**Studienarbeit**

# Agile Weiterentwicklung eines Software-Werkzeuges zur verteilten, kollaborativen Programmierung in Echtzeit

Marc ′BlackJack′ Rintsch

Matrikelnummer: 3429607

marc@rintsch.de

Betreuer:
Christopher Özbek
Stephan Salinger

19. Juni 2009

# Inhaltsverzeichnis

# 1 Einleitung

Der Inhalt der vorliegenden Arbeit ist die agile Weiterentwicklung des Software-Werkzeuges *Saros*[1], das die *verteilte Paarprogrammierung* bzw. auch *Side-by-Side-Programmierung* ermöglicht. Der Fokus liegt dabei auf *Präsenzinformationen* und der *Serialisierung* von Ereignissen.

In Abschnitt 1.2 wird ein kurzer Überblick über die Software und ihre bisherige Entwicklung gegegen. Danach folgt in Abschnitt 1.3 auf der nächsten Seite eine Beschreibung der drei Arbeitspakete. Wie diese Umgesetzt wurden, wird in Kapitel 2 behandelt. Abschliessend gibt es in Kapitel 3 ein Resümee und ein Ausblick auf mögliche künftige Aufgaben.

## 1.1 Konventionen / Notationen

Klassennamen werden in diesem Dokument kursiv gesetzt. In der Regel werden sie ohne die Paketstruktur angegeben, da die Namen innerhalb des *Saros*-Projekts normalerweise eindeutig sind. Das gleiche gilt für prominente Klassen aus der Java-Standardbibliothek. Beispiele wären *EditorManager* oder *NullPointerException*.

Dateinamen werden ebenfalls kursiv gesetzt. Wenn sich der Text auf den Inhalt einer Datei zu einem bestimmten Zeitpunkt bezieht, wird zusätzlich die Revisionsnummer des Subversion-Repositories mit einem kleinen "*r*" davor, und insgesamt in Klammenrn gesetzt, angegeben.

## 1.2 Was ist Saros?

Saros ist ein *Eclipse*-Plugin, das mehreren Teilnehmern das verteilte, kollaborative Bearbeiten eines gemeinsamen Projekts ermöglicht. In der Vergangenheit gab es schon vier Diplom- und Studienarbeiten, *DPP I* bis *DPP IV*[2], die sich mit der Entwicklung des Projekts beschäftigten.

**[Dje06]** Die erste Version wurde im Rahmen einer Diplomarbeit von *Riad Djemili* geschrieben. Sie ermöglichte das gemeinsame Arbeiten von zwei Personen an einem

[1] http://dpp.sourceforge.net/

[2] *DPP* steht hier für *Distributed Pair Programming*, also *Verteilte Paarprogrammierung*.

Projekt als klassische Paarprogrammierung. Es gab zwei Teilnehmer, bei denen einer die Rolle des Schreibers (*Driver*) und der Andere die des Beobachters (*Observer*) übernahm. Die Rollen konnten während einer Sitzung getauscht werden.

**[Gus07]** In einer Studienarbeit hat *Björn Gustavs* das Plugin so erweitert, dass es mehr als zwei Teilnehmer geben konnte, wobei weiterhin nur einen Schreiber zur gleichen Zeit möglich war. Diese exklusive Schreiberrolle konnte aber jedem der Beobachter übertragen werden.

**[Rie08]** *Oliver Rieger* hat in seiner Diplomarbeit dann die Mehr-Schreiber-Funktionalität hinzugefügt. Der *Jupiter*-Algorithmus sorgt dabei für die notwendige Nebenläufigkeitskontrolle. Ferner wurde eine effizientere Dateiübertragung über das *Peer-to-Peer*-Protokoll *Jingle* implementiert.

**[Jac09]** Die Diplomarbeit von *Christoph Jacob* beschäftigte sich unter anderem mit der allgemeinen Verbesserung der Codequalität des Projekts und einer Konsistenzsicherung.

Die letzte Arbeit überlappte zeitlich mit dieser, war aber abgeschlossen, bevor die Arbeit an diesem Dokument begann. Deshalb wird sie hier zur “Vergangenheit” gezählt. Parallel zu dieser Studienarbeit laufen bereits vier weitere Arbeiten, die sich auf das *Saros*-Projekt beziehen, von *Lisa Dohrmann*, *Edna Rosen*, *Sandor Szücs*, und *Sebastian Ziller*.

## 1.3 Arbeitspakete

Die drei Arbeitspakete wurden zusammen mit den beiden Betreuern festgelegt. Nur das erste Paket wurde direkt nach der Einarbeitungsphase bestimmt. Die beiden anderen ergaben sich im Laufe der Arbeit aus der Notwendigkeit heraus, Probleme zu beseitigen, die sich auf die Umsetzung vom ersten Arbeitspaket bezogen. Daher das Adjektiv “agil” im Titel dieser Arbeit.

### 1.3.1 AP1 Präsenzinformationen

Unter Präsenzinformation fallen die grafischen Hinweise in der Benutzeroberfläche, die anzeigen in welcher Datei und an welcher Stelle innerhalb der Datei, sich ein anderer Benutzer befindet. Ebenso die Anzeigen und Benachrichtigungn über Zustände und Zustandsänderungen der Benutzer, wie zum Beispiel die Rolle, oder ob jemand die Sitzung betritt oder verlässt. Im weiteren Sinne kann man noch den Verfolgermodus diesem Arbeitspaket zuordnen. Des weiteren gehört die Erfassung und Übertragung der nötigen Informationen dazu.

Die Umsetzung dieses Arbeitspakets ist in Abschnitt 2.2 auf Seite 10 beschrieben.

### 1.3.2 AP2 Sequenzialisierung von Nachrichten

In diesem Arbeitspaket geht es darum, dass die Nachrichten bei den Empfängern auch in der Reihenfolge ausgeführt werden, wie sie beim Sender erzeugt wurden. Insbesondere bei asnychron versandten Nachrichten.

Es stellte sich im Laufe der Bearbeitung von Arbeitspaket 1 heraus, dass die Implementierung der Sequenzialisierung im *ActivitySequencer* nicht mehr funktionierte, weil dort immer noch von nur jeweils *einem* Schreiber und Beobachter ausgegangen wurde.

In Abschnitt 2.3 auf Seite 19 befindet sich eine Beschreibung der Änderungen zu diesem Arbeitspaket.

### 1.3.3 AP3 Serialisierung von Nachrichten

Unter *Serialisieren* versteht man die Umwandlung des Zustands eines Objekts in einen Bytestrom, aus dem durch *Deserialisieren* eine Kopie des Objekts erzeugt werden kann [RSL99, Seite 3-4]:

> To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object. [...] "Deserialization" is the process of converting the serialized form of an object back into a copy of the object. When an object is serialized, the entire tree of objects rooted at the object is also serialized. When it is deserialized, the tree is reconstructed.

Dabei werden nur die Daten und Typinformationen des Objekts, und aller Objekte die davon referenziert werden, in einen Bytestrom umgewandelt, aber nicht der Code oder Bytecode der Objekte. Die Klassen für die Daten der serialisierten Objekte müssen also beim Deserialisieren verfügbar sein.

*Saros* verwendet nicht den von Java standardmässig zur Verfügung gestellten Mechanismus über das "tagging"-Interface *java.io.Serializable*, sondern ein von der Programmiersprache unabhängiges XML-Format. Der Grund dafür ist das zur Kommunikation verwendete XMP-Protokoll, das Nachrichten selber als XML durch das Netz schickt. Es ist einfacher und effizienter XML, anstelle von Binärdaten in XML einzubetten. Ferner erleichtert das Format auch die Fehlersuche, weil man sich die übertragenen Nachrichten im Klartext anschauen kann. *Saros* bietet dafür eine Option, die das in Abbildung 1.1 auf der nächsten Seite gezeigte Fenster mit den versendeten und empfangenen XMPP-Nachrichten aktiviert.

Die Serialisierung der Nachrichten entspricht nicht ganz der Beschreibung aus [RSL99], da XML aus Zeichen und nicht aus Bytes besteht. Die Umwandlung zwischen Zeichen und Bytes übernimmt die verwendete XMPP-Bibliothek *Smack*[3] vor dem Verschicken

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

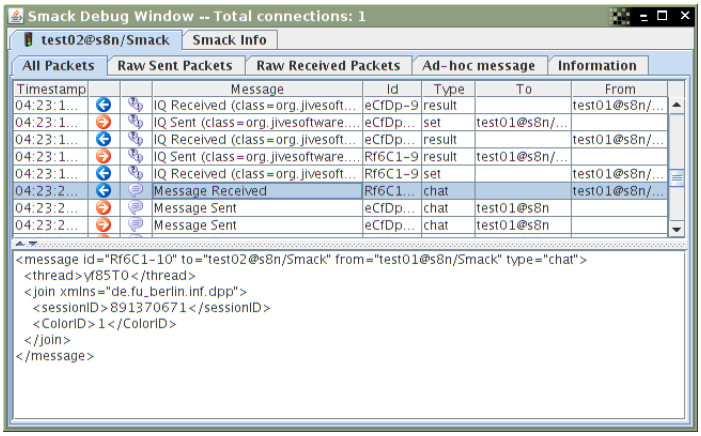


**Abbildung 1.1:** *Smack*-Debug-Fenster

und nach dem Empfangen der Nachrichten.

Das Arbeitspaket bestand darin, die Erzeugung des XML robuster zu gestalten, da das XML vorher recht naiv und fehleranfällig mittels Zeichenketten manuell zusammen gebaut wurde. Hauptsächlich durch Konkatenation von Zeichenketten und mit der *StringBuilder*-Klasse aus der Standardbibliothek.

# 2 Umsetzung

## 2.1 Einarbeitung

Vor der Verständigung mit den beiden Betreuern über die Arbeitspakete, gab es eine Einarbeitungsphase, bei der ich, unterstützt durch Christoph Jacob, die Quelltextbasis kennenlernen sollte, und kleine Veränderungen vornahm.

Ein erstes, grobes Bild der Architektur und des Datenflusses der Informationen, die von einem Teilnehmer zum anderen übertragen werden, entstand. Aktionen im Editor werden dabei

- vom *EditorManager* verarbeitet, der Schnittstelle zu Eclipse,
- an die Nebenläufigkeitskontrolle – den *Jupiter*-Algorithmus – übergeben,
- und weiter über den *ActivitySequencer*, der mittels Zeitstempeln die Sende- und Empfangsreihenfolge kontrolliert,
- an den *XMPPChatTransmitter* geleitet, der die Daten verschickt.

Beim Empfänger der Daten werden diese vier Stationen in umgekehrter Reihenfolge durchlaufen.

Als Nachschlagewerk für *Eclipse*-Plugins hat sich [Dau08] bewährt. Das Buch behandelt zwar noch *Eclipse* 3.3, aber auf der Webseite zum Buch[1] ist ein PDF-Dokument mit Aktualisierungen zu *Eclipse* 3.4 zu finden.

Die ersten Änderungen am Programm waren sehr einfacher Natur. Zum Beispiel wurde beim Wegnehmen der Schreiber-Rolle über den entsprechenden Eintrag im Kontextmenü des Benutzers im Session-View, dieser Eintrag nicht ausgegraut, sondern stand weiterhin zum Anklicken zur Verfügung. Ein erneutes Anwählen war allerdings nicht wirkungslos, da der Rollenwechsel zu dem Zeitpunkt als Umschaltoperation zwischen den beiden Rollen Schreiber und Beobachter implementiert war, und nicht als "Setter". So war "zweimal Schreiber-Rolle wegnehmen" im Endeffekt einmal zur Beobachter-Rolle und wieder zurück zur Schreiber-Rolle schalten. Die Klassen, welche die GUI-Aktionen implementieren, haben alle eine Methode, die überprüft, ob die jeweilige Aktion im aktuellen Programmzustand möglich ist, und dementsprechend den Status auf verfügbar oder ausgegraut setzt. Die Lösung war, einfach diese Methode am Ende der *run()*-Methode der Aktion aufzurufen.

[1] http://www.bdaum.de/bookeclipse33.HTM

Nach der "Solo-Aktion" gab es eine Paarprogrammierungs-Sitzung mit Christoph Jacob. Zusammen haben wir das Problem gelöst, dass bei jedem Teilnehmer die Zuordnung von Farben zu Benutzern lokal getätigt wurde – die Farbvergabe also nicht global eindeutig war. Mit der Ausnahme, dass die erste Farbe bei allen Teilnehmern für den Host reserviert wurde. Das funktioniert nur bei maximal zwei Teilnehmern korrekt. In der Benutzerliste, die nach einer angenommenen Einladung an einen neuen Teilnehmer versandt wird, war die Farb-ID für alle bereits in der Sitzung aktiven Teilnehmer, die der Host bei sich lokal vergeben hatte, interessanterweise schon enthalten. Beim Empfänger wurde diese Information aber ignoriert. Wir haben das Programm dann so verändert, dass die Farbzuordnung nur noch beim Host passiert, und von dort diese Informationen bei der Einladung und mit erwähnter Benutzerliste an die Teilnehmer übermittelt wird. Somit sind die Farb-IDs überall gleich zugeordnet und man konnte sinnvoll über diese Farben mit anderen Teilnehmern kommunizieren.

Die nächste Änderung betraf die Anzeige der Teilnehmer im Session-View. Der Name des Benutzers wurde dort fett und ohne besondere Hintergrundfarbe angezeigt, während die Namen der entfernten Benutzer mit ihrer jeweiligen Farbe hinterlegt waren. Der Benutzer konnte nicht sagen, welche Farbe ihm selbst vom Programm zugeteilt war, und die Farbe deshalb nicht den anderen Teilnehmer einer Sitzung weiterkommunizieren. Zum Beispiel in der Art "Ich habe jetzt den relevanten Teil des Quelltextes markiert, der müsste bei Ihnen nun grün hinterlegt sein". Auch hier war wieder eine leichte Umsetzung der Lösung möglich, da man sich an dem bereits vorhandenem Quelltext für die Hintergrundfarbe orientieren konnte, um beim Namen des lokalen Benutzers, die Schriftfarbe auf "seine" Farbe zu setzen. So kann der Benutzer immer noch sich selbst leicht identifizieren, da die Zeile ohne Hintergrundfarbe deutlich auffällt, und trotzdem erkennen, welche Farbe ihm zugeteilt wurde.

## 2.2 Präsenzinformationen (AP1)

### 2.2.1 Zuteilung der Benutzerfarben

Die Farbe eines Benutzers wird über eine Farb-ID festgelegt, die im *User*-Objekt gespeichert ist, welches einen Benutzer repräsentiert. Der Code der Benutzeroberfläche holt sich die konkreten Farben für einen Benutzer anhand dieser ID aus den Einstellungen von Eclipse. Die Farben werden über die *plugin.xml* voreingestellt. Die ID, eine Nummer zwischen 0 und 4, denn es gibt fünf Farben in den Einstellungen, ist Teil der Schlüssel für diese Einstellungen. Bei den Schlüsseln geht die Nummerierung allerdings von 1 bis 5.

In Listing 2.1 auf der nächsten Seite sieht man am Beispiel der Annotation für die Auswahl von den ersten beiden Benutzern, wie die Daten in der *plugin.xml* hinterlegt sind. Unter dem *"extension point" org.eclipse.ui.editors.annotationTypes* werden die Markierungstypen bekannt gemacht, und unter *org.eclipse.ui.editors.markerAnnotation-*

**Listing 2.1:** Definition der Benutzerfarben in der *plugin.xml* (r431)

```
<plugin>
   ...
   <extension
         point="org.eclipse.ui.editors.annotationTypes">
      ...
      <type
            markerType="de.fu_berlin.inf.dpp.annotations.selection.1"
            name="de.fu_berlin.inf.dpp.annotations.selection.1"/>
      <type
            markerType="de.fu_berlin.inf.dpp.annotations.selection.2"
            name="de.fu_berlin.inf.dpp.annotations.selection.2"/>
      ...
   </extension>
   ...
   <extension
         point="org.eclipse.ui.editors.markerAnnotationSpecification">
      ...
      <specification
            annotationType="de.fu_berlin.inf.dpp.annotations.selection.1"
            colorPreferenceValue="255,0,0"
            ...
            verticalRulerPreferenceValue="false"/>
      <specification
            annotationType="de.fu_berlin.inf.dpp.annotations.selection.2"
            colorPreferenceValue="0,255,0"
            ...
            verticalRulerPreferenceValue="false"/>
   </extension>
   ...
</plugin>
```

*Specification* sind die Einstellungen für die Darstellung, die auch in den Einstellungen von Eclipse auftauchen, hinterlegt. Neben der Farbe sind dort auch noch andere Einstellungen für die Anzeige zu finden, wie zum Beispiel die Art der Darstellung der Markierung und der Tooltip-Text.

Die Zuteilung einer freien Farbe und die Freigabe dieser Farbe, wenn der Benutzer die Sitzung verlässt, passiert im *SharedProject*-Objekt und ist in Listing 2.2 auf der nächsten Seite zu sehen. Mit diesem Quelltext war ich unzufrieden. Die *int*s in *colorlist* wurden als Wahrheitswerte missbraucht, im gesamten Quelltext war keine Stelle zu finden, bei der die Farb-ID des *user*-Objekts auf den Wert -1 hätte gesetzt werden können, und der Rückgabewert von *assignColorId()* wurde nirgends verwendet. Ausserdem ist das Array ein Element grösser als es sein müsste.

Dieser Quelltext wurde durch die Klasse *FreeColors* ersetzt, welche die freien Farb-IDs verwaltet. Der Konstruktor nimmt die maximale Anzahl von verschiedenen IDs entgegen und bietet Methoden zum "ziehen" einer unbenutzten ID und zum zurückgeben

**Listing 2.2:** Verwaltung der Benutzerfarben in *SharedProject.java* (r431)

```
// ...
    private static final int MAX_USERCOLORS = 5;
    private final int colorlist[] = new int[SharedProject.MAX_USERCOLORS + 1];
// ...
    boolean assignColorId(User user) {

        // already has a color assigned
        if (user.getColorID() == -1) {
            return true;
        }

        for (int i = 0; i < SharedProject.MAX_USERCOLORS; i++) {
            if (this.colorlist[i] == 0) {
                user.setColorID(i);
                this.colorlist[user.getColorID()] = 1;
                return true;
            }
        }

        return false;
    }

    public void removeUser(User user) {
        this.participants.remove(user);

        // free colorid
        this.colorlist[user.getColorID()]
        // ...
```

einer ID wenn diese nicht mehr benötigt wird. Das *SharedProject* bietet die gleichen Operationen, die an ein Exemplar der neuen Klasse delegiert werden.

Intern werden die IDs in einer Warteschlange gehalten. Das hat gegenüber dem alten Vorgehen mit dem Array den Vorteil, dass eine freigewordene Farbe nicht sofort wieder vergeben wird – zumindest solange noch genügend freie IDs vorhanden sind. Bei der alten Strategie konnten die Teilnehmer jemanden der die Sitzung verlassen hat, farblich nicht von jemandem unterscheiden, der direkt danach dazugestossen ist, was zu Irritationen führen kann, wenn man den Personenwechsel nicht mitbekam.

Bei der alten Farbvergabestrategie wurde die Benutzerfarbe gar nicht gesetzt, falls alle IDs bereits vergeben waren, was dazu führte, das ab dem sechsten Teilnehmer einer Sitzung alle die erste Farbe bekamen, da die Farb-ID im *User*-Konstruktor mit 0 initialisiert wurde. Weil der erste Benutzer in einer Sitzung immer der Host ist, der technisch und semantisch eine besonderere Bedeutung hat, als die anderen Teilnehmer, liefert die neue Strategie immer die höchste Farb-ID, falls die anderen alle aufgebraucht sind. So bleibt die Farbe mit der ID 0 immer dem Host vorbehalten.

Die *FreeColors*-Klasse sieht jetzt ein wenig überdimensiert aus für das kleine Problem, so dass man durchaus argumentieren könnte, man sollte sie wieder im *SharedProject* aufgehen lassen. Zu dem Zeitpunkt als sie entstand, war mir noch nicht klar, dass es nicht einfach möglich ist, die Farben für die Annotationen auf eine andere Art als über die *Eclipse*-Einstellungen zu setzen. Ich ging davon aus, dass die *FreeColors*-Klasse später einmal noch mehr Funktionalität zum Verwalten von Farben erhalten würde, statt nur einfache ganze Zahlen zwischen 0 und 5 zu speichern. Andererseits könnte man auch Methoden zum Abbilden einer Farb-ID auf ein Farb-Objekt in diese Klasse verlegen, um ihr mehr Daseinsberechtigung zu verleihen.[2]

Unschön an der Verwaltung der Farben ist die Indirektion über IDs auf die Werte in den Einstellungen. Nur die IDs werden zu anderen Teilnehmern übertragen, nicht die Farben selbst. Damit sind die Farben in einer Sitzung nur global eindeutig, solange die Teilnehmer sie nicht in den Einstellungen abweichend voneinander verändern.

Die Beschränkung auf maximal fünf verschiedene Farben ist dagegen nicht so tragisch, da auch bei diesen Fünf schon das Problem besteht, Farben zu finden, die sich deutlich genug, sowohl voneinander, als auch von den anderen Farben für Annotation und Syntax-Hervorhebungen in Eclipse unterscheiden.

Damit die Annotationen nicht so sehr im Vordergrund stehen, und damit vom Quelltext ablenken, sind die Farben mittlerweile heller als am Anfang dieser Arbeit; ähneln im Farbton allerdings teilweise immer noch anderen wichtigen, farbigen Markierungen und Hervorhebungen in *Eclipse*, beziehungsweise dem Java-Editor. Beim Aufhellen muss man überdies berücksichtigen, dass bei LCDs die Farbwahrnehmung oft sehr stark vom Blickwinkel auf das Display abhängt, und gerade Pastelltöne schnell "unsichtbar" werden. Bei einem Laptop einer Kommilitonin konnte sogar der umgekehrte Effekt beobachtet werden – von schräg unten liess sich eine Selektions-Annotation deutlich erkennen, während sie bei fast senkrechtem Blick auf das Display Schwierigkeiten hatte die farbige Hinterlegung zu erkennen.

### 2.2.2 *ViewPort*-Annotationen

Nachdem nun die Farben für alle Benutzer global eindeutig waren, oder zumindest die Farb-IDs, wie in Abschnitt 2.2.1 auf Seite 10 beschrieben, folgte die Überarbeitung der Annotation für die Position und die Ausmasse des Darstellungsfeldes von den Teilnehmern. Bis zu dem Zeitpunkt wurden die Darstellungsfelder der Schreiber alle in der gleichen Farbe dargestellt.

Dazu mussten erst einmal die Typen und Einstellungen für die neuen, verschiedenfarbigen Markierungen in der *plugin.xml* hinterlegt werden. Und zwar unter dem "Grundnamen" *de.fu_berlin.inf.dpp.annotations.viewport.* und einer Ziffer von 1 bis 5 für die ver-

[2] Man könnte die Klasse auch in *ColorManager* umbenennen, um der Java-Konvention gerecht zu werden, überall das schwammige Wort "Manager" dran zu pappen. ;-)

schiedenen Farb-IDs. Die Einträge in der XML-Datei erfolgten analog zu denen für die Selektions-Annotation, die man in Listing 2.1 auf Seite 11 sehen kann.

Danach war nur noch eine kleine Anpassung der Klasse *ViewportAnnotation* nötig, um statt der einen Farbe aus den Einstellungen, die Farbe des Benutzers, zu dem die Annotation gehört, zu verwenden. Der entsprechende Quelltext in der *SelectionAnnotation*-Klasse diente dabei als Vorlage.

Zu Beginn einer Sitzung wurden oft die Viewport-Annotation nicht angezeigt, darum werden nun beim Einladungsvorgang nicht mehr nur die Information, welche Editoren beim Host gerade offen sind übertragen, sondern auch die Position und Ausmasse des jeweiligen Darstellungsfeldes.

Das in allen Editoren die Viewport-Annotationen der anderen Teilnehmer zu sehen sind, und nicht nur in dem Editor, den der jeweilige Teilnehmer bei sich aktiviert hat, wurde in einem Eintrag im Bugtracker kritisiert.[3] Das sah ich zunächts nicht als Problem, da man den "aktiven" Editor ja zusätzlich noch an der Annotation für die Schreibmarke erkennen kann. Dazu habe ich deshalb die Meinung der anderen "Sarosianer" eingeholt. Für Edna Rosen war der Punkt nicht so wichtig, da in dem von ihr beobachteten Einsatzszenario nur zwei Teilnehmer vorkommen. Christopher Oezbek empfand es als störend und merkte an, dass sich die Schreibmarke nicht innerhalb des Darstellungsfeldes befinden muss. Damit ist dieser visuelle Hinweis natürlich leichter zu übersehen als ich annahm. Letztendlich überzeugt hat mich dann eine *Side-By-Side*-Programmiersitzung mit *Saros* zu dritt, mit Lisa Dohrmann und Christopher Oezbek, bei der eine als *deprecated* gekennzeichnete, und über das Projekt verteilt zahlreich aufgerufene Methode (*Saros.getDefault()*), durch Alternativen ersetzt werden sollte. Der Vorteil der Verwendung von *Saros* für diese Aufgabe gegenüber des tradionellen Vorgehens ist, das weniger Kommunikation ausserhalb der IDE stattfinden muss. Ohne diese Unterstützung hätte man, um Konflikte zu vermeiden, entweder die Aufrufe der Methode vorher unter den Beteiligten aufteilen müssen, oder aber mit Ansage vor jeder Aufrufstelle, die ein einzelner Sitzungsteilnehmer verändert, arbeiten müssen. Mit dem Plugin konnte man im Editor sofort sehen, ob sich schon jemand anders um einen Aufruf kümmert. Allerdings musste man nach der Annotation der Schreibmarke Ausschau halten, und konnte sich nicht an der deutlicheren Viewport-Annotation orientieren. In Zukunft sollte also nur noch für den auf der Gegenseite aktiven Editor die Viewport-Annotation angezeigt werden, oder zumindest sollte das Verhalten über die Einstellungen beeinflussbar sein.

### 2.2.3 *Contribution*-Annotationen

Die Contribution-Annotation, also die farbige Hinterlegung von Text, den ein Schreiber beigetragen hat, war schwieriger umzusetzen als Annotationen des Darstellungsfeldes.

[3] *#2707089 – Multiple viewports of driver cannot be distinguished.*

Da es früher nur einen, exklusiven Schreiber gab, war hierfür auch nur eine Farbe vorgesehen. Verschiedene Farben für die einzelnen Teilnehmer bekannt zu machen funktioniert analog zu den anderen Annotationen über die *plugin.xml*.

Die Farb-ID ist ein Merkmal des einzelnen Benutzers, und Benutzer werden über ihre Jabber-ID als Quellen in den Aktivitäten angegeben, die als Nachrichten zwischen den Teilnehmern ausgetauscht werden. Es wäre also naheliegend, die Quelle von *TextEditActivity*s zu verwenden um den Text aus diesen Aktivitäten mit der passenden Farbe zu hinterlegen. Allerdings gehen bei Schreibern diese Aktivitäten an den Host, der sie durch den *Jupiter*-Algorithmus schickt und dann erst wieder an die einzelnen Teilnehmer. Womit alle Textveränderungen, die durch *Jupiter* verarbeitet wurden, den Host als Quelle in den Nachrichten eingetragen haben, wenn sie bei den Teilnehmern ankommen.

Zu dem Zeitpunkt war der Mehr-Schreiber-Modus nicht die Voreinstellung und die erste Näherungslösung für die Aufgabe war ein "Hack". Nämlich dass die Quelle in der Nachricht ignoriert wurde und einfach irgend ein Benutzer mit der Schreiber-Rolle zur Farbgebung herangezogen wurde. Solange es nur einen exklusiven Schreiber gibt, ist das natürlich die richtige Farbe.

Für eine saubere Lösung hätte man die Jabber-ID der originalen Quelle der Textveränderung beim Host durch die Transformationen im *Jupiter*-Algorithmus schleusen müssen, was ich mir zu dem Zeitpunkt nicht zutraute, da dort einfach zu viel unbekannt war.

### 2.2.4 *Contribution*-Verlauf

Das die Contribution-Annotationen von Schreibern solange sie diese Rolle inne haben, bleibend angezeigt werden, führt in der Praxis dazu, dass immer mehr Text farbig hinterlegt wird. Damit erfüllen sie nicht den angestrebten Zweck von Präsenzinformationen – dass man sieht wo sich der andere zur Zeit im Quelltext aufhält, und was er aktuell verändert. Deshalb sollten diese Annotation nicht dauerhaft sein, sondern ältere Markierungen wieder verschwinden.

Im Quelltext schon vorhanden war eine Klasse *ContributionHelper* mit Hilfsfunktionen zum Einfügen von Contribution-Annotationen in den Editor, die zu einem *ContributionAnnotationManager* erweitert wurde. Mit einer *Map*, die Benutzer auf Warteschlangen mit *ContributionAnnotation*-Exemplaren abbildet, um für jeden Benutzer eine geordnete Menge von Annotationen zu verwalten, bei der man effizient an einem Ende Elemente hinzufügen, und am anderen entfernen kann. Die alten Methoden zum Einfügen von Annotationen zeichnen jetzt zusätzlich den Verlauf für jeden Benutzer auf, und entfernen ältere Annotationen. Das Kriterium hierfür ist die Länge des Verlaufs. Momentan werden maximal 20 Annotationen angezeigt. Ausserdem wird ein "Listener" beim *SharedProject* registriert, damit der Verlauf für Benutzer, welche die Sitzung verlassen, gelöscht werden kann.

Zu einem Eintrag im Verlauf führt jede *TextEditActivity*, die Text hinzufügt. Unglücklicherweise wird irgendwo auf dem Weg vom originalen Schreiber, zum Host, durch den *Jupiter*-Algorithmus, und zu den Empfängern, der Text in einzelne Buchstaben zerhackt, ausser bei *TextEditActivity*s die auch wirklich beim Host erzeugt wurden. Damit sind ein Verlauf von 20 farbig hinterlegte Änderungen vom Host in der Regel deutlich umfangreicher als die selbe Anzahl von Änderungen von anderen Teilnehmern. Das Problem ist noch offen.

Sanftes Ausblenden der älteren Contribution-Annotationen, anstelle des abrupten Verschwindens, wäre ein schöner Effekt, aber die von *Eclipse* erzwungene Indirektion über die Einträge in den *Eclipse*-Einstellungen hätte für jeden aufgehellten Farbton für jede Farb-ID ein eigenes Schlüssel/Wert-Paar erforderlich gemacht. Diese "Explosion" der Zahl der Konfigurationseinträge wäre schlecht manuell zu warten, und würde es dem Benutzer auch nahezu unmöglich machen, die Farben in den Einstellungen zu ändern, da er jeder Helligkeitsstufe einzeln eine Farbe zuordnen müsste.

### 2.2.5 Verfolgung des Darstellungsfeldes (*Viewport*)

Das Darstellungsfeld bei einem Beobachter im Verfolgermodus sollte den Aktivitäten des verfoltgen Schreibers möglichst "intelligent" folgen. Relevante Informationen, die dazu übertragen werden, sind zum einen Anfang und Länge des beim Schreiber angezeigten Darstellungsfeldes und die Position der Schreibmarke oder der Auswahl. Dies geschieht mit den Aktivitäten *ViewportActivity* und *TextSelectionActivity*.

Nur starr dem Darstellungsfeld des Schreibers zu folgen, wie das ganz am Anfang einmal gemacht wurde, ist nicht optimal. Denn wenn das Darstellungsfeld des Verfolgers weniger Zeilen umfasst, als das beim Beobachteten, kann die Schreibmarke des Verfolgten das Darstellungsfeld beim Verfolger verlassen und es lässt sich nicht mehr sehen, was gerade geschrieben oder durch Auswahl hervorgehoben wird.

Ein erster Ansatz mit dieser verschwindenden Schreibmarke umzugehen war es, auch die Position der Schreibmarke, beziehungsweise die von Selektionen des Schreibers zu verwenden, um die Position des Darstellungsfeldes beim Verfolger zu verändern. Die erste Implementierung hat dazu geprüft, ob sich die Schreibmarke des Verfolgten innerhalb des Darstellungsfeldes des Verfolgers befindet, und falls nicht, dessen Darstellungsfeld so positioniert, dass sich die verfolgte Schreibmarke in der mittleren Zeile befindet. Das hat die Situation verbessert, ist aber auch noch nicht optimal, da so bei kontinuierlicher, vertikaler Bewegung der Schreibmarke in eine Richtung, beim Verfolger mit einem kleineren Darstellungsfeld selbiges immer um die halbe Höhe in Zeilen "weiterspringt", wenn die verfolgte Schreibmarke den oberen oder unteren Rand überschreitet.

Beim Berücksichtigen der Position der Schreibmarke muss man beachten, dass es für den Verfolgten weiterhin möglich ist, sein Darstellungsfeld mittels Rollbalken verändern, um zum Beispiel an anderer Stelle im Quelltext etwas zu zeigen, und damit auch

dafür zu sorgen, dass sich die Schreibmarke nicht mehr in seinem Darstellungsfeld, und auch nicht in denen der Verfolger, befindet. In diesem Fall darf man beim Verfolger natürlich das Darstellungsfeld nicht um die Schreibmarke des Verfolgten zentrieren.

In der Horizontalen folgt das Darstellungsfeld der Schreibmarke des Verfolgten übrigens nicht. Die API von *Eclipse* bietet dafür keine Ereignisse wie bei der vertikalen Änderung des Darstellungsfeldes an, und man kann auch nur den Versatz des linken Randes zum Zeilenanfang in Zeichen abfragen, nicht aber wie breit das Darstellungsfeld ist. Zumindest bei Quelltexten in Java, und auch den anderen üblichen Programmiersprachen, schätze ich den Handlungsbedarf hier nicht so hoch ein, da oft eine Stil-Richtlinie existiert, die maximal 80 Zeichen pro Zeile empfiehlt. Das wird auch in den meisten Fällen von der automatischen Formatierung des Quelltextes im Editor forciert.

### 2.2.6 Übertragung von Rollenaktivitäten

Die Darstellung der Rollenverteilung bei den einzelnen Teilnehmern war sehr instabil. Die Meinung wer Schreiber und wer Beobachter ist, entwickelte sich oft bei den jeweiligen Teilnehmern auseinander. Der vermutete Grund dafür war die zustandsbehaftete Übermittlung von Rollenwechseln. Die *RoleActivity* übertrug als einziges Datum die Jabber-ID des betroffenen Benutzers und die API des *SharedProject* kannte nur eine *toggleUserRole()*-Methode, welche für den angegebenen Benutzer die Rolle zwischen Schreiber und Beobachter aufgrund des bestehenden Zustands umschaltete.

Nachdem die tatsächliche Rolle in der *RoleActivity* übertragen, und die *toggleUserRole()* durch eine *setUserRole()* ersetzt wurde, war die gemeinsame Sicht auf die Rollenverteilung innerhalb der Sitzung stabil. Neben der *RoleActivity*-Klasse und der bereits genannten Methode, mussten dazu noch die GUI-Klassen, welche die Aktionen des Kontextmenüs im Session-View implementieren und Auswirkungen auf den Rollenstatus von Teilnehmern haben, angepasst werden, und die Rolleninformation musste für die Nachrichtenübertragung in XML (de)serialisiert werden.

### 2.2.7 Darstellung der Schreibmarken

Die Information, in welcher Zeile sich andere Benutzer befinden, wird als Annotation am rechten Rand angezeigt. Nun sollte die Position der Schreibmarken der Teilnehmer auch innerhalb der Zeile dargestellt werden. Die Position wird als Selektions-Annotation übertragen. Selektionen der Länge Null werden dabei als Position der Schreibmarke interpretiert.

Genau wie bei den übertragenen Nachrichten wird für die Anzeige der Schreibmarke die vorhandene Infrastruktur für Selektions-Annotationen verwendet. Falls die Länge

der Selektion Null beträgt, wird sie beim Empfänger einfach auf Eins erhöht und die Beschriftung der Annotation wird von *"Selection of . . . "* in *"Cursor of . . . "* geändert. Dabei gibt es allerdings noch Randfälle, die man berücksichtigen muss. Falls die Schreibmarke sich am Ende einer Zeile befindet, also hinter dem letzten Zeichen, kann dort kein Zeichen mit der Annotation hervorgehoben werden. In diesem Fall wird die Position um ein Zeichen nach links versetzt. Allerdings auch nur, wenn da überhaupt ein Zeichen existiert. Bei Leerzeilen ist also nichts vorhanden, was man mit einer Annotation im Textfeld des Editors versehen könnte, und daher wird die Schreibmarke in Leerzeilen, die wirklich gar nichts enhalten, überhaupt nicht dargestellt. Das ist etwas unschön. Ebenfalls ein kleiner Schönheitsfehler ist die Darstellung wenn sich die Schreibmarke auf einem Tabulatorzeichen befindet. Auf diesem einen Zeichen wird die Annotation für die Schreibmarke nämlich über die gesamte Breite des "Tabs" angezeigt.

Längere Zeit gab es Probleme mit der Darstellung von Schreibmarken weil die *TextEditActivity* wegen dem *Jupiter*-Algorithmus mit einem Umweg über den Host geschickt wurde, die Position der Schreibmarke aber mit einer *SelectionActivity* direkt zu den Sitzungsteilnehmern. So kam die Position, die von einer bereits erfolgten Textänderung ausgeht, vor der Textänderung selbst an, was am Zeilenende zu einem ständigen "Springen" der Schreibmarken-Annotation zwischen zwei benachbarten Zeilen führte, während ein Schreiber etwas am Ende einer Zeile tippte. Das Problem wurde durch Sebastian Ziller's Umwandlung von *Jupiter-Requests* in Aktivitäten behoben. Der *ActivitySequencer* wartet beim Empfänger nun auf das Eintreffen der *TextEditActivity* bevor die *SelectionActivity* ausgewertet wird.

### 2.2.8 Benachrichtigungen über Sitzungsereignisse


**Abbildung 2.1:** Eine "Sprechblase" mit einer Information über die Sitzung.

Benutzer wurden bereits mit einer "Sprechblase" wie sie in Abbildung 2.1 zu sehen ist, darüber informiert, wenn sich die Rolle eines Benutzers änderte. Das Betreten und Verlassen einer Sitzung durch andere Benutzer, war dagegen leicht zu übersehen. Nach dem beseitigen von redundantem Code – das *BalloonWindow* wurde für den Rollenwechsel an zwei verschiedenen Stellen im Quelltext mit fast identischem Quelltext aufgerufen – werden die Teilnehmer nun auch über diese beiden Ereignisse informiert. Zusätzlich zur Jabber-ID wird jetzt ausserdem der Nickname in der Sprechblase angezeigt.

Zuwenig Information war nicht gut, aber jetzt ist es teilweise etwas zuviel Information, zum Beispiel wenn der Host die Schreiber-Rolle von einem Teilnehmer auf einen anderen überträgt. Dann werden zeitgleich zwei Sprechblasen übereinander dargestellt. Eine mit der Information, dass der alte Schreiber nun in den Beobachtermodus gewechselt ist, und eine Weitere die mitteilt, dass der neue Schreiber ab sofort diese Rolle innehat. Ein möglicher Lösungsansatz wäre es, bei jeder Meldung, die angezeigt werden

soll, erst zu schauen, ob nicht schon eine Sprechblase angezeigt wird, und dort die Information hinzuzufügen und die Anzeigedauer entsprechend zu verlängern.

## 2.3 *ActivitySequencer* (AP2)

Der *ActivitySequencer* soll dafür sorgen, dass die aus dem Netz ankommenden Nachrichten über Aktivitäten in der Reihenfolge ausgeführt werden, in der sie bei den Absendern erzeugt wurden. Die vorhandene Implementierung mittels einer Lamport-Uhr [Lam78] funktionierte nur mit zwei Teilnehmern zuverlässig, da jeder Teilnehmer nur einen Zeitstempel verwaltete.

Ein Mechanismus zur Sequenzialisierung ist nötigt, weil neben dem XMP-Protokoll, welches die Beibehaltung der Reihenfolge von Nachrichen garantiert [SA04, Seiten 8 und 58], Daten über *Peer-to-Peer*-Verbindungen übertragen werden. Typisches Beispiel ist das Anlegen einer neuen Datei. Deren Inhalt wird, falls das möglich ist, asynchron übertragen, direkt gefolgt von einer *EditorActivity* für das Editorfenster für die neue Datei. Ohne funktionsfähige Sequenzialisierung kam es oft vor, dass die *EditorActivity* vor dem Dateiinhalt ankam und versucht wurde, für eine noch nicht existierende Datei einen Editor zu öffnen.

### 2.3.1 Mehr als zwei Teilnehmer berücksichtigen

Die vorgefundene Implementierung war wirklich total kaputt. In der Warteschlange konnte grundsätzlich maximal eine Aktivität stehen, und alle Aktivitäten die einen höheren Zeitstempel besassen, als der Erwartete, wurden einfach verworfen. Aus diesem Grund war die erste Massnahme, diese Implementierung komplett zu entfernen, und einfach alle Aktivitäten sofort nach dem Eintreffen auszuführen. Dadurch wurden einige Operationen in der falschen Reihenfolge ausgeführt, aber insgesamt sind weniger Aktivitäten verlorengegangen, weil nichts mehr ignoriert wurde.

Die Zeitstempel wurden durch fortlaufende Sequenznummern und eine Warteschlange für eingehende Nachrichten pro anderem Teilnehmer ersetzt. Für die Sequenznummern werden bei jedem Teilnehmer für jeden Anderen zwei Zähler geführt – einer, der zur Nummerierung der ausgehenden Nachrichten verwendet wird, und ein Zweiter, über den Buch geführt wird, welche Sequenznummer vom jeweiligen Teilnehmer als nächstes erwartet wird.

Die Informationen werden in Exemplaren vom Typ *ActivityQueue* gespeichert. Diese kennen den Teilnehmer, für den sie verantwortlich sind, in Form seiner Jabber-ID, enthalten die beiden Felder *nextSequenceNumber* und *expectedSequenceNumber*, eine Prioritätswarteschlange für die eingegangenen Aktivitäten, und den ältesten Zeitstempel dieser Aktivitäten. Hier ist jetzt mit Zeitstempel eine reale Zeit, nämlich die lokale “wall clock” gemeint, zu der die Aktivität in der Warteschlange ankam. Diese Angabe wird verwendet, um zu entscheiden, wann eine Aktivität zu alt ist, als dass noch

Hoffnung bestünde, dass fehlende Aktivitäten, die zeitlich davor lagen, doch noch eintreffen. Ausserdem wird noch ein Verlauf von Aktivitäten geführt, der allerdings aktuell nicht genutzt wird. Für die Prioritätswarteschlange wurde die Klasse *TimedActivity*[4] um eine Implementierung der *Comparable*-Schnittstelle erweitert.

Verwaltet werden die *ActivityQueue* Exemplare von einem *ActivityQueuesManager*, der Jabber-IDs auf *ActivityQueue*-Objekte abbildet, und Operationen zum Erzeugen von *TimedActivity*s für gegebene Empfänger bereitstellt, und zum Einfügen von empfangenen Aktivitäten in die Warteschlange und der Abfrage von solchen, die Ausgeführt werden können. Das ist immer dann der Fall, wenn keine Aktivitäten, die zeitlich davor liegen, fehlen, oder eine Wartezeit überschritten ist. Eine Zeitüberschreitung ist nämlich der Weg, mit dem im Moment mit verlorengegangenen Aktivitäten umgegangen wird. Die Wartezeit beträgt 30 Sekunden, bzw. das doppelte, falls zum Zeitpunkt der Überprüfung gerade ein eingehender Dateitransfer von dem Teilnehmer, zu dem die Warteschlange gehört, läuft. Dateiübetragungen, die länger als eine Minute benötigen, führen also zum Verwerfen von Aktivitäten. Das ist an der Stelle ein Kompromiss, da ansonsten "hängende" Übertragungen, alle anderen Aktivitäten von dem entsprechenden Teilnehmer blockieren würden. Es folgt eine kleine Beschreibung der Methoden von dem *ActivityQueuesManager*:

***getActivityQueue()*** ist eine interne Methode und holt eine *ActivityQueue* für einen gegebenen Teilnehmer aus der Abbildung von Jabber-IDs auf Warteschlangen. Sollte keine solche Warteschlange existieren, wird eine neue angelegt und in die Abbildung eingetragen.

***createTimedActivities()*** erzeugt für einen gegebenen Teilnehmer und eine Liste mit Aktivitäten, eine Liste mit *TimedActivity*-Exemplaren. Diese Methode steht über eine Methode, die den Aufruf nur weiterleitet, auf dem *ActivitySequencer* nach "aussen" zur Verfügung. Sie ruft für jede einzelne Aktivität eine Methode auf der zum Teilnehmer gehörigen *ActivityQueue* auf, welche die Aktivität in eine mit einer Sequenznummer versehenen *TimedActivity* verpackt.

***add()*** wird vom *ActivitySequencer* mit empfangenen Aktivitäten aufgerufen und fügt diese der Queue der Quelle der Aktivität hinzu, wo ihr noch ein Zeitstempel mit der "wall clock time" verpasst wird. Aktivitäten vom Typ *TextEditActivity* werden gesondert behandelt, da diese zusätzlich zum Sender der Nachricht, noch die originale Quelle der Aktivität enthalten. Die beiden Angaben unterscheiden sich voneinander bei Textaktivitäten, die über den Host durch den *Jupiter*-Algorithmus gegangen sind.

***removeQueue()*** entfernt die Warteschlange für den gegebenen Teilnehmer. Diese Methode wird aufgerufen, wenn der *ActivitySequencer* darüber informiert wird, dass der Teilnehmer die Sitzung verlassen hat.

[4] Der Klassenname stimmt nicht mehr, da die Aktivitäten mittlerweile keine Zeitstempel mehr tragen, sondern Sequenznummern erhalten. Mir fiel allerdings noch kein passenderer, akzeptabler Name für diese Klasse ein.

***removeActivities()*** erfragt von allen Warteschlangen die Aktivitäten, welche sofort ausgeführt werden können. Der Aufruf erfolgt von zwei Stellen innerhalb des *ActivitySequencer*. Einmal direkt, nach dem Empfang und hinzufügen einer Aktivität, weil diese bisher blockierte Aktivitäten freigeben kann. Die andere Stelle ist ein *TimerTask*, der alle $\frac{1}{2}$ Sekunde schaut, ob schon zu lange auf ausstehende Aktivitäten gewartet wird, und deshalb eventuell wartende Aktivitäten zur Ausführung frei gegeben werden. Siehe die Beschreibung von *checkForMissingArguments()* weiter unten.

***getHistory()* & *getExpectedSequenceNumbers()*** sind zwei Methoden, die nicht verwendet werden. Früher war einmal vorgesehen, dass ein Teilnehmer einen Anderen nach allen Aktivitäten ab einer gegebenen Sequenznummer, bzw. damals noch ab einem Zeitstempel, fragen konnte, und diese Aktivitäten erneut zugesandt bekam. Zum Beispiel um nach Verbindungsabrüchen Versäumtes wieder aufzuholen. Die Aufrufe an den entsprechenden Stellen waren schon zu Beginn dieser Arbeit alle auskommentiert, ich wurde aber angehalten, die Funktionalität grundsätzlich im vorhandenen Rahmen aufrecht zu erhalten.

***checkForMissingArguments()*** wird von einem *TimerTask* jede halbe Sekunde aufgerufen (siehe auch *removeActivities()* weiter oben) und ruft die gleichnamige Methode auf den verwalteten *ActivityQueue*-Exemplaren auf. Dort wird überprüft, ob die älteste Aktivität in der Warteschlange das "Mindesthaltbarkeitsdatum" überschritten hat, und gegebenenfalls wird der Zähler für die nächste zu erwartende Sequenznummer erhöht und somit eine nicht empfangene Nachricht "verworfen".

Die eben beschriebenen Klassen, Felder, und Methoden kann man im Klassendiagramm in Abbildung 2.2 auf der nächsten Seite – mit einem Vergrösserungsglas bewaffnet – wiederfinden.

### 2.3.2 Aufgabentrennung

Die Aufgabentrennung bei den Objekten rund um den *ActivitySequencer* war nicht besonders sauber. Das *SharedProject*-Exemplar besass einen *TimerTask*, der in regelmässigen Abständen das Versenden von gepufferten, ausgehenden Aktivitäten anstiess. Wie man im Sequenzdiagramm in Abbildung 2.3 auf Seite 23 sehen kann, wurden dazu vom *ActivitySequencer* die Aktivitäten abgefragt und dem *XMPPChatTransmitter* zum Versenden übergeben. Zusätzlich bekam der eine Referenz auf den *ActivitySequencer* um mit dessen Hilfe aus den *IActivity*-Exemplaren *TimedActivity*s für die jeweiligen Empfänger zu erstellen, die er vom *SharedProject* erfragte.

Die drei Objekte waren also recht stark gekoppelt und haben Aufgaben übernommen, die besser beim *ActivitySequencer* aufgehoben sind. Insbesondere steckte beim Versenden im *XMPPChatTransmitter* zu viel Wissen über die Aktivitäten und wie man sie mit Sequenznummern versehen muss. Dort wurden sogar *TextEditActivity*s noch einmal besonders behandelt, um den originalen Sender zu setzen – Wissen, welches eindeutig in

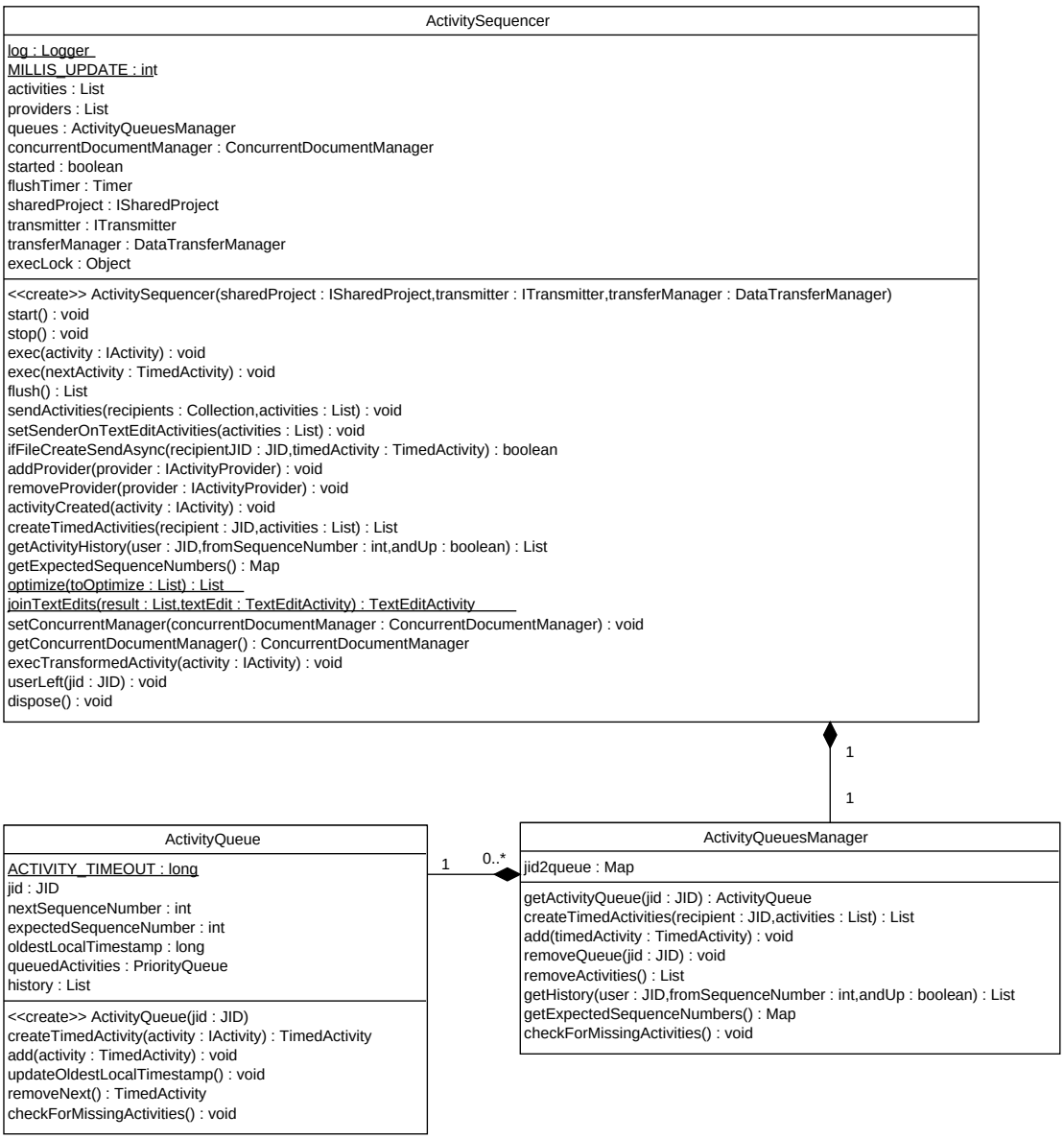


**Abbildung 2.2:** Die Klasse *ActivitySequencer* und die beiden inneren Klassen *ActivityQueuesManager* und *ActivitySequencer*

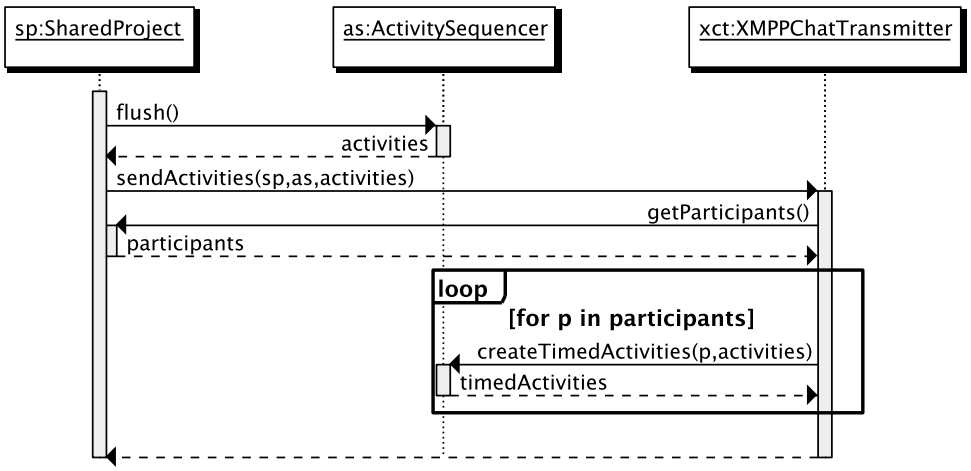


**Abbildung 2.3:** Sequenzdiagramm für das periodisch aufgerufene Versenden von ausgehenden Nachrichten vor der Umverteilung der Verantwortlichkeiten. Das *SharedProject* ist die Treibende kraft. Der *XMPPChatTransmitter* ist auch sehr aktiv beteiligt. (r1053)

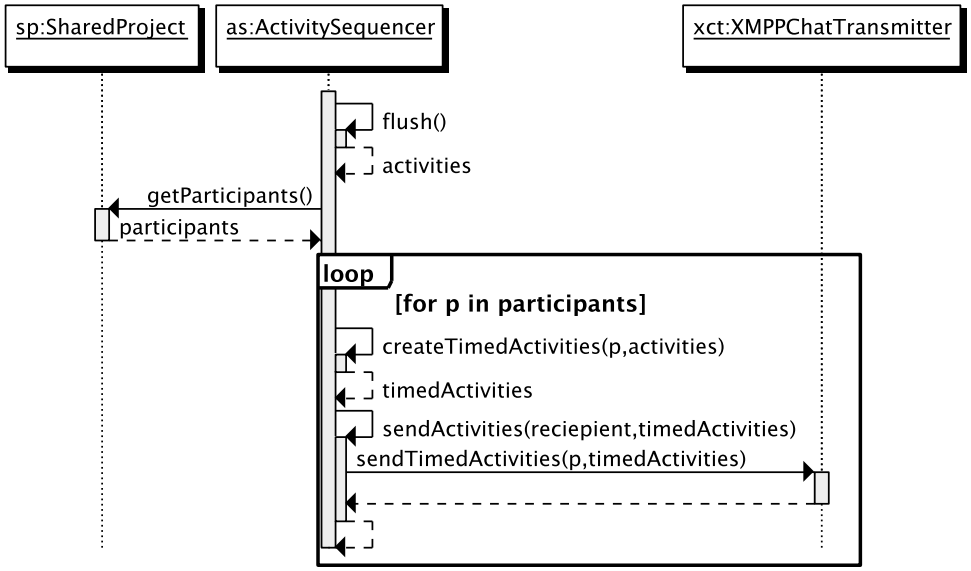


**Abbildung 2.4:** Sequenzdiagramm für das periodisch aufgerufene Versenden von ausgehenden Nachrichten nach der Umverteilung der Verantwortlichkeiten. Der *ActivitySequencer* hat die Führungsrolle übernommen.

den Verantwortungsbereich des *ActivitySequencer* gehört.

Nach der Überarbeitung wird die Hauptarbeit im *ActivitySequencer* verrichtet, wie man im Sequenzdiagramm in Abbildung 2.4 auf der vorherigen Seite sehen kann. Der *TimerTask* gehört nun zum *ActivitySequencer* und die Operationen laufen grösstenteils intern ab. Das *SharedProject* wird nur noch nach den Teilnehmern der aktuellen Sitzung gefragt, und der *XMPPChatTransmitter* bietet eine einfache Schnittstelle zum Versenden von Aktivitäten an einen gegebenen Empfänger.

## 2.4 XML-Erzeugung (AP3)

Viele von den XML-Nachrichten, die über das XMP-Protokoll übertragen werden, wurden Anfangs "von Hand" mit *StringBuilder*-Exemplaren, oder auch nur mittels Konkatenation von Zeichenketten oder *String.format()* erstellt. Dabei konnte sehr einfach nicht-valides XML entstehen. In Listing 2.3 auf der nächsten Seite ist die alte Vorgehensweise Anhand von Beispielen aus den Aktivitäten zu sehen. Die abstrakte Basisklasse *AbstractActivity* bietet eine Methode um die Aktivität als XML-Fragment darzustellen. Dort wird ein *StringBuilder* erstellt, der dann von der konkreten Unterklasse mit Inhalt gefüllt wird. In *EditorActivity* werden die einzelnen Teile der XML-Nachricht nacheinander angefügt, während *TextEditActivity* den Weg über *String.format()* geht.

Zu diesem Zeitpunkt wurde immerhin schon der Inhalt der Textknoten bei der *TextEditActivity* ordentlich mit Hilfe von CDATA-Abschnitten vor dem XML-Parser geschützt. Ein *CDATA*-Abschnitt sieht wie folgt aus: <![**CDATA**[ … ]]>. Anstelle der Auslassungspunkte können nahezu beliebige Zeichenfolgen stehen, ausser der Endkennzeichnung für *CDATA*-Abschnitte und Zeichen, die im XML-Standard nicht erlaubt sind. Vor der Einführung dieser Schutzmassnahme konnte man durch Eingabe der Zeichenfolge ]]> in einem Texteditor ganz leicht und zuverlässig eine Inkonsistenz erzeugen.

| **Zeichenname** | | **Wert** | |
|---|---|---|---|
| horizontal tab | (HT) | 9 | 0x09 |
| line feed | (LF) | 10 | 0x0A |
| carriage return | (CR) | 13 | 0x0D |

**Tabelle 2.1:** In XML erlaubte ASCII-Zeichen mit Wert $< 32$.

*CDATA*-Abscnitte schützen den Inhalt von Textknoten nahezu ausreichend. Was damit nicht erfasst wird, sind Zeichen beziehungsweise Bytewerte, welche in XML-Dokumenten nicht vorkommen dürfen, die aber manchmal in Textdateien enthalten sind. Von den Zeichen der ASCII-Kodierung mit Bytewerten unter 32, sind nur der horizontale Tabulator, der Zeilenvorschub, und der Wagenrücklauf erlaubt. Manche Programmierer setzen aber auch andere "whitespace"-Zeichen zur Formatierung oder Gliederung ein, wie zum Beispiel einen Blattvorschub[5] um grössere Einheiten wie Funktionen, oder Methoden voneinander zu trennen.

Völlig ungeschützt waren Attributwerte, in denen von *Saros* zum Beispiel Dateipfade kodiert werden. Was Dateinamen enthalten dürfen, hängt vom verwendeten Betriebs-

[5] auf englisch *form feed* (FF), Bytewert 12 (0x0C)

**Listing 2.3:** Die *toXML()*-Methoden von *AbstractActivity*, *EditorActivity*, und *TextEditActivity* (r983)

```
1  public abstract class AbstractActivity implements IActivity {
2      // ...
3      public String toXML() {
4          // TODO Escape data in XML in subclasses, for example paths.
5          StringBuilder sb = new StringBuilder();
6          this.toXML(sb);
7          return sb.toString();
8      }
9  }
10
11 public class EditorActivity extends AbstractActivity {
12     // ...
13     public void toXML(StringBuilder sb) {
14         sb.append("<editor ");
15         sb.append("path=\"").append(getPath()).append("\" ");
16         sb.append("type=\"").append(getType()).append("\" ");
17         sb.append("checksum=\"").append(getChecksum()).append("\" ");
18         sb.append("/>");
19     }
20 }
21
22 public class TextEditActivity extends AbstractActivity {
23     // ...
24     public void toXML(StringBuilder sb) {
25
26         String result = String
27             .format(
28                 "<edit path=\"%s\" offset=\"%d\" source=\"%s\"><text>%s</text
                       ><replace>%s</replace></edit>",
29                 getEditor(), offset, getSource(), Util.escapeCDATA(text),
                       Util
30                     .escapeCDATA(replacedText));
31
32         sb.append(result);
33     }
34 }
```

und Dateisystem ab. Die kleinsten Einschränkungen, wenn man die von *Saros* unterstützten Plattformen betrachtet, gibt es bei *Linux*, wo bei vielen gängigen Dateisystemen bei den erlaubten Bytewerten in Dateinamen nur das Nullbyte und das Zeichen für den Pfadtrenner ausgeschlossen werden.[6] Dateipfade können also auch die oben schon erwähnten Zeichen enthalten, die nicht in XML-Dokumenten vorkommen dürfen.

Selbst Pfadnamen, welche nur erlaubte Zeichen enthalten, können nicht immer verlustfrei als ungeschützter Attributwert übertragen werden, weil XML-Parser den Attributwert normalisieren dürfen. Das bedeutet,

- sämtliche "whitespace"-Zeichen dürfen durch Leerzeichen ersetzt werden,
- diese dürfen am Anfang und am Ende des Wertes entfernt werden,
- und mehrere aufeinanderfolgende Leerzeichen dürfen zu Einem zusammengefasst werden.

### 2.4.1 Explizit "escapen"

In einem ersten Anlauf die erzeugten XML-Fragmente robuster zu machen, wurden die gefährdeten Attributwerte mit der *Prozent-Kodierung* gemäss [BLFM05, Seite 12] kodiert:

> A percent-encoded octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing that octet's numeric value.

Beispiel: Aus der Eingabe `''Ein dümmliches/Beispiel''` wird durch die Prozent-Kodierung `''Ein%20d%C3%BCmmliches%2FBeispiel''`.

Bei Pfadnamen wurde neben der Prozent-Kodierung auch an allen Stellen sichergestellt, dass nicht mehr die *toString()*- und *fromString()*-Methode auf *IPath*-Exemplaren verwendet wurde, sondern *toPortableString()* und *fromPortableString()*, um die Interoperabilität über Plattformgrenzen hinweg zu sichern.

Nicht betroffen sind Attributwerte, bei denen sicher ist, dass sie keine verbotenen Zeichen oder signifikante "whitespace"-Zeichen enthalten, wie zum Beispiel numerische Werte.

Listing 2.4 auf der nächsten Seite zeigt wieder am Beispiel der Klassen *EditorActivity* und *TextEditActivity* die Veränderungen in der *toXML()*-Methode. Die Methoden *Util.urlEscape()* und *Util.urlUnescape()* stützen sich auf die *Apache Commons*-Bibliothek der *Apache Software Foundation*.[7]

[6] Die Aufnahme von *MacOS* in die Liste der unterstützten Plattformen wird daran nichts ändern.
[7] http://commons.apache.org/codec/

**Listing 2.4:** Die *toXML()*-Methoden von *EditorActivity* und *TextEditActivity* (r1006)

```
1  public class EditorActivity extends AbstractActivity {
2      // ...
3      public void toXML(StringBuilder sb) {
4          sb.append("<editor ");
5          sb.append("path=\"").append(
6                  Util.urlEscape(getPath().toPortableString())).append("\" ");
7          sb.append("type=\"").append(getType()).append("\" ");
8          sb.append("/>");
9      }
10 }
11
12 public class TextEditActivity extends AbstractActivity {
13     // ...
14     public void toXML(StringBuilder sb) {
15
16         String result = String
17             .format(
18                 "<edit path=\"%s\" offset=\"%d\" source=\"%s\" sender=\"%s
                       \"><text>%s</text><replace>%s</replace></edit>",
19                 Util.urlEscape(getEditor().toPortableString()), offset, Util
20                     .urlEscape(getSource().toString()), Util
21                     .urlEscape(getSender().toString()), Util.escapeCDATA(text
                           ),
22                 Util.escapeCDATA(replacedText));
23
24         sb.append(result);
25     }
26 }
```

### 2.4.2 Serialisierung mit *XStream*

Schöner als das explizite Schützen von Inhalten beim Zusammenbauen von Zeichenketten, ist natürlich eine API zu verwenden, welche das Erstellen von XML-Fragmenten einfacher und sicherer macht. Dazu habe ich mir *JDOM*[8] und *XStream*[9] näher angeschaut. Ersteres ist eine Java-API, die eine etwas leichtgewichtigere und besser auf die Sprache Java zugeschnittene Alternative zum sprachunabhängigen *Document Object Model* (*DOM*) vom *W3C*[10] sein möchte. Mit der API lassen sich leicht valide XML-Fragmente erstellen, wobei der Programmierer von der Verantwortung befreit wird, selber dafür zu sorgen, dass Tags nicht falsch verschachtelt werden, und es zu jedem öffnenden Tag auch ein Schliessendes gibt. Inhalte von Attributwerten und Textknoten werden als Zeichenketten übergeben und müssen nicht vorher "escaped" werden. *XStream* macht es dem Programmierer noch einfacher – man zeichnet Klassen und Attribute mit Annotationen aus, registriert die Klassen zur Laufzeit bei einem *XStream*-Exemplar und kann fortan darüber Objekte von registrierten Typen in XML umwandeln, sowie XML zurück in Objekte. Die Entscheidung viel zu Gunsten von *XStream* aus.

Beim Zurückwandeln von XML in Objekte ergab sich eine kleine Hürde. *XStream* bietet zwar eine Adapterklasse für einen *XmlPullParser*[11], der uns von der XMPP-Bibliothek *Smack* übergeben wird, aber dieser Adapter geht davon aus, dass der Parser noch nicht mitten im Dokument "steht". Da unsere Nachrichten in den XMPP-Nachrichten eingebettet sind, hat *Smack* mit dem *XmlPullParser* natürlich schon einen Teil der Nachricht geparst und auch schon das *START_NODE*-Ereignis für das Element, mit dem der für uns interesannte Teil beginnt, gesehen. Ein eigener Adapter, der den Parser kapselt und dieses Ergeignis beim ersten Aufruf erneut simuliert, war aber schnell geschrieben.

In Listing 2.5 auf der nächsten Seite sind die *XStream*-Auszeichnungen für die *TextEditActivity* zu sehen. Die *toXML()*-Methode, wie man sie noch in Listing 2.4 auf der vorherigen Seite sieht, ist ersatzlos gestrichen. Es werden folgende Annotationen im gezeigten Quelltextschnippsel verwendet:

***XStreamAlias*** gibt dem Tag oder Attribut einen anderen Namen als den Namen im Java-Programm. Besonders bei Klassen wichtig, weil dort sonst der vollständige Name, inklusive des Package-Namens, als Tag-Name verwendet wird, und dieser dadurch sehr lang wird. Das resultierende XML wäre für den Menschen nicht mehr so leicht zu lesen. Bei den Attributen ist das Umbenennen nicht unbedingt erforderlich, wird bei den Aktivitäten aber oft gemacht, um die Namensgebung durch die Umstellung auf *XStream* möglichst unverändert zu lassen, da in einem ersten Schritt erst einmal nur das Serialisieren umgestellt wurde, und das Deserialisieren noch mit dem vorhandenen Code passierte, bis das weiter oben be-

[8] http://jdom.org/
[9] http://xstream.codehaus.org/
[10] http://www.w3.org/DOM/
[11] http://www.xmlpull.org/

**Listing 2.5:** *XStream*-Annotationen für die *TextEditActivity*

```
1  @XStreamAlias("textEditActivity")
2  public class TextEditActivity extends AbstractActivity {
3
4      @XStreamAsAttribute
5      public final int offset;
6
7      /** ... */
8      public final String text;
9
10     @XStreamAlias("replaced")
11     public final String replacedText;
12
13     @XStreamAsAttribute
14     protected final IPath editor;
15
16     @XStreamAsAttribute
17     @XStreamConverter(UrlEncodingStringConverter.class)
18     protected String sender;
19     // ...
```

**Listing 2.6:** XML für eine *TextEditActivity*

```
1  <textEditActivity source="test02%40s8n%2FSmack" offset="554"
2                    editor="src%2FTest.java" sender="test02%40s8n%2FSmack">
3      <text>baz</text>
4      <replaced>foobar</replaced>
5  </textEditActivity>
```

schriebene Problem mit dem *XmlPullParser* behoben war.

***XStreamAsAttribute*** weist *XStream* an, das Feld als Attribut und nicht als Element zu serialisieren. Das geht nur bei einfachen, nicht zusammengesetzten Objekten, beziehungsweise bei Solchen, für deren Typ ein Konverter registriert ist, der Exemplare in einfache Zeichenketten und zurück wandeln kann. Siehe *XStreamConverter* weiter unten.

***XStreamConverter*** gibt explizit einen Konverter für ein Feld vor. Normalerweise sucht sich *XStream* anhand des Typs eines Feldes den passenden Konverter unter den vorher Registrierten aus. Manchmal möchte man aber von diesem abweichen, zum Beispiel um eine Zeichenkette so zu "escapen", dass sie als Attributwert verwendet und verlustfrei wieder zurückgewonnen werden kann. Warum das nötig sein kann, wird im Abschnitt über das Normalisieren von Attributwerten auf Seite 26 erläutert.

Das *XStream*-Exemplar bei dem alle Aktivitäten und eine *IPathConverter*-Klasse registriert sind, erzeugt mit seiner *toXML()*-Methode XML-Fragmente wie in Listing 2.6 ge-

**Listing 2.7:** *IPathConverter* – Konvertiert Pfade in plattformunabhängige Zeichenketten.

```
1  public class IPathConverter extends AbstractSingleValueConverter {
2
3      @SuppressWarnings("unchecked")
4      @Override
5      public boolean canConvert(Class clazz) {
6          return clazz.isAssignableFrom(IPath.class);
7      }
8
9      @Override
10     public Object fromString(String path) {
11         return Path.fromPortableString(Util.urlUnescape(path));
12     }
13
14     @Override
15     public String toString(Object obj) {
16         return Util.urlEscape(((IPath) obj).toPortableString());
17     }
18 }
```

zeigt. Das *source*-Attribut stammt aus der Basisklasse *AbstractActivity*.

Selbst geschriebene Konverter-Klassen gibt es zwei Stück. Eine für *IPath*-Exemplare, die von *XStream* aufgrund des Typs von Feldern ausgewählt wird, und zum anderen den *UrlEncodingStringConverter*, der explizit per Annotation für Felder vom Typ *String* ausgewählt werden kann. Diese Klassen erben von einem, von *XStream* zur Verfügung gestellten, *AbstractSingleValueConverter* und müssen nur drei recht einfache Methoden implementieren. Eine *canConvert()*, welche eine Klasse erwartet und prüft, ob der Konverter ein Exemplar dieser Klasse verarbeiten kann oder nicht, und das Methodenpaar *toString()* und *fromString()*, welche so ein Exemplar in eine Zeichenkette umwandeln, und eine Zeichenkette wieder in ein Objekt vom entsprechenden Typ. Listing 2.7 zeigt den *IPathConverter*, der dafür sorgt, dass Pfade als plattformübergreifende und für Attributwerte geeignete Zeichenketten kodiert werden.

Beim Umstellen der (De)Serialisierung von Aktivitäten ist die Klasse *ActivityRegistry* obsolet geworden. Dort wurden *IActivityProvider*-Objekte registriert, die eine *fromXML()*-Methode anboten, um am übergebenen Parser ihr Glück zu versuchen, das heisst sie haben geschaut, ob das erste Element einen Tag-Namen von einer Aktivität besass, die sie deserialisieren können. Namentlich waren das

- der *EditorManager* (*EditorActivity*, *TextEditActivity*, *TextSelectionActivity*, und *ViewportActivity*),
- der *RoleManager* (*RoleActivity*),
- und der *SharedResourcesManager* (*FileActivity* und *FolderActivity*).

Nachdem die einzelnen Aktivitäten nun umgestellt waren, ging es in der Hierarchie der

**Listing 2.8:** *FileList*

```
1  @XStreamAlias("fileList")
2  public class FileList {
3      /**
4       * @invariant Contains all entries from this.added, this.altered, and
5       *            this.unaltered.
6       *
7       * @see #readResolve()
8       */
9      @XStreamOmitField
10     private Map<IPath, Long> all = new HashMap<IPath, Long>();
11
12     private final Map<IPath, Long> added = new HashMap<IPath, Long>();
13     private final Map<IPath, Long> removed = new HashMap<IPath, Long>();
14     private final Map<IPath, Long> altered = new HashMap<IPath, Long>();
15     private final Map<IPath, Long> unaltered = new HashMap<IPath, Long>();
16
17     // ...
18
19     /**
20      * This is called after deserialization by XStream.
21      *
22      * Initializes {@link #all} (which is not sent via XML) to ensure the
23      * invariant of this class.
24      */
25     private Object readResolve() {
26         all = new HashMap<IPath, Long>();
27         all.putAll(added);
28         all.putAll(altered);
29         all.putAll(unaltered);
30         return this;
31     }
32     // ...
```

verschachtelten Teile einer Nachricht eine Stufe nach oben zur *ActivitiesPacketExtension*, die nun komplett über *XStream*(de)serialisiert wird.

Nach den Aktivitäten kam die *FileList* an die Reihe. Die Klasse ist ein Container für *IPath*-Objekte, für die Prüfwerte berechnet werden. Ausserdem werden Mengenoperationen zur Verfügung gestellt, die es erlauben die Unterschiede und Gemeinsamkeiten zwischen zwei *FileList*-Exemplaren zu ermitteln. Diese Listen werden zum Beispiel beim Einladungsvorgang zwischen Host und potenziellem neuen Teilnehmer ausgetauscht. Eine Besonderheit in Bezug auf die Serialisierung ergibt sich dadurch, dass die Daten redundant sind. Es gibt Felder für hinzugefüge, veränderte, und unveränderte Dateien und ein Feld, welches alle Daten dieser drei Felder zusammen noch einmal enthält. Diese Redundanz möchte man natürlich nicht bei der Übertragung haben, deshalb ist das Feld *all* mit der Annotation *XStreamOmitField* von der Serialisierung durch *XStream* ausgenommen. Damit der Inhalt beim Deserialisieren wieder hergestellt

wird, gibt es die *readResolve()*-Methode, die von *XStream* nach der normalen Deserialisierung aufgerufen wird. Der Name der Methode scheint nicht besonders sprechend gewählt zu sein – es gab auf der *Saros*-Mailingliste nachfragen deswegen – die *XStream*-Entwickler haben hier aber einfach den Namen übernommen, den die Serialisierungs-API aus der Standardbibliothek für so eine Methode vorsieht. Die relevanten Quelltextausschnitte sieht man in Listing 2.8 auf der vorherigen Seite.

Parallel zu diesen Änderungen hat Sebastian Ziller die *Jupiter-Request*s zu Aktivitäten umgebaut. Die *XStream*-Annotationen für die Klasse *Request* und den davon referenzierten Klassen, haben wie gemeinsam in einer Paarprogrammierungs-Sitzung vorgenommen. Auf die altmodische Art – nicht mit *Saros*.

Es sind noch nicht alle *PacketExtensions*, also Objekte, die in XMPP-Nachrichten eingebettete XML-Dokumente repräsentieren, auf *XStream* umgestellt. Eher durch Zufall habe ich entdeckt, dass wir das nicht nur mit den Klassen machen sollten, in denen *wir* XML naiv und fehleranfällig mit Zeichenketten zusammenbasteln, sondern auch mit Unterklassen von der *DefaultPacketExtension*, die *Smack* zu Verfügung stellt, und die eine sehr einfache API bietet, um Schlüssel und dazugehörige Werte, beides Zeichenketten, in einer Nachricht unterzubringen. Erschreckenderweise wird dort intern nämlich auch nur ein *StringBuilder* verwendet und die gegebenen Zeichenketten werden ohne weitere Behandlung einfach eingesetzt. Stilistisch sehr unschön ist auch das dynamische Erzeugen von durchnummerierten Tag-Namen in XML, was zum Beispiel in der *UserListExtension* gemacht wird, um eine variable Anzahl von Benutzern zu kodieren. Das ist in etwa vergleichbar mit dem Anlegen von durchnummerierten Namen in dynamischen Programmiersprachen, anstelle eine Liste oder ein Array zu verwenden.

## 2.5 Sonstiges

In diesem Abschnitt werden Änderungen erwähnt, die nicht zu einem der drei Arbeitspakete passen, und noch nicht im Abschnitt 2.1 auf Seite 9 über die Einarbeitungsphase beschrieben wurden.

### 2.5.1 Quadratisch, unpraktisch, gar nicht gut...

Noch relativ am Anfang dieser Arbeit stiessen Christoph Jacob und ich bei etwas grösseren Projekten auf ein problematisches Laufzeitverhalten beim Starten und Beenden einer Sitzung. Der erste Verdacht war eine langsame Übermittlung der Dateien, eventuell in Verbindung mit dem Erstellen einer Archivdatei, in welcher die Dateien übertagen werden. Damit konnte man aber die Verzögerung am Ende der Sitzung nicht erklären. Nach Tests mit Projekten verschiedener Grösse, zeichnete sich sehr deutlich ab, dass die “Denkpause” wohl in einem quadratischen Verhältnis zur Anzahl der Dateien

**Listing 2.9:** *SharedProject.setProjectReadonly()* (r456)

```
public void setProjectReadonly(final boolean readonly) {
    // ...
            FileList flist;
            try {
                flist = new FileList(SharedProject.this.project);

                monitor.beginTask("Project settings ...",
                        flist.getPaths().size());
                // ...
                for (int i = 0; i < flist.getPaths().size(); i++) {
                    IPath path = flist.getPaths().get(i);
                    // ...
                }
    // ...
```

**Listing 2.10:** *FileList.getPaths()* (r456)

```
public class FileList {
    private Map<IPath, Long> all = new HashMap<IPath, Long>();
    // ...
    public List<IPath> getPaths() {
        return sorted(this.all.keySet());
    }
    // ...
```

im Projekt steht. Ein Projekt mit ungefähr 7.000 Dateien sorgte noch vor dem ersten Einladungsvorgang und nach dem Beenden einer Sitzung jeweils ca. 20 Minuten für 100% Prozessorauslastung, in denen aber augenscheinlich nicht viel passierte.

Mittels Debugger und Haltepunkten stellte sich die Methode *setProjectReadonly()* in der Klasse *SharedProject* (Listing 2.9) als die Quelle der übertriebenen Wartezeit heraus. *FileList*-Exemplare ermöglichen das Erstellen von Dateilisten aus den Resourcen eines *Eclipse*-Projekts und bieten Mengenoperationen auf solchen Listen. Man kann schon an dem Quelltextschnippsel erkennen, dass die *getPaths()*-Methode bei jedem Schleifendurchlauf, also für jede Datei im Projekt zweimal aufgerufen wird. Dabei wird jedesmal eine Liste mit allen Dateien erstellt. Ein Blick in den Quelltext von der Methode *getPaths()* (Listing 2.10) zeigt, dass sogar noch mehr passiert: Die Liste wird aus den Schlüsseln einer Hashmap erstellt und sortiert. Für *jede* Datei im Projekt, nur um die Grösse der Liste abzufragen – die sich nicht verändert, und *einen* Pfad aus dieser Liste zu nehmen und etwas damit zu tun. Wahrscheinlich gibt es bei jeder grösseren Quelltextbasis früher oder später Code, der ein Fall für *The Daily WTF*[12] ist. Die triviale Lösung dieser Bremse ist eine einfache *foreach*-Schleife über die Liste, welche nur *einmal* vor der Schleife zusammengestellt wird.

[12] http://thedailywtf.com/

### 2.5.2 Mehr-Schreiber-Modus

Der experimentelle Mehr-Schreiber-Modus konnte zwar schon aktiviert werden, aber das Projekt befand sich noch in einer Umbruchphase. So gab es einige Stellen im Projekt, die noch von einem einzigen, exklusiven Schreiber ausgingen. Teilweise nur in den Bezeichnern, wie Namen im Singular, hinter denen nicht mehr einzelne Objekte steckten, sondern *Collection*s oder *Map*s. An einigen Stellen gab es aber auch noch Code, der nur einen einzigen Schreiber berücksichtige.

Beispielhaft für die nicht mehr treffenden Namen sei hier das Feld *activeDriver* in der Klasse *DriverDocumentManager* genannt, welches mittlerweile an eine ganze Liste mit Jabber-IDs gebunden war.

Beim Code war zum Beispiel das Feld *driverTextSelection* der Klasse *EditorManager* betroffen, in dem die Textauswahl "des" Schreibers hinterlegt wurde. Diese eine *ITextSelection* musste durch eine Abbildung von Benutzern auf solche Objekte ersetzt werden.

### 2.5.3 *BlockingProgressMonitor*

*Eclipse* stellt für Plugin-Programmierer viele Methoden für Aktionen zu Verfügung, die etwas länger dauern können, und deshalb eine Schnittstelle zum Überwachen des Fortschritts der Aktion bieten. Ausserdem kann man eine laufende Aktion darüber auch abbrechen. Wenn man an der Fortschrittsinformation kein Interesse hat, und auch sicher ist, dass man die Aktion nicht abbrechen möchte, kann man dort einen *NullProgressMonitor* übergeben, der zur Bibliothek von *Eclipse* gehört. Das wurde an einigen Stellen bei *Saros* so gemacht. Die Aktionen laufen allerdings asynchron, so dass es bei Aktionen wie dem Anlegen oder Löschen von Dateien und Ordnern zu Wettlaufsituationen kam. Um Diese zu verhindern gibt es nun einen *BlockingProgressMonitor*, der den *NullProgressMonitor* erweitert, und eine blockierende Methode bietet, welche auf das Ende oder den Abbruch der Aktion wartet.

### 2.5.4 Java 1.5 Kompatibilität

Als Mindestanforderung für *Saros* gibt die Projektwebseite eine Java-Laufzeitumgebung der Version 1.5 an. Da bei den Entwicklern aber in der Regel schon 1.6 installiert ist, können sich Inkompatibilitäten einschleichen, vor denen *Eclipse* nicht zuverlässig warnt. So passiert bei der Verwendung des Konstruktors von *IOException* der neben der Nachricht auch noch ein *Throwable* entgegennimmt. Diesen Konstruktor gibt es in der Standardbibliothek von Java 1.5 noch nicht. Das Problem ist zufällig aufgefallen, nachdem ich den Entwicklungsrechner gewechselt hatte, auf ein System, das als Laufzeitumgebung auch eine JRE 1.5 zur Verfügung stellte. Dort hat *Eclipse* den Konstruktoraufruf als Fehler erkannt und als solchen markiert.

Die Basisklasse *Exception* kennt auch bei Java 1.5 schon die Möglichkeit eine geschachtelte Ausnahme aufzunehmen, also war die Lösung einfach eine Klasse *CausedIOException* zu implementieren, welche von *IOException* erbt, einen entsprechenden Konstruktor zur Verfügung stellt, und die übergebene Ausnahme über *Exception.initCause()* setzt.

Sobald *Saros* auf Java 1.6 als Mindestanforderung umsteigt, ist diese Klasse überflüssig und kann wieder entfernt werden. *Sun*'s Unterstützung für Java 1.5 läuft laut der *Java Technologie EOL Policy*[13] am 30. Oktober 2009 aus, und wenn bis dahin eine 64-Bit-Version von Java 1.6 für *MacOS* verfügbar ist, ist ein Umstieg geplant.

### 2.5.5 Unit-Tests

Es waren zwar Unit-Tests in Form eines separaten Projekts vorhanden, diese wurden aber schon sehr lange nicht mehr gepflegt. Es wurde beschlossen neue Tests innerhalb des *Saros*-Projekts anzulegen. Meine ersten Tests betrafen die Aktivitäten und deren Serialisierung (siehe Abschnitt 2.4). Bei diesen Tests hauptsächlich "round trips" getestet, also ob sich aus den serialisierten Aktivitäten wieder äquivalente Objekte erstellen lassen.

Wenn man Klassen testen möchte, welche etwas komplexer als die Aktivitäten sind und Exemplare von anderen Klassen benötigen um zu funktionieren, braucht man *Mock-Objekte*. Der *ActivitySequencer* interagiert zum Beispiel mit einem *SharedProject* und dem *XMPPChatTransmitter*. Ausserdem wurde intern ein *ConcurrentDocumentManager*-Exemplar erstellt und verwendet.

Im Projekt war bereits die Bibliothek *EasyMock* für das Erstellen von Mock-Objekten integriert. Nachdem ich mir mittels eines Tutorials über *JUnit* in Verbindung mit *EasyMock* ([Min]) einen kleinen Überblick über die Verwendung verschafft hatte, stellte ich fest, dass *EasyMock* nur Interfaces "mocken" kann. Da *Saros* viele Singletons besitzt, hätte man also für "reines" *EasyMock* eine Menge, sonst unnötiger, Interfaces schreiben müssen. Das *EasyMock*-Projekt bietet zwar eine *EasyMock Class Extension* mit der es möglich ist auch Klassen zu "mocken", aber deren aktuelle Versionen setzen *JUnit 4* voraus, während *Saros* noch *JUnit 3* verwendet. Daraufhin habe ich als Alternative *Mockito*[14] angeschaut, das auch grundsätzlich einen netten Eindruck hinterlassen hat. Allerdings reagieren dessen Mock-Objekte auf unvorhergesehene Methodenaufrufe einfach mit der Rückgabe eines Standardwertes, der zur Signatur der aufgerufenen Methode passt, statt eine Ausnahme auszulösen. Man bekommt also unter Umständen solche Aufrufe gar nicht mit. Daraufhin habe ich die letzte *EasyMock*-Version plus *Class Extension* installiert, die noch mit *JUnit 3* zusammenarbeitet. Das war zwar auch die allererste *EasyMock Class Extension* und sie trägt den Zusatz "prerelease" in der Versionsangabe, aber bisher ist das noch nicht negativ aufgefallen.

[13] http://java.sun.com/products/archive/eol.policy.html
[14] http://mockito.googlecode.com/

Es scheint bei Mocking-Bibliotheken für Java nicht üblich zu sein, oder vielleicht auch nicht so ohne weiteres möglich, dass man beliebige Typen durch Mock-Objekte ersetzen kann, so das ein **new** SomeClass() kein Exemplar von *SomeClass*, sondern ein vorher festgelegtes Mock-Objekt ist. Deshalb ist es schwer isolierte Tests für Klassen zu schreiben, die intern komplexere Objekte erzeugen und verwenden, weil man Diese dann immer mit testet. Beim *ActivitySequencer* betraf das den *ConcurrentDocumentManager*, der jetzt nicht mehr intern erzeugt, sondern über eine "Setter"-Methode von aussen zugeführt wird. Grundsätzlich hat man aber ein Problem mit Singletons, die bei *Saros* recht zahlreich verwendet werden.

Unit-Tests im gleichen Projekt in *Eclipse* zu verwalten, hat sowohl Vor- als auch Nachteile. Ein Vorteil ist, dass es nicht mehr so einfach passieren kann, dass die Tests nicht mehr zu dem zu testenden Quelltext "passen", weil zum Beispiel bei Refaktorisierungen, wie umbenennen von Klassen oder Methoden, die Änderungen bei den Tests auch gleich vorgenommen werden. Man sieht auch schnell an den Fehler-Anzeigen von *Eclipse*, wenn man etwas verändert hat, was dazu führt, dass ein Test nicht mehr kompilierbar ist. Das die Klassen und Methoden der Tests als Treffer bei den verschiedenen Suchmöglichkeiten von *Eclipse* auftauchen und damit das Ergebniss "verwässern", empfinde ich dagegen als Nachteil. Beim suchen in der Klassen- oder Aufrufhierarchie, sind in der Regel nur die Treffer innerhalb des Produktivcodes von Interesse.

# 3 Nachgedanken

## 3.1 Fazit

Der Einarbeitung durch einen Vorgänger ist sicher eine gute Sache. Das Projekt hat doch schon eine Grösse und Komplexität erreicht, dass ich alleine wesentlich mehr Zeit bei der Orientierung im Quelltext benötigt hätte. Ein Dank an Christoph Jacob an dieser Stelle.

Bei dieser Arbeit habe ich einige neue Techniken kennengelernt, unter anderem Teile der *Eclipse*-Plugin-Architektur, die nützlichen Bibliotheken der Apache-Commons, *PicoContainer* als praktischer Aufbewahrungsort für Singletons[1], XML-Bibliotheken wie *JDOM* und *XStream*, und mit *EasyMock* und *Mockito* zwei Vertreter von "mocking"-Bibliotheken.

*SourceForge* musste ich in der täglichen Arbeit kennenlernen und muss sagen, dass es teilweise nicht besonders angenehm ist. Operationen auf dem Subversion-Repository sind unglaublich langsam. Da ich für die Ausarbeitung viel mit "Diffs" zwischen Revisionen gearbeitet habe, um noch einmal im Detail nachzuvollziehen, was für Änderungen das jeweils waren, habe ich mir lokal eine Kopie des Repositories anlegen müssen. Beim Hochladen von Dateien auf den Webspace oder für Releases war die Übertragung zu den *SourceForge* Servern nicht nur unglaublich langsam, sondern es gab auch des öfteren Zeitüberschreitungen und Verbindungsabrüche. Auch sehr negativ aufgefallen ist die Suche im Bugtracker, die nicht auf den "live"-Daten basiert. So findet man aktuelle neue oder geschlossene Einträge nicht über die Suche, oder die Treffer sind nicht mehr aktuell, werden zum Beispiel noch als offen angezeigt, obwohl sie schon geschlossen sind. Das ist besonders unangenehm beim Vor- und Nachbereiten von Testsitzungen aufgefallen.

Das Testdokument ist unpraktisch. Eine Datei im Word-Format, die sich auch nur in Word bearbeiten lässt, und selbst dort recht fragil ist.

[1] Und wenn man der Beschreibung auf der Webseite des Projekts glauben schenkt, ist das sowieso eine Wunderdroge, die auf magische Weise sämtliche Probleme mit Komplexität löst. ;-)

## 3.2 Ausblick

Eine universell einsetzbare Lösung für den Dateiabgleich zwischen Host und anderen Teilnehmern wäre praktisch, so dass man für den Einladungsprozess und die Synchronisation, die von der Konsistenzsicherung (*ConsistencyWatchdog*) im Falle von Inkonsistenzen angestossen wird, gemeinsamen Code verwenden kann. So eine Art *rsync*[2] für Java.

Neben dem im letzten Absatz von Abschnitt 2.2.5 auf Seite 16 besprochenen horizontalen Verfolgen des Darstellungsfeldes, beziehungsweise der Schreibmarke, verschwindet Letztere beim Beobachter zum Beispiel auch beim Code-Folding. Beispielsweise in eingeplappten Blöcken mit Import-Anweisungen, auch wenn der Beobachtete diesen Block bei sich im Editor ausgeklappt hat. Dieses Verhalten sollte man korrigieren.

*Saros* geht bisher immer davon aus, dass die Pfadangabe einer Ressource den Editor eindeutig identifiziert. Es gibt aber Ressourcentypen, für die es mehrere geöffnete Editoren, auch gleichzeitig, geben kann. Da wären XML- oder HTML-Editoren zu nennen, die neben dem Quelltexteditor noch einen Struktureditor oder einen WYSIWYG-Editor bieten. Dieser Editortyp muss mit in die übertragenen Informationen einfliessen, damit der Empfänger zuverlässig den richtigen Editor auswählen kann.

Benutzer werden als *User*-Exemplare modelliert und werden sowohl über *JID*-Exemplare als auch über Zeichenketten identifiziert, die beide letztendlich Jabber-IDs repräsentieren. Es wäre zu prüfen, in wie weit man Zeichenketten vermeiden, und möglichst konsequent auf *JID*s setzen kann.

Bei der Verwaltung der Warteschlangen und Zeitstempeln im *ActivitySequencer* wird noch ein nicht mehr verwendeter Verlauf von versendeten Nachrichten mitgeführt. Siehe die Beschreibung zur *getHistory()*-Methode auf Seite 21. Das sollte man endlich einmal entsorgen (dürfen). Diese Aufzeichnungen der Aktivitäten war einmal für den Fall gedacht, dass jemand die Verbindung verliert und etwas später wieder aufnehmen kann, und die versäumten Aktivitäten von den anderen Teilnehmern erneut anfordert. Ein Abgleich der Dateien, vielleicht über die Konsistenzsicherung, wäre vom Prinzip her einfacher und würde die Haltung des Verlaufs überflüssig machen. Zudem werden bei den Dateiaktivitäten die Datei*inhalte* nicht im Verlauf gespeichert. Die liessen sich also sowieso nicht erneut übertragen.

Wenn mehr Unit-Tests geschrieben werden, ergeben sich wahrscheinlich auch weitere Probleme mit Code, der nicht mit leichter Testbarkeit im Hinterkopf geschrieben wurde. Hier wäre ausserdem zu überdenken, ob man nicht auf *Junit 4* umsteigen sollte – auch vor dem Hintergrund, dass die aktuellen Versionen der Erweiterung für das "mocken" von Klassen für *EasyMock* diese Abhängigkeit haben. Siehe auch Abschnitt 2.5.5 auf Seite 35.

[2] http://rsync.samba.org/

Die Dokumentation in Form von Javadoc könnte verbesstert werden, insbesondere fehlen Informationen über das grössere Bild – das Zusammenspiel der Klassen. Dokumentation auf Package-Ebene ist überhaupt nicht vorhanden.

*Eclipse* kann von Haus aus mit seiner statischen Codeanalyse einige problematische Code-Konstellationen aufzeigen und auch Hinweise auf schlechten Programmierstil geben. Während meiner Arbeit hatte ich zusätzlich einmal *FindBugs*[3] über das Projekt laufen lassen und damit Probleme identifizieren können, die von Eclipse nicht erkannt wurden. Es wäre nützlich zu prüfen, welche Werkzeuge zur statischen Codeanalyse helfen könnten die Code-Qualität von *Saros* zu verbessern. Möglichst Werkzeuge, die sich einfach in Eclipse integrieren lassen. Auch Erweiterbarkeit von Tests wäre ein Kriterium, so dass man von den üblichen Empfehlungen abweichende Richtlinien oder Projektspezifisches automatisch überprüfen lassen kann. Beispielsweise unsere **private**/**protected**-Richtlinie, oder auch strukturelles, wie das Rücksetzen des "interrupt flag" bei der Behandlung von *InterruptedException*.

Ferner sollte man die alte und neue "Konkurrenz" im Auge behalten. Tut sich bei den alten, "toten" Projekten wieder etwas? Was gibt es Neues, wie zum Beispiel *Bespin* von den *MozillaLabs*[4]? Ist das Vergleichbar? Wo liegen die Unterschiede in der Zielsetzung? Kann man etwas von denen lernen?

Last but not least könnte man sich die Anpassung von *Eclipse* an andere Programmiersprachen anschauen, zum Beispiel die Plugins für *PHP*[5] (*PDT*[6]), *Python*[7] (*PyDev*[8]), oder *Scala*[9], und überprüfen wieviel Aufwand man betreiben muss, damit *Saros* auch mit deren Editoren zusammenarbeitet. So könnte man den Kreis potentieller Nutzer vergrössern. An dem Plugin für *C*- und *C++*-Entwicklung (*CDT*[10]) konnte man ja schon sehen, dass man vielleicht für jeden weiteren Editortyp für andere Programmiersprachen eine Anpassung im *Saros*-Plugin vornehmen muss.

[3] http://findbugs.sourceforge.net/
[4] http://labs.mozilla.com/projects/bespin/
[5] http://www.php.net/
[6] http://www.eclipse.org/pdt/
[7] http://www.python.org/
[8] http://pydev.sourceforge.net/
[9] http://www.scala-lang.org/
[10] http://www.eclipse.org/cdt/

# Literaturverzeichnis


[BLFM05] BERNERS-LEE, T., R. FIELDING und L. MASINTER: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard), Januar 2005.

[Dau08] DAUM, BERTHOLD: *Java Entwicklung mit Eclipse 3.3*. dpunkt.verlag, Heidelberg, 5. aktualisierte und erweiterte Auflage, 2008.

[Dje06] DJEMILI, RIAD: *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung*. Diplomarbeit, Freie Universität Berlin, 2006.

[Gus07] GUSTAVS, BJÖRN: *Weiterentwicklung des Eclipse-Plug-Ins Saros zur Verteilten Paarprogrammierung*. http://www.inf.fu-berlin.de/inst/ag-se/theses/Gustavs_Saros_DPPII.pdf, Oktober 2007.

[Jac09] JACOB, CHRISTOPH: *Weiterentwicklung eines Werkzeuges zur verteilten, kollaborativen Softwareentwicklung*. Diplomarbeit, Freie Universität Berlin, 2009.

[Lam78] LAMPORT, LESLIE: *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM, 21(7):558–565, 1978.

[Min] MINELLA, MICHEAL: *Unit testing with JUnit and EasyMock*. http://www.michaelminella.com/testing/unit-testing-with-junit-and-easymock.html. Online, 2009-04-24.

[Rie08] RIEGER, OLIVER: *Weiterentwicklung einer Eclipse-Erweiterung für verteilte Paar-Programmierung im Hinblick auf Kollaboration und Kommunikation*. Diplomarbeit, Freie Universität Berlin, 2008.

[RSL99] RYAN, V., S. SELIGMAN und R. LEE: *Schema for Representing Java(tm) Objects in an LDAP Directory*. RFC 2713 (Informational), Oktober 1999.

[SA04] SAINT-ANDRE, P.: *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 3920 (Proposed Standard), Oktober 2004.