Bachelor-Arbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Unterstützung mehrerer Projekte in einer Saros-Sitzung

Christian Dohnert
Matrikelnummer: 4196798
donut87@googlemail.com

Betreuer: Dr. Karl Beecher
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Elfriede Fehr

10. März 2011

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 ahnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Christian Dohnert

Berlin, 07. März 2011

0 Verwendete Notationen und Begriffe

0.1 Notationen

Englische Begriffe lassen sich teilweise nur schwer und manchmal überhaupt nicht sinnvoll in die deutsche Sprache übersetzen. Solche Begriffe sind *kursiv* dargestellt. Dazu zählen vor allem die Begriffe aus 0.2

Neue Begriffe und Abkürzungen sind **fett** markiert.

0.2 Begriffe

Workspace Ein *Workspace* ist ein Ordner in Projekte und Einstellungen gespeicher werden. Man kan mehrere *Workspaces* haben. Das ist zum Beispiel sinnvoll, wenn man Projekte in verschiedenen Programmiersprachen hat und Eclipse für jede Sprache ein anderes Aussehen haben soll.

Debugging ist die systematische Defektsuche in Software.

Buddy wird ein Kontakt in einer Kontaktliste genannt.

Wizard Ein *Wizard* ermöglicht es einem Nutzer schrittweise Daten einzugeben, mit denen Eclipse dann Aktionen ausführt. Auch schrittweise Anleitung lassen sich mit *Wizards* realisieren.

WizardPage Ein *Wizard* enthält immer mindestens eine (*WizardPage*). Diese Seiten enthalten dann die notwendigen Felder um Daten einzugeben (beispielsweise Textfelder). Auch Informationen können dem Nutzer mitgeteilt werden. Ein gutes Beispiel dafür ist die erste Seite des 'JoinSessionWizard' aus Saros (vgl. Abbildung 3 in Abschnitt 3.2).

ZipStream ist eine in Java enthaltene Klasse, die zum komprimierten Übertragen von Datenströmen verwendet wird.

Repository Das *Repository* ist das Projektarchiv eines Versionskontrollsystems.

Patch Ein *Patch* ist eine Änderung am Quelltext (vom englischem: Flicken).

Branch In Versionskontrollsystemen können Verzweigungen (englisch: *Branches*) erstellt werden. In diesen wird dann unabhängig vom Hauptentwicklungszweig programmiert.

Plug-in Ein *Plug-in* ist eine Software aber kein eigentständiges Programm. Es erweitert die Funktionalität oder das Aussehen ein bestehendes Computerprogramm.

X-Arbeiter X-Arbeiter sind die Studenten, die gerade eine Abschlussarbeit zum Thema Saros schreiben. Das 'X' ist der Platzhalter für Diplom, Bachelor oder Master.

Inhaltsverzeichnis

1 Einleitung

Das Ziel dieses Kapitels ist es die wesentlichen Grundlagen auf denen meine ganze Arbeit aufbaut zu erklären. Es gibt einen kurzen Einblick in die Oberfläche von Eclipse und eine kompakte Einführung in zwei Praktiken kollaborativen arbeitens. Zum Schluss gehe ich dann darauf ein wie Saros kollaboratives arbeiten mit Parteien möglich macht, die auf der ganzen Welt verteilt sein können.

1.1 Eclipse

Eclipse ist eine integrierte Entwicklungsumgebung (**IDE**[1]), die ursprünglich nur für die Programmiersprache Java gedacht war. Seit Version 3.0 aus dem Jahr 2004 ist Eclipse eine durch *Plug-ins* erweiterbare Plattform. Durch die Erweiterbarkeit von Eclipse kann nun jede beliebige Programmiersprache unterstüzt werden, wobei für die meisten der verbreitetesten Programmiersprachen (vgl. Abbildung 6) bereits *Plug-ins* existieren. *Plug-ins* dienen aber nicht nur der Unterstützung weiterer Programmiersprachen. So gibt es Erweiterungen, die Versionskontrollsysteme einbinden[2], Aufgaben organisieren[3] oder auch WYSIWYG-Designer[4] für graphische Komponenten[5].

Die Oberfläche von Eclipse wird von den ***Views*** gestaltet. Ein *View* dient hierbei einem speziellen Zweck oder Funktionalität. Eine sehr wichtige *View* ist der „Project Exlporer“. In diesem *View* werden alle Projekte des *Workspace* mit den dazugehörigen Dateien dargestellt Je nach Programmiersprache ändert sich hier die Darstellungsweise.

Ein Editor wird auch in einem *View* dargestellt. Hat man mehrere Dateien gleichzeitig geöffnet ist normalerweise trotzdem nur ein Editor zu sehen. Die anderen sind im Hintergrund und über Reiter zu erreichen. Die meisten *Plug-ins* erweitern Eclipse um mindestens einen *View*. So ist fügt das *Plug-in* zum einbinden eines Versionskontrollsystems *Subclipse* u.a. einen *View* ein, mit der man die Verzeichnisstruktur des Archivs durchsuchen kann.

1.2 Paarprogrammierung und Side-by-Side Programming

Paarprogrammierung und *Side-by-Side Programming* sind beides Praktiken kollaborativem arbeitens. Die Paarprogrammierung wird von Kent Beck in seinem 1999 erschienen Buch „Extreme Programming Explained “ beschrieben. Alistair Cockburn schlägt in [Coc05] das *Side-by-Side programming* als eine andere Art gemeinsam zu programmieren vor.

Beide möchte ich hier kurz erklären und einige Vor- und Nachteile nennen.

1.2.1 Paarprogrammierung

Paarprogrammierung ist eine Praktik des *Extreme Programming*[6]. Sie sieht vor, dass jeglicher Quelltext eines Projekts von jeweils zwei Programmierern

[1] Abkürzung für den englischen Begriff: Integrated Developement Environment
[2] Sublcipse: http://subclipse.tigris.org/
[3] Task Tag Decorator: http://sourceforge.net/projects/tasktagdecorat/
[4] WYSIWYG steht für „What you see is what you get„— meint also, was man bearbeitet sieht dann später auch genau so aus
[5] Jigloo: http://www.cloudgarden.com/jigloo/
[6] http://www.extremeprogramming.org/

erstellt wird. Beide sitzen vor dem gleichen Computer. Es darf jedoch nur einer von beiden Tastatur und Maus bedienen und somit tatsächlich Quelltext produzieren, während der andere überlegt, z.B. ob der Ansatz der richtige ist oder ob es vielleicht bessere Lösungen geben würde [Bec99].

Die Kontrolle über die Eingabegeräte wechselt zwischen den zwei Programmierern. Der Zeitpunkt an dem gewechselt wird, variiert je nach angewandtem Prinzip. Üblich ist es diesen Wechsel nach logischen Abschnitten aufzuteilen. Ein logischer Abschnitt kann hier ein zu lösendes Problem wie ein Bug oder eine Teilfunktion sein. Nach Ablauf einer festen Zeit (z.B. 60 Minuten) zu wechseln ist auch möglich. Somit sind beide Programmierer für den erstellten Quelltext verantwortlich.

Die Paarprogrammierung hat das Potenzial viele Vorteile mit einem Nutzen für ein gesamtes Software-Projekt zu bringen[7]. Ein typischer positvier Effekt ist eine geringere Zahl an Defekten. Da jedes Fragment des Quelltextes von mindestens zwei Leuten bearbeitet wurde, ist das Verständnis für den Quelltext weiter verteilt. Die Zahl derer die ein Projekt gefährden können, wenn sie es verlassen, verringert sich dadurch.

In kommerziellen Projekten stehen diesen Vorteilen höhere Kosten gegenüber. Es müssen zwei Programmierer dafür bezahlt werden an einer Aufgabe gemeinsam zu arbeiten, die (theoretisch) auch ein einzelner Entwickler bewältigen könnte. Ebenfalls können Schwierigkeiten entstehen, wenn Menschen mit unterschiedlichen Arbeitsgewohnheiten zusammenarbeiten sollen. Konfliktpotenzial ergibt sich auch aus den Persönlichkeiten der Programmierpartner. Passen die Partner auf der Ebene der Persönlichkeit nicht zueinander, wird das gemeinsame Arbeiten erschwert.

1.2.2 Side-by-Side programming

Als einen „in den ersten Monaten eines Projekts [...] guten Ausgangspunkt“ werden in [Coc05] neun Techniken genannt. Eine davon ist das *Side-by-Side programming*.

Dabei arbeiten wie bei der Paarprogrammierung zwei Entwickler als Team. Im unterschied dazu darf aber jeder Entwickler an einem eigenen Computer arbeiten. Diese Computer stehen aber so nahe beieinander, dass einer dem anderem „mit einer leichten Drehung des Kopfes“ [Coc05] schauen kann was der andere gerade tut. Ein Abstand von ca. 60 Zentimetern ermöglicht genau das.

Es ist natürlich nicht der Sinn zu schauen ob der andere auch tatsächlich arbeitet. Vielmehr wird versucht positive Auswirkungen der Paarprogrammierung mit persönlichen Arbeiten und Aufgaben (aller Art) zu verbinden. Beide sollen jederzeit ansprechbar sein und sich so gegenseitig helfen können. Hilfe ist in diesem Kontext vielseitig sein. Beide können sich z.B. zu einer kurzen Paarprogrammierungs-Sitzung zusammentun oder gemeinsam Defekte suchen.

An Grenzen stößt man hier, wenn man versucht dieses Prinzip auf eine Gruppe mit mehr als zwei Personen zu erweitern.

Drei Monitore lassen sich zwar nebeneinander hinstellen, oder auch kreisförmig anordnen damit jeder neben jedem sitzt. Es gibt aber keine Möglichkeit, wie jeder mit nur einer leichten Kopfdrehung auf einen anderen als den eigenen

[7] Alle hier aufgezählten Vor- und Nachteile habe ich [AB] entnommen

Monitor schauen kann. Drei, Vier oder gar noch mehr Entwickler können mit dieser Technik nicht kollaborativ arbeiten.

Das *Side-by-Side programming* eigenet sich also nur für Gruppen mit zwei Personen.

1.3 Saros

Wie oben erwähnt funktionieren die beiden Praktiken Paarprogrammierung und Side-by-Side programming nur, wenn die beiden Programmierer sich gleichzeitig im selben Raum aufhalten. Nun gibt es aber Szenarien in denen es nicht möglich ist, dass sich Personen, die zusammen programmieren wollen, nicht treffen können. So zum Beispiel in einer Firma mit verschiedenen Standorten.

Auch bei Projekten im Bereich *Open Source* ist das ein Problem, da auch hier die Programmierer auf dem gesamten Globus verteilt sein können.

Saros ist ein *Plug-in* für Eclipse für verteiltes kollaboratives Arbeiten. Verteilte Teams können damit im Sinne der *Paarprogrammierung* oder des *Side-by-Side Programming* arbeiten. Das Problem verschiedener Zeitzonen kann Saros jedoch nicht lösen.

Einen weiteren Vorteil gegenüber des konventionellen *Side-by-Side Programming* hat Saros. Es ermöglicht neben dem verteilten Arbeiten zu zweit, auch die kollaborative Arbeit mit mehreren Personen. Um das näher zu erklären gehe ich jetzt auf die einzelnen Funktionen und das Aussehen von Saros ein. Da ich die Erstellung einer Saros-Sitzung später noch erkläre, gehe ich hier nur auf den Zustand einer bereits existierenden Sitzung mit drei Benutzern ein.

1.3.1 Die Oberfläche von Saros

Den bisherigen Sitzungsaufbau beschreibt die Arbeit „Einladungsprozess in Saros“ [Só09] im Detail. Auch ich werde im Hauptteil dieser Arbeit noch einmal darauf eingehen und die von mir vorgenommenen Veränderungen darstellen. Deshalb beschreibe ich hier welche Funktionen und Möglichkeiten eine Saros-Sitzung mit drei Personen bietet.

Die zwei wichtigsten Elemente sind der *View Saros View* und das Editorfenster. In der linken Hälfte von *Saros View* (**1**) werden alle Benutzer der Sitzung und alle Kontakte der Freundesliste (*Buddy list*) (**2**) angezeigt. Rechts ist ein *Multi User Chat* (**3**) über den alle Teilnehmer einer Saros-Sitzung miteinander (schriftlich) kommunizieren können. Im Editor wird mit Hilfe von farbigen Annotationen verdeutlicht

- wer welchen Quelltext verfasst hat (**4**)
- für wen gerade welcher Abschnitt im Dokument sichtbar ist (**5**)
- die Cursorposition jeder Person(**6**)
- welche Textstellen markiert sind (**6**)

1.3.2 Wie Saros arbeitet

Jeder Teilnehmer einer Saros-Sitzung hat eine eigene Kopie jedes bearbeiteten Projekts. Ebenso werden die Änderungen die man macht (schreiben, löschen,

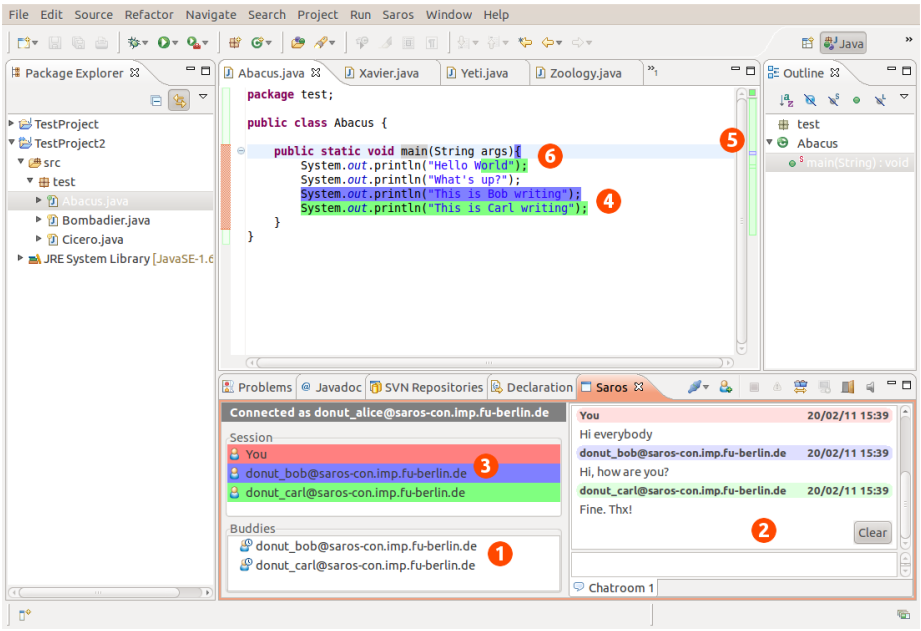

Abbildung 1: Saros aus der Sicht des Host

verschieben etc.), zunächst lokal ausgeführt. Die Änderungen werden dann an alle Sitzungsteilnehmer verteilt und dort ausgeführt. Änderungen am Quelltext dürfen natürlich nur Teilnehmer mit Schreibberechtigung machen.

Nicht jede Aktion (Änderungen am Quelltext, eine andere Datei öffnen, Text markieren etc.) wird sofort übertragen. Da Sarosnachrichten über das *Extensible Messaging and Presence Protocol* (XMPP) verschickt werden, wird zwischen den einzelnen Übermittlungen ein Zeitabstand eingehalten um zu vermeiden wegen *Floodings*[8] vom XMPP-Server verbannt zu werden.

Bis zum Ende 2010 wurden alle Nachrichten zwischengespeichert und im Intervall von 1000 Millisekunden versendet. Verglichen mit konventioneller *Paarprogrammierung* ist das eine große Verzögerung. Deshalb ist es inzwischem jedem Nutzer möglich diesen Wert zu verändern. Dabei sieht die Voreinstellung vor 300 Millisekunden zu warten (vgl. 5.2.6).

Außerdem gibt seit dieser Umstellung kein festes Intervall mehr. Statt immer nach Ablauf der Zeit alle im Speicher befindlichen Aktionen zu senden, wird nachdem einmal etwas gesendet wurde ein 'Timer' mit dem vom Benutzer angegebenen Wert gestartet. Alle Aktionen die in dieser Zeitspanne gesendet werden sollen, werden zwischengespeichert und erst nach Ablauf der 'Frist' gesendet. Sollte der Nutzer innerhalb dieser Zeitspanne nichts senden wollen, werden die nächtsen Aktionen sofort übertragen und der 'Timer' wird erneut gesetzt.

[8] Als Flooding bezeichnet man das überschwemmen des Servers mit Nachrichten

1.3.3 Was Saros möglich macht

Ich habe nun kurz zusammengefasst wie man mit Saros kollaborativ arbeiten kann. Die relativ geringe Latenz zwischen den lokalen und entfernten Aktionen (schlimmstenfalls 300 Millisekunden) lässt die Unterschiede zwischen echtem nebeneinander arbeiten und arbeiten mit verteilten Teilnehmern geringer werden. *Paarprogrammierung* oder *Side-by-Side Programming* sind so auch möglich, wenn die Partner nicht im selben Raum sitzen.

Ich hatte auch erwähnt, dass das kollaborative Arbeiten mit Saros nicht mehr nur auf zwei Programmierer beschränkt ist. Mit einem Doppelklick auf einen Sitzungsteilnehmer gelangt man zu der Datei, die derjenige gerade ansieht oder bearbeitet. Jemandem „über die Schulter zu schauen“ohne sich dabei aus seinem Stuhl zu bewegen ist damit also noch einfacher und lässt sich auf beliebig viele Teilnehmer anwenden. Die „Kopfdrehung“ wird hier durch einen Mausklick ersetzt.

2 Motivation

Die letzte Anfrage ob Saros mehr als ein Projekt unterstütze erreichte das Saros-Team mitte November 2010. Anfragen von Anwendern waren Motivation zur Erweiterung der Funktionalität (vgl 5.4.1). Ich möchte noch auf die Entwicklung von Saros eingehen um die Sinnhaftigkeit dieser Anforderung darzustellen. Ein Blick auf den Status quo wird dann noch unterstreichen, dass eine solche Veränderung notwendig war.

2.1 Die Entwicklung von Saros

Zu einer Entwicklung gehören sowohl Verbesserungen als auch Erweiterungen. Alle Veränderungen dieser Kategorie hier darzustellen sprengt jedoch den Rahmen dieser Arbeit. Ich möchte hier einige ausgewählte Verbesserungen und Erweiterungen erwähnen. Mein Ziel ist es dabei zu zeigen als was Saros begann und in welche Richtung es sich verändert hat.

2.1.1 Vor meiner Arbeit

Saros war ursprünglich ein Werkzeug für die verteilte Paarprogrammierung. Es entstand im Rahmen der Diplomarbeit „Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung“ [Dje06] und sollte zwei Benutzern an verschiedenen Orten ermöglichen ein Projekt im Sinne der *Paarprogrammierung* (vgl. 1.2.1) zu bearbeiten. Deshalb sah das technische Design nur einen schreibberechtigten Benutzer (damals noch *Driver* genannt) vor. Auf mehrere Observer wurde in „Anbetracht des engen Zeitplans“ [Dje06] vorerst verzichtet. Dennoch sah „die Implementierung diese Funktionalität bereits an allen Punkten bereits vor.“ [Dje06]. Eine Erweiterung auf mehrere Observer wurde dann im April 2007 umgesetzt und bis zum August 2007 verbessert (vgl. 5.2.1). Die Möglichkeit viele schreibberechtigte Benutzer zu haben wurde aber erst mehr als ein Jahr später eingeführt. Die Nutzer konnten in den Eclipse-Einstellungen festlegen ob mehrere Schreiber zugelassen sind oder nicht. Ein halbes Jahr später wurde diese Einstellung zur Standardoption (vgl. 5.2.2). Saros wurde auch um Möglichkeiten der Kommunikation erweitert. Um zu kommunizieren, waren anfangs noch externe Instant-Messenging- und VoIP-Programme notwendig. Sowohl ein Chat als auch eine fernmündliche Kommunikation wurden im März 2010 zu Saros hinzugefügt (vgl. 5.2.3). Die Abhängigkeit der Nutzer von externen Programmen wurde damit verringert. Auch die Option einem Sitzungsteilnehmer einen Einblick auf den eigenen Bildschirm zu geben ist seit Mai 2010 eine Funktion von Saros (vgl. 5.2.5).

2.1.2 Während meiner Arbeit

Mit der Version 11.1.28 von Saros wurde auch eine Whiteboardfunktionalität in Saros integriert. Riad Djemili stellt in seiner Arbeit [Dje06] fest, dass diese Funktion „von den Benutzern gewünscht wird“. Nun können Teilnehmer einer Sitzung auch Skizzen gemeinsam anfertigen.

Die Funktionen eines Versionskontrollsystem in Saros einzubinden war schon seit Mitte 2010 in Arbeit. Seit Version 11.1.7 wird, sobald ein Versionskontrollsystem zu einem Projekt gehört, dieses auch automatisch genutzt. Das heißt,

dass nicht mehr das gesamte Projekt vom Einladenden zum Eingeladenen transportiert werden muss. Vielmehr wird zuerst die Revision des Einladenden aus dem Versionskontrollsystem kopiert und dann nur die Dateien übertragen die der Einladende lokal verändert hat. Das kann Zeit während des Einladungsprozesses sparen, wenn die verschiedenen Teilnehmer über eine asynchrone Internetverbindung miteinander kommunizieren und sich das Versionskontrollsystem auf einem Server befindet, der über eine schnelle Datenanbindung verfügt. Alle Veränderungen am Quelltext in Verbindung mit der Versionskontrolle können in umgewandelt werden. Die Veränderungen wird dann so übersetzt, dass für Teilnehmer ohne Anbindung an das Versionskontrollsystem das Ganze so aussieht, als hätte der Sitzungsersteller alles per Hand verändert. Diese Funktion ist wichtig, da nicht jeder Sitzungsteilnehmer zwangsläufig Zugriff auf das Versionskontrollsystem hat. Trotzdem an der Sitzung teilnehmen. Auch wenn ein eingeladener Teilnehmer diese Funktionen freiwillig deaktiviert oder kein Zugriff auf das *Repository* hat, greift die Sicherung und er kann trotzdem in vollem Umfang an der Sitzung teilnehmen.

2.1.3 Nach meiner Arbeit

An Saros wird weiterhin gearbeitet. So wird es demnächst möglich sein gemeinsam ein Programm zu *debuggen*. In Arbeit ist auch noch eine Verbesserung der VoIP-Unterstützung. Zur weiteren Verbesserung der Kommunikation soll es auch möglich werden mit beliebigen Personen aus der *Buddy*-Liste eine Unterhaltung zu starten (völlig unabhängig davon ob einer der beiden Teilnehmer einer Saros-Sitzung ist oder nicht). Auch die kürzlich eingeführten Neuerungen wie das Whiteboard lassen noch Raum für Verbesserungen und Erweiterungen.

2.2 Status quo

Ein Projekt zu einer laufenden Sitzung hinzuzufügen ist bereits seit Anfang 2010 möglich aber sehr umständlich. Um noch einmal die Notwendigkeit meiner Arbeit zu unterstreichen möchte ich hier einmal die Anleitung preisgeben, wie mehr als ein Projekt in einer Saros-Sitzung zu realisieren war.

1. Der Einladende der Sitzung fügt zuerst das Projekt hinzu.
    - Er sollte spätestens danach (besser aber schon im Vorfeld) alle Sitzungsteilnehmer darüber informiert haben.
2. Diese müssen nun in ihrem *Workspace* ein Projekt wählen und ebenfalls zur Sitzung hinzufügen.
    - Dieses kann ein neu anglegtes oder auch schon existierendes sein. Wählt der 'Client' ein bereits existierendes Projekt, sollte dies auch dem vom 'Host' hinzugefügtem Projekt entsprechen.
3. Daraufhin wird man als Eingeladener Sitzungsteilnehmer aufgefordert den Namen des Projekts auf der Seite des Einladenden anzugeben.
    - Deshalb muss der 'Host' auch schon vorher alle Teilnehmer darüber informiert haben.

4. Nachdem der 'Host' eine beliebige Projektdatei verändert hat, meldet der 'Consistency-Watchdog'[9] eine Inkonsistenz, sofern das Projekt nicht mit dem auf Host-Seite übereinstimmt. Diese wird nun nachdem der Prozess angestoßen ist, aufgelöst.
5. Alle Teilnehmer können in der um ein Projekt erweiterten Sitzung weiterarbeiten.

Dieser Vorgang muss für jedes Eclipse-Projekt und jeden 'Client' durchgeführt werden. Stößt ein neuer Teilnehmer zu Saros-Sitzung hinzu, erhält dieser beim Einladungsprozess nur das ursprünglich geteilte Projekt. Für jedes weitere Projekt muss er seinen Teil des Prozesses erneut durchführen. Diese Anleitung ist nur einer denkbar ungünstigen Stelle dokumentiert, dem Text im entsprechenden Eintrag des Versionskontrollsystems. Als Benutzer ist es jedoch schwer dies herauszufinden.

2.3 Zusammenfassung

Ich habe hier nun versucht die Entwicklung des Funktionsumfangs von Saros darzustellen. Schaut man sich diese bisherige und die geplante Entwicklung[10] an, merkt man, dass Saros bereits jetzt wesentlich mehr ist, als ein Werkzeug zur verteilten Paarprogrammierung. Es ist ein Werkzeug zum kollaborativen Arbeiten. Man stelle sich vor man wolle zu viert oder fünft gemeinsam wie beim *Side-by-Side Programming* arbeiten. Das ist mit Saros möglich, aber nur, wenn alle am selben Eclipseprojekt arbeiten wollen. Mehr als ein Projekt je Sitzung möglich zu machen erscheint sinnvoll. Die bisherige Lösung ist jedoch nicht für eine produktive Arbeit geeignet.

3 Mehr als ein Projekt gemeinsam bearbeiten

In diesem Abschnitt beschreibe ich chronologisch meine Änderungen an Saros. Um den Überblick nicht zu verlieren, ist an bestimmten Stellen noch einmal kurz der aktuelle Zustand des Quelltexts und des Verhaltens von Saros zusammengefasst.

3.1 Die Anforderungen

Die funktionalen Anforderungen an meine Implementierung:

- Eine Sitzung kann n Projekte beinhalten ($n \geq 0$)
- Eine Sitzung kann mit n Projekten gestartet werden ($n \geq 0$)
- Zur Sitzung können n Projekte gleichzeitig hinzugefügt werden ($n > 0$)
- Es können beliebig viele Projekte gleichtzeitig aus der Sitzung entfernt werden

Nebenbedingungen und nicht funktionale Anforderungen:

- Jeder Teilnehmer einer Saros-Sitzung muss an der Bearbeitung aller Projekte der Sitzung teilnehmen können.

[9] Mechanismus in Saros um Inkonsistenzen zu entdecken und beheben
[10] https://www.inf.fu-berlin.de/w/SE/ThesesHome

3.2 Aufteilen des Einladungsprozesses

Anhand der Arbeit von Tas Sóti [Só09] habe ich mir zuerst einen Überblick verschafft wie eine Saros-Sitzung überhaupt zu Stande kommt. Hier eine kurze Zusammenfassung:

1. Der Einladende lädt einen Benutzer ein.
   - Entweder durch die Aktion 'Share Project...' oder durch die Aktion 'Invite Buddy...' (siehe Abbildung 5)
   - Der Einladende hat ab diesem Punkt keine Kontrolle mehr über den Prozess, abgesehen von der Möglichkeit ihn abzubrechen.
2. Beim Eingeladenen erscheint die Einladung.

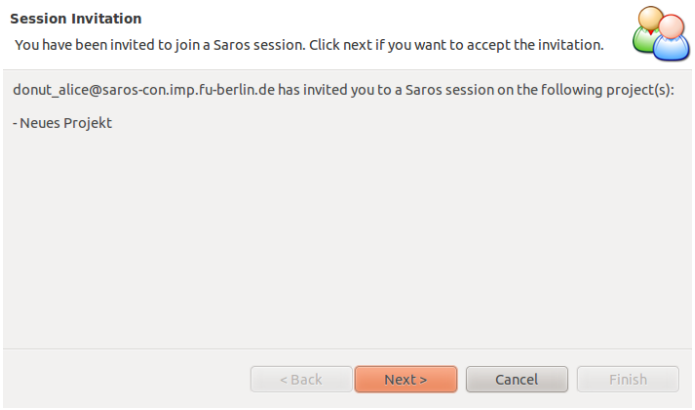

Abbildung 2: Seite 1 — Sie wurden zu einer Saros-Sitzung eingeladen.

3. Klickt der Eingeladene auf 'Next', akzeptiert er die Einladung. Jetzt wird eine Nachricht an den Einladenden verschickt. Diese Nachricht ist die Anforderung der vollständigen Dateiliste zum Projekt. Das signalisiert dem Einladenden, dass die Einladung angenommen wurde.
4. Saros generiert nun auf der Seite des Sitzungserstellers die Dateiliste und schickt sie zum Eingeladenen.
5. Dieser muss nun auf der zweiten Seite des *Wizards* entscheiden, welches Projekt er für die Sitzung verwenden möchte.
   - Zur Auswahl stehen hier:
     (a) Ein neues Projekt anlegen.
     (b) Ein bereits existierendes Projekt als Basis verwenden.
     (c) Ein bereits existierendes Projekt als Basis verwenden, aber eine Kopie davon erstellen, damit das Original unverändert bleibt.
6. Nachdem der Benutzer mit einem Klick auf 'Finish' bestätigt hat, wird anhand der vollständigen Dateiliste vom Einladenden eine Liste mit den Dateien erstellt, die sich unterscheiden.

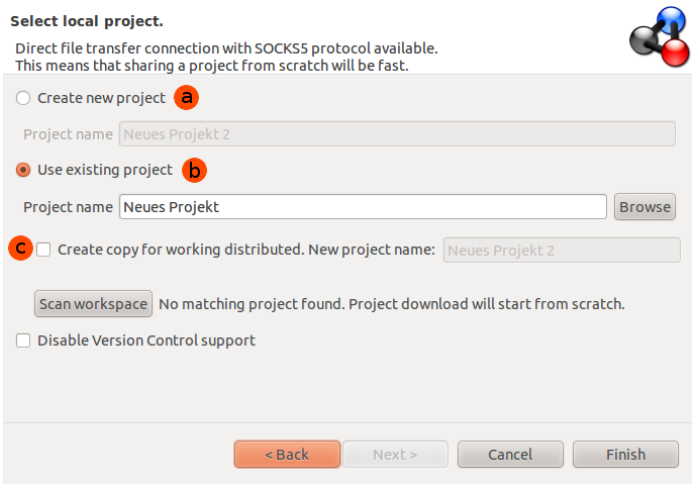

Abbildung 3: Seite 2 — Welche Basis soll für das neue Projekt genutzt werden?

7. Diese Liste wird wieder zurück an den Sitzungsersteller gesendet.
8. Die fehlenden Dateien werden dann vom Sitzungsersteller zum engeladenen Benutzer (je nach Einstellung einzeln oder als Archiv) gesendet.
9. Die Einladung ist abgeschlossen sobald das lokale Projekt mit dem Projekt der Sitzung synchronisiert ist. Der Eingeladene ist nun Teilnehmer der Saros-Sitzung.

Hier erkennt man bereits, dass ein Benutzer erst zum Sitzungsteilnehmer werden kann nachdem er das Projekt synchronisiert hat. Schaut man sich den Ablauf[11] der Methode 'start' in der Klasse OutgoingInvitationProcess (**OIP**) an, wird das noch deutlicher.

```
checkAvailability(monitor.newChild(1));

checkVersion(monitor.newChild(1));

sendInvitation(monitor.newChild(1));

getFileList(monitor.newChild(1));

if (!doStream) {
    createArchive(monitor.newChild(3));
    sendArchive(monitor.newChild(90));
} else {
    streamArchive(monitor.newChild(93));
}

completeInvitation(monitor.newChild(3));
```

[11] Dieser Ablauf ist wie alle anderen in diesem Abschnitt vereinfacht. Fehlerbehandlungen u.ä. wurden ausgelassen um nur das Wesentliche darzustellen

Die Methode 'sendArchive' oder 'streamArchive' wird vor der Methode 'completeInvitation' ausgeführt. Der Benutzer wird zwar schon vorher zur Sitzung hinzugefügt. Aber erst wenn die Einladung auch abgeschlossen ist, kann er an der Sitzung aktiv teilnehmen. Somit ist es weder möglich ein Projekt unabhängig von einer Sitzungseröffnung zu verteilen noch gibt es die Option eine Sitzung ohne Projekt zu starten. Auch auf der Gegenseite dem IncomingInvitationProcess (**IIP**) sind die Methoden zum Entpacken und Einordnen in den Prozess eingebunden.

Meine erste Aufgabe lautete nun also nicht „Füge ein Projekt zur Sitzung hinzu“, sondern „Teile den Einladungsprozess in zwei Teile auf“.

Die Methoden zum Erstellen und Senden des Archivs und Entpacken und Einsortieren der Dateien sollten nun jeweils in einer eigenen Klasse und somit in einem eigenständigen Prozess organisiert werden. Den OIP habe ich in die Klassen OutgoingSessionNegotiation (**OSN**) und OutgoingProjectNegotiation (**OPN**) geteilt. Aus dem IIP habe ich die Klassen IncominSessionNegotiation (**ISN**) und IncomingProjectNegotiation (**IPN**) gebildet. Bei dieser Neustrukturierung wollte ich natürlich bereits gemachte Verbesserungen nicht zunichte machen.

Ein wesentliche Verbesserung durch die Arbeit [Só09] war es, den Einladungsprozess jederzeit abbrechen zu können. Da ein Abbruch immer auf die gleiche Art und Weise behandelt wird (siehe: 1), war mein erster Ansatz eine gemeinsame (abstrakte) Oberklasse für alle Prozesse zu schaffen und in dieser einen Abbruch zu behandeln. Außerdem erschien mir die Idee sinnvoll die Prozesse soweit zu abstrahieren, dass an jeder Stelle im Saros-Quelltext an der auf einen der Prozesse zugegriffen wird, nur die abstrakte Oberklasse verfügbar ist.

Die Idee scheiterte an der Tatsache, dass die Prozesse doch zu unterschiedlich sind. Es gabe zu viele Stellen wie diese:

```
@Override
protected String getProjectName(){
    // we don't have a project name
    // in session negotiation
    return null;
}
```

Wie der Kommentar schon aufzeigt, ergibt es keinen Sinn einen Projektnamen anzufordern, wenn es im aktuellen Vorgang nur um die Eröffnung einer Sitzung geht. Nun sind in jedem Prozess die gleichen Methoden zum Abbruch implementiert. Das war zunächst nur als eine Übergangslösung gedacht. Da mir aber am Ende die Zeit fehlte über eine sorgfältigere Lösung nachzudenken, bleibt diese Lösung bis auf Weiteres bestehen.

Nachdem ich fertig war die beiden Klassen aufzuteilen, ergab sich ein neues Problem. Sobald eine Saros-Sitzung besteht, bekommt jeder Teilnehmer alle Aktivitäten der anderen Teilnehmer zugeschickt. Jeder neue Sitzungsteilnehmer erhält bereits Nachrichten die u.a. offene Dateien und Cursorpositionen betreffen, obwohl das korrespondierende Element (ein Projekt oder eine Datei) noch nicht in seinem *Workspace* vorhanden ist. Problematisch ist das nur bei solchen Aktivitäten, die ein gemeinsames Projekt betreffen. Um das Problem zu lösen, habe ich eine Warteschlange eingeführt, die eine zu einem bestimmten Projekt gehörende Aktivität solange erst einmal zurückhält. Wird ein Projekt

hinzugefügt, werden alle Aktivitäten für dieses Projekt aus der Warteschlange ausgeführt.

Solange keine Fehler während einer Sitzung auftreten, kann es nicht passieren, dass bei einem Teilnehmer Aktivitäten während der ganzen Sitzung zurückgehalten werden, da jeder Teilnehmer laut den Anforderungen (siehe: 3.1) an allen Projekten der Sitzung beteiligt ist.

Der Ablauf der OSN sieht nun so aus:

```
checkAvailability(monitor.newChild(10));

checkVersion(monitor.newChild(10));

sendInvitation(monitor.newChild(50));

addUserToSession();

completeInvitation(monitor.newChild(30));
```

Am Ende der Funktion 'completeInvitation' steht der Aufruf:

```
sarosSessionManager.startSharingProjects(peer)
```

Ist die Sitzung erfolgreich erstellt werden die Projekte[12] der Sitzung mit dem Eingeladenen geteilt. Die Erstellung der Sitzung ist hiermit beendet. Die Projekte mit dem Eingeladenen zu teilen ist ein neuer Prozess, für den der neue Sitzungsteilnehmer eine seperate Nachricht während der OPN zugeschickt bekommt. Hier der Ablauf:

```
sendFileList(monitor.newChild(1));

getRemoteFileList(monitor.newChild(2));
if (doStream) {
    streamArchive(monitor.newChild(97));
} else {
    // pack each project into one archive
    List<File> projectArchives = createProjectArchives(
        monitor.newChild(8), this.projectFilesToSend);
    File zipArchive = null;
    if (projectArchives.size() > 0) {
        // pack all archive files into one big archive
        zipArchive = File
            .createTempFile("SarosSyncArchive", ".zip");
        FileZipper.zipFiles(projectArchives, zipArchive,
            monitor.newChild(2));
     }
     // send the big archive
     sendArchive(monitor.newChild(87), zipArchive,
         processID);
```

Betrachtet man die Klasse OPN genauer (bzw. die Methode 'sendFileList' in Listing 4), sieht man, dass dem neu zur Sitzung gestoßenen Teilnehmer vorgegaukelt wird, die Projekte seien eben erst hinzugefügt worden. Ein Projekt wird

[12] im Moment nur eins, aber im Hinblick auf die kommenden Veränderungen bereits im Plural addressiert

also automatisch nach dem Beitreten eines Teilnehmers hinzugefügt. Da jetzt das verteilen eines Projekts losgelöst von der Eröffnung einer Sitzung abläuft, kann ein Nutzer das auch manuell ausführen.

Nach dieser Umgestaltung des Prozesses ist es auch notwendig die graphische Oberfläche anzupassen.
Wie bereits erwähnt und in den Abbildungen 2 und 3 zu sehen, gab es einen *Wizard* (der JoinSessionWizard), der durch die Sitzungserstellung geführt hat. Nach der Aufspaltung des Einladungsprozesses musste auch der JoinSessionWizard in zwei *Wizards* aufgeteilt werden. Dafür habe ich die zweite *WizardPage* in einen neuen *Wizard* übertragen. Diesen habe ich entsprechend seiner Funktion 'AddProjectToSessionWizard' genannt. Der erste *Wizard* dient jetzt nur noch zum Aktzeptieren einer Einladung, während der zweite die Einstellungen für das gemeinsame Projekt (also die zweite *WizardPage*) anzeigt.

Es existiert nun für beide Prozesse je ein eigener *Wizard*. Der 'JoinSessionWizard' hat nur noch Zugriff auf die Klasse IncominSessionNegotiation und der 'AddProjectToSessionWizard', darf nur noch den Prozess 'IncominProjectNegotiation' steuern.

3.3 Mehrere Projekte gleichzeitig hinzufügen

Eine Einladung zu einer Saros-Sitzung läuft nun so ab:

1. Der Einladende wählt ein Projekt aus, klick auf 'Share Project...' und wählt dann die einzulandenden Benutzer aus.
2. Die Einladung kommt bei allen Empfängern an.
3. Alle, die die Einladung angenommen haben, treten der Sitzung bei.
   - Die Eingeladenen sind jetzt Teilnehmer der Sitzung. Sarosfunktionen wie der Chat-Raum und das Whiteboard können genutzt werden.
4. Jetzt wird das Projekt mit allen Teilnehmern geteilt.
   - Tatsächlich ist der Eingeladene während er ein Projekt empfängt nicht in der Lage Eclipse zu benutzen, da dieser Prozess alle anderen Aktionen blockiert.

Eine Sitzung kann nun mehrere Projekte beinhalten. Jedes Projekt muss aber einzeln hinzugefügt werden. Nachdem eine Sitzung mehr als ein Projekt enthält sollte jetzt aber niemand mehr eingeladen werden. Es ist bis jetzt unmöglich dem Eingeladenen mehr als ein Projekt zu schicken.

Das nächste Ziel ist nun mit mehreren Projekten zu starten. Da das aber nichts anderes ist als direkt nach Sitzungsbeginn mehrere Projekte hinzuzufügen, geht das einher mit der Möglichkeit mehrere Projekte zu einer laufenden Sitzung zu einem beliebigen Zeitpunkt hinzuzufügen.

Auch hier ein paar kurze technische Details zu Saros ohne die es schwer ist meine weiteren Veränderungen zu erklären.

Dateiliste In 3.2 hatte ich bereits die Dateiliste erwähnt, aber noch nicht näher erklärt. Die Dateiliste eines Projekts wird in der Klasse *FileList* repräsentiert. Sie enthält folgende für den Austausch von Projekten wichtige Daten:

- Die Namen aller Dateien im Projekt mit einem dazugehörigen Hash-Wert, der anzeigt, ob die Datei bereits synchron mit der des Einladenden ist.
- Informationen zur Nutzung eines Versionskontrollsystems.

Sie enthält jedoch keine Information darüber zu welchem Projekt sie eigentlich gehört.

projectID ist eine für jedes in der Sitzung enthaltene Projekt identische Kennung. Nachdem ein Projekt zur Sitzung hinzugefügt wurde, wird der Projektname als *projectID* festgelegt. Diese Kennung ist u.a. wichtig, wenn es darum geht Änderungen am Quelltext der richtigen Datei im richtigen Projekt zuzuordnen.

Verschicken der Projekte Projekte (bzw. die benötigten Dateien) können auf zwei Arten übertragen werden.

1. Ein Archiv mit allen benötigten Dateien wird gepackt und an den Empfänger gesendet.
   Leider kann das bei zu großen Projekten dazu führen, dass die gesamte Anwendung abstürzt, da der Speicher der JVM[13] zu klein ist.
2. Jede Datei wird einzeln in einem Datenstrom gesendet.

Möchte man nun ein Projekt mit anderen Personen gemeinsam bearbeiten, verschickt man eine Dateiliste und eine *projectID*. Dass diese beiden Objekte zueinander gehören, ist zwar nur implizit klar, aber das reicht völlig aus. Es ist außerdem kein Problem ein Archiv oder einen Datenstrom im Austauschprozess dem korrekten Projekt zuzuordnen. Es gibt ja nur eins. Da ich nun aber mindestens zwei Projekte zur gleichen Zeit der Sitzung hinzufügen wollte, stand ich vor folgenden Problemen:

1. Eine Dateiliste muss eindeutig einem Projekt zuzuordnen sein.
2. Ein eingehendes Archiv muss eindeutig einem Projekt zuzuordnen sein.
3. Ein eingehender Datenstrom muss eindeutig einem Projekt zuzuordnen sein.

Problem 1 ließ sich leicht lösen. Der Klasse *FileList* habe ich das zusätzliche Attribut 'projectID' gegeben. Ohne dass ich weitere Änderungen vornehmen musste, hat auch die XML-Darstellung keine Probleme bereitet.

Um Problem 2 zu lösen hatte ich zwei Ideen. Die erste sah vor jedes zu versendende Projekt in ein eigenes Archiv zu packen. Der Empfänger signalisiert wenn er bereit ist die Pakete zu empfangen. Nachdem ein Paket übermittelt und verarbeitet wurde, wird eine Nachricht an den Versender geschickt, die ihn auffordert das nächste Paket zu schicken. Das kann dann solange fortgeführt werden, bis alle Pakete übermittelt wurden.

Dem gegenüber stand die Variante erst alle Pakete in je ein Archiv zu packen und dann diese vielen Archive zu einem großen zusammenzupacken. Das große Paket wird dann verschickt und der Empfänger muss zuerst aus dem einen

[13] Java Virtual Machine

großen Archiv die vielen kleineren Archive extrahieren. Die kleinen Archive werden dann verarbeitet.

Ich entschied mich nach Absprache mit meinem Betreuer für Variante 2. In der Übertragungsmethode liegt hier der entscheidende Vorteil. Diese musste nicht geändert werden, da immernoch ein einzelnes Archiv übetragen wird.
Um die kleinen Archive einem Projekt zuzuordnen erschien es mir als sinnvoll die *projectID* als Namen des Archives zu verwenden. Der Empfänger muss also aus dem Namen der Datei die *projectID* auslesen.

Ein Projektarchiv wird aber als temporäre Datei angelegt. Dafür wird die Java-Methode *File.createTempFile*[14] benutzt. Sie fügt aber an den gewählten Dateinamen eine Zufallszahl an. Um die *projectID* auszulesen muss die Zufallszahl eindeutig zu erkennen sein, damit sie ignoriert werden kann.

Das Projektarchiv nicht als temporäre Datei zu speichern hätte andere viel größere Probleme nach sich gezogen. Ein geeigneter Speicherort in den die JVM Dateien schreiben darf muss systemunabhängig festgelegt werden. Auch das Löschen der Datei muss nun selbst organisiert sein. All dies entfällt, wenn man eine temporäre Datei erzeugen lässt.

Dass die *projectID* immer der Name des Projekts ist, war ein Hindernis. Es war mir nicht möglich eine Zeichenkette zu finden, die in jedem Fall das Ende des Projeknamen kennzeichnen kann, da Eclipse für Projektnamen nur solche Zeichen verbietet, die auch in Dateinamen verboten sind. Somit besteht bei jeder noch so langen und zufälligen Zeichenkette die (zwar nur sehr geringe) Wahrscheinlichkeit, dass sie bereits im Projektnamen auftaucht. Die Lösung war die *projectID* anders festzulegen.

Dass die *projectID* der Projektname ist, war eine willkürlich Festlegung. Diese eindeutige Kennung wird nur einmal festgelegt, wenn ein Projekt zur Sitzung hinzugefügt wird. Der *projectID* einen anderen Wert zuzuweisen, stellte kein Problem dar.
Ich entschied mich für eine große Zufallszahl. Die Projekte einfach bei eins beginnend durchzunummerieren, da Java für eine temporäre Datei einen Namen mit mindestens drei Zeichen verlangt.

Der Archivname für ein Projekt setzt sich jetzt wie folgt zusammen: *projectID* + '&&&&' + eine von Java festgelegte Zufallszahl + '.zip'.
Die Zuordnung der Dateien im Datenstrom ist vergleichsweise einfacher. Die dafür verwendete Klasse *ZipStream* ermöglicht es, an eine zu verschickende Datei zusätzliche Informationen anzuhängen.

Es ist jetzt möglich mehrere Projekte gleichzeitig zu versenden und zu empfangen. Kommen nun mehrere Projekte zugleich an, kann die Entscheidung ein Projekt als Grundlage zu benutzen oder ein neues zu erstellen, für jedes Projekt unterschiedlich ausfallen. Im 'AddProjectToSessionWizard' können aber nur Einstellung für ein Projekt vorgenommen werden. Die graphische Oberfläche benötigt also eine erneute Anpassung.

Die Idee den entsprechenden *Wizard* mehrfach anzuzeigen habe ich verworfen. Mehrere sich öffnende Fenster sind für einen Benutzer nur schwer zu überschauen. Um die Sache übersichtlich zu halten, sind die Projekte im entsprechenden *Wizard* mit Reitern organisiert. Alle Einstellungen für ein Projekt werden in einem

[14] http://download.oracle.com/javase/1.5.0/docs/api/java/io/File.html
#createTempFile%28java.lang.String,%20java.lang.String%29

seperaten Reiter angezeigt. Das Ergebnis erfüllt seinen Zweck, lässt aber noch Spielraum für optische Verbesserungen. Das neue Aussehen im Überblick:

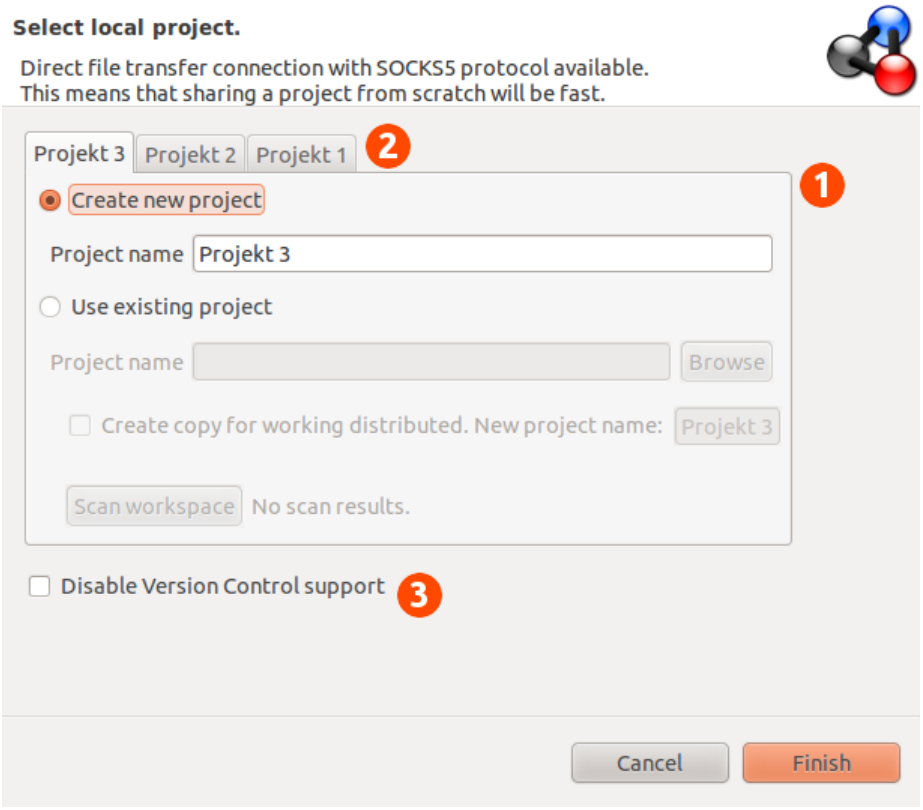

Abbildung 4: Zur Sitzung werden 3 Projekte hinzugefügt.

1. Der Bereich für projektspezifische Einstellungen.
2. Reiter mit Namen der Projekte auf der Seite des Einladenden.
3. Projektunabhängige Einstellungen für den gesamten Austauschvorgang.
    - Im Moment betrifft die einzige Einstellung hier nur die Option das Versionskontrollsystem explizit nicht zu nutzen.

Um die Reiter mit dem Projektnamen zu beschriften muss diese Information zum Eingeladenen gesendet werden. Da ich aber gerade die Entscheidung getroffen habe, für die *projectID* nicht mehr den Projektnamen zu verwenden, muss diese Information zusätzlich übertragen werden. Ohne den Projektnamen zu wissen wird es für einen zur Saros-Sitzung eingeladenen Teilnehmer schwierig zu entscheiden, welches Projekt in seinem *Workspace* er welchem eingehenden Projekt zuordnen soll. Da sowohl Dateilisten als auch Namen einem Projekt zugeordnet werden müssen, hielt ich es für sinnvoll eine neue Klasse zu erstellen, die alle Daten enthält, die beim Hinzufügen eines Projekts zur Sitzung notwendig sind. In der Klasse 'ProjectExchangeInfo' (siehe: 3) sind diese Informationen

zusammen mit der *projectID* gebündelt. Um die Daten zu versenden müssend diese Informationen noch serialisiert werden. Die Klasse 'ProjectExchangeInfoDataObject', enthält diese serialisierten Informationen. Das bedeutet in diesem Zusammenhang nur, dass die Dateiliste in XML-Form dargestellt ist (vgl. 2).

3.4 Was noch zu tun ist

Die Anforderung beliebig viele der Projekte aus einer Sitzung zu entfernen (vgl. 3.1) habe ich bisher nicht weiter erwähnt. Ich habe sie durch einen Mangel an Zeit nicht umsetzen können. Obwohl es anfangs recht einfach erschien, traten bei der Implementierung Probleme auf.

Das Projekt aus der Liste von Projekten zu löschen, funktioniert nur solange wie maximal einer der Teilnehmer eine der Dateien im Projekt bearbeitet bzw. geöffnet hat. Problematisch sind in diesem Fall die Annotationen (Farbkodierungen zu Cursorpositionen u.ä. – siehe 1.3.1). Diese müssen gelöscht werden und das Editorfenster muss aus der Liste der zur Sitzung gehörenden Editoren gelöscht werden. Das klingt zunächst recht einfach, ist aber komplizierter als gedacht. In der Klasse *EditorManager* werden alle Aktionen an diesen Fenstern gesteuert. Es existiert aber noch kein Mechanismus um gezielt ein solches Fenster aus der Liste zu entfernen. Das ist auch verständlich, da das bisher noch nicht nötig war. Aufgrund meiner sehr knappen Zeit (ca. ein bis zwei Tage), habe ich es nicht mehr geschafft mich in die Mechanismen dieser sehr großen[15] und nicht überall gut dokumentierten Klasse einzuarbeiten. Eine andere Anforderung habe ich nicht komplett erfüllt. Theoretisch ist zwar möglich eine Sitzung ohne ein Projekt zu starten, praktisch hat ein Benutzer aber keine Möglichkeit dazu. Auch hier verweise ich wieder auf die fehlende Zeit. Vorgesehen hatte ich, dass bei der Aktion 'Invite Buddy' (siehe Abbildung 5) ein Dialog erscheint, der nur Projekte aussuchen lässt. Eclipse bietet selbst keinen solchen Dialog. Man müsste also einen eigenen Dialog schreiben um den Benutzer darauf zu beschränken. Das erfordert einiges an Hintergrundwissen, was ich nicht habe.
Die Verwaltung der Projekte einer Saros-Sitzung obliegt dem Sitzungsersteller. Gedacht war aber, dass jeder Teilnehmer mit Schreibrechten die Möglichkeit Projekte hinzuzufügen. Genauso habe ich das auch implementiert, jedoch nicht ausreichend getestet. Es kam zu Fehlern bei diesem Prozess. Diese Fehler wurden erst sichtbar als die neue Version von Saros veröffentlicht werden sollte. Zu diesem Zeitpunkt habe ich aber bereits daran gearbeitet diese Arbeit zu verfassen. Dadurch fehlte mir die Zeit mich damit genauer zu befassen. Die Limitierung ist also eine Zwischenlösung.

3.5 Zusammenfassung und Ausblick

Die Neuerungen im Überblick:

1. Eine Saros-Sitzung kann beliebig viele Projekte enthalten.
2. Eine Saros-Sitzung kann mit beliebig vielen Projekten gleichzeitig gestartet werden.
3. Es können beliebig viele Projekte (auch gleichtzeitig) zu einer Sitzung hinzugefügt werden.

[15] zur Zeit 1800 Zeilen

So zusammengefasst scheinen die Neuerungen geringfügig. Die Zahl der möglichen Anwendungsszenarien ist aber gestiegen. Hier 2 Beispielszenarien die vorher nicht möglich waren.

In einer E-Mail (siehe 5.4.1) erreichte das Saros-Team eine Anfrage, ob die Möglichkeit besteht mehrere Projekte mit Saros gemeinsam zu bearbeiten.
Dieses Entwicker-Team arbeitet mit vielen Projekten an vielen Standorten. Sie können jetzt auch Paarprogrammierung und *Side-by-Side Programming* einsetzen, wenn mehrere Projekte zu bearbeiten sind.

Ein weiterer Anwendungsfall:
Alistair Cockburn berichtet in seinem Buch [Coc05] von einem leitendem Entwickler, der nur noch selten zum Programmieren kommt, da er viele Berichte schreiben muss und Unterhaltungen zu führen hat. Da er selbst wenig Quelltext schreibt, aber über einen guten Überblick und Erfahrung verfügt, setzt er sich (ganz im Sinne des *Side-by-Side Programming* neben einen Entwickler und fungiert hier als Ratgeber. Mit Saros ist es nun überhaupt kein Problem eine solche Person „neben“ mehrere Entwicker zu setzen. Auch wenn alle ihr eigenes Projekt haben, stellt dies jetzt kein Problem dar. Es ermöglicht dem leitenden Entwickler sogar noch eher Fragen zu deligieren, falls er selbst die Antwort nicht kennt oder gerade doch keine Zeit hat.

4 Rückblick auf einen Prozess

Das Ziel dieses Kaptitels ist es den Prozess meines Arbeitens zu reflektieren. Als Leitfaden für den ersten Teil dienen mir hier u.a. die Regeln für Abschlussarbeiten in der Arbeitsgruppe Software Engineering[16].

4.1 Einem rotem Faden folgen

Am Anfang dieser Arbeit standen klar definierte Anforderungen (siehe: 3.1). Das unterscheidet meine Arbeit von manch anderer Arbeit. Es gab kaum Platz für Interpretationen der Aufgabe.

4.1.1 Mit den Anforderungen vertraut machen

Da ich die Software Saros schon aus dem Kurs „Softwareprojekt: Softwaretechnik“[17] kannte, verkürzte sich die Einarbeitungszeit in das Projekt. Mir blieb also vermeintlich mehr Zeit als üblich um meine Anforderungen umzusetzen und die damit einhergehenden Probleme zu lösen.

4.1.2 Meinen Plan präsentieren

Nachdem mir klar war was ich zu tun hatte, sollte ich das in einer kurzen Präsentation im Seminar „Beträge zum Software Engineering“ vortragen, ohne dabei auf zu viele technische Details einzugehen. Das stellte mich vor eine Herausforderung. Von mir wurde erwartet, dass ich deutlich darlege was ich vorhabe, aber trotzdem nicht zu viel über technischen Details erzähle. Ich habe in zwei Punkten von dieser Präsentation profitiert.

1. Ich habe meinen Plan visualisiert.
2. Ich habe Rückmeldungen zu diesem Plan erhalten.

Die Visualisierung war während des Programmierens recht nützlich. Ich konnte immer wieder meine eigenen Entwicklungen mit meiner Idee abgleichen. Die Rückmeldung war bei mir im Wesentlichen positiv. Es war angenehm zu wissen, dass ich mich nicht schon am Anfang in eine Sackgasse verrenne.

4.1.3 Meinen Plan nicht aus den Augen verlieren

Meinen Plan im Auge zu behalten und mich auch weiterhin nicht zu verrennen, war ein Ziel der regelmäßigen Treffen mit meinem Betreuer. Laut den bereits erwähnten Regeln zu Abschlussarbeiten gibt es neben den regelmäßigen Treffen noch eine andere Art der Betreuung. Eine regelmäßige stichpunktartige Zusammenfassung des eigenen Fortschritts schien mir und meinem Betreuer aber weniger effektiv als die regelmäßigen Treffen. Das allein wäre sicher nicht ausreichend gewesen um eine gute Betreuung zu garantieren. Dass ich mich gut betreut gefühlt habe, liegt vor allem daran, dass ich auch außerhalb dieser regelmäßigen Termine mit meinem Betreuer kommuniziert habe. Es gab Entscheidungen, die ich nicht allein treffen wollte oder konnte. Hätte ich bis zum nächsten regulärem Treffen warten müssen, hätte ich vermutlich nicht so viel von meiner Aufgabe geschafft.

[16] https://www.inf.fu-berlin.de/w/SE/ThesisRules
[17] https://www.inf.fu-berlin.de/w/SE/SoftwaretechnikProjekt2010

4.1.4 Teil eines Prozesses sein

In meinem Leitfaden steht, meine Arbeit solle „aus Sicht der Arbeitsgruppe teil eines Prozesses sein, der vor ihr begann und nach ihr weitergeht“ [Eng]. Der Prozess 'Entwicklung von Saros' hat zweifelsfrei vor meiner Arbeit begonnen und wird mit Sicherheit auch danach noch weitergehen. Die Frage ist also „Ist meine Arbeit Teil dieses Prozesses?“ Aus meiner Sicht ist sie das. Ich habe Saros verändert bzw. erweitert. Da ich nicht alle Anforderungen implementiert habe, werde ich auch über den Rahmen dieser Bachelor-Arbeit hinaus noch an dieser Software arbeiten.

4.2 Arbeiten in einem Team

Fünf Abschlussarbeiten die Verbesserungen und Neuerungen in Saros bewirken, waren in Arbeit als ich mit meiner angefangen habe. Davon sind einige bereits fertiggestellt und andere noch in Arbeit. Weitere wurden begonnen während meiner Arbeit an meiner Aufgabe. Diese Gegebenheiten stellen klar, dass an Saros mehrere Personen gleichzeitig arbeiten. Wir sind also eine Gruppe mit einem gemeinsamen Ziel: Die Verbesserung von Saros. Wir sind also ein Team.

4.2.1 Teamfähigkeit

Was hat Teamfähigkeit nun mit einer Bachelor-Arbeit im Studienfach Informatik zu tun? Die Antwort auf diese Frage ist: „In einem Team sollten alle Mitglieder fähig sein in diesem Team zu arbeiten.“ Und was bedeutet es *teamfähig* zu sein? Zur zweiten Frage habe ich eine (meiner Meinung nach) gelungene Definition gefunden: „Teamfähigkeit bedeutet also, mit anderen zu kooperieren und im Hinblick auf ein angestrebtes Ziel effektiv Ergebnisse zu erzielen.“ [See06] Ich werde also versuchen zu beleuchten wie in diesem Team kooperiert wird um Ergebnisse zu erzielen. Deshalb verliere ich ein paar Worte zur Arbeitsumgebung mit ihren Möglichkeiten und Schwächen (4.2.2). Danach möchte ich etwas darüber sagen wie Änderungen am Quelltext von Saros zu Stande kommen sollen (4.2.3) und welche Schwierigkeiten es dabei geben kann. Zuletzt hinterfrage ich noch die zusätzlichen Aufgaben im Saros-Team.

4.2.2 Der Saros-Raum

Als X-Arbeiter im Saros-Team hat man die Möglichkeit im studentischen Arbeitsraum des Sarosprojekt (kurz: der Saros-Raum) zu arbeiten. Dort sind 5 Arbeitsplätze mit Computern vorhanden. Diese kann man benutzen oder seinen eigenen Laptop mitbringen und ggf. an einen der Monitore anschließen. Niemand wird verpflichtet dort zu arbeiten. Aus meiner Sicht ist es aber von Vorteil es trotzdem zu tun. Hat man eine Frage, kann man sie direkt an die anderen Teammitglieder stellen. Vermutlich kann sie jemand direkt beantworten. Wäre man zu Hause, müsste man eine E-Mail schreiben und hoffen, dass sie jemand auch schnell beantwortet. Genauso kann man anderen mit dem eigenen Wissen weiterhelfen.

Diese direkte und synchrone Kommunkikation ist nicht nur bei Wissensfragen hilfreich. Um nach längerem Programmieren sichergehen, dass man keinen Fehler in den eigenen Entscheidungen hat, kann man auch jemanden bitten, sich erklären zu lassen was man getan hat. Das kann dabei helfen eigene Fehler

zu finden oder sich zu überzeugen, dass man alles genau so gemacht hat wie geplant.

Trotz der Vorteile sind die Arbeitsbedingungen nicht immer ideal. Diskusionen über Lösungswege zu Problemen, erklärende Gespräche und andere Unterhaltungen tragen manchmal zu einer Ablenkung von der eigenen Arbeit bei. Vor allem am Dienstag und Donnerstag ist der Raum meist stark frequentiert. An diesen Tagen sind wir aufgefordert mindestens um 14:00 Uhr anwesend zu sein. Dann findet nämlich das *Stand-Up-Meeting*[18] statt. Dazu mehr in 4.2.4. An den beiden 'Randtagen' der Woche Montag und Freitag ist es für gewöhnlich sehr ruhig im Saros-Raum. Man hat hier also die Chance in Ruhe zu arbeiten, gleichzeitig aber bei Problemen direkt mit dem eigenen Betreuer sprechen zu können. Als Frühaufsteher ergibt sich aber auch die Möglichkeit an den höher frequentierten Tagen ruhig arbeiten zu können. Vor zehn Uhr ist selten jemand im Saros-Raum.

4.2.3 Änderungen vornehmen und einbringen

Der Quelltext von Saros wird mit dem Versionskontrollsystem Subversion[19] verwaltet. Durch paralleles arbeiten an dem Quelltext von Saros können Konflikte zwischen den einzelnen Veränderungen entstehen. Es gibt zwei Möglichkeiten mit diesen Konflikten umzugehen.

1. Man erstellt sich einen Branch
2. Man arbeitet stets auf der aktuellen Version des Hauptentwicklungszweig.

Variante eins hat einen großen Vorteil. Von Änderungen im Hauptentwicklungszweig ist man nicht unmittelbar betroffen. Je länger sich diese Arbeit im eigenen Entwicklungszweig hinzieht, desto schwieriger wird es im Anschluss die eigene Entwicklung in den Hauptentwicklungszweig einzupflegen. Dieser Nachteil ist bei Variante zwei nicht gegeben. Vielmehr ist man dazu gezwungen die eigene Arbeit forlaufend anzupassen.

Eine weitere Aufgabe für jeden X-Arbeiter ist das Überprüfen der Veränderungen.

Zum einen gehört es zu den Aufgaben jedes Teammitglieds die *Patches* der anderen zu begutachten, zum anderen müssen natürlich auch die eigenen *Patches* begutachtet[20] werden. Zur Unterstützung dieses Prozesses gibt es das *Review Board*. Hier soll jeder seine *Patches* einstellen und mit einer Erläuterung versehen. Alle Mitglieder der Mailingliste *dpp-devel*[21] werden dann über neue Anfragen zu Durchsichten, Kommentare zu einzelnen Quelltextpassagen, Bewertungen und allen Aktualisierungen informiert. Je mehr Änderungen man also zeitgleich vornimmt, desto schwieriger wird es jemanden zu finden der die Änderungen durchsieht und prüft. Da zwei positiv bewertete Durchsichten notwendig sind um die Änderungen in Saros einpflegen zu dürfen, sinkt die Wahrscheinlichkeit gute Bewertungen zu erhalten mit Größe und Umfang des *Patches*. Auch mit kleineren Veränderungen kann man aber Probleme haben. Diese

[18] nach dem Vorbild der agilen Softwareentwicklung – http://www.extremeprogramming.org/rules/standupmeeting.html

[19] http://subversion.apache.org/

[20] http://www.saros-project.org/HowToDevel#How_to_contribute_code_to_Saros

[21] dpp-devel@lists.sourceforge.net — Alle X-Arbeiter sollten auf dieser Liste eingetragen sein

bauen in der eigenen Arbeit meist aufeinander auf. Trotzdem muss jede einzelne Veränderung oder Neuerung von der Gruppe für (mindestens) akzeptabel befunden werden. Das kann die eigene Arbeit ins Stocken bringen.

Anfangs habe ich versucht in einem eigenen *Branch* zu entwickeln. Nachdem ich aber gemerkt habe wie schnell sich Saros verändert, habe ich meine Entscheidung revidiert. So habe ich immer auf einer möglichst aktuellen Version gearbeitet. Für Tage an denen ich zu Hause gearbeitet habe, habe ich einen *Patch* erstellt und per E-Mail an mich selbst zugeschickt.

Da ich selbst sehr viele Stellen in Saros bearbeitet habe, gab es auch Tage an denen ich die Hälfte meiner Zeit damit verbracht habe Versionskonflikte zu lösen. An diesem Punkt hätte mir auffallen können, dass meine Änderungen zu groß sind und in kleine Teile hätten aufgeteilt werden sollen. Im Anschluss Bewertungen für meine Arbeit zu bekommen, war dementsprechend schwierig. Kaum jemand hatte Zeit sich wirklich mit meinem *Patch* auseinanderzusetzen.

An einigen Stellen musste ich zudem Fehler mehrfach berichtigen. Einige Quelltextpassagen, die ich zu Hause bearbeitet hatte, waren auf meinem Computer im Saros-Raum wieder im Urzustand. Ich kann mir das nur dadurch erklären, dass ich mit meinen *Patches* durcheinandergekommen bin.

Trotz dieser Probleme bin ich mit meiner Entscheidung nicht in einem *Branch* zu entwickeln zufrieden. Dadurch bin ich immer mit der aktuellen Version des Quelltextes vertraut gewesen.

4.2.4 Das Meeting

Wie bereits erwähnt, findet jeden Dienstag und Donnerst um 14 ein Treffen des Saros-Teams statt und alle X-Arbeiter sind aufgefordert teilzunehmen. Eingeleitet wird das Treffen mit dem Erscheinen eines Mitarbeiters der Arbeitsgruppe. Dann stehen alle auf und erzählen der Reihe kurz die folgenden drei Dinge:

1. Was ich seit dem letzten Teammeeting getan habe.
2. Was ich geplant habe bis zum nächsten Teammeeting zu tun.
3. Auf welche Probleme ich gestoßen bin.

Ein Ziel dieser Treffen ist, dass keiner der X-Arbeiter, isoliert von allen anderen seiner Arbeit nachgeht, sondern jeder (wenigstens grob) weiß, was die anderen gerade tun (wir sind immerhin ein Team). Erreicht wird das nur, wenn auch alle zu dieser Zusammenkunft erscheinen. Ein weiteres Ziel ist es Denkanstöße zu angesprochenen Problemen zu geben. Das funktioniert meiner Erfahrung nach gut. Manchmal funktioniert es auch zu gut. Obwohl bei einer längeren Erlärung der Lösung auf den Zeitpunkt nach dem Meeting verwiesen werden soll, wird direkt während des Meetings über das Problem diskutiert. Ein Hinweis auf den Inhalt dieses Meetings ist ein gutes Mittel eine solche Diskusion zu beenden.

Manchmal habe ich mich durch diese Treffen gestört gefühlt. Das lag bei diesen wenigen Gelegenheiten aber daran, dass ich gerade in meine Arbeit vertieft war und nicht auf die Zeit geachtet hatte. Ich finde die Uhrzeit auch nicht geeignet um ein solches Treffen abzuhalten. Durch ein Gespräch mit einem Mitarbeiter der Arbeitsgruppe habe ich aber erfahren, dass dieses Uhrzeit gewählt wurde, damit möglichst viele X-Arbeiter anwesend sind.

4.2.5 Weitere Aufgaben im Team

Neben der Aufgabe eigene Veränderungen einzubringen und die der anderen zu bewerten, hat jeder X-Arbeiter auch noch andere Aufgaben. Ich meine hier nicht „Blumen gießen“oder „Kaffee kochen“. Die wichtigste Nebenaufgabe fällt einmal im Monat an und betrifft vier Teammitglieder. Da Anwender von Saros nicht genötigt werden sollen sich den Quelltext herunterzuladen und dann zu kompilieren, wird einmal im Monat (in der letzten Woche, die einen Freitag enthält) eine neue Version von Saros freigegeben. Als Test-Manager (**TM**) und Release-Manager (**RM**) sind zwei X-Arbeiter für diesen Prozess verantwortlich. Beide bekommen einen „Assistenten“ zur Seite gestellt. Die Assistenten haben den Prozess schon einmal in der entsprechenden Rolle selbst erlebt und können Hilfestellung bei Problemen geben. Die Arbeit für die vier Verantwortlichen beginnt am Dienstag Nachmittag und endet am Freitag Nachmittag, wenn eine neue Version zum installieren bereitsteht. Der genaue Ablauf ist unter `http://www.saros-project.org/HowToDevel#Our_Release_Schedule` beschrieben. Problematisch an diesem Ablauf ist nur, dass man vor allem als Hauptverantwortlicher ständig etwas zu tun hat. Man kommt also mir der eigentlichen Aufgabe im Rahmen der X-Arbeit in diesen Tagen nur langsam voran.

5 Anhang

5.1 Listings

5.1.1 Abbruch des Einladungsprozesses

Listing 1: Diese Methoden steuern das Abbrechen des Einladungsprozess

```
public void localCancel(String errorMsg, CancelOption
    cancelOption) {
    if (!cancelled.compareAndSet(false, true))
        return;
    log.debug("Inv" + Util.prefix(peer) + ": localCancel:
        " + errorMsg);
    if (monitor != null)
        monitor.setCanceled(true);
    cancellationCause = new LocalCancellationException(
        errorMsg,
        cancelOption);
}

@Override
public void remoteCancel(String errorMsg) {
    if (!cancelled.compareAndSet(false, true))
        return;
    log.debug("Inv" + Util.prefix(peer) + ": remoteCancel
        : " + errorMsg);
    if (monitor != null)
        monitor.setCanceled(true);
    cancellationCause = new RemoteCancellationException(
        errorMsg);
}

protected void executeCancellation() throws
    SarosCancellationException {

    log.debug("Inv" + Util.prefix(peer) + ":
        executeCancellation");
    if (!cancelled.get())
        throw new IllegalStateException(
            "executeCancellation should only be called
                after localCancel or remoteCancel!");

    String errorMsg;
    String cancelMessage;
    if (cancellationCause instanceof
        LocalCancellationException) {
        LocalCancellationException e = (
            LocalCancellationException) cancellationCause;
        errorMsg = e.getMessage();

        switch (e.getCancelOption()) {
        case NOTIFY_PEER:
            transmitter.sendCancelInvitationMessage(peer,
                errorMsg);
```

```
            break;
        case DO_NOT_NOTIFY_PEER:
            break;
        default:
            log.warn("Inv" + Util.prefix(peer)
                + ": This case is not expected here.");
        }

        if (errorMsg != null) {
            cancelMessage = "Invitation was cancelled
                locally"
                + " because of an error: " + errorMsg;
            log.error("Inv" + Util.prefix(peer) + ": " +
                cancelMessage);
            monitor.setTaskName("Invitation failed. (" +
                errorMsg + ")");
        } else {
            cancelMessage = "Invitation was cancelled by
                local user.";
            log.debug("Inv" + Util.prefix(peer) + ": " +
                cancelMessage);
            monitor.setTaskName("Invitation has been
                cancelled.");
        }

    } else if (cancellationCause instanceof
        RemoteCancellationException) {
        RemoteCancellationException e = (
            RemoteCancellationException) cancellationCause
            ;

        errorMsg = e.getMessage();
        if (errorMsg != null) {
            cancelMessage = "Invitation was cancelled by
                the buddy "
                + " because of an error on his/her side:
                    " + errorMsg;
            log.error("Inv" + Util.prefix(peer) + ": " +
                cancelMessage);
            monitor.setTaskName("Invitation failed.");
        } else {
            cancelMessage = "Invitation was cancelled by
                the buddy.";
            log.debug("Inv" + Util.prefix(peer) + ": " +
                cancelMessage);
            monitor.setTaskName("Invitation has been
                cancelled.");
        }
    } else {
        log.error("This type of exception is not expected
            here: ",
            cancellationCause);
        monitor.setTaskName("Invitation failed.");
    }
```

```
        sarosSession.returnColor(this.colorID);
        invitationProcesses.removeInvitationProcess(this);
        throw cancellationCause;
    }

    protected void checkCancellation(CancelOption
        cancelOption)
        throws SarosCancellationException {
        if (cancelled.get()) {
            log.debug("Inv" + Util.prefix(peer) + ":
                Cancellation checkpoint");
            throw new SarosCancellationException();
        }

        if (monitor == null) {
            log.warn("Inv" + Util.prefix(peer) + ": The
                monitor is null.");
            return;
        }

        if (monitor.isCanceled()) {
            log.debug("Inv" + Util.prefix(peer) + ":
                Cancellation checkpoint");
            localCancel(null, cancelOption);
            throw new SarosCancellationException();
        }
    }
```

5.1.2 Eine Dateiliste

Listing 2: Eine Dateiliste für ein Projekt mit 3 Dateien

```
<fileList>
  <useVersionControl>false</useVersionControl>
  <entries>
    <entry>
      <path>
        <segments>
          <string>src</string>
          <string>test</string>
          <string>Cicero.java</string>
        </segments>
        <separators>-516145488</separators>
      </path>
      <null/>
    </entry>
    <entry>
      <path>
        <segments>
          <string>.classpath</string>
        </segments>
        <separators>626474272</separators>
      </path>
```

```
      <null/>
    </entry>
    <entry>
      <path>
        <segments>
          <string>src</string>
          <string>test</string>
          <string>Abacus.java</string>
        </segments>
        <separators>-1934149680</separators>
      </path>
      <null/>
    </entry>
    <entry>
      <path>
        <segments>
          <string>.settings</string>
          <string>org.eclipse.jdt.core.prefs</string>
        </segments>
        <separators>1474381856</separators>
      </path>
      <null/>
    </entry>
    <entry>
      <path>
        <segments>
          <string>src</string>
          <string>test</string>
          <string>Bombadier.java</string>
        </segments>
        <separators>-1154734768</separators>
      </path>
      <null/>
    </entry>
    <entry>
      <path>
        <segments>
          <string>.project</string>
        </segments>
        <separators>-1070404352</separators>
      </path>
      <null/>
    </entry>
  </entries>
  <projectID>1320472574</projectID>
</fileList>
```

5.1.3 Die Klasse ProjectExchangeInfo

Listing 3: Die Klasse ProjectExchangeInfo — Alle Informationen zum versenden eines Projekts

```
package de.fu_berlin.inf.dpp.activities;
```

```
import de.fu_berlin.inf.dpp.FileList;

/**
 * This class contains all the information that an
    invited user needs. The
 * {@link FileList} of the whole project, the projectName
    and the session wide
 * projectID.
 */
public class ProjectExchangeInfo {
    protected FileList fileList;
    protected String projectName;
    protected String projectID;

    // The description is not used yet, but there was a
       description field all
    // the time and I didn't want to delete it. This
       field could be useful
    // sometime...
    protected String description;

    /**
     *
     * @param projectID
     *            session wide ID of the project. This ID
        is the same on all
     *            clients/buddies
     * @param projectName
     *            Name of the project on inviter side.
     * @param fileList
     *            Complete List of all Files in the
        project.
     */
    public ProjectExchangeInfo(String projectID, String
       description,
        String projectName, FileList fileList) {
        this.fileList = fileList;
        this.projectName = projectName;
        this.description = description;
        this.projectID = projectID;
    }

    public FileList getFileList() {
        return fileList;
    }

    public String getProjectName() {
        return projectName;
    }

    public String getDescription() {
        return description;
    }
```

```
    public String getProjectID() {
        return projectID;
    }

    public ProjectExchangeInfoDataObject
        toProjectInfoDataObject() {
        return new ProjectExchangeInfoDataObject(
            projectID, description,
            projectName, fileList.toXML());
    }
}
```

5.1.4 Die Methode sendFileList

Listing 4: Die Methode 'sendFileList' — Der start eines Projektaustausch

```
private void sendFileList(SubMonitor subMonitor)
        throws LocalCancellationException {

        subMonitor.setTaskName("Creating file list...");
        subMonitor.worked(50);
        useVersionControl = sarosSession.
            useVersionControl();
        List<ProjectExchangeInfo> pInfos = new ArrayList<
            ProjectExchangeInfo>(
            this.projects.size());
        try {
            for (IProject project : projects.values()) {
                FileList fileList = new FileList(project,
                    useVersionControl);
                fileList.setProjectID(sarosSession.
                    getProjectID(project));
                ProjectExchangeInfo pInfo = new
                    ProjectExchangeInfo(
                    sarosSession.getProjectID(project), "
                        ", project.getName(),
                    fileList);
                pInfos.add(pInfo);
            }
        } catch (CoreException e) {
            throw new LocalCancellationException(e.
                getMessage(),
                CancelOption.NOTIFY_PEER);
        }
        log.debug("Inv" + Utils.prefix(peer) + ": Sending
            file list...");
        subMonitor.setTaskName("Sending file list...");
        this.sarosSession.sendActivity(sarosSession.
            getUser(peer),
            new ProjectsAddedActivity(sarosSession.
                getLocalUser(), pInfos,
                processID, this.doStream));
```

```
        subMonitor.done();

    }
```

5.2 Changelogs

5.2.1 Mehrere Observer

```
r91 | ornis | 2007-04-02 23:10:39 +0200 (Mon, 02 Apr 2007)
multiple observer (some open TODOs!)
```

5.2.2 Mehrere Driver

```
r726 | chris_fu | 2009-02-22 17:15:56 +0100 (Sun, 22 Feb 2009)
[FEATURE] introduce multi driver support (EXPERIMENTAL),
the user can enable this feature in the preferences
(needs restart of eclipse or reopen of session view)

r798 | coezbek | 2009-02-26 09:11:12 +0100 (Thu, 26 Feb 2009)
[FEATURE] Multi-Driver mode does not need a restart
now to be disabled/enabled.

r1774 | marrin | 2009-09-21 16:05:20 +0200 (Mon, 21 Sep 2009)
[FIX] #2863403 Multi-Driver the default option.
```

5.2.3 Kommunkation

```
r2089 | testvogel | 2010-03-23 21:22:17 +0100 (Tue, 23 Mar 2010)
[FEATURE] experimental VoIP function added.
Two users of a shared project can start a simple
VoIP session now.

r2088 | testvogel | 2010-03-23 20:08:25 +0100 (Tue, 23 Mar 2010)
[FEATURE] MultiUserChat added, but it should be
marked experimental.
```

5.2.4 Mehrere Projekte

```
r2002 | coezbek | 2010-02-17 15:06:59 +0100 (Wed, 17 Feb 2010)
[INTERAL] Saros now uses SPath internally whenever possible.
This enables multi-project support.
[TODO] We need user interface to support this feature.
At the moment there is an experimental hack,
to connect additional projects to existing sessions.
In a session right click on an additional project as
host and "add to session..."
Each client needs to do this as well (yeah, it is a hack)
```

5.2.5 Screensharing

```
r2154 | s-lau | 2010-05-05 08:11:27 +0200 (Wed, 05 May 2010)
[FEATURE] screensharing among two users in a saros-session
```

5.2.6 Verringerte Latenz

```
r2892 | k_beecher | 2011-01-04 15:24:51 +0100 (Tue, 04 Jan 2011)
[FEATURE] Duration between sending of edits is now configurable
via the advanced prefs. It has also been reduced to 300ms,
which should greatly improve perceived latency.
```

5.3 Bidler

5.3.1 Jemanden Einladen

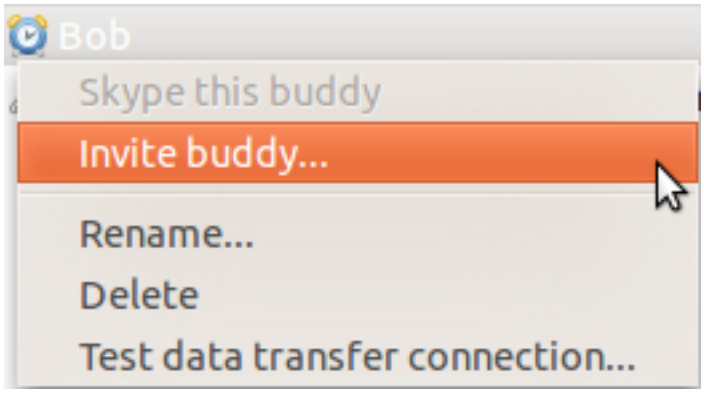

Abbildung 5: Jemanden zu einer Saros-Sitzung einladen

5.3.2 Tiobe Index

5.3.3 Eclipse

5.4 Verschiedenes

5.4.1 E-Mail

Die (anonymisierte) E-Mail eines Anwenders, der sich wünscht mehr als ein Projekt zu mit Saros bearbeiten zu können.

```
Hallo liebes Saros Team,

wir arbeiten aktuell mit sehr verstreuten
Entwicklerteams und arbeiten daher ständig daran, unsere
Entwicklungsumgebung für diese Gegebenheit zu optimieren.
Leider haben wir bisher Pair-Programming immer wieder genau
dann verworfen, wenn die Entwickler nicht
mehr in einem Raum saßen.
Statt dessen haben beschränken wir uns in diesen Fällen auf
```

Position Feb 2011	Position Feb 2010	Delta in Position	Programming Language	Ratings Feb 2011	Delta Feb 2010	Status
1	1	=	Java	18.482%	+1.13%	A
2	2	=	C	14.986%	-1.62%	A
3	4	↑	C++	8.187%	-1.26%	A
4	7	↑↑↑	Python	7.038%	+2.72%	A
5	3	↓↓	PHP	6.973%	-3.03%	A
6	6	=	C#	6.809%	+1.79%	A
7	5	↓↓	(Visual) Basic	4.924%	-2.13%	A
8	12	↑↑↑↑	Objective-C	2.571%	+0.79%	A
9	10	↑	JavaScript	2.558%	-0.08%	A
10	8	↓↓	Perl	1.907%	-1.69%	A
11	11	=	Ruby	1.615%	-0.82%	A
12	-	=	Assembly*	1.269%	-	A-
13	9	↓↓↓↓	Delphi	1.060%	-1.60%	A
14	19	↑↑↑↑↑	Lisp	0.956%	+0.39%	A
15	37	↑↑↑↑↑↑↑↑↑↑	NXT-G	0.849%	+0.58%	A--
16	30	↑↑↑↑↑↑↑↑↑↑	Ada	0.805%	+0.44%	A--
17	17	=	Pascal	0.735%	+0.13%	A
18	21	↑↑↑	Lua	0.714%	+0.21%	A--
19	13	↓↓↓↓↓↓	Go	0.707%	-1.07%	A--
20	32	↑↑↑↑↑↑↑↑↑↑	RPG (OS/400)	0.626%	+0.27%	A--

Abbildung 6: Die Verbreitung 20 bekanntesten Programmiersprachen nach TIOBE (www.tiobe.com)

häufige Reviews, die über "normales" Screensharing
gut realisierbar sind.

Nun bin ich auf Saros gestoßen und finde,
dass es eigentlich genau das ist, was wir gesucht haben.
Bis auf eine Kleinigkeit?:
Unser Code verteilt sich auf sehr viele Maven Module
und damit auch sehr viele Eclipse Projekte.
Um also ordentlich zusammenarbeiten zu können, müsste
die Freigabe auf jeden Fall über mehrere Projekte möglich sein.

Und nun endlich meine Frage: Ist diese Erweiterung bereits
geplant oder aus bestimmten Gründen evtl. gar nicht möglich?
Oder habe ich diese Funktionalität womöglich einfach übersehen?

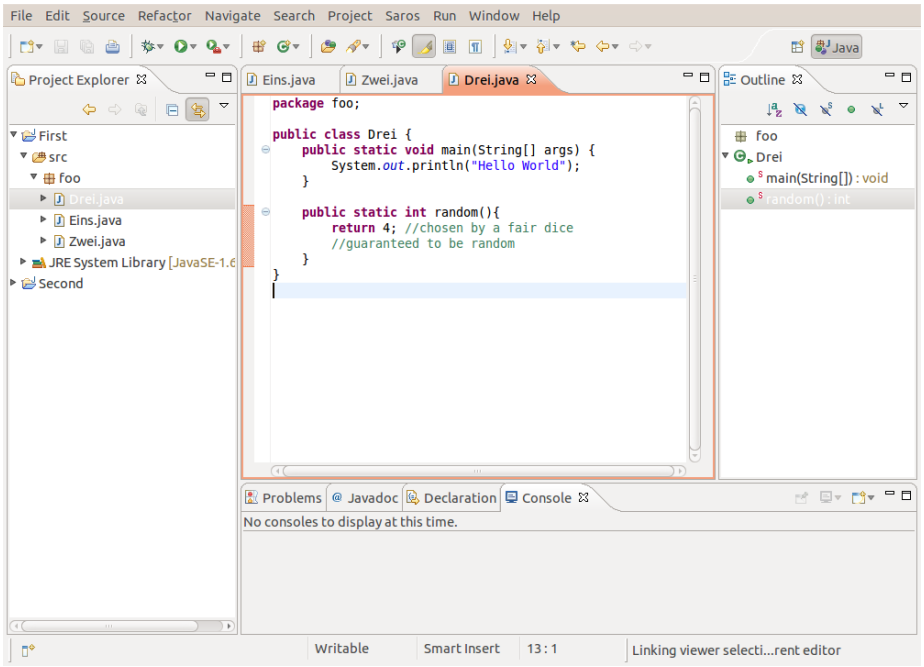

Abbildung 7: Eclipsefenster mit drei offenen Editoren

`Vielen Dank für das vielversprechende Plugin!`

Literatur

[AB] Nachiappan Nagappan Andrew Begel. Perceptions of agile software development at microsoft: A horizontal case study.

[Bec99] Kent Beck. Extreme programming explained, 1999.

[Coc05] A. Cockburn. *Crystal clear: agile Software-Entwicklung für kleine Teams ; [Strategien, Rollen und Prozesse ; typische Fehler erkennen und vermeiden ; zahlreiche Praxisbeispiele und eine detaillierte Fallstudie].* Mitp, 2005.

[Dje06] Riad Djemili. Entwicklung einer eclipse-erweiterung zur realisierung und protokollierung verteilter paarprogrammierung, August 2006.

[Eng] AG Software Engineering. Thesis rules. https://www.inf.fu-berlin.de/w/SE/ThesisRules.

[See06] Tanja Seelheim. Teamfähigkeit und performance: Auswirkungen der gruppenzusammensetzung nach persönlichkeitsmerkmalen auf die leistung einer gruppe, Juli 2006.

[Só09] Tas Sóti. Einladungsprozess in saros, Oktober 2009.