die fehlende Stabilität an, was auf das Versagen der Software an einigen Stellen während des Experimentes zurückzuführen ist. Einer der Teilnehmer gab für seine positive Bewertung an, dass Saros die Vorteile von Paarprogrammierung vereint mit der Möglichkeit sich aus der gemeinsamen Sitzung zu lösen und wieder getrennt zu arbeiten.

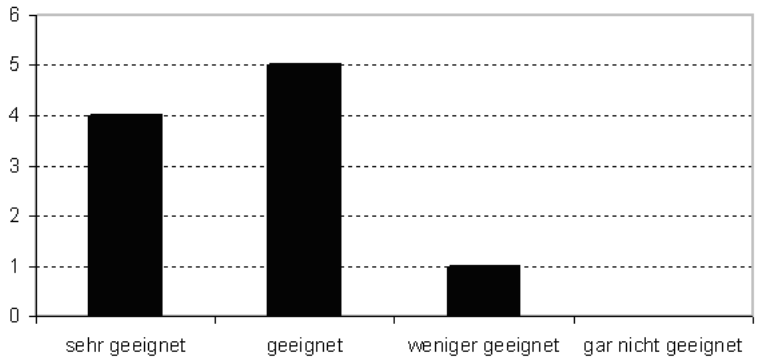

Abbildung 11: Antworten auf die Frage, für wie geeignet die Teilnehmer Saros für eine gemeinsame Programmiersitzung halten

Die Meinung darüber, ob das Werkzeug für **Code-Durchsichten** geeignet ist, gehen dagegen ein wenig auseinander. Vier von den zehn Teilnehmern hielten das Werkzeug für *sehr geeignet*, ebenso viele für *weniger geeignet* und zwei weitere für *geeignet* (siehe Abbildung 12). Von denjenigen, die das Werkzeug für *weniger geeignet* hielten, wurde am meisten eine unterschiedliche Lesegeschwindigkeit als Begründung angeben. Einer der beiden Teilnehmer, die *geeignet* angaben, schränkte seine Aussage ein, indem er eine Phase des Code-Kennenlernens vor der Durchführung der Begutachtung empfahl.

Auf die Frage, ob sie sich das Werkzeug für den Einsatz in **Schulungen** vorstellen könnten, antworteten sieben der zehn Teilnehmer mit *sehr geeignet* (4) oder *geeignet* (3). Drei Teilnehmer gaben an, das Werkzeug hierfür als *weniger geeignet* zu halten. Als Begründung gab einer dieser drei an, dass es keine Möglichkeit gebe, Bilder oder Skizzen zu übertragen.

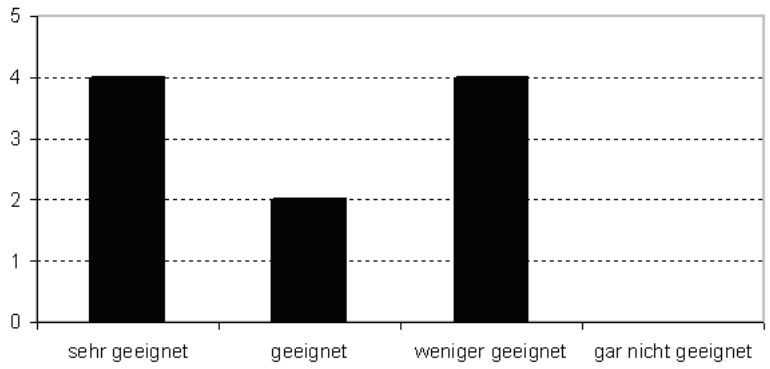

Abbildung 12: Antworten auf die Frage, für wie geeignet die Teilnehmer Saros für Code-Durchsichten halten

Auch einer der Teilnehmer, der das Werkzeug für *sehr geeignet* hielt, empfahl zusätzlich eine Präsentation, um zum Beispiel UML-Diagramme den Schülern zu zeigen. Von den Befürwortern wurde positiv angemerkt, dass damit ein interaktiver Unterricht möglich sei, der sich von dem klassischen Frontalunterricht unterscheide. So könnte zum Beispiel der Lehrer den Schülern einfach schnell etwas in den Quelltext schreiben, ohne es diktieren zu müssen. Die Abbildung 13 zeigt die Verteilung der Antworten auf diese Frage.

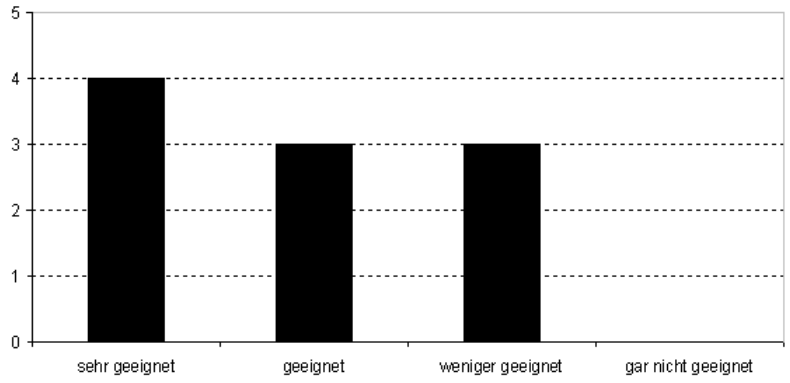

Abbildung 13: Antworten auf die Frage, für wie geeignet die Teilnehmer Saros für den Einsatz in Schulungen halten

Allgemeine Ergebnisse

Die Teilnehmer sollten in einer Frage insgesamt die Softwareentwicklung mit Saros bewerten. Die Antwortmöglichkeiten waren *positiv*, *eher positiv*, *eher negativ*, *negativ* und *weiß nicht*. Insgesamt 80 Prozent der Teilnehmer bewerteten die Softwareentwicklung mit Saros insgesamt als *eher positiv* und 20 Prozent sogar als *positiv* (siehe Abbildung 14). Die Begründungen, warum das Werkzeug nur mit *eher positiv* und nicht mit *positiv* bewertet wurde, hingen meistens mit der noch nicht zu hundertprozentig gegebenen Stabilität der Software zusammen. Die Ergebnisse sind eventuell von der Tatsache beeinflusst, dass die Teilnehmer dem Projekt Saros sowie meiner Person nicht unbefangen waren.

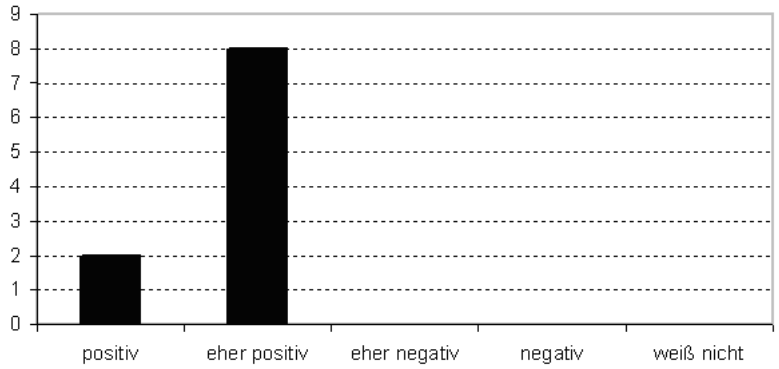

Abbildung 14: Antworten auf die Frage, wie die Teilnehmer insgesamt die Softwareentwicklung mit Saros bewerten

10.3 Zusammenfassung

Zusammenfassend hat die Mini-Fallstudie folgende Ergebnisse:

1. Die Interaktionsmöglichkeiten wurden von allen Teilnehmern als ausreichend für ein gemeinsames Arbeiten am Quelltext empfunden, eine Videoübertragung wurde sich nur sehr selten gewünscht. Dafür wünschte sich eine große Mehrheit eine Möglichkeit, Skizzen gemeinsam zu erarbeiten oder zumindest übertragen zu können.

2. Die Mehrschreiberfunktionalität wurde sehr unterschiedlich von den Teilnehmern verwendet und bewertet. Das Spektrum reicht von gar nicht benutzt und als unnötig empfunden, bis hin zu ständig verwendet und als unabdingbar empfunden. Die Faktoren, die diese stark unterschiedlichen Auffassungen beeinflussten, konnten nicht identifiziert werden. Eine Vermutung von mir ist allerdings, dass die Erfahrungen mit Paarprogrammierung und verteilter Paarprogrammierung einen nicht unerheblichen Einfluss auf die Bewertung dieser Funktionalität haben.

3. Als mögliche Einsatzszenarien des Werkzeuges wurde die gemeinsame Programmiersitzung sowie der Einsatz in Schulungen identifiziert.

11 Ausblick

Zu guter Letzt werde ich hier einen Ausblick wagen, wie die nähere und etwas fernere Zukunft von Saros aussehen könnte.

Während ich diese Ausarbeitung schreibe, werden die letzten von Herrn Salinger organisierten Funktionstests durchgeführt, die evaluieren sollen, ob das Produkt Saros eine genügende Qualität für ein erstes öffentliches Beta-Release besitzt. Es ist damit zu rechnen, dass dieses Release in den folgenden Wochen erscheinen wird. Ich vermute, dass ein solches Release dem Projekt eine neue Dynamik verleihen könnte. So könnte sich die Anzahl der Personen, die sich das Werkzeug anschauen und ausprobieren, stark erhöhen. Dadurch könnte man wahrscheinlich neue Benutzer der Software gewinnen, welche sicherlich auch neue nützliche Fehlermeldungen im öffentlich zugänglichen Bug-Tracker schreiben würden. Da es sich bei Saros um Open Source Software handelt, ist auch nicht ausgeschlossen, dass externe Entwickler Interesse anmelden, sich an der Entwicklung von Saros zu beteiligen. Eine solche Community von neuen Benutzern sowie Entwicklern würde ich zumindest begrüßen und ich denke, dies würde für das Projekt Saros neue Zukunftsmöglichkeiten eröffnen.

Durch die geplante Kooperation mit einem Wirtschaftsunternehmen wird sich die Software in einem betrieblichen Umfeld beweisen müssen. Dies stellt eine hervorragende Art der Validierung der Software dar und ermöglicht der Arbeitsgruppe neue Forschungen im Bereich der empirischen Bewertung von verteilter, kollaborativer Softwareentwicklung.

Der Fokus bei diesem Einsatzszenario liegt zunächst auf der Basisfunktionalität von Saros. So sind erweiterte Funktionen, wie die Mehrschreiberfunktionalität, zunächst nicht unbedingt notwendig. Dies ermöglicht, dass die in Kapitel 8 beschriebenen offenen Probleme in folgenden Diplom- und Masterarbeiten von Studenten in Angriff genommen werden können, ohne dass davon der Einsatz in diesem betrieblichen Umfeld zu sehr abhängig ist.

Es sind einige sinnvolle Erweiterungen und Verbesserungen im Bereich der Benutzbarkeit von Saros denkbar. Gerade für die Parallelität auf Schreibebene müssen einige neue Konzepte gefunden werden, um die Awareness bei mehreren Drivern zu verbessern. Die empirische Bewertung in Kapitel 10 hat zudem gezeigt, dass von Seiten der Benutzer sich eine Funktion gewünscht wird, um gemeinsam Zeichnungen zu erarbeiten. Eine solche Skizzen-Funktion würde die Interaktionsmöglichkeiten zwischen den Teilnehmer der Programmiersitzung und damit auch die Brauchbarkeit der Software erheblich verbessern.

12 Zusammenfassung

Durch die gründlich durchgeführte Anforderungsbestimmung wurde eine stabilere Basis für die weitere Entwicklung an Saros geschaffen. Als Ergebnis dieser Bemühungen existiert nun in dem Projekt ein zentrales Anforderungsdokument. Die Anforderungen werden in Form eines Anforderungskataloges sowie durch Anwendungsfälle beschrieben. Dies ermöglicht erst ein effektives Anforderungsmanagement in der Zukunft, insbesondere was die Nachverfolgbarkeit (traceability) von Anforderungen betrifft.

Es wurden mehrere Testexperimente, Funktions- und Strukturtests sowie Code-Durchsichten durchgeführt und so das Versagen der Software und die zugrunde liegenden Defekte bestimmt. Ein großer Teil der gefundenen Probleme und Defekte konnten behoben werden.

Durch die zusätzliche Verwendung von UDP als Transportprotokoll bei Direktverbindungen wurde eine Verbesserung der NAT-Traversierung erreicht, auch wenn es bei der praktischen Umsetzung noch einige Probleme gibt, die genauer untersucht werden müssen. Die geleistete Neuimplementierung einer erweiterten Konsistenzsicherung stellt einen wichtigen Schritt in die Richtung zu einem robusteren Werkzeug dar.

Um eine Parallelität auf Schreibebene mit Saros auf eine geeignete Art und Weise zu ermöglichen, müssen noch einige Herausforderungen bewältigt werden. Diese sind zum Teil konzeptioneller Art, wie im Bereich der Awareness-Informationen, aber auch technischer Natur, wie dem Realisieren einer brauchbaren Undo-Funktion. Für das zuletzt genannte Problem halte ich die Erweiterung der Nebenläufigkeitskontrolle für den am Erfolg versprechendsten Ansatz.

Die durchgeführte Mini-Fallstudie hatte als Ergebnis, dass die durch Saros, in Verbindung mit einer zusätzlichen Audioübertragung, gebotenen Interaktionsmöglichkeiten von den Entwicklern als grundsätzlich ausreichend für ein gemeinsames Arbeiten am Quelltext empfunden wurde. Eine Videoübertragung, um die entfernten Teilnehmer der Programmiersitzung sehen zu können, wurde sich nur sehr selten gewünscht. Dafür wünschte sich eine große Mehrheit eine Möglichkeit, Skizzen gemeinsam zu erarbeiten oder zumindest übertragen zu können. Die Möglichkeit des gleichzeitigen Schreibens wurde sehr unterschiedlich von den Teilnehmern verwendet und bewertet. Die bei dem empirischen Experiment teilnehmenden Softwareentwickler hielten das Werkzeug für gemeinsame Programmiersitzungen als geeignet und konnten sich zusätzlich einen Einsatz bei Schulungen vorstellen.

13 Literaturverzeichnis

Literatur

[1] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *Software, IEEE*, 17(4):19–25, 2000.

[2] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming, June 2000.

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

[4] Y. M. Mansour B. George. A multidisciplinary virtual team. In *Systemics, Cybernetics and Informatics (SCI)*, 2002.

[5] 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.

[6] David Stotts, Laurie Williams, Nachiappan Nagappan, Prashant Baheti, Dennis Jen, and Anne Jackson. Virtual teaming: Experiments and experiences with distributed pair programming. Technical Report TR03-003, Department of Computer Science, University of North Carolina - Chapel Hill, February 28 2003. Tue, 1 Apr 2003 14:17:15 GMT.

[7] E. Gehringer D. Stotts P. Baheti, L. Williams and J.Smith. Distributed pair programming: Empirical studies and supporting environments. Technical report, UNC-CH Technical Report TR02-010, March 15, 2002.

[8] Riad Djemili, Christopher Oezbek, and Stephan Salinger. Saros: Eine Eclipse-Erweiterung zur verteilten Paarprogrammierung. In Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven, editors, *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 317–320. GI, 2007.

[9] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung. Master's thesis, Freie Universität Berlin, 2006.

[10] B. Gustavs. Weiterentwicklung des Eclipse-Plug-Ins Saros zur Verteilten Paarprogrammierung. `http://www.inf.fu-berlin.de/inst/ag-se/theses/Gustavs_Saros_DPPII.pdf`. (Online; accessed 19-March-2009).

[11] O. Rieger. Weiterentwicklung einer Eclipse-Erweiterung für verteilte Paar-Programmierung im Hinblick auf Kollaboration und Kommunikation. Master's thesis, Freie Universität Berlin, 2008.

[12] Berthold Daum. *Java-Entwicklung mit Eclipse 3.3*. dpunkt.verlag, 2008.

[13] OSGi Service Platform Release 4 Version 4.1. `http://www.osgi.org/Download/Release4V41`. (Online; accessed 17-March-2009).

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[15] XMPP Extensions. `http://xmpp.org/extensions`. (Online; accessed 17-March-2009).

[16] History of XMPP. `http://xmpp.org/about/history.shtml`. (Online; accessed 17-March-2009).

[17] Extensible Messaging and Presence Protocol (XMPP): Core. `http://www.ietf.org/rfc/rfc3920.txt`. (Online; accessed 17-March-2009).

[18] Smack API. `http://www.igniterealtime.org/projects/smack/`. (Online; accessed 17-March-2009).

[19] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[20] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, 1989.

[21] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. 2004.

[22] 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.

[23] 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.

[24] C. Sun C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. `http://www.cit.gu.edu.au/~scz/conferences/cscw98.ppt`, 2004. (Online; accessed 17-March-2009).

[25] Project ACE on Sourceforge. `http://sourceforge.net/projects/aceoperator`. (Online; accessed 17-March-2009).

[26] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*. Prentice Hall PTR, 2 edition, July 2001.

[27] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems, 1996.

[28] Bryan Ford, Dan Kegel, and Pyda Srisuresh. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Technical Conference*, 2005.

[29] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats), 2003.

[30] Jürgen Schmidt. The hole trick, How Skype & Co. get round firewalls. http://www.h-online.com/security/How-Skype-Co-get-round-firewalls--/features/82481/. (Online; accessed 30-March-2009).

[31] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). Internet-Draft draft-ietf-behave-turn-10, Internet Engineering Task Force, September 2008. Work in progress.

[32] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. Internet-Draft draft-ietf-mmusic-ice-19, Internet Engineering Task Force, October 2007. Work in progress.

[33] J. Rosenberg. TCP Candidates with Interactive Connectivity Establishment (ICE). Internet-Draft draft-ietf-mmusic-ice-tcp-07, Internet Engineering Task Force, July 2008. Work in progress.

[34] ACM Programming Contest. Why Johnny Can't Count. http://acmicpc-live-archive.uva.es/nuevoportal/data/p2553.pdf. (Online; accessed 23-March-2009).

[35] LimeWire Project. Entwicklerdokumentation der LimeWire API. http://wiki.limewire.org/index.php?title=Javadocs. (Online; accessed 27-March-2009).

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

[37] Laurie Williams, Anuja Shukla, and Annie I. Anton. An initial exploration of the relationship between pair programming and brooks' law. In *ADC '04: Proceedings of the Agile Development Conference*, pages 11–20, Washington, DC, USA, 2004. IEEE Computer Society.

[38] ACM Programming Contest. Doors and Penguins. http://acmicpc-live-archive.uva.es/nuevoportal/data/p3581.pdf. (Online; accessed 23-March-2009).

[39] ACM Programming Contest. Count. http://contests.vanb.org/2007regionals/10-7/count.pdf. (Online; accessed 27-March-2009).

A Anforderungskatalog

A.1 Funktionale Anforderungen

ID	*Titel*	*Beschreibung*
FR1	Kollaboration	Das Werkzeug muss mehreren verteilten Entwicklern (auch mehr als 2) ein gemeinsames Arbeiten am Quelltext innerhalb eines Projektes ermöglichen. Alle Entwickler müssen sowohl als *Driver* als auch als *Observer* an der Programmiersitzung teilnehmen können.
FR2	Kommunikation	Es muss eine Kommunikation zwischen den Teilnehmern der Programmiersitzung möglich sein. Als Mindestanforderung muss diese Kommunikation über Textnachrichten geschehen können. Dabei soll es möglich sein, eine Nachricht an alle oder nur an einen bestimmten Teilnehmer zu schicken. Die Kommunikation über Sprache und/oder Video muss nicht durch das Werkzeug selber geleistet werden, hierfür können externe Programme herangezogen werden.
FR4	Parallelität auf Sichtebene	Den Observern ist es nicht nur möglich andere Programme neben der Entwicklungsumgebung auszuführen (Parallelität auf Programmebene) sondern auch innerhalb der Entwicklungsumgebung von dem Aufmerksamkeitsbereich des/der Driver abzuweichen und eigenständig Dateien eines Projektes anzuschauen.
FR5	Parallelität auf Schreibebene	Das Werkzeug muss es ermöglichen, dass es zur gleichen Zeit mehrere Driver geben kann, d.h. es kann mehr als ein Entwickler Schreibzugriff auf den Quelltext besitzen. Die Driver sollen einfache Editieroperationen blockierungsfrei durchführen können. Für Dateioperationen und komplexere Editieroperationen (Refactoring) können temporär einzelne Dateien gesperrt werden (exklusives Schreibrecht). Die Undo-Funktion (der eigenen Änderungen) muss weiterhin funktionieren.
FR6	Awareness	Es muss zu jeder Zeit visualisiert werden, wer gerade an welcher Datei arbeitet (z.B. durch Markierung in der Dateiliste). Es muss die aktuelle Cursorposition der Driver übertragen werden und die jeweiligen Sichtbereiche und Änderungen dieser visuell erkennbar sein (z.B. durch global eindeutige Hintergrundfarben). Die Observer sollen die Möglichkeit besitzen, Textpassagen zu selektieren und dies soll für die Driver sichtbar sein.

FR7	Verfolgermodus	Es muss den Observern möglich sein, die Veränderungen von den Drivern am Quelltext automatisch zu beobachten, d.h. es wird dem Sichtbereich eines Drivers gefolgt. Hierzu kann man den zu beobachtenden Driver auswählen. Der Verfolgermodus soll vom Observer selbständig aktiviert und deaktiviert werden können.
FR8	Rollenwechsel	Der die Programmiersitzung initiierende Entwickler hat zu jeder Zeit die Möglichkeit, den Teilnehmern die Driver-Rolle zu geben und wieder zu entziehen.
FR9	Übersetzbarkeit des Quelltextes	Bei allen Teilnehmern der Programmiersitzung muss der Quelltext genau so übersetzt werden können als wäre es eine normale nicht verteilte Programmiersitzung.
FR10	Projektfreigabe	Als Zugriffskontrolle muss eine Freigabe auf Projektebene möglich sein.
FR11	Auswahl von Programmier-partnern	Es muss eine Liste von bekannten Programmierpartnern angezeigt werden und ob diese gerade für eine Programmiersitzung verfügbar sind.
FR12	Integration in ElectroCodeo-Gram (ECG)	Es muss ein ECG-Sensor vorhanden sein, der die Rollenwechsel und Veränderungen aufzeichnet. Des Weiteren muss erkennbar sein, wer welche Änderungen durchgeführt hat. Als Mindestanforderung muss aus einem Log von einem Entwickler ersichtlich sein, welche Änderungen am Quelltext von ihm selber und welche Änderungen von anderen kommen.
FR13	Zustandskontrolle	Es muss eine Zustandskontrolle geben, die den aktuellen Zustand der Verbindung überwacht. Dem Benutzer soll der Durchsatz (In/Out) der aktuellen Verbindung angezeigt werden.

A.2 Nichtfunktionale Anforderungen

ID	*Titel*	*Beschreibung*
NFR1	Zielplattform	Das Werkzeug muss in Eclipse (3.4) unter Windows und Linux mit installiertem JDK (ab 1.5) lauffähig sein.
NFR2	Installation	Die Installation muss über eine Eclipse Updatesite in weniger als 5 Minuten möglich sein.

NFR3	Dokumentation	Es muss Dokumentation für folgende Personengruppen existieren: Entwickler, Administratoren und Benutzer. Alle externen Dokumente müssen in englischer Sprache vorliegen.
NFR4	Dokumentation des Quelltextes	Der Quelltext muss auf Methoden-, Klassen- und Paketebene dokumentiert sein.
NFR5	Initiierung einer Programmiersitzung	Voraussetzung für das Initiieren einer Programmiersitzung ist, dass Saros bei allen Teilnehmern installiert ist, alle Teilnehmer online sind und sich gegenseitig in Saros als Kontakte hinzugefügt haben. Die Synchronisation muss in Abhängigkeit der Differenz der Dateien effizient sein. Die gesamte Zeit berechnet sich aus folgenden Laufzeiten: T1 = Checksummen berechnen T2 = Checksummen übertragen T3 = Differenz berechnen T4 = Differenz übertragen Die benötigte Zeit für das Berechnen einer Checksumme und der Differenz soll vergleichbar (asymptotisch) zu denen der gängigen Tools wie *rsync* sein (wahrscheinlich $nlogn$). Die Übertragungszeiten hängen von der verfügbaren Bandbreite ab, sollte aber effizient sein, indem, wenn möglich, die Daten vor der Übertragung komprimiert werden.
NFR6.1	Netzwerk I	Das Werkzeug *muss* in allen Netzwerktopologien eingesetzt werden können, sofern eine Verbindung zum Jabber-Serververbund aufgebaut werden kann.
NFR6.2	Netzwerk II	Für die Kommunikation zwischen zwei Nutzern *soll* eine Direktverbindung genutzt werden.
NFR6.3	Netzwerk III	Für den Austausch von Dateien *muss* eine Direktverbindung genutzt werden, sofern diese durch Verwendung der letzten veröffentlichten, stabilen Version der Bibliothek Smack möglich ist. Sollte eine Direktverbindung nicht möglich sein, so muss es eine Rückfallstrategie geben, die trotzdem das Funktionieren des Werkzeuges sicherstellt.

NFR6.4	Netzwerk IV	Es *muss* für jeden Benutzer möglich sein, den Zustand seiner Verbindung bzw. seiner Verbindungen zu den anderen Benutzern einzusehen. Dies beinhaltet: 1. (*muss*) Direktverbindung oder IBB bzgl. Austausch von Dateien 2. (*soll*) Direktverbindung oder Transport über Server bzgl. der restlichen Kommunikation 3. (*muss*) verstrichene Zeit seit Eintreffen des letzten Datenpaketes 4. (*soll*) Größe oder Anzahl der Daten, die noch auf Versand warten 5. (*soll*) durchschnittliches Datenaufkommen in KB pro Sekunde gemittelt über die letzten 60 Sekunden und über die gesamte Sitzung 6. (*soll*) totales Datenaufkommen in MB
NFR6.5	Netzwerk V	Eine Chatkommunikation muss zwischen den Teilnehmern möglich sein, selbst falls der Jabber-Server des Hosts keine Möglichkeit zur Erstellung eines Multi-User Chats besitzt.
NFR7	Konsistenz	Jeder Teilnehmer der Programmiersitzung muss ein konsistentes Abbild aller Dateien des gemeinsamen Projektes besitzen. Inkonsistenzen müssen innerhalb von 15 Sekunden erkannt und gelöst werden.
NFR8	Konsistenzmodell	Die Nebenläufigkeitskontrolle für den Gruppeneditor hat folgenden Anforderungen zu genügen: • Konvergenz • Kausalitätserhaltung • Intentionserhaltung
NFR9	Echtzeit	Die lokalen Änderungen müssen verzögerungsfrei wie bei einem Single-Editor durchgeführt werden. Änderungen von entfernten Entwicklern können mit einer Verzögerung von ein paar wenigen Sekunden nachvollzogen werden.
NFR10	Bugtracker	Für das Projekt muss es einen Bug/Issuetracker existieren.
NFR11	Öffentliche Mailingliste	Für das Projekt muss eine öffentliche Mailingliste existieren.
NFR12	Projektseite	Für das Projekt muss eine öffentliche Projektseite im Internet existieren.

B Anwendungsfälle

In diesem Kapitel werden Anwendungsfälle von Saros beschrieben. Diese ergänzen nicht nur den Anforderungskatalog für eine genauere Produktdefinition sondern liefern auch mögliche Testszenarien.

Das folgende Diagramm gibt lediglich eine Übersicht über die Use-Cases. Die eigentlichen Use-Cases werden danach in einer halb-formalen Art und Weise in Textform beschrieben.

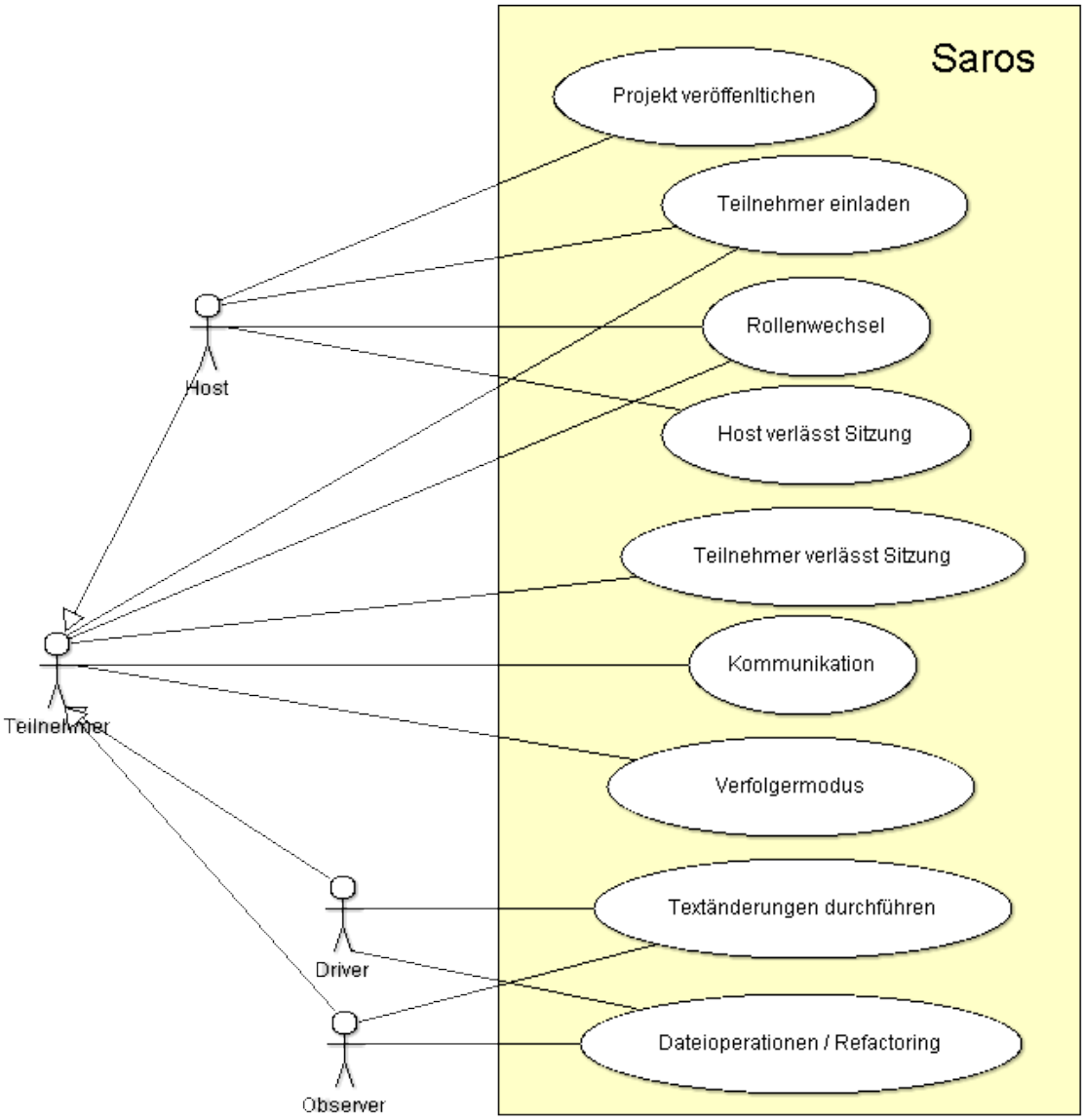

Abbildung 15: Übersicht über die aufgestellten Anwendungsfälle und deren Akteure

Use Case 1

Name: **Projekt veröffentlichen**

Primärakteur: Host

Stakeholder und Interessen:

- Host: möchte eine verteilte Programmiersitzung erstellen

Vorbedingungen:

- Saros muss korrekt installiert sein
- Host muss mit Kommunikationsnetz verbunden sein (Jabber)

Nachbedingungen:

- verteilte Programmiersitzung ist erstellt
- Host hat zunächst die Rolle Driver und wird im View *Shared Project Session* als solcher angezeigt
- die Möglichkeit ein weiteres Projekt zu veröffentlichen ist im Kontextmenü deaktiviert (ausgegraut)

Standardablauf:

1. Host wählt im Kontextmenü eines Projektes *Share Project* aus
2. Einladungsdialog wird angezeigt
3. Teilnehmer werden eingeladen (siehe Use Case *Teilnehmer einladen*)

Erweiterungen / alternative Abläufe: keine

spezielle Anforderungen: keine

Häufigkeit des Auftretens: mittel

Use Case 2

Name: **Teilnehmer einladen**

Primärakteur: Host

Stakeholder und Interessen:

- Host: möchte Teilnehmer zu einer verteilten Programmiersitzung einladen
- Teilnehmer: möchten an der Programmiersitzung teilnehmen

Vorbedingungen:

- Projekt muss bereits veröffentlicht sein (siehe Use Case *Projekt veröffentlichen*)

Nachbedingungen (bei Erfolg):

- neue Teilnehmer nehmen an der Programmiersitzung teil und werden im View *Shared Project Session* angezeigt

- neue Teilnehmer haben zunächst die Rolle Observer

Nachbedingungen (bei Misserfolg):

- falls es bei der Synchronisation Probleme gab, muss der Originalzustand (vor der Einladung) hergestellt werden

Standardablauf:

1. Host öffnet den Einladungsdialog (Button im View *Shared Project Session*)
2. Liste von verfügbaren Teilnehmern wird angezeigt
3. Host wählt den einzuladenden Teilnehmer aus und klickt auf den Button *Invite*
4. dem einzuladenden Teilnehmer wird eine Einladung geschickt und in der Liste wird der Status (warte auf Bestätigung) angezeigt
5. eingeladener Teilnehmer akzeptiert die Einladung
6. Dateiliste aller Dateien im Projekt werden zum eingeladenen Teilnehmer verschickt (Status in der Liste wird aktualisiert)
7. eingeladener Teilnehmer wählt aus, ob in seinem Workspace ein neues Projekt angelegt oder ob ein bestehendes Projekt als Ausgangsbasis genommen werden soll
8. Synchronisation der Dateien findet statt, so dass auf beiden Seiten der gleiche Datenbestand vorliegt (Status in der Liste wird aktualisiert)
9. nach erfolgreicher Synchronisation ist der Einladungsvorgang abgeschlossen und in der Liste wird als Status der Erfolg der Einladung angezeigt
10. Host wählt einen weiteren Teilnehmer aus (springe zu Schritt 3) oder schließt den Einladungsdialog

Erweiterungen / alternative Abläufe:

- Alternativer Anfang 1: gerade zuvor wurde ein Projekt veröffentlicht (Use Case 1) und der Einladungsdialog ist dadurch bereits geöffnet (Schritt 1 entfällt)
- Alternativer Anfang 2: im Kontextmenü des Teilnehmers (View *Roster*) wurde *Invite user to shared project* ausgewählt (Schritte 1-3 entfallen)
- 7a: Der eingeladene Teilnehmer wählt ein bestehendes Projekt als Ausgangsbasis und lässt für die verteilte Programmiersitzung eine Kopie des Projektes erstellen.
- 7b: Der eingeladene Teilnehmer lässt vom Werkzeug ein bestehendes Projekt finden, dass vom Datenbestand am ähnlichsten ist. Der Grad der Übereinstimmung wird angezeigt. Danach hat er alle Möglichkeiten wie in 7/7a beschrieben.
- 5a: Der eingeladene Teilnehmer lehnt die Einladung ab, dies wird als Status in der Liste angezeigt und zu Schritt 10 gesprungen.
- 5b: Der Einzuladende ist bereits in einer anderen Programmiersitzung. Er wird über die Einladung informiert und kann entscheiden, ob er die Sitzung wechseln möchte.

spezielle Anforderungen:

- Für eine schnelle Synchronisation ist eine Peer-to-Peer Verbindung zwischen den Teilnehmern nötig.

Häufigkeit des Auftretens: mittel

Use Case 3

Name: **Teilnehmer verlässt Sitzung**

Primärakteur: Teilnehmer

Stakeholder und Interessen:

- Teilnehmer: will Programmiersitzung verlassen

Vorbedingungen:

- Teilnehmer nimmt an Sitzung teil (wird im View *Shared Project Session* angezeigt)

Nachbedingungen:

- Teilnehmer nimmt an Sitzung nicht mehr teil (wird bei allen Teilnehmern im View *Shared Project Session* nicht mehr angezeigt)

Standardablauf:

1. Teilnehmer drückt den Button zum Verlassen der Sitzung im View *Shared Project Session*

Erweiterungen / alternative Abläufe: siehe Use Case 4 (*Host verlässt Sitzung*)

spezielle Anforderungen: keine

Häufigkeit des Auftretens: mittel

Offene Fragen: keine

Use Case 4

Name: **Host verlässt Sitzung**

Primärakteur: Host

Stakeholder und Interessen:

- Host: will Programmiersitzung verlassen

Vorbedingungen:

- Programmiersitzung wurde vom Host initiiert

Nachbedingungen:

- Programmiersitzung wurde beendet, alle Teilnehmer haben Sitzung verlassen
- der Datenbestand ist bei allen Teilnehmern konsistent

Standardablauf:

1. Host drückt den Button zum Verlassen der Sitzung im View *Shared Project Session*
2. alle Teilnehmer werden darüber informiert, dass der Host Sitzung verlassen will, und werden aufgefordert, ihre Aktivitäten einzustellen
3. alle Teilnehmer bestätigen das Beenden der Programmiersitzung und verlassen damit die Sitzung
4. Datenbestand wird in einen konsistenten Zustand gebracht und die Programmiersitzung beendet
5. Erfolgsmeldung beim Host über das erfolgreiche Beenden der Sitzung
6. alle Teilnehmer können am Projekt lokal alleine weiterarbeiten

Erweiterungen / alternative Abläufe:

- 3a: obwohl einige Teilnehmer das Beenden der Sitzung nicht bestätigten, erzwingt der Host den Prozess des Beendens
- alternativer Ablauf (**Host ist über das Netz nicht mehr erreichbar**)
    1. ist der Host für eine bestimmte Zeit nicht erreichbar, wird eine Warnmeldung bei allen Teilnehmern angezeigt
    2. Funktionen für die Interaktion (Rollenwechsel, Nachrichten, usw.) werden deaktiviert
    3. Teilnehmer können weiterarbeiten und die Veränderungen werden zwischengespeichert
    4. Nachdem wieder eine Verbindung zum Host besteht, werden die Teilnehmer darüber informiert, die zwischengespeicherten Änderungen werden abgearbeitet und die Funktionen für die Interaktion wieder aktiviert.

spezielle Anforderungen: keine

Häufigkeit des Auftretens: mittel

Use Case 5

Name: **Textänderungen durchführen (einfache Editieroperationen)**

Primärakteur: Driver

Stakeholder und Interessen:

- Driver: will gemäß seiner Rolle Änderungen am Quelltext vornehmen
- Observer / andere Driver: wollen gemäß ihrer Rolle Änderungen am Quelltext mitbekommen

Vorbedingungen:

- es muss eine verteilte Programmiersitzung existieren, die gewünschten Teilnehmer bereits daran teilnehmen

- Teilnehmer, der Änderungen durchführen will, muss Rolle Driver besitzen

Nachbedingungen:

- Änderungen müssen bei allen Teilnehmer vollzogen sein

Standardablauf:

1. Driver modifiziert den Quelltext (Lösch- oder Einfügeoperation)
2. Veränderung werden bei allen anderen Teilnehmern nachvollzogen (evtl. mit Transformation der Operation)
3. Awareness-Informationen (Veränderungen hervorheben, evtl. Sichtbereichsanzeige ändern , usw.) werden bei allen anderen Teilnehmern dargestellt

Erweiterungen / alternative Abläufe:

- falls sich ein Observer im Verfolgermodus für diesen Driver befindet, wird evtl. Fensterbereich gescrollt

spezielle Anforderungen:

- Nebenläufigkeitskontrolle muss die Konsistenz (Konvergenz, Kausalitäts- und Intentionserhaltung) sicherstellen

Häufigkeit des Auftretens: sehr häufig

Use Case 6

Name: **Verfolgermodus**

Primärakteur: Teilnehmer

Stakeholder und Interessen:

- beobachtender Teilnehmer: will einen anderen Teilnehmer (Driver oder Observer) verfolgen
- der zu beobachtende Teilnehmer: will im Rahmen einer kollaborativen Programmiersitzung beobachtet werden

Vorbedingungen:

- verteilte Programmiersitzung muss existieren, die gewünschten Teilnehmer bereits daran teilnehmen

Nachbedingungen:

- Fensterbereich des Editors beim Beobachter wird kontinuierlich so angepasst (durch evtl. scrollen), dass die Veränderungen am Quelltext bzw. das Selektieren von Textstellen beobachtet werden können
- neben dem Namen des beobachtenden Teilnehmer wird im View *Roster* angezeigt, dass dieser verfolgt wird

Standardablauf:

1. beobachtender Teilnehmer: Rechtsklick im View *Shared Project Session* auf den zu beobachtenden Teilnehmer (Kontextmenü) und Auswählen der Aktion *follow*

Erweiterung (**der beobachtete Teilnehmer verlässt Sitzung**):

- im View *Shared Project Session* wird signalisiert, dass der beobachtete Teilnehmer die Sitzung verlassen hat

Alternativen: keine

spezielle Anforderungen: keine

Häufigkeit des Auftretens: häufig

Use Case 7

Name: **Rollenwechsel**

Primärakteur: Host

Stakeholder und Interessen:

- Host: ist der Einzige, der die Rollenzuweisungen in der Programmiersitzung ändern darf
- Teilnehmer: möchte, dass seine Rolle gewechselt wird

Vorbedingungen:

- es muss eine verteilte Programmiersitzung existieren, an der der Teilnehmer teilnimmt

Nachbedingungen:

- Rollenwechsel für den Teilnehmer wurde durchgeführt und wird im View *Shared Project Session* entsprechend angezeigt
- Schreibberechtigung wird entsprechend der Rolle gesetzt

Standardablauf:

1. Host wählt im View *Shared Project Session* den Teilnehmer aus, deren Rolle gewechselt werden soll
2. im Kontextmenü wählt der Host eine der Aktionen zum geben oder zum wegnehmen der Driver Rolle aus

Erweiterungen / alternative Abläufe: keine

spezielle Anforderungen: keine

Häufigkeit des Auftretens: häufig

Use Case 8

Name: **Kommunikation**

Primärakteur: Teilnehmer

Stakeholder und Interessen:

- Teilnehmer: möchte mit anderen Teilnehmern kommunizieren

Vorbedingungen:

- verteilte Programmiersitzung muss existieren, die gewünschten Teilnehmer bereits daran teilnehmen

Nachbedingungen:

- Nachricht ist an alle anderen Teilnehmern verschickt worden
- Nachricht wurden bei allen anderen Teilnehmern angezeigt

Standardablauf:

1. Teilnehmer: öffnet den View *Chat* (falls dieser nicht geöffnet ist)
2. View zum Empfangen und Versenden von Textnachrichten wird angezeigt
3. im unteren Teil des Views wird die zu versendende Nachricht eingeben
4. bei allen anderen Teilnehmern wird (falls noch nicht bereits geöffnet) der View geöffnet und die Nachricht im oberen Teil angezeigt (mit Nickname des Senders und dessen Hintergrundfarbe)
5. Empfänger kann auf die Nachricht antworten (gehe zu Schritt 3) oder nicht antworten und den View entweder offen halten oder schließen

Schritte 1-2 können entfallen, falls der View bereits geöffnet ist

Alternativer Ablauf (**Skype Kommunikation**):

1. Teilnehmer: Rechtsklick im View *Shared Project Session* auf den gewünschten Kommunikationspartner (Kontextmenü) und Auswählen der Aktion *Skype this User*
2. zu dem ausgewählten Teilnehmer wird eine Verbindung mit Skype aufgebaut (es wird das Programm Skype aufgerufen und ein Gruppenunterhaltung mit dem ausgewählten Teilnehmer geöffnet)

spezielle Anforderungen: keine

Häufigkeit des Auftretens: häufig

Use Case 9

Name: **Durchführung von Dateioperationen oder komplexen Editieroperationen (Refaktorisierungen)**

Primärakteur: Driver

Stakeholder und Interessen:

- Driver: will eine Dateioperation oder komplexere Editieroperation (Refaktorisierung) durchführen

Vorbedingungen:

- verteilte Programmiersitzung muss existieren, die gewünschten Teilnehmer bereits daran teilnehmen
- Teilnehmer, der Änderungen durchführen will, muss Rolle Driver besitzen

Nachbedingungen:

- bei allen Teilnehmer müssen die Veränderungen vollzogen worden sein

Standardablauf:

1. Driver wählt wie in Eclipse gewohnt Dateioperationen oder Refaktorisierung aus
2. falls kein anderer Driver betroffene Datei geöffnet hat, springe zu Schritt 5
3. Alle anderen Driver, die eine betroffene Datei geöffnet haben, werden darüber informiert, dass eine Dateioperation oder eine Refaktorisierung durchgeführt werden soll, welche ein exklusives Schreibrecht erfordert.
4. alle anderen Driver geben ihre Zustimmung
5. für alle anderen Driver werden betroffene Dateien gesperrt (d.h. der Schreibzugriff wird entzogen)
6. Veränderungen werden bei allen Teilnehmern durchgeführt
7. während der Durchführung werden die Dateien in der Dateiliste als gesperrt markiert
8. nach der Durchführung bekommen alle Driver wieder für die betroffenen Dateien Schreibzugriff
9. Markierungen in der Dateiliste werden entfernt

Erweiterung (**während der Durchführung will jemand eine betroffene Datei öffnen**):

- derjenige wird darüber informiert, dass gerade eine Sperre auf der Datei ist und er diese nur lesend öffnen kann

alternative Abläufe: keine

spezielle Anforderungen: keine

Häufigkeit des Auftretens: mittel

C Aufgaben aus dem empirischen Experiment

C.1 Vorbereitungsaufgabe: Count

Input: *count.in*
Output: *count.out*

This problem is strictly to acclimate teams to the contest environment. We strongly suggest you first finish this problem, and then attempt the more complex practice problem.

Input

The input file will contain an unknown number of lines with at most 100 characters on each line. All the characters will be printable ASCII characters. Note that the input file will always exist, but may be empty.

Output

The number of lines in the input file.

Sample input

one
and two
three

Output for sample input

3

Anmerkung: Diese Aufgabe stammt aus einem ACM Programming Contest [39].

C.2 Aufgabe 1: Doors and Penguins

Input: *doors.in*
Output: *doors.out*

The organizers of the Annual Computing Meeting have invited a number of vendors to set up booths in a large exhibition hall during the meeting to showcase their latest products. As the vendors set up their booths at their assigned locations, they discovered that the organizers did not take into account an important fact - each vendor supports either the Doors operating system or the Penguin operating system, but not both. A vendor supporting one operating system does not want a booth next to one supporting another operating system.

Unfortunately the booths have already been assigned and even set up. There is no time to reassign the booths or have them moved. To make matter worse, these vendors in fact do not even want to be in the same room with vendors supporting a different operating system.

Luckily, the organizers found some portable partition screens to build a wall that can separate the two groups of vendors. They have enough material to build a wall of any length. The screens can only be used to build a straight wall. The organizers need your help to determine if it is possible to separate the two groups of vendors by a single straight wall built from the

portable screens. The wall built must not touch any vendor booth (but it may be arbitrarily close to touching a booth). This will hopefully prevent one of the vendors from knocking the wall over accidentally.

Input

The input consists of a number of cases. Each case starts with 2 integers on a line separated by a single space: D and P, the number of vendors supporting the Doors and Penguins operating system, respectively ($1 <= D, P <= 500$). The next D lines specify the locations of the vendors supporting Doors.

This is followed by P lines specifying the locations of the vendors supporting Penguins. The location of each vendor is specified by four positive integers: x_1, y_1, x_2, y_2. $(x1, y1)$ specifies the coordinates of the southwest corner of the booth while $(x2, y2)$ specifies the coordinates of the northeast corner.

The coordinates satisfy $x1 < x2$ and $y1 < y2$. All booths are rectangular and have sides parallel to one of the compass directions. The coordinates of the southwest corner of the exhibition hall is $(0, 0)$ and the coordinates of the northeast corner is $(15000, 15000)$. You may assume that all vendor booths are completely inside the exhibition hall and do not touch the walls of the hall. The booths do not overlap or touch each other.

The end of input is indicated by $D = P = 0$.

Output

For each case, print the case number (starting from 1), followed by a colon and a space. Next, print the sentence:

It is possible to separate the two groups of vendors.

if it is possible to do so. Otherwise, print the sentence:

It is not possible to separate the two groups of vendors.

Print a blank line between consecutive cases.

Sample input

```
3 3
10 40 20 50
50 80 60 90
30 60 40 70
30 30 40 40
50 50 60 60
10 10 20 20
2 1
10 10 20 20
40 10 50 20
25 12 35 40
0 0
```

Output for sample input

Case 1: It is possible to separate the two groups of vendors.

Case 2: It is not possible to separate the two groups of vendors.

Anmerkung: Diese Aufgabe stammt aus einem ACM Programming Contest [38].

C.3 Aufgabe 2: Durchführung einer Code-Durchsicht

Entpacken Sie die Datei *JUnit_Error.zip* und importieren Sie das Projekt in den Workspace von Eclipse (dies muss nur der Einladende tun und dieses Projekt anschließend mit den anderen „sharen“).

a)

Führen Sie eine Code-Durchsicht (Code Review) der Klasse *junit.framework.TestSuite* durch. Notieren Sie sich die gefundenen Defekte und korrigieren Sie diese.

- Ihr Fokus liegt auf dem Entdecken von Defekten, die durch Programmierfehler entstanden sind.
- Falls Sie bestimmte Codeteile nicht verstehen, dann ziehen Sie ggf. den Code anderer Klassen oder die JavaDocs von Junit (http://junit.sourceforge.net/javadoc/) zu Rate.

b)

Betrachten Sie nun die Klasse *junit.swingui.TestRunner*. Soviel sei verraten: Diese Klasse enthält zumindest einen Defekt, den Sie finden sollen.

Allerdings ist diese Klasse ziemlich umfangreich. Entscheiden Sie nun selbst, ob Sie eine Durchsicht oder eine andere Methode zur Lokalisierung des Defektes verwenden wollen.

Eine andere Möglichkeit wäre z.B. die Durchführung von Anwendungstests auf der Klasse. Stoßen Sie hierbei auf ein Versagen, können Sie versuchen, den zugehörigen Defekt z.B. unter Zuhilfenahme eines Debuggers zu lokalisieren. Die Schwierigkeit liegt hier in der Konstruktion geeigneter Testfälle. Allerdings sollten Sie mit hier nahe liegenden Testfällen schon ein Versagen erzeugen können. Sollte Ihnen dies nicht gelingen, lesen Sie den nachstehenden Tipp. Eines sollten Sie in diesem Fall auf jeden Fall zuerst tun: Beheben Sie die unter Aufgabenteil a) gefundenen Fehler.

Tipp: Schreiben Sie sich eine einfache Klasse und zugehörig eine Testklasse mit zwei Testmethoden, die sicher fehlschlagen und zwei Testmethoden, die sicher nicht fehlschlagen. Fügen Sie die Testfälle zu einer Suite zusammen und führen Sie diese mit dem Swingui-Testrunner aus. Sehen Sie sich das Ergebnis genau an.

D Fragebögen

D.1 Fragebogen 1

Frage 1: Wie viel Erfahrung hast Du mit Pair Programming (PP)?

- ☐ noch nie etwas über Pair Programming gehört
- ☐ bis jetzt nur in einer Vorlesung darüber gehört
- ☐ schon ein paar Mal (< 5) mit PP entwickelt
- ☐ schon öfters (5 bis 20 Mal) mit PP entwickelt
- ☐ schon oft (> 20 mal) mit PP entwickelt

wenn mehr als 20, schätze wie oft:

Frage 2: Sind Deine Erfahrungen mit PP positiv oder negativ?

- ☐ keine Ahnung, habe noch nie mit PP entwickelt
- ☐ positiv
- ☐ eher positiv
- ☐ eher negativ
- ☐ negativ
- ☐ weiß nicht

Begründe Deine Antwort:

Frage 3: Wie viel Erfahrung hast Du mit Distributed Pair Programming (DPP)?

- ☐ noch nie etwas über Distributed PP gehört
- ☐ bis jetzt nur in einer Vorlesung darüber gehört
- ☐ schon ein paar Mal (< 5) mit DPP entwickelt
- ☐ schon öfters (5 bis 20 Mal) mit DPP entwickelt
- ☐ schon oft (> 20 mal) mit DPP entwickelt

wenn mehr als 20, schätze wie oft:

Frage 4: Sind Deine Erfahrungen mit DPP positiv oder negativ?

- ☐ keine Ahnung, habe noch nie mit DPP entwickelt
- ☐ positiv
- ☐ eher positiv
- ☐ eher negativ
- ☐ negativ
- ☐ weiß nicht

Begründe Deine Antwort:

Frage 5: Hast Du schon mal einen collaborative real-time editor verwendet?

- ☐ ja
- ☐ nein

wenn ja, welche?

Fragen zu Deiner Person:

Student?

- ☐ ja
- ☐ nein
- ☐ ehemaliger Student

Fortschritt des Studiums:

- ☐ Grundstudium
- ☐ Hauptstudium
- ☐ abgeschlossenes Studium

Fachrichtung und Art des Studiums (Diplom oder Bachelor/Master):
ungefähre Programmiererfahrung (in Jahren):

D.2 Fragebogen 2

Teil A (Ablauf der Programmiersitzung):

Frage 1: In welcher Gruppe hast Du am Experiment teilgenommen?

Frage 2: Wie oft habt Ihr in der Programmiersitzung von der Möglichkeit des zeitgleichen Schreibens Gebrauch gemacht?

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Frage 3: Warst Du in der Programmiersitzung hauptsächlich Driver oder Observer (in Saros, das bedeutet hattest Du Schreibrechte)?

- ☐ Driver (und oft geschrieben)
- ☐ Driver (aber wenig geschrieben)
- ☐ Observer

Frage 4: Wie oft habt Ihr in Saros die Rollen (Driver/Observer) getauscht?

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Frage 5: Wie oft hast du den Follow Mode verwendet?

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Frage 6: Wie oft hast Du den Follow Mode verlassen, um Dir andere Code- Stellen anzuschauen?

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Frage 7: welche Möglichkeiten der Interaktion hast Du benutzt?

Sprache:

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Selektionen im Editor:

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Textnachrichten (Chat):

- ☐ viel
- ☐ wenig
- ☐ gar nicht

Teil B (Bewertung von Saros):

Frage 1: Wie bewertest Du insgesamt die Softwareentwicklung mit Saros?

- ☐ positiv
- ☐ eher positiv
- ☐ eher negativ
- ☐ negativ
- ☐ weiß nicht

Begründe Deine Antwort:

Frage 2: Wie würdest Du die Möglichkeit der Interaktion zwischen den Entwicklern einstufen?

- ☐ gut
- ☐ ausreichend
- ☐ nicht ausreichend
- ☐ viel zu wenig Möglichkeiten

Begründe Deine Antwort:

Frage 3: Würdest Du Dir Video als Art der Interaktion wünschen?

- ☐ ja
- ☐ nein

wenn ja, warum?

Frage 4: Wie gut hat Dir die Möglichkeit des gleichzeitigen Schreibens (mehrere Driver) gefallen?

- ☐ positiv
- ☐ eher positiv
- ☐ eher negativ
- ☐ negativ
- ☐ weiß nicht

Begründe Deine Antwort:

Frage 5: für welche Tätigkeiten kannst Du Dir den Einsatz von Saros vorstellen, für welche weniger?

Gemeinsame Programmiersitzung

- ☐ sehr geeignet
- ☐ geeignet
- ☐ weniger geeignet
- ☐ gar nicht geeignet

Begründe Deine Antwort:

Peer Review (Code-Durchsicht)

- ☐ sehr geeignet
- ☐ geeignet
- ☐ weniger geeignet
- ☐ gar nicht geeignet

Begründe Deine Antwort:

Schulungen

- ☐ sehr geeignet
- ☐ geeignet
- ☐ weniger geeignet
- ☐ gar nicht geeignet

Begründe Deine Antwort:

Frage 6: Was hat Dir beim Entwickeln mit Saros besonders gefallen?

Frage 7: Was hat Dir beim Entwickeln mit Saros nicht gefallen?

Frage 8: Hast Du Verbesserungsvorschläge?