Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Verbesserung der Action Awareness im Open Source Plug-in Saros

Damla Durmaz
Matrikelnummer: 4197120
ddurmaz88@gmail.de

Gutachter: Prof. Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Elfriede Fehr
Betreuerin: Julia Schenk

Berlin, 24. Oktober 2014

0.1 Eidesstattliche Erklärung

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

24. Oktober 2014

Damla Durmaz

Für Ferhat, der mich immer unterstützt und zu mir gehalten hat.

Zusammenfassung

Action Awareness bedeutet das Verstehen der Aktivitäten anderer Nutzer in einer Kollaboration. Ist diese Kollaboration verteilt, so ist Action Awareness nicht mehr wegzudenken, da Nutzer nicht mehr mitbekommen würde, wer was gemacht hat. Insgesamt kann also Action Awareness die Qualität der Kollaboration beeinflussen. Saros, ein Eclipse-Plugin zur verteilten, kollaborativen Programmierung, bietet bereits Action Awareness Informationen, wie wer schreibt gerade was und wo. Allerdings fehlen jegliche Informationen über die Interaktion mit der IDE. Ziel dieser Arbeit ist, die Action Awareness zu verbessern. Dazu wurden Aufzeichnungen von Saros-Sitzungen mithilfe von Techniken der Grounded Theory Methodology untersucht, um Stellen zu finden, die zeigen, dass mehr Action Awareness zu einer verbesserten Arbeit geführt hätten. Auf Grundlage dieser Beobachtungen wurde ein Konzept für einen ersten Prototypen entwickelt, welches anschließend implementiert und in Nutzertests evaluiert wurde. Mithilfe des Feedbacks wurde der Prototyp erweitert und nach Überarbeitung der Darstellung wurde ein zweiter Prototyp entwickelt. Das Ergebnis ist eine neue Activity Log, ein neuer Reiter in der Saros-View, welches alle bisherigen und die aktuelle Aktivität der Sitzungsteilnehmer anzeigt. Zudem wurde eine neue Implementierungsweise im Rahmen dieser Arbeit ausprobiert, bei dem der Inhalt der neuen Activity Log nicht nur mit reiner GUI-Programmierung, sondern auch mit einem HTML-basierten Ansatz umgesetzt wurde. Auf diese Weise ist es möglich, beide Vorgehensweisen zu vergleichen und für zukünftige Entwicklungen im Bereich der Unterstützung weiterer Entwicklungsumgebungen von Saros einen ersten Prototypen zu erhalten.

Inhaltsverzeichnis

0.1 Eidesstattliche Erklärung i

1 Einleitung **1**
1.1 Motivation 1
1.2 Problemstellung 3
1.3 Ziel der Arbeit 4
1.4 Verwandte Arbeiten und Abgrenzung 4
1.5 Aufbau dieser Arbeit 6

2 Grundlagen **8**
2.1 Saros 8
2.2 Awareness 10
2.2.1 Situation Awareness 10
2.2.2 Workspace Awareness 11
2.2.3 Awareness in Saros 15
2.2.4 Action Awareness 17
2.3 Grounded Theory 19
2.3.1 Vorgehensweise bei der Grounded Theory 20
2.3.2 Methoden zur Analyse 20

3 Qualitative Analyse der Videos **22**
3.1 Vorhandenes Datenmaterial 22
3.2 Vorgehen bei der Analyse 23
3.3 Erarbeitete Kategorien 25
3.4 Probleme beim Konzeptentwurf 27
3.5 Ergebnis der qualitativen Analyse 31

4 Implementierung des ersten Prototyps **32**
4.1 Technische Realisierbarkeit 32
4.1.1 Listener 32
4.1.2 Extensions 36
4.2 Konzept für den ersten Prototyp 38
4.3 Skalierbarkeit 40
4.3.1 Dialoge und Views 40
4.3.2 Erzeugungsdialoge 40
4.3.3 Suchdialoge 40
4.3.4 JUnit Testläufe 40
4.3.5 Refactorings 40
4.4 Erste Version des Prototyps 42
4.4.1 Anzeigeorte 42
4.4.2 Dialoge und Views 45
4.4.3 Erzeugungsdialoge 48
4.4.4 Suchdialog 50

4.4.5 Refactorings . . . 53
4.4.6 JUnit Testläufe . . . 55
4.5 Zweite Version des Prototyps . . . 56
4.5.1 Dialoge und Views . . . 56
4.5.2 JUnit Testläufe . . . 57
4.5.3 Refactorings . . . 57
4.5.4 Erzeugte Datei . . . 59

5 Evaluation des ersten Prototyps 61
5.1 Problemstellung . . . 61
5.2 Leitfragen . . . 62
5.3 Aufgabenstellung . . . 63
5.3.1 Definition des Ziels . . . 63
5.3.2 Benötigte Informationen . . . 63
5.3.3 Vorgehen . . . 64
5.3.4 Entwicklung der Aufgabenstellung . . . 65
5.4 Interviewleitfaden . . . 69
5.5 Ablauf des Nutzertests . . . 70
5.6 Auswertung der Nutzertests . . . 71
5.6.1 Vorgehen . . . 72
5.6.2 Beobachtung . . . 72
5.6.3 Kategorisierung . . . 73
5.6.4 Auswertung . . . 76

6 Implementierung des zweiten Prototyps 78
6.1 Technische Realisierbarkeit . . . 78
6.1.1 Dialoge und Views . . . 78
6.1.2 JUnit Testläufe . . . 79
6.1.3 Refactorings . . . 82
6.1.4 Erzeugte Dateien . . . 83
6.1.5 Visuelle Hilfsmittel . . . 83
6.1.6 Funktionen des zweiten Prototyps . . . 84
6.2 Gründe für einen neuen Reiter: Activity Log . . . 85
6.3 HTML-basierte Implementierung . . . 89
6.3.1 Vorgehensweise . . . 90
6.3.2 Erster Versuch ohne Plug-in Anbindung . . . 92
6.4 Implementierung mithilfe von SWT . . . 94
6.4.1 Technische Umsetzung . . . 94
6.4.2 Funktionen . . . 96
6.5 HTML-basierte Implementierung mit Plug-in . . . 98
6.5.1 Versuch unter Windows . . . 98
6.5.2 Versuch unter Mac OSX . . . 100
6.6 Vergleich beider Ansätze . . . 108
6.6.1 Erzeugung der Gesamtstruktur . . . 109

6.6.2 Gestaltung des Browser-Inhaltes 109
6.6.3 Aufwand bei Änderungen 110
6.6.4 Testen 110
6.6.5 Komplexität der Umsetzung und Threadhandling . . . 110
6.6.6 Fazit 111

7 Mitarbeit im Saros-Projekt 112
7.1 Entwicklungsprozess im Saros-Projekt 112
7.2 Reviewprozess 113
7.2.1 Reviews von eigenen Änderungen 113
7.2.2 Reviews von fremden Änderungen 114
7.2.3 Hürden und Gewinn 114
7.3 Zeitlicher Verlauf 116
7.3.1 Erster Monat 116
7.3.2 Zweiter Monat 119
7.3.3 Dritter Monat 120
7.3.4 Vierter Monat 121
7.3.5 Fünfter Monat 121
7.3.6 Sechster Monat 122
7.3.7 Rückblick 123

8 Fazit und Ausblick 124

A Aufgabenstellung für die ersten Nutzertests 126

B Interviewleitfaden 128

C Einladung zum Nutzertest 132

D Notierte Beobachtungen 133

E Entwickelte Kategorien 134

1 Einleitung

Diese Arbeit beschäftigt sich mit der Verbesserung der Action Awareness in Saros. Action Awareness beschreibt dabei das Verstehen der Aktivitäten und Absichten anderer in einem gemeinsamen Arbeitsbereich während der kollaborativen Arbeit. Dieses Kapitel dient der Einführung in diese Arbeit. Es wird beschrieben, welche Motivation hinter dieser Arbeit steckt. Diese Motivation dient dazu, im darauffolgenden Kapitel besser erläutern zu können, welches Problem mit dieser Arbeit gelöst werden soll. Anschließend wird erklärt, was das Ziel dieser Arbeit ist, worin sich diese von anderen Arbeiten abgrenzt und zum Schluss wird erläutert, wie diese aufgebaut ist.

1.1 Motivation

Bei der klassischen Paarprogrammierung bearbeiten zwei Programmierer nebeneinander an einem Computer denselben Code [WKCJ00]. Der eine Programmierer schreibt den Code (**Driver**), während der andere Teilnehmer zuschaut, sich Gedanken über das Problem macht, den Quellcode kontrolliert und über Fehler spricht, falls diesem welche auffallen (**Observer**) [SSB+02]. Diese Rollenverteilung ist jedoch nicht fest. Zum einen macht nach Stotts et al. in [SSB+02] eine effektive Paarprogrammierung aus, dass diese Rollen in periodischen Zeitabständen wechseln. Zum anderen herrscht diese Rollenverteilung nur beim Programmieren selbst. Denn Chong et al. haben in [CH07] beschrieben, dass außerhalb des Tippens die beiden Programmierer auf demselben Abstraktionsgrad miteinander diskutieren und Entscheidungen treffen, wobei auch der Faktor Expertise eine wichtige Rolle spielt.

Nichtsdestotrotz nehmen beide Programmierer die Aktivitäten des jeweils anderen unterbewusst wahr und die eigenen Eindrücke, Gedanken und Aktionen hängen von denen des anderen ab [Sch12]:

- Wo im Quellcode befindet sich die Person?
- Was macht diese Person an der entsprechenden Stelle?
- Was interessiert die Person gerade?
- Was sind die Absichten?

Die Aktionen des Beobachters sind abhängig von diesen Informationen. Dieses Wissen über die Aktionen und den Status anderer Personen in der Gruppenarbeit wird auch als*Awareness* bezeichnet und ist hrundlegend für die eigenen Aktionen [GGGG95]. Es handelt sich dabei *um auf die Minute aktuelles Wissen um die Aktivitäten anderer Menschen in der Gruppe, das vom einzelnen benötigt wird, um sich mit diesen zu koordinieren und die eigenen Aufgaben innerhalb der Gruppe zu erfüllen* und ist wichtig für eine effektive und effiziente Zusammenarbeit [GGR96a].

Bei der kollaborativen Arbeit ist eine spezielle Form von Awareness von Bedeutung, und zwar *Workspace Awareness*. Das gemeinsam genutzte Medium ist der Arbeitsbereich und die Awareness Informationen sind hier die Informationen über die Anwesenheit, die Aktivitäten, die Absichten der Nutzer sowie die Änderungen an den Artefakten in diesem gemeinsamen Arbeitsbereich [Gut96].

Wie wichtig Awareness Informationen sind, soll folgendes Beispiel verdeutlichen: Zwei Studenten möchten gemeinsam ein Plakat für eine Präsentation vorbereiten. Dabei teilen sie die Arbeit auf, wobei jeder von ihnen eine Hälfte des Plakats bearbeitet. Während jeder Student an seiner Hälfte arbeitet, schaut jeder von ihnen gelegentlich über die Schulter zum anderen Studenten und kann dabei folgende Dinge in Erfahrung bringen:

- Was macht der andere gerade?
- Wo auf dem Plakat ist er gerade? Wo darf der beobachtende Student also nicht schreiben?
- Was könnte er gleich machen? Braucht er gleich eine Schere oder den Stift, den der Beobachtende gerade nutzt?
- Hat der beobachtende Student Anmerkungen zu der Arbeit des anderen?

Wenn ein Student weiß, wo auf dem Plakat sich der andere befindet, kann er z.B. nachfragen, wie der andere seinen Teil gestalten möchte oder wie weit dieser mit seiner Arbeit ist. Evtl. hat der Student eine Idee, wie man Elemente anordnen könnte. Würden die Studenten nichts von dem wissen, würden sie nicht wissen, wie weit der andere ist und könnten ihre Arbeit nicht so gut koordinieren und aufeinander abstimmen. Es könnte sogar so weit kommen, dass sie am Ende ihrer Arbeit die beiden Hälften des Plakats zusammenlegen und feststellen, dass diese gar nicht zusammenpassen. Würden die Studenten Awareness Informationen erhalten, die jedoch unpräzise sind, also zu grob oder schwer verständlich sind, würde das auch wenig weiterhelfen. Wenn sie sich z.B. gegenseitig nicht sehen, jedoch erfahren, wenn der andere mit einer neuen Spalte beginnt, können sie daraus nicht schlussfolgern, was sie jetzt schreiben werden, wo die neue Spalte auf dem Plakat angeordnet werden soll usw. Das Beispiel zeigt insofern, dass es nicht nur wichtig ist, Awareness Informationen zu erhalten, sondern wie diese Informationen gestaltet oder präsentiert werden.

Bei der verteilten Zusammenarbeit sind die Awareness Informationen auf die Software beschränkt, die für die Kollaboration genutzt wird, und können nicht mehr über die natürlichen Mechanismen wie z. B. Sehen oder Hören wahrgenommen werden. Das kann die Qualität der Kollaboration beeinflussen [GG96]. Wenn keine Workspace Awareness Mechanismen zur Verfügung stehen würden, würden die Teilnehmer nicht mehr wissen, woran die anderen

arbeiten, worauf sie sich konzentrieren und was ihre Absichten im Arbeitsbereich sind [GGR96b]. Demnach muss also Software, die für die verteilte kollaborative Arbeit eingesetzt wird, Mechanismen zur Verfügung stellen, die Workspace Awareness Informationen in einer angemessenen Form übermitteln kann, da sonst die verteilte Kollaboration kaum oder gar nicht möglich wäre. Gutwin et al. haben z. B. in [Gut96] Werkzeuge für die Übertragung von Workspace Awareness Informationen vorgestellt und konnten nachweisen, dass diese nicht nur konsumiert wurden, sondern dass die beobachteten Nutzer diese nicht mehr missen möchten. Eine in [DOS07] vorgestellte Umfrage bestätigt zudem, dass verteilt arbeitende Entwickler Informationen über die aktuellen Aktivitäten anderer im gemeinsamen Arbeitsbereich als sehr wichtig empfinden.

1.2 Problemstellung

Das letzte Kapitel hat beschrieben, was bei der Entwicklung von Software für die verteilte Kollaboration essentiell ist. Sie muss stets mit der Motivation entwickelt oder weiterentwickelt werden, den Nutzern Workspace Awareness Informationen bereitzustellen. Denn nur mit diesen Informationen können sie sich im gemeinsamen Arbeitsbereich besser zurecht finden, Aufgaben mit den Teilnehmern koordinieren und insgesamt besser verteilt arbeiten.

Saros ist solch eine Software. Es handelt sich dabei um ein für die Entwicklungsumgebung Eclipse entwickeltes Open-Source-Plugin für die verteilte, kollaborative Programmierung [SOBS10] und überträgt Awareness Informationen aus dem Code-Editor (wer tippt gerade was und wo). Diese Art von Workspace Awareness wird auch als „Action Awareness“ bezeichnet [GG02] und gibt Informationen über die Aktivitäten und Absichten der anderen Teammitglieder wieder. Action Awareness deckt Fragen ab, wie „Was passiert?“, „Welche Aktionen werden von wem ausgeführt?“ und auch „Was ist bisher passiert“ [**?**]. Saros überträgt jedoch keine Informationen über die Aktivitäten, die in anderen Anwendungen, anderen Eclipse Views als dem Editor oder Dialogen (Suche, Refactoring etc.) stattfinden. Dabei sind gerade solche Informationen wichtig, um nachvollziehen zu können was der entfernte Entwickler gerade außer dem Editieren tut oder aber auch was er bisher getan hat.

Action Awareness kann auf unterschiedlichen Ebenen bestehen, entweder ganz allgemein „Der andere ist in einer anderen Anwendung“ oder sehr detailliert „Der andere macht gerade ein Refactoring in Datei X in Zeile Y“. Je nach Situation kann diese Information relevant sein, wenn z.B. ein Programmierer bei der verteilten Paarprogrammierung nicht weiß, dass der Partner gerade bereits den Namen der Klasse refactort und dies zur selben Zeit ebenfalls durchführt, sodass es zu Konflikten kommen könnte.

1.3 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Erarbeitung und Umsetzung eines Konzepts zur Verbesserung der Action Awareness in Saros. Der Fokus liegt hierbei nicht auf dem Mechanismus zur Übertragung und Darstellung, sondern auf der Art der Action Awareness Informationen an sich. Als Grundlage für die Erarbeitung des Konzepts werden Aufzeichnungen analysiert, in denen mehrere Saros-Sitzungen zwischen zwei Programmierern aufgezeichnet wurden, die an unterschiedlichen Standorten arbeiten. Durch diese Analyse soll herausgefunden werden, an welchen Stellen in Saros Bedarf für mehr Action Awareness vorhanden ist und das Ergebnis soll ein Konzept sein, welches implementiert und mithilfe von Nutzertests evaluiert wird. Das Feedback der Nutzertests soll genutzt werden, um zu sehen, ob die Action Awareness Informationen wahrgenommen werden, ob sich insgesamt die Action Awareness verbessert hat und was man noch verbessern könnte.

1.4 Verwandte Arbeiten und Abgrenzung

In der Literatur sind einige Mechanismen verbreitet, die Action Awareness unterstützen. Gutwin und Greenberg haben in [GG02] und Schümmer und Lukosch in [SL07] einige vorgestellt:

- **Indikator für Aktivitäten und Änderungen**: Hier werden auf der Benutzeroberfläche Symbole als Indikatoren für Aktivitäten oder Änderungen genutzt. Ackerman et al. haben nämlich in [AS95] erklärt, dass Menschen generell wissen möchten was um sie herum geschieht. Sie wollen nicht mal immer wissen, was genau der- oder diejenigen gemacht hat. Es reiche aus, dass sie wissen, dass generell gerade eine Aktivität stattfindet. Solch ein Indikator kann z. B. ein Zahlenwert sein, um anzugeben, wie oft eine Datei verändert wurde.

- **Telepointer**: In der natürlichen Kommunikation wird der Mauszeiger verwendet, um als Ersatz für den Finger zu dienen. Oft wird der Zeiger auch für deiktische Ausdrücke („das da“, „hier“, „dort“) verwendet und sagt aus, wo sich der Akteur befindet und auch was er tut [Gre91].

- **Entfernter Mauszeiger**: Im Gegensatz zum Telepointer ist ein entfernter Mauszeiger mit dem Eingabegerät des Nutzers verbunden [Sch07]. Mit dem entfernten Mauszeiger kann man Artefakte im entfernten Arbeitsbereich verschieben und verändern, während der Telepointer lediglich dazu dient, auf einen Punkt im Arbeitsbereich zu zeigen. Er kann aber auch zusätzlich zum eigenen Mauszeiger stets präsent auf der Arbeitsfläche angezeigt werden, um neben dem entfernten Arbeiten auch einfach als Indikator für Aktivitäten präsent zu sein [HG04]. Solch ein Indikator ist nicht gleich ein Indikator, da meistens

Telepointer nicht immer präsent sind, sondern erst aktiviert werden müssen. Allerdings wird in der Literatur solch ein Mauszeiger, der also nur präsent und nicht sofort mit dem Eingabemedium verbunden ist, oft auch als Telepointer bezeichnet. Nichtsdestotrotz wird in [Sch07] der Telepointer und der entfernte Mauszeiger voneinander unterschieden.

- **Sichtbarkeit der Aktivitäten**: Hier werden die Aktivitäten der anderen Teilnehmer sichtbar gemacht, indem z.B. um den Aufenthaltsort herum ein kleines Sichtfeld angezeigt wird, sodass man sehen kann, wo diese Person im Arbeitsbereich ist. Daraus kann schlussgefolgert werden, was diese Person gerade tut. Eine andere Idee ist, die Aktivität in Form von Text mitzuteilen. Aus früheren Messengern oder heutigen Messenger-Apps kennt man, dass Informationen wie „... schreibt gerade“ angezeigt werden, um zu verdeutlichen, was der andere gerade tut.

- **Radar Views**: Eine andere Möglichkeit, um Aktivitäten sichtbar zu machen, sind die sogenannten Radar Views, bei denen der gesamte Arbeitsbereich der anderen oder ein Ausschnitt davon in einer kleinen Miniatur angezeigt wird, die wiederum selbst Awareness Informationen enthalten kann [Gut96]. Diese können Wissen über die Artefakte, die die anderen nutzen, geben, aber auch die Aktionen mit diesen Artefakten und die Absichten, die mit der Arbeit an diesen Artefakten verfolgt werden.

Alle oben genannten Mechanismen sind Ideen, Action Awareness Informationen zu übermitteln. Allerdings beschäftigen sich diese verwandten Arbeiten vor allem mit der Darstellung der Action Awareness Informationen. Sie überlegen sich, wie die Informationen am besten übertragen werden sollten bzw. wie sie beim Empfänger präsentiert werden und die Frage nach dem „Was“ steht nicht im Vordergrund. Diese Arbeit grenzt sich in der Hinsicht von den oben genannten und ähnlichen Arbeiten in der Literatur dahingehend ab, dass sie sich nicht nur damit beschäftigt, wie die Informationen am besten angezeigt werden sollen, sondern vor allem was übertragen werden soll. Wie Kapitel 1.3 bereits zusammengefasst hat, soll mithilfe der Videoanalyse ermittelt werden, welche Informationen für die Nutzer hilfreich wären. Auch wenn das Ergebnis der Arbeit ein neuer Reiter in der Saros-View ist, der eine Aktivitätenanzeige darstellt, so ist es nicht das Ziel gewesen, die bestmögliche Darstellung zu finden, sondern herauszufinden, was am besten übertragen werden sollte. Daher wurden zu Beginn der Arbeit bereits vorhandene Mechanismen benutzt, wie die Saros Sitzungsübersicht oder der Saros Chat. Außerdem sind obigen Mechanismen zwar interessante und teilweise bewährte Ansätze, aber sie sind nicht darauf angelegt, solche feingranularen Informationen über Aktionen zu übertragen, wie sie in Kapitel 1.2 genannt wurden, also Informationen über

was gerade refaktorisiert wurde, welcher Dialog offen ist, welche View gerade aktiviert wurde usw..

1.5 Aufbau dieser Arbeit

Im Folgenden soll beschrieben werden, wie diese Arbeit aufgebaut ist.

Nachdem im ersten Kapitel das grundlegende Problem, die Motivation und das Ziel dieser Arbeit beschrieben wurden, werden im zweiten Kapitel die Grundlagen erörtert. Dabei werden drei wichtige Bereiche abgedeckt: Zuerst wird das Tool Saros beschrieben. Anschließend werden die für diese Arbeit wichtige Begriffe, wie Awareness, Workspace Awareness und Action Awareness erörtert. Zum Schluss wird die Methodik vorgestellt, mit der die Sitzungsaufzeichnungen qualitativ analysiert wurden, um mögliche Problemfelder in Saros zu finden.

Das dritte Kapitel beschäftigt sich mit der qualitativen Datenanalyse an sich. Es wird kurz auf das Datenmaterial eingegangen und danach beschrieben, wie bei der Analyse vorgegangen wurde. Anschließend werden die gefundenen Problemkategorien beschrieben. Da nach der qualitativen Analyse einige Fragen im Raum standen, die nicht pauschal beantwortet werden konnten, war es nicht möglich, ein finales Konzept zu entwickeln, was ebenfalls in diesem Kapitel erläutert wird. Zum Schluss werden die Ergebnisse der qualitativen Analyse geschildert.

Das vierte Kapitel behandelt die Entwicklung und Implementierung des ersten Prototyps. Die Bezeichnung „erster“ rührt daher, dass in dieser Arbeit nach der Methodik der agilen Softwareentwicklung vorgegangen wurde und der Prototyp durch Einholen von Feedback durch Nutzertests in Form eines zweiten Prototyps verbessert werden sollte. In diesem Kapitel werden zu Beginn die Ergebnisse der Recherche zur technischen Realisierbarkeit dargestellt. Parallel zur Recherche fand die Implementierung der ersten Version des Prototyps statt, der jedoch aufgrund gewonnener Kenntnisse zur Besseren Umsetzung und auch aufgrund einiger technischen Schwierigkeiten verworfen wurde. Die Beschreibung dieser ersten Version wird im Anschluss an die Ergebnisse der Recherche gezeigt. Danach wird das Konzept für die zweite Version des Prototyps vorgestellt sowie auf die Skalierbarkeit des Konzepts eingegangen. Skalierbarkeit bedeutet hier, ob der entwickelte Prototyp nur in Saros bei der Entwicklung in Java oder aber auch in der Entwicklung mit anderen Programmiersprachen funktioniert. Zum Schluss des Kapitels wird die Umsetzung der zweiten Version, also der finalen Version des ersten Prototyps vorgestellt.

Das fünfte Kapitel beschreibt die Evaluation des ersten Prototyps in Form von Nutzertests. Die Problemstellung, also was mithilfe der Tests herausgefunden werden soll, wird zuerst erörtert. Anschließend werden die Leitfragen aufgezählt, die zur Gestaltung der Testaufgaben formuliert wurden. Danach wird vorgestellt, welche Aufgaben für die Nutzertests in Erwägung kamen

und welche letztendlich eingesetzt wurde. Danach folgt die Beschreibung des Interviewleitfadens, denn zusätzlich zu der Durchführung von Aufgaben wurde mit den Testnutzern Interviews durchgeführt, um Eindrücke, Wünsche und persönliches Feedback einzuholen. Zum Schluss wird geschildert, wie die Nutzertests abgelaufen sind und was letztendlich das Ergebnis war.

Das sechste Kapitel geht auf die Umsetzung des Feedbacks in Form eines zweiten Prototyps ein. Auch hier wird es wieder ein Kapitel zur Recherche der technischen Realisierbarkeit geben und anschließend wird erläutert, warum es nicht nur notwendig war, einen neuen Reiter in der Saros Chat-Ansicht zur Darstellung der neuen Action Awareness Informationen zu implementieren, sondern welche Vorteile das mit sich bringt. Außerdem wurde ein neuer Ansatz zur Darstellung des Inhaltes des Reiters getestet. Neben der Implementierung der Anzeige mithilfe der Java *SWT* Bibliothek wurde der Inhalt des Reiters mithilfe einer browser-basierten Vorgehensweise gestaltet. Dieser Versuch hat das Ziel zu sehen, ob dieser Ansatz prototypisch funktioniert und welche Vor- und Nachteile dieser Ansatz im Vergleich zur reinen GUI-Programmierung mittels SWT hat.

Im siebten Kapitel wird die Arbeit innerhalb des Saros-Projekts beschrieben. Das Ziel dieses Kapitels ist die Darstellung des Entwicklungs- und des Reviewprozesses in Saros. Dort wird auch darauf eingegangen, welche Hürden und welcher Gewinn dieser Prozess für eine Abschlussarbeit bedeuten. Im Anschluss wird der zeitlichen Verlauf dieser Arbeit geschildert.

Am Ende folgt ein Fazit dieser Arbeit. Dabei der aktuelle Stand beschrieben, inwieweit die Ziele der Arbeit umgesetzt wurden und in einem Ausblick erörtert, was in der Zukunft noch alles in diesem Bereich gemacht werden kann.

2 Grundlagen

In diesem Kapitel werden die Grundlagen vorgestellt, die für die Bearbeitung dieser Arbeit notwendig waren. Zunächst wird Saros in Kapitel 2.1 beschrieben, wobei neben einer kurzen Vorstellung die Einsatzszenarien vorgestellt werden und im Anschluss ein technischer Überblick gegeben wird. Anschließend werden in Kapitel 2.2 die Begriffe „Awareness“, „Workspace Awareness“ sowie „Action Awareness“näher beschrieben. Abschließend wird in Kapitel 2.3 die *Grounded Theory* Methodik vorgestellt, mit der die qualitative Analyse der Sitzungsaufzeichnungen durchgeführt wurde.

2.1 Saros

Saros[1] ist ein für die Entwicklungsumgebung Eclipse entwickeltes Open-Source-Plugin für die verteilte, kollaborative Programmierung, auch *verteilte Partyprogrammierung* (engl. *Distributed Party-Programming, DPP*) genannt. Verteilt bedeutet hierbei, dass die Bearbeitung des Codes in räumlich voneinander getrennten Orten stattfindet [SS00]. Die kollaborative Arbeit erfolgt dabei in Echtzeit in einem Eclipse-Editor und wird in einer sogenannten *Saros-Sitzung* gekapselt. Jeder Teilnehmer einer Saros-Sitzung besitzt dabei eine identische Kopie des verteilten Eclipse Projekts und kann in Echtzeit die Dateien des Projekts gemeinsam mit anderen Sitzungsteilnehmern bearbeiten. Die Änderungen eines Nutzers am Projekt werden lokal, also an der eigenen Kopie des Projekts, gespeichert und im Hintergrund kümmert sich Saros um die Synchronisation der Kopien, indem es jede Änderungsinformationen an alle anderen Teilnehmer der Saros-Sitzung verteilt werden. Ein Nutzer kann also sofort die Änderungen an einem Dokument durch einen anderen Sitzungsteilnehmer bei sich nachvollziehen und weiß auch, wer diese Änderungen vorgenommen hat.

Ursprünglich wurde Saros 2006 im Rahmen einer Diplomarbeit für die kollaborative Arbeit von zwei Programmierern entwickelt [Dje06], um die Arbeit bei der verteilten Paarprogrammierung (engl. *Distributed Pair-Programming*) zu erforschen [DOS07], aber das wurde allmählich ausgeweitet auf die allgemeine verteilte, kollaborative Programmierung und daher wurde Saros seitdem in der Arbeitsgruppe „Software Engineering“ der Freien Universität Berlin stets weiterentwickelt, sodass es mittlerweile in seiner aktuellsten Version bis zu fünf Teilnehmer in einer Sitzung verwalten kann.

Der Arbeitsablauf läuft bei jeder Saros-Sitzung gleichermaßen ab: Eine Person ist der Koordinator der Sitzung bzw. initiiert diese und wird damit als *Host* bezeichnet. Der Host sendet einem Teilnehmer eine Einladung zu einem gemeinsam zu bearbeitenden Projekt. Wenn der Teilnehmer diese Einladung annimmt, wird eine Sitzung zwischen dem Host und dem Teilnehmer aufgebaut. Dabei wird das gemeinsame Projekt auf beiden Rechnern synchronisiert,

[1] http://www.saros-project.org

sodass beide danach eine identische Kopie des Projekts besitzen. Nun können der Host und die Teilnehmer gemeinsam am Projekt arbeiten. Beendet der Host die Sitzung, wird diese für allen anderen Teilnehmer ebenfalls beendet. Der Host hat zudem Privilegien, wie die Verteilung von Rechten an die Teilnehmer. Standardmäßig haben alle Sitzungsteilnehmer Schreibrechte am gemeinsamen Projekt, doch der Host kann beliebigen Teilnehmern die Schreibrechte entziehen, sodass der entsprechende Nutzer Änderungen zwar sieht, jedoch selbst nichts editieren kann.

Nach [SOBS10] gibt es vier Szenarien bei der kollaborativen Arbeit, in denen sich der Einsatz von Saros eignet. Als Voraussetzungen für alle Szenarien ist das Vorhandensein einer Audioverbindung (z.B. über Telefon oder VoIP), da das die kollaborative Arbeit ungemein erleichtert.

Code Review Beim Code Review agiert eine Person als „Treiber“ und begutachtet den Quellcode, während andere „Beobachter“ diesen Prozess beobachten. Treiber und Beobachter können gemeinsam die Codequalität überprüfen oder nach Fehlern suchen. Auch wenn der Treiber den Prozess leitet, kann hier ein Wissensaustausch unter allen Teilnehmern der Sitzung stattfinden [SOBS10]. Der Treiber kann z. B. einen Fehler an einer Codestelle beheben, während die Beobachter das kontrollieren und ggf. Verbesserungen oder Anmerkungen ausdrücken. Hier eignet sich der *Follow Modus* von Saros, bei dem jede Aktion des Treibers, wie das Öffnen einer Datei oder die Ansicht einer bestimmten Textstelle, an den Beobachter übermittelt wird. So weiß der Beobachter immer, auf was oder wo der Treiber zur Zeit den Fokus legt. Wenn der gemeinsame Code Review beendet ist und die Teilnehmer unabhängig voneinander unterschiedliche Codestellen bearbeiten wollen, können sie den Follow-Modus einfach verlassen.

Einführung von Neulingen Das Szenario der Einführung eines Neulings in eine Codestelle oder ein Projekt wird durch Saros ebenfalls unterstützt. Die Rollenverteilung ist ähnlich der beim Code Review mit dem einzigen Unterschied, dass der Treiber nicht den Code begutachtet, sondern den Beobachter in den Quellcode einführt als eine Art Lehrer. Auch hier eignet sich der im vorigen Punkt beschriebene Follow-Modus besonders gut, da der Neuling nicht nur Erklärungen erhält, sondern auch die Erklärungen anhand des Quellcodes nachvollziehen kann.

Verteilte Paarprogrammierung Saros eignet sich gut, wenn zwei Entwickler Paarprogrammierung betreiben wollen, jedoch nicht am selben Ort sind. Durch die Synchronisation der Änderungen in Echtzeit und den Kontextinformationen, die Saros bietet, können beide entweder in den typischen Rollen eines Drivers und Observers (siehe Kapitel 1.1) arbeiten oder jeder arbeitet unabhängig voneinander an unterschiedli-

chen Stellen des Quellcodes. In diesem Szenario wird häufiger auch der Follow-Modus eingesetzt.

Verteilte, kollaborative Programmierung In diesem Szenario arbeiten mehr als zwei Entwickler am selben Projekt. Sie arbeiten kollaborativ entweder an der selben Codestelle oder unabhängig voneinander an unterschiedlichen Stellen und müssen meistens nicht unbedingt mitbekommen, was genau die anderen gerade tun. Sie wissen meistens lediglich durch die Rosterview von Saros, wer noch mitarbeitet. In diesem Szenario wird in den wenigsten Fällen der Follow-Modus eingesetzt.

2.2 Awareness

Bei der kollaborativen Arbeit führen die Teilnehmer nicht nur ihre Aufgaben aus, sondern beobachten auch die Aktivitäten der anderen Teilnehmer. Sie schauen darauf, worauf sich die anderen konzentrieren, was sie gerade tun und auf was sich ihre Aufmerksamkeit richtet [Sch12]. Nach diesen Informationen können sich auch eigene Aktivitäten richten. Dieses kontextsensitive Wissen über die Aktionen und den Status anderer Personen in der Gruppenarbeit wird auch als *Awareness* bezeichnet. Gutwin et al. definieren es in [GGGG95], was übersetzt Folgendes bedeutet: „Es handelt sich um auf die Minute aktuelles Wissen um die Aktivitäten anderer Menschen, das vom einzelnen benötigt wird, um sich mit diesen zu koordinieren und die eigenen Aufgaben innerhalb der Gruppe zu erfüllen“. Nach Dourish et al. sind das Informationen, durch die die Aktivitäten der anderen verstanden werden und die Grundlage für eigene Aktivitäten bilden [DB92]. Gutwin et al. fasst es in [GG02] zusammen: Wissen, was um einen herum passiert.

Awareness Informationen können die Qualität der kollaborativen Arbeit beeinflussen [GGR96a]. Gutwin et al. haben in [GG98] ein Experiment durchgeführt, bei dem Testnutzer drei Aufgaben lösen mussten und zwei Aufgaben, bei denen Awareness Informationen angeboten wurden, konnten schneller gelöst werden als die, wo keine Awareness Informationen angeboten wurden.

Wenn sich die Teilnehmer der Kollaboration am selben Ort befinden, werden diese Informationen als selbstverständlich angenommen bzw. wahrgenommen. Findet diese kollaborative Arbeit jedoch an räumlich voneinander getrennten Orten statt, kann das die Qualität der Kollaboration beeinflussen, denn die Teilnehmer wissen nicht mehr, wer was, wo und warum gemacht hat [GG96] [GGR96b].

2.2.1 Situation Awareness

Awareness spielt in vielen Gebieten eine wichtige Rolle. Ursprünglich wurden die ersten Forschungen im Bereich Awareness bei der Analyse vom Flugverhalten von Piloten im Militär durchgeführt [GG02]. Ein Kampfpilot muss im Luftraum eine Menge an Informationen über seine Umgebung und seine

Abbildung 1: Situation Awareness ist nicht nur das auf die Minute aktuelle Wissen über das Geschehen um uns herum, sondern ein Prozess.

Kameraden aufnehmen und verarbeiten. Die Umgebung ist komplex, kann sich schnell ändern und die Aktionen der Kameraden sind aufgrund des Sichtwinkels und der schnellen Bewegungen nicht ohne Probleme ersichtlich. Die Autoren nennen in [GG02] noch weitere Bereiche, in denen Awareness eine wichtige Rolle spielt, wie die Anästhesiologie, bei der Koordination von Flügen oder bei der Kontrolle des Flugverkehrs. All diese Bereiche haben gemeinsam, dass die Informationen, die aus der Umgebung (sei es physikalische Umgebung oder den Menschen in der Umgebung) aufgenommen werden können, komplex sind, sich schnell ändern können und die Menge an Informationen sehr hoch sein kann.

Diese Art der Awareness wird in der Literatur auch als *Situation Awareness* bezeichnet und ist die auf die Minute aktuelle Kenntnis, die erforderlich ist, um zu wissen, was gerade passiert und was zu tun ist [ATP95]. Endsley et al. beschreiben in [End95] Situation Awareness eher als Prozess. Abbildung 1 beschreibt, wie dieser Prozess zu verstehen ist: Zuerst werden die Informationen aus der Umgebung wahrgenommen. Dabei werden die relevanten Informationen für die Aufgaben und von den irrelevanten Informationen selektiert. Die aufgenommenen Informationen werden verstanden. Anschließend wird anhand der Änderungen in der Umgebung der Zustand der Informationen in der nahen Zukunft vorhergesagt.

2.2.2 Workspace Awareness

Definition

In der kollaborativen Arbeit ist das gemeinsam genutzte Medium der Arbeitsbereich. Dieser wird von allen Teilnehmern geteilt und stets mit den Artefakten, die sich in diesem Arbeitsbereich befinden, von allen Teilnehmern genutzt, um miteinander zu arbeiten. Die Informationen über die Anwesenheit, die Aktivitäten und die Absichten der Nutzer in diesem gemeinsamen Arbeitsbereich sind Awareness Informationen und werden von allen Teilnehmern genutzt, um effektiver miteinander zu arbeiten bzw. die Aktivitäten zu koordinieren. Diese Form von Awareness wird auch als *Workspace Awareness* bezeichnet [GG96] [Sch12] und ist das auf die Minute aktuelle Wissen um die Aktivitäten anderer Menschen in einem gemeinsam genutzten Arbeitsbereich. Es hat dabei dieselben Eigenschaften, wie sie Situation Awareness hat (siehe

Kapitel 2.2.1): Es ist eine Umgebung, die sich ständig ändert, in der die Informationen aus dem gemeinsamen Arbeitsbereich wahrgenommen und verstanden werden. Gutwin et al. sehen in [GG02] Workspace Awareness daher als spezielle Form von Situation Awareness. Der wichtige Unterschied zwischen Situation Awareness und Workspace Awareness ist, dass Situation Awareness Informationen hochkomplex sein können, die Menge an sich kann extrem groß sein und die Umgebung und damit auch die Informationen können sich rasant ändern, was in einem gemeinsamen genutzten Arbeitsbereich meistens nicht der Fall ist.

Wichtig ist das Verständnis, dass es sich hierbei nicht nur um das Wissen über den Arbeitsbereich an sich handelt, also was dort vorzufinden ist, wie es präsentiert wird oder von wem es erstellt wurde, sondern es umfasst auch das Wissen über die Interaktionen der Nutzer mit diesem, die wiederum Grundlage für eigene Entscheidungen und Aktionen bilden.

Relevanz

Workspace Awareness ist vor allem deshalb wichtig, da bei der kollaborativen Arbeit die Teilnehmer immer wieder ihre Aufmerksamkeit zwischen der gemeinsamen Arbeit und der individuellen, unabhängigen Arbeit wechseln und jeder Teilnehmer unabhängig von der Arbeit (gemeinsam oder individuell) für die eigenen Aktivitäten Wissen über die Interaktionen anderer mit dem Arbeitsbereich benötigt, also woran diese gerade arbeiten, worauf sie sich konzentrieren und was ihre Absichten sind. Workspace Awareness ist also essentiell für die Koordination von Aufgaben bei der kollaborativen Arbeit im gemeinsamen Arbeitsbereich. Mehr als nur für die eigenen Aufgaben kann Workspace Awareness genutzt werden, um anderen Teilnehmern bei ihren Aufgaben zu helfen, ihre Absichten nachzuvollziehen oder ihre deiktischen Referenzen zu Artefakten und Objekten im Arbeitsbereich zu verstehen [GG02] [GG96]. Zusammengefasst ist Workspace Awareness in folgenden Situationen wichtig:

- Koordination der Aktivitäten
- Vereinfachung der verbalen Kommunikation
- Anderen Nutzern Hilfe leisten bei ihrer Arbeit
- Wechsel zwischen individueller und gemeinsamer Arbeit

Insgesamt trägt Workspace Awareness zum bessren Verständnis der Aktivitäten im gemeinsamen Arbeitsbereich bei und verbessert somit die Qualität der kollaborativen Arbeit [GG98].

Abbildung 2 fasst die relevanten Punkte visuell zusammen: Die Nutzer haben ihre eigenen Aufgaben, die sie individuell bearbeiten. Dazu müssen sie sich jedoch ab und zu absprechen und daher die Aktivitäten koordinieren.

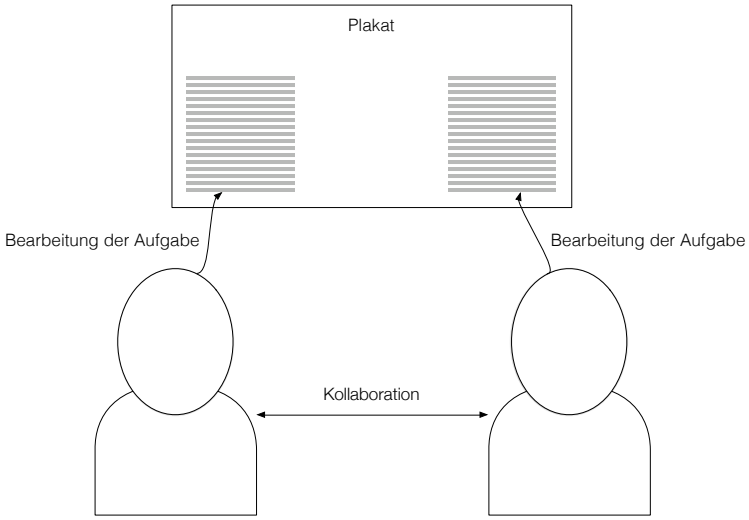


Abbildung 2: Die Teilnehmer bewältigen bei einer Kollaboration zwei Aufgaben, und zwar erfüllen sie zum einen ihre Aufgaben und zum anderen führen beide eine Kollaboration aus.

Diese Wechsel zwischen gemeinsamer und individueller Arbeit ist durch die vorhandene Workspace Awareness möglich.

Probleme bei verteilter Kollaboration

Sind alle Teilnehmer einer kollaborativen Sitzung am selben Ort, ist Workspace Awareness automatisch gegeben, da man den Arbeitsbereich mit den Artefakten und den Interaktionen der Teilnehmer mit diesen sehen kann. Problematischer wird es, wenn es sich um eine verteilte, kollaborative Arbeit handelt und die Teilnehmer einen Arbeitsbereich gemeinsam nutzen, sich aber an unterschiedlichen Orten befinden [SPS13], wie es in Abbildung 3 verdeutlicht wird. Die Nutzer bewältigen zwar ihre eigenen Aufgaben, jedoch sehen sie nicht, wer was, wann, wo und warum gemacht hat. Damit fehlen dem Nutzer eine Workspace Awareness Informationen [GGR96b], sodass eine Kollaboration verhindert wird.

Verteilt arbeitende Nutzer müssen sich also auf die Workspace Awareness verlassen, die die zur Kollaboration genutzte Software übermittelt. Allerdings bringt dieser Umstand einige Einschränkungen mit sich:

- Der Arbeitsbereich ist größer als der Bildschirm und die Nutzer navigieren unabhängig voneinander über den Arbeitsbereich. Der Nutzer kann nicht mehr sehen, auf welchen Teil des Arbeitsbereiches sich die anderen fokussieren, wo sie sich befinden, was sie gerade bearbeiten und was ihre Absichten sind.

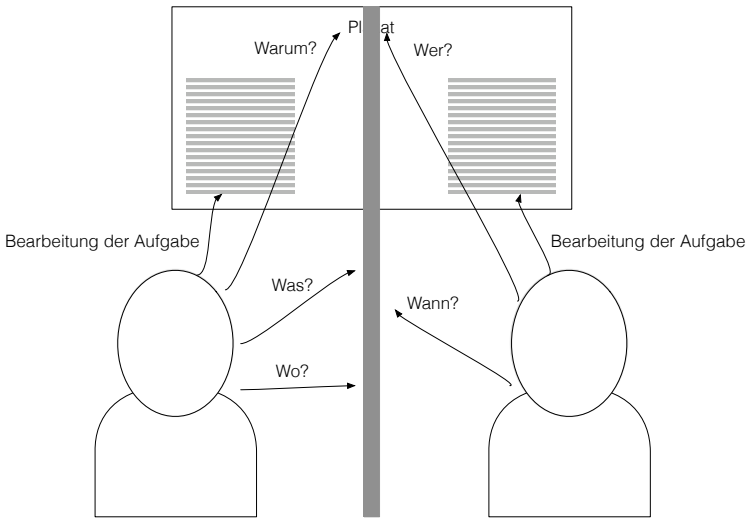


Abbildung 3: Bei einer verteilten Kollaboration ist eine räumliche Barriere zwischen den Nutzern. Die Nutzer können Aktivitäten und den Sinn hinter ihnen nicht mehr nachvollziehen.

- Es fehlt jegliche Art von Kommunikation.
- Die Bewegungen und die Interaktionen der Teilnehmer mit dem Arbeitsbereich sind nicht zu sehen.
- Selbst wenn ein Teil des Arbeitsbereiches durch die Software übertragen wird, kann ein Nutzer nur den Teil des Arbeitsbereiches sehen, welches auf dem Bildschirm aktuell angezeigt wird. Als Beispiel dient das von Roseman et al. in [RG96] vorgestellte Tool *GroupSketchpad*, ein Whiteboard für verteilte, kollaborative Arbeit (siehe Abbildung 4): Sobald eine Person auf dem gemeinsamen Whiteboard zeichnet, werden diese Zeichnungen auf dem Bildschirm des anderen Nutzers synchronisiert. Auf der Abbildung ist zu auf der rechten Seite zu sehen, dass der andere Nutzer weiter nach rechts gescrollt hat, sodass sich die Sichtbereiche beider Nutzer nur teilweise überlappen. Daher sieht ein Nutzer nur einen Teil des Arbeitsbereiches des jeweils anderen Nutzers.
- Die Handbewegungen eines Nutzers werden nur durch den Mauszeiger bzw. den Telepointer übertragen.
- Die Sprachqualität hängt vom Kommunikationsmedium ab und kann bei der Störung der Audioqualität beeinflusst werden.
- Die Videos bei der Videoübertragung können Auflösungsprobleme haben, sodass z. B. die Handbewegungen nicht gut zu erkennen sind.

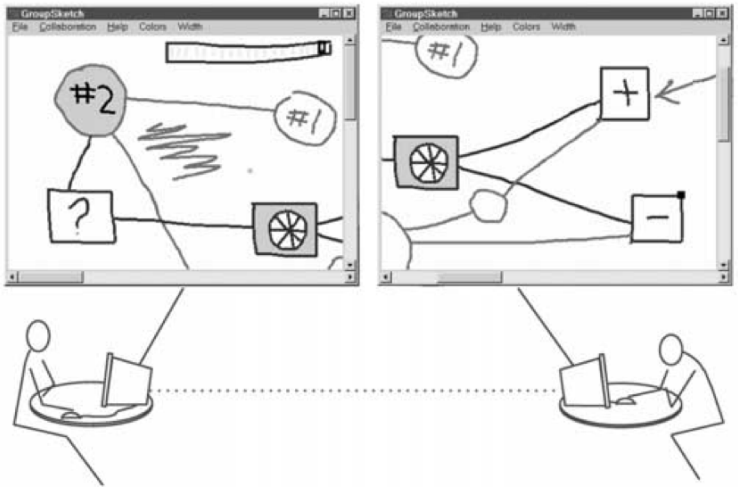


Abbildung 4: *GroupSketchpad*, ein Whiteboard-Tool für die verteilte, kollaborative Arbeit. Der rechte Nutzer ist in seinem Arbeitsbereich weiter nach rechts gescrollt, sodass sich die Sichtbereiche beider Nutzer nur teilweise überlappen. Somit sieht ein Nutzer nur einen Teil des Arbeitsbereiches des jeweils anderen Nutzers.

Das bedeutet für eine Person, dass diese nicht mehr so einfach zwischen der individuellen, von den anderen unabhängigen Arbeit und der gemeinsamen, kollaborativen Arbeit wechseln kann, da beim Wechsel Kontextinformationen fehlen und die Person nun nicht mehr weiß, wo die anderen Personen im Arbeitsbereich sind, was sie gerade tun und was ihre Absichten sind [GGR96b].

Es wird also deutlich, dass Workspace Awareness Mechanismen bei der verteilten, kollaborativen Arbeit nicht wegzudenken sind bzw. das Fehlen dieser Informationen die Qualität der kollaborativen Arbeit beeinflussen kann [GG02]. Gutwin et al. schlagen daher in [GG02] eine Liste an Fragen vor, die einem Entwickler bei der Gestaltung einer Software für die verteilte, kollaborative Arbeit helfen soll, ein angemessenes Konzept für die Ermittlung der Workspace Awareness Informationen und die Anzeige dieser zu erstellen. Denn je nach Domäne, Informationsressourcen und Anforderungen kann sich das Konzept ändern und ein allgemein richtiges Konzept ist hier nicht möglich. Kapitel 2.2.4 wird jedoch etwas näher auf dieses Thema eingehen.

2.2.3 Awareness in Saros

Saros bietet bereits eine Menge an Workspace Awareness Mechanismen, die das kollaborative Arbeiten ungemein erleichtern. Abbildung 5 zeigt die wichtigsten Workspace Awareness Informationen[2]:

[2]Die Ansicht entspricht dem aktuellen Stand im Git-Repository zum Datum der Abgabe dieser Arbeit

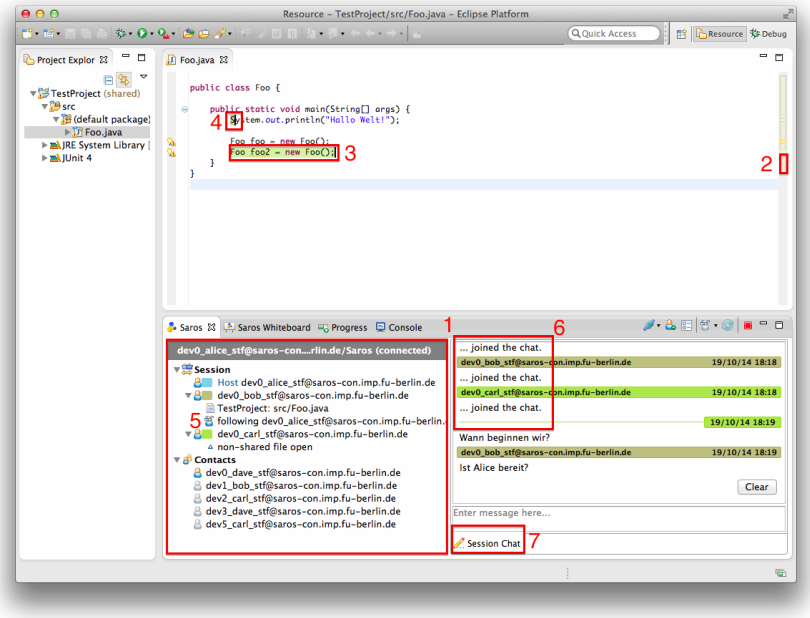


Abbildung 5: Saros bietet eine Menge an Workspace Awareness Mechanismen, die die verteilte, kollaborative Arbeit ungemein erleichtern.

1. Die Rosterview zeigt eine Buddyliste und eine Session Overview, also eine Liste an allen Teilnehmern der aktuellen Sitzung. Außerdem wird in der Buddyliste angezeigt, welcher von den Kontakten online oder offline ist. Ein online Buddy kann z. B. per Rechtsklick in die aktuelle Sitzung aufgenommen werden. Zudem zeigt diese View an, wer einen im Follow-Modus beobachtet. Außerdem wird angezeigt, welche Datei eine Person offen hat. Handelt es sich dabei um eine nicht in der Sitzung geteilte Datei, so wird dies entsprechend mitgeteilt. Hat ein Nutzer über einen längeren Zeitraum (zurzeit ist dies auf 5 Minuten gesetzt) Eclipse nicht mehr im Fokus, so wird dies durch einen Overlay symbolisiert.

2. Mithilfe von Viewports wird dargestellt, wo sich ein Sitzungsteilnehmer in der Datei befindet. Dabei wird die Farbe genutzt, die der Buddy eingestellt hat.

3. Markiert ein Teilnehmer Code, wird diese Markierung in den Editor der anderen Teilnehmer in der Farbe des markierenden Nutzers übertragen. Solange der Nutzer die Markierung beibehält, bleibt diese im Editor auch sichtbar.

4. Die Position des Mauszeigers wird übertragen.

5. Hat ein Nutzer den Follow-Modus aktiviert, so ist dies in der Session

Overview zu sehen. Jede Aktivität, wie Öffnen einer geteilten Datei oder Scrollen an eine bestimmte Stelle wird dabei an die beobachtende Person versendet und bei diesem ebenfalls ausgeführt.

6. Der Synchronisierungsprozess wird sichtbar gemacht, indem mitgeteilt wird, wenn jemand erfolgreich der Sitzung beitreten konnte.

7. Tippt ein Nutzer im Eingabefeld des Saros-Chats eine Nachricht ein, so ändert sich das Icon des Chat-Reiters von einem Sprechblasen-Symbol zu einem Stift-Symbol.

Die Liste verdeutlicht, dass Saros bereits eine Menge Workspace Awareness Mechanismen besitzt. Einige wurden in der Abbildung nicht dargestellt. Z.B. werde die Codeschnipsel, die von einem Nutzer geschrieben wurden, in seiner Farbe an der entsprechenden Stelle auch hervorgehoben. Was aussieht wie eine Markierung hilft vor allem nachvollziehen zu können, wer was geschrieben hat. Es gibt noch einige weitere Workspace Awareness Mechanismen, die hier jedoch nicht weiter aufgeführt wird, da es sich nur noch um Kleinigkeiten handelt.

2.2.4 Action Awareness

Im Allgemeinen werden in Gruppenarbeiten Awareness Informationen immer benötigt, egal ob es sich um eine verteilte oder nicht verteilte Gruppenarbeit handelt [DB92]. Je nach Anwendungsgebiet kann Awareness anders aussehen, anders dargestellt werden und unterschiedlich ermittelt werden. Daher ist es nach Gutwin und Greenberg in [GG02] schwer, eine allgemeine Lösung dafür zu finden, welche Informationen für jede Domäne Awareness bieten und wie diese präsentiert werden können. Ein Designer einer Groupware kann demnach ein Konzept für sein System nicht einfach auf ein anderes System übertragen, sondern muss erneut überlegen, was in der neuen Domäne Awareness bedeutet, wie es ermittelt wird und wie es den Nutzern des Systems helfen soll.

Gutwin und Greenberg haben sich mit dieser Problematik beschäftigt und in [GG02] ein konzeptionelles Framework vorgeschlagen. Das Framework ist vor allem für verteilte, kollaborative Arbeit gedacht, in der der Arbeitsbereich in einer mittleren Gruppengröße gemeinsam genutzt wird. Dieses Framework wird auch als *Workspace Awareness Framework* bezeichnet und besteht aus drei Aspekten:

- Welche Art von Informationen nehmen Personen auf, wenn sie gemeinsam einen Arbeitsbereich nutzen?

- Wie erfassen die Personen Workspace Awareness Informationen?

- Wie nutzen Personen Workspace Awareness Informationen in einer Kollaboration?

Das Framework ist keine Richtlinie, sondern eher ein Leitfaden, sodass man sich bei der Entwicklung oder Erweiterung von Systemen zur kollaborativen, verteilten Nutzung von Arbeitsbereichen Gedanken machen kann darüber,

- welche Informationen man sammeln und weitergeben muss,
- wie diese Informationen präsentiert werden können und
- wann diese Informationen am sinnvollsten sind.

Welche Informationen gerade Workspace Awareness darstellen, kann mit den Fragen „Wer, Was, Wo, Wann und Wie“ ermittelt werden. Bei der Arbeit in einem gemeinsamen Arbeitsbereich wissen die Teilnehmer, mit wem sie zusammen arbeiten, was sie gerade oder insgesamt tun, wo sie sich befinden, wann etwas geschehen ist und wie etwas verändert wurde. Diese Fragen können sowohl Ereignisse umfassen, die in der Vergangenheit stattgefunden haben als auch Ereignisse, die in der Gegenwart stattfinden.

Die Frage nach dem „Was“ bezieht sich auf Informationen über

- was die Teilnehmer machen (*Aktion*),
- warum sie das machen (*Intention*)
- und womit oder worauf sie die Aktion ausführen (*Artefakt*).

Diese Art von Wissen über Aktionen anderer im gemeinsamen Arbeitsbereich wird auch als *Action Awareness* bezeichnet und ist eine spezielle Art von Workspace Awareness. Dieses Wissen kann auf unterschiedlichen Ebenen bestehen, wie ganz allgemein „Der andere ist in der Anwendung X“ oder sehr detailliert, wie “Der andere macht gerade ein Refactoring in Datei X in Zeile Y“. Werden Action Awareness Informationen übertragen, die Ereignisse betreffen, die bereits längst vergangen sind, beschreiben diese Informationen eine Art Aktivitäten-Historie. So kann nicht nur nachvollzogen werden, welche Aktivitäten ausgeführt wurden, sondern auch wie sie zeitlich zu ordnen sind.

Fehlen diese Informationen über Aktivitäten anderer, kann nicht mehr nachvollzogen werden, wer in der Kollaboration was und warum getan hat. Entweder werden Änderungen an Artefakten nicht mehr sichtbar oder nur schwer nachvollziehbar. Das Wissen um die Aktionen anderer dient auch zur Orientierung der eigenen Aktivitäten. Wenn z. B. zwei verteilt arbeitende Paarprogrammierer für ein Modul Tests schreiben und einer der beiden sehen kann, dass die andere Person für Klasse X gerade einen Test schreibt, so weiß der Nutzer, was er zu tun hat (Test für Klasse Y, andere Aufgaben etc.).

Wie in Kapitel 1.2 bereits beschrieben, bietet auch Saros bereits einige Action Awareness Mechanismen an, wie wer was geschrieben hat und wo (siehe Kapitel 2.2.3). Allerdings bietet Saros keine weiteren Informationen darüber an, mit welchen Dialogen oder Views interagiert wird oder aber auch, was außerhalb von Saros geschieht. So kann es z. B. geschehen, dass ein

Nutzer einen Dialog offen hat und eine Weile in diesem Dialog verharrt. Der andere entfernte Nutzer würde dies nicht mitbekommen und würde entweder denken, dass der andere ein Problem hat oder sogar Eclipse nicht mehr im Fokus hat.

2.3 Grounded Theory

Zur Erstellung eines Konzeptes zur Verbesserung der Action Awareness in Saros wurden Aufzeichnungen zu Paarprogrammiersitzungen zur Hilfe genommen. Für die Analyse des Videomaterials wurden einige Methoden der *Grounded Theory* verwendet, die in diesem Abschnitt beschrieben werden soll.

Die Grounded Theory, übersetzt auch *gegenstandsverankerte Theorie*, ist eine Vorgehensweise aus der Sozialwissenschaft, um qualitative Daten auszuwerten [SC96]. Dabei kann es sich um Interviewtranskripte, Beobachtungen oder aber auch Video- und Audiomaterial handeln, in denen solche aufgenommen wurden. Dabei ist die Grounded Theory keine einzelne Methode, sondern eine Sammlung von Techniken, um unterschiedliche Formen von qualitativen Daten auszuwerten und aus diesen „induktiv“ [SC96] eine Theorie abzuleiten. Die Techniken können nicht nur auf qualitative Daten aus der Sozialforschung, sondern auch auf Daten anderer Domänen angewendet werden. So wird Grounded Theory auch in der Politik, in Wirtschaftswissenschaften oder auch in der Informatik [Car07] verwendet.

Die Grounded Theory wurde mit dem Ziel entwickelt, die induktiv entstandene Theorie zu einer „grounded“ Theorie zu machen. Das soll bedeuten, dass die induktiv gewonnene Theorie auf den Gegenstand, in diesem Fall auf die Daten, beruht. Somit ist die Theorie entstanden durch Daten natürlichen Ursprungs und soll somit verständlich, nachvollziehbar und allgemeingültig sein. Die Grounded Theory ist somit ein wissenschaftliches Vorgehen und bietet einen Satz an Techniken und Methoden an, um genau diese Ziele zu erreichen.

Der Satz an Techniken dient als Leitfaden für die qualitative Analyse und ist keine Richtlinie. Jede Technik an sich ist auch kein starres, vorgegebenes Vorgehen und kann mit anderen Techniken kombiniert werden. Somit können Techniken in unterschiedlicher Reihenfolge und Kombination unterschiedlich verschiedene Ebenen an Präzision ergeben. Die qualitativen Daten zeichnen sich dabei dadurch aus, dass sie nicht durch quantifizierbare Methoden und Verfahren entstanden sind. Meistens handelt es sich um sprachlich vermittelte Daten, wie Interviews, Beobachtungsprotokolle oder Video- bzw. Audioschnitte von Gesprächen und Versuchsreihen. Natürlich können auch statistisch erhobene Daten vorhanden sein, wie die Anzahl an Neuversuchen in einem Experiment, aber die eigentliche Analyse zur induktiven Theorienbildung ist eine qualitative Analyse.

2.3.1 Vorgehensweise bei der Grounded Theory

Die qualitative Analyse erfolgt in drei Schritten: Zu Beginn werden die Daten erhoben. Danach werden die Daten analysiert bzw. interpretiert. Der zweite Schritt wird auch *Kodieren* genannt und dient der Konzeptualisierung der Daten. Das Ergebnis dieses iterativen Prozesses ist eine *Grounded Theory*, eine gegenstandsverankerte Theorie, die aus den qualitativen Daten erhoben ist und sich auf das Phänomen bezieht, welches die Daten wiederspiegeln sollen. Anschließend wird die durch den iterativen Prozess der Konzeptualisierung entstandene Theorie mündlich oder schriftlich veröffentlicht und kann in einer Diskussion erneut analysiert oder angepasst werden. Erfüllt die Grounded Theory die vier nach Strauss et al. geforderten Kriterien **Übereinstimmung**, **Verständlichkeit**, **Allgemeingültigkeit** und **Kontrolle** [SC96], so handelt es sich um eine Theorie, die das zu beobachtende und untersuchende Phänomen abbildet, auf die sich die qualitativen Daten beziehen.

Wichtig bei der Anwendung von Techniken der Grounded Theory ist, *theoretisch sensibel* zu sein. Für die Anwenderin oder den Anwender der Techniken bedeutet dies, die Konzepte hinter den zu analysierenden Daten nicht nur rein deskriptiv zu beschreiben, sondern die Daten auf einer abstrakten Ebene zu interpretieren. Das Ziel der Grounded Theory ist nicht, die zugrundeliegenden Daten einfach oberflächlich zu beschreiben, sondern ein abstraktes Konzept oder abstrakte Konzepte mit Beziehungen zueinander zu finden und diese zu einer allgemeine Theorie zu formen.

2.3.2 Methoden zur Analyse

In diesem Abschnitt sollen die wichtigsten Methoden der Grounded Theory vorgestellt werden, die auch so in dieser Arbeit zur qualitativen Analyse des Videomaterials angewendet wurden.

Kodieren Kodieren bedeutet, dass man Wörtern und Sätzen der qualitativen Daten ein Wort, eine Überschrift oder eine Beschreibung zuordnet. Damit konzeptualisiert man die Daten. Weitere Daten, die dem gleichen Konzept zugrunde liegen, werden diesem auch zugeordnet. So hat man am Ende der Kodierung eine Menge an Kategorien, sogenannte *Kodes*, die die Daten auf einer abstrakteren Ebene beschreiben und über die es einfacher ist, eine Theorie zu bilden. Der Prozess des Kodierens ist also ein Analyseprozess. Dabei wird zwischen *offenem*, *axialem* und *selektivem* Kodieren unterschieden. *Offenes* Kodieren bedeutet, dass die Daten unmittelbar kodiert werden, also die Daten direkt Konzepten zugeordnet werden. Beim *axialen* Kodieren werden die ermittelten Kodes miteinander in Beziehungen gebracht und beim *selektiven* Kodieren werden die Kodes und ihre Relationen zueinander benutzt, um auf einer abstrakteren Ebene eine Theorie zu bilden.

Permanenter Vergleich Bei der Methode des permanenten Vergleiches werden die Kodierung, die Analyse und die Sammlung der Daten parallel durchgeführt und die Konzepte werden in Beziehung gesetzt, um neue Kodes zu finden [SPP07].

Theoretisches Sampling Während des Kodierens wird Wissen hinzugewonnen und das abstrakte Konzept über den Daten nimmt immer mehr Form an. Das Wissen um die bereits gefundenen Kodes beeinflusst die Erhebung neuer Daten. Diese Erhebung neuer Daten wird auch als *theoretisches Sampling* bezeichnet [Rug09].

Memos Diese Technik dient dazu, während der Analyse Gedanken, Ideen und weitere Hypothesen über Beziehungen zwischen den Konzepten zu notieren. Das stete Notieren und Überdenken der bisherigen Analyse verbessert auch die tiefe und Präzision der Reflexion [PBH04].

3 Qualitative Analyse der Videos

Für die Entwicklung eines Konzepts zur Verbesserung der Action Awareness in Saros wäre es kein wohlüberlegtes Vorgehen, blind ein Konzept zu entwickeln. Es macht mehr Sinn, sich Saros-Sitzungen anzusehen und Stellen zu identifizieren, wo Bedarf für mehr Action Awareness vorhanden ist und zu überlegen, inwieweit an diesen Stellen Hilfe geleistet werden kann. Nichtsdestotrotz helfen die Analyse von Aufzeichnungen, ein Gefühl dafür zu entwickeln, welche Stellen generell mehr Action Awareness erfordern bzw. nach welchen Aktivitäten Nutzer diese Informationen benötigen.

Dieses Kapitel beschreibt, was für ein Material zur qualitativen Analyse und Entwicklung eines Konzepts zur Verfügung stand, wie bei der Analyse vorgegangen wurde und was das Ergebnis der Analyse war. Die in Abschnitt 2.3.2 beschriebenen Techniken der Grounded Theory wurden dabei teilweise bewusst, teilweise unterbewusst angewendet bis zu einem Grad, bei dem sicher entschieden werden konnte, dass das erarbeitete Konzept allgemein ist und eine angemessene Präzision vorweisen konnte.

3.1 Vorhandenes Datenmaterial

Das Material zur qualitativen Datenanalyse wurde von der AG Software Engineering zur Verfügung gestellt. Dabei handelt es sich um neun Saros-Sitzungen, die jeweils JA1 bis JA9 benannt sind, wobei JA9 aus vier Videos besteht. Insgesamt gibt es also zwölf Videos im .avi-Format. In diesen Videodateien wurden mehrere Saros-Sitzungen aufgezeichnet, bei denen zwei Entwickler J2[3] und J1 gemeinsam an einem größeren Refactoring arbeiten. Während der Sitzungen refaktorisieren beide Entwickler Stück für Stück den Code und entscheiden sich dann dazu, das gesamte Plug-in neu zu programmieren. J1 hat viel Domänewissen und hat fortgeschrittene Programmierfähigkeiten. J2 hat dagegen wenig Domänewissen, aber sehr gute Programmierfähigkeiten. Dies hat dazu geführt, dass J1 öfters J2 im Follow-Modus beobachtet hat und es immer wieder zum Wissensaustausch zwischen J1 und J2 kam. In den Aufzeichnungen sind sie räumlich voneinander getrennt und kommunizieren über Skype. Sowohl ihr Bildschirm, ihr Audio als auch ihre Webcam werden während der gesamten Sitzung mithilfe von *Camtasia Studio 8* aufgezeichnet. Eine genaue Beschreibung einer solchen Sitzungsaufzeichnung wird in Kapitel 5.5 beschrieben. Einige Sitzungen finden noch am selben Tag mit einer Pause zwischendurch statt, andere wiederum finden an unterschiedlichen Tagen statt. Einige Videos haben eine Länge von ca. einer Stunde, andere wiederum sind zwei Stunden lang. Insgesamt umfasst das Material ca. 1044 Minuten.

[3] Da es sich um Aufzeichnungen mit Firmen- und Produktnamen handelt, dürfen hier die realen Namen nicht genannt werden. Diese Bezeichnung wurde von der AG vergeben und können bei Bedarf dort nachgefragt werden.

3.2 Vorgehen bei der Analyse

Zu Beginn stand zur Debatte, ob zur qualitativen Datenanalyse nicht eine QDA-Software (*QDA* steht hierbei für *qualitative Datenanalyse*) genutzt werden sollte. QDA-Software soll den Forschenden bei der qualitativen Datenanalyse unterstützen, um z. B. Audio- und Videomaterial zu transkribieren, um Notizen in Dokumenten oder Beziehungen zwischen Dokumenten zu setzen oder um Techniken der Grounded Theory anzuwenden (kodieren, Memos erstellen, usw.). Auf dem Markt gibt es bereits viele verschiedene freie, proprietäre sowie webbasierte Softwarelösungen. Auf [Wik14] ist eine Liste von QDA-Softwarelösungen, gruppiert nach Lizenz.

Webbasierte Lösungen kamen nicht in Frage, da vor der Ausgabe des Videomaterials eine Nutzervereinbaerung unterschrieben wurde, in der es darum ging, dass ich as Material nicht an Dritte weitergeben darf und die Daten somit nicht auf einen Server hochgeladen werden durften. Die Aufmerksamkeit fiel bei den restlichen QDA-Softwarlösungen vor allem auf zwei proprietäre Softwarelösungen, und zwar *MAXQDA*[4] und auf *Atlas.ti*[5]. Atlas.ti war deshalb interessant, da während der Bearbeitung dieser Arbeit ein anderer Student ebenfalls Atlas.ti in seiner Bachelorarbeit genutzt hat und somit Feedback zur Tauglichkeit geben konnte. Zum anderen war MAXQDA interessant, da es in manchen Foren für die Nutzung unter Mac OSX immer wieder empfohlen wurde. Atlas.ti habe ich schnell verworfen, da es dieses zur damaligen Zeit noch nicht für Mac OSX angeboten wurde[6]. Daher fiel die Wahl auf die 30-tätige Demoversion von MAXQDA 11 (siehe Abbildung 6[7]). Jedoch wurde schon beim ersten Einpflegen des Datenmaterials schnell bemerkbar, dass die Software mit so großem Datenmaterial nicht immer mithalten konnte. Es mag an der darunterliegenden Maschine gelegen haben, am Arbeitsspeicher oder auch an der Software, die so großes Videomaterial nicht effizient verarbeiten konnte. Neben Videomaterial wurden auch Paper und andere Dokumente eingepflegt und erste Versuche, Paper mithilfe von Kodes zu versehen, zeigten, dass hierbei viel Zeit in Anspruch genommen wurde, um allein Videomaterial und Paper zu kodieren. Von daher wurde zuerst darauf verzichtet die Software für die Untersuchung und Kodierung von wissenschaftlichen Quellen. Allerdings fror das darunterliegende Betriebssystem Mac OSX immer wieder ein und irgendwann musste überlegt werden, ob sich der Mehraufwand am Ende bei der Analyse lohnen würde. Letztendlich wurde aufgrund der technischen Schwierigkeiten auf die Nutzung dieser und ähnlicher QDA-Software auch für die Videoanalyse verzichtet und die Analyse erfolgte unkonventionell mit der Erstellungen von Notizen.

[4] http://www.maxqda.de

[5] http://atlasti.com/de/

[6] Inzwischen wurde jedoch eine Mac OSX Version veröffentlicht, siehe http://atlasti.com/product/mac-os-edition/

[7] http://www.maxqda.com/products/maxqda

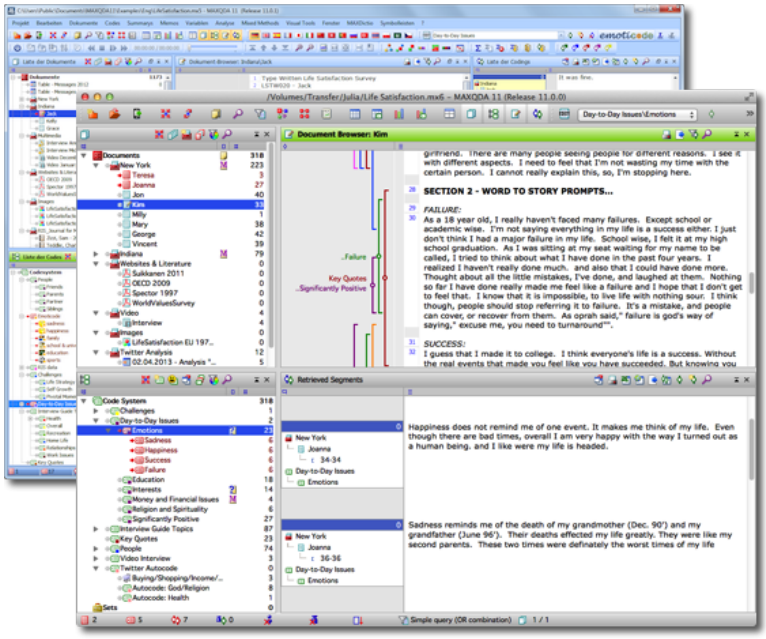


Abbildung 6: MAXQDA - Eine QDA-Software, die mithilfe von vier verschiedenen Arbeitsbereichen das Pflegen des qualitativen Datenmaterials, das Transkribieren und Kodieren dieses unterstützen soll.

Bei der Analyse mussten Stellen im Videomaterial gefunden werden, die zeigen sollten, dass wegen fehlenden Action Awareness Informationen eventuell Probleme entstehen würden oder sogar entstanden sind. Solche Stellen geben Hinweise darauf, woran das liegt und welche Informationen in diesen Situationen hilfreich gewesen wären. Ich musste also nach Stellen suchen, die Bedarf für mehr Action Awareness aufgezeigt haben. Ebenso wichtig waren aber auch Stellen, wo die Sitzungsteilnehmer ihre Handlungen verbalisiert haben. Wenn ein Nutzer beschreibt, dass er gerade einen Dialog in Eclipse geöffnet hat und eine neue Java-Klasse anlegen möchte, tut er dies vielleicht, da der andere Nutzer das sonst nicht mitbekommen würde und sich fragen würde, was der andere wohl macht. Auch Stellen, in denen IDE-Elemente sehr häufig verwendet wurden, wurden festgehalten. Solche Stellen weisen zwar nicht direkt Probleme aufgrund fehlender Action Awareness auf, aber könnten allein schon aufgrund der Häufigkeit ihrer Nutzung in der Zukunft Bedarf für mehr Action Awareness erfordern.

Vor der Analyse habe ich die Tabelle 1 mit folgenden Spalten erstellt: **Video** steht für die Nummer des Videos. Insgesamt gab es zwölf Videos. **Zeit** beschreibt den Zeitpunkt im Video, in der diese Beobachtung gemacht wurde, und unter **Beobachtung** habe ich notiert, was genau beobachtet wurde, was für Probleme zu sehen waren und auch erste Ideen für Action Awareness habe ich hier festgehalten.

Video	Zeit	Beobachtung
...	...	...

Tabelle 1: Tabelle zum Festhalten der Beobachtungen bei der Analyse des Videomaterials

3.3 Erarbeitete Kategorien

Wie im vorigen Abschnitt beschrieben, habe ich eine Tabelle angelegt, in der die Videonummer, die Zeit und das notiert wurde, was beobachtet werden konnte. Schon während der ersten Analyse des Videomaterials fiel auf, dass sich die bisherigen und die vermutlich darauffolgenden Beobachtungen in drei Kategorien einteilen lassen konnten:

- Eclipse Dialoge
- Eclipse Views
- Andere Anwendungen

Eclipse Dialoge bezeichnet dabei Dialoge, bei deren Nutzung Bedarf für mehr Action Awareness untersucht werden konnte. Es waren z. B. Situationen zu beobachten, wo ein Nutzer den anderen immer wieder gefragt hat, was der andere tut. Der andere hatte zu dem Zeitpunkt einen Eclipse Dialog offen und hat das dann mitgeteilt. Auch wurde beobachtet, wie eine der Personen auf die andere gewartet hat und als die andere Person die Aktivität in einem Dialog abgeschlossen hat und die Person das Ergebnis innerhalb des Editors sehen konnte, hat dieser mit Interjektionen, wie „Ah“ oder „Aha“, reagiert. Bei den *Eclipse Views* konnte ich ebenfalls interessante Beobachtungen machen. Ich habe z. B. bei den JUnit Views beobachten können, wie beide Nutzer immer wieder dieselben Tests ausgeführt haben, ohne zu wissen, dass der jeweils andere die Tests bereits ausführt oder ausgeführt hat. Außerdem wurde sehr häufig für Dateioperationen der Package Explorer verwendet. Dort trat es einige Male auf, dass der eine dort eine neue Datei erstellt hat und der andere erst durch Nachfrage in Erfahrung bringen konnte, was er dort tut. Die dritte Kategorie *Andere Anwendungen* beschreibt Situationen, wo außerhalb von Eclipse gearbeitet wurde und die Nutzer z. B. durch Nachfragen erst feststellen konnten, dass der jeweils andere außerhalb von Eclipse ist. Das wiederum zeigt, der jeweilige Nutzer gar nicht wusste, dass der andere nicht mehr im Eclipse Workspace ist.

Während der Beobachtung des Videomaterials habe ich daher der Tabelle eine weitere Spalte, namens **Kategorie**, hinzugefügt. Weiterhin stellte sich heraus, dass in jeder Kategorie mehrere Beobachtungen teilweise in denselben Eclipse Dialogen oder Eclipse Views gemacht werden konnten. So wurden z. B. in der Kategorie *Dialog* 80 Beobachtungen bei der Nutzung

Dialoge	Dateioperation	33
	Refactoring	33
	Suche	10
	Projekteigenschaften (Project Properties)	5
Views	Package Explorer	12
	JUnit	8
	Search	8
	Console	4
	Outline	4
	Task List	3
	Tasks	3
	Saros Whiteboard	2
	Coverage	1
Andere Anwendungen	Webbrowser	13
	Putty	7
	Dateimanager	4
	Email-Client	3
	Windows-Kommandozeile	3
	Nicht am Rechner	3
	FileZilla	2
Summe aller Beobachtungen		181

Tabelle 2: Gefundene Kategorien, Unterkategorien und Unterunterkategorien

der Refactoring-Tools von Eclipse notiert. Ähnlich sieht es mit der Nutzung von Erzeugungsdialogen, wie „New Java Class“ oder aber auch die Nutzung der Suchfunktion von Eclipse aus. Von daher habe ich nach der ersten Videoanalyse alle notierten Beobachtungen innerhalb einer Kategorie in eine Unterkategorie zugeordnet. Erzeugungsdialoge wurden z. B. in die Unterkategorie *Dateioperation* zugeordnet und ergeben somit 32 Beobachtungen. Ebenso wurden in der Kategorie *Eclipse Views* Unterkategorien gebildet, die jeweils immer eine View umfassen. Z.B. habe ich mehrere Probleme bzgl. Action Awareness in der JUnit-View sehen können. Die letztendlichen Namen der Dialoge pro Unterkategorie wurden am Ende noch einmal in einer Unterunterkatgeorie festgehalten. Insofern hat die Kategorie Eclipse Views keine Unterunterkategorie. Alle gefundenen Kategorien Unterkategorien sind der Tabelle 2 zu entnehmen.

Insgesamt habe ich das gesamte Videomaterial zwei Mal durchgesehen. Bei der ersten Durchsicht erfolgte die Untersuchung grob und diente vor allem der Kategorisierung und der Kodierung des Videomaterials. Als ich mein Gespür für Stellen mit mehr Bedarf für Action Awareness gestärkt habe und somit theoretisch sensibler für Beobachtungen wurde (siehe Kapitel 2.3.1), habe ich die Videos ein zweites Mal angesehen und dieses Mal wurden nicht

Video	Zeit	Beobachtung	Kategorie	Unterkategorie	Unterunterkategorie
4	56:47	J2 hat über den Package Explorer eine neue Datei geöffnet. Diese wird nicht bei J1 angezeigt, da sie aus einem nicht geschartem Projekt stammt. Daher sieht J1 die Datei nicht. J1 musste über den Klassenkopf, wo steht, dass es RingBuffer extended, die Klasse anzeigen lassen, denn er weiß über J2 (er hat den Namen der Klasse erwähnt, als er diese geöffnet hatte), wie die Klasse heißt. Beide haben nun dieselbe Datei offen (Klasse RingBuffer), aber sehen den jeweils anderen darin nicht, weil es eine nicht geteilte Datei ist.			
5	35:02	J2 hat in einem nicht geteilten Projekt nach einer Datei gesucht. Diese ist für J1 eigentlich nicht wichtig. J2 möchte nur etwas nachsehen. Interessant wäre jedoch trotzdem die Meldung „J2 hat gerade eine nicht geteilte Datei offen. Dies gibt es in Saros bereits (Steht unter dem Buddy im Session-Feld). Wäre es hier nicht zu viel, wenn man überträgt, dass sich derjenige im package-Explorer befindet?			
5	40:51	J2 hat gerade etwas eingecheckt. J1 weiß das, sieht das aber nicht direkt. Ist es nötig, dies mitzuteilen? Vielleicht ist es hilfreich, da man das Projekt dann updaten könnte. Andererseits ist es ja ein geteiltes Projekt, man muss also nicht updaten. Was bringt das also? Vielleicht nur eine Kurzinfo, dass gerade eingecheckt wird.			
5	46:44	J2 hat erneut das Feld „Organize Imports“ geöffnet. J1 ist sowieso im following-Modus und schaut nur zu. Selbst wenn er nicht zuschauen würde, bringt es ihm etwas, zu wissen, dass J2 gerade die Imports organisiert? Denn das ist letztendlich nur wie programmieren… Welchen Mehrwert hat es?			
5	1:05:10	J2 hat J1 während einer Saros-Sitzung zu einem anderen Projekt hinzugefügt bzw. eingeladen. Dazu ging er in den Package Explorer und hat ein weiteres Projekt angeklickt und auf "Add to Saros" geklickt. Kann man sowas anzeigen? Immerhin bewegt er sich dabei im Package Explorer herum. Durch die Einladung hat sich bei J1 ein Dialog geöffnet mit dem Titel "Add Projects". Man kann bei J2 auch diesen Dialog als Info über J1 anzeigen.			
7	34:02	J2 hat etwas committet und J1 Bescheid gesagt, dass er mal kurz warten soll. Anschließend hat er im gesagt, dass er updaten kann. Frage: Wäre die Info, dass jemand committet hat usw. nötig? Allein schon, dass er im Package Explorer war, ist Information genug. Wnen zu viele Informationen kommen, wird J1 zugespammt.			

Abbildung 7: Einige Beobachtungen wurden keinen Kategorien zugeordnet, sondern lediglich notiert, um Ideen festzuhalten.

nur weitere Kodes gefunden und offen kodiert, sondern erste Kodes konnte ich auch axial zuordnen, sodass bereits Relationen zwischen den gefundenen Kategorien gefunden werden konnten.

Alles in allem habe ich 181 Beobachtungen gemacht, wobei die Summe aller Beobachtungen in den drei Hauptkategorien nicht der Gesamtsumme entspricht, denn einige Beobachtungen konnten mehreren Unterkategorien zugeordnet werden, andere wiederum wurden keiner Kategorie zugewiesen. Abbildung 7 zeigt eine Auswahl an Beobachtungen, die keinen Kategorien zugeordnet wurden. Es handelt sich um Beobachtungen während eines Commits, der Organisation von importierten Paketen oder aber auch beim Hinzufügen eines weiteren Projekts in die Sitzung. Hier habe ich lediglich Ideen für die Übertragung weiterer Action Awareness Informationen festgehalten und in einer Reflexion überdacht. Die gesamte Tabelle mit allen Beobachtungen ist hier zu finden.

3.4 Probleme beim Konzeptentwurf

Die gefundenen Kategorien in Kapitel 3.3 geben Aufschluss über mögliche Stellen mit fehlender Action Awareness in Saros. Allein in der Kategorie „Dialoge“ wurden bei Dateioperationen oder Dialogen zu Refactorings je 33 Beobachtungen festgehalten (siehe Tabelle 2). In der Kategorie „Views“ konnte ich ebenfalls viele möglicherweise problematische Stellen ermitteln, die im Package Explorer, in der JUnit oder Search View beobachtet wurden. Allerdings ist der Entwurf eines Konzepts anhand dieser Beobachtungen bzw. Kategorien noch kein sinnvoll durchdachtes Konzept. Die Tabelle zeigt

nämlich, dass es einige Dialoge oder Views gibt, bei denen mehrere Beobachtungen festgehalten wurden, während ich bei anderen eher weniger beobachten konnte.

Warum der Entwurf eines Konzepts schwerfiel, soll im Folgenden anhand der Refactoring Dialoge beschrieben werden. Die 33 Beobachtungen beziehen sich auf unterschiedliche Refactoring Dialoge. Bei unterschiedlichen Refactoring Dialogen konnte ich jeweils unterschiedlich viele Beobachtungen machen, wobei nicht zu jedem Dialog jede Beobachtung zusammengefasst wird, da das einfach die Seitenzahl sprengen würde.

Extract Constant Dieser Dialog bewirkt, dass aus dem markierten Ausdruck eine lokale, private final Konstante erzeugt wird. Diese Funktion wurde nur ein einziges Mal benutzt. Und bei dem Mal, wo es genutzt wurde, war einer der beiden Nutzer auch nicht anwesend. Spätestens wenn diese Person zurückkehrt, wird sie die neue Konstante sehen.

Extract Method Diese Funktion bewirkt, dass aus dem markierten Bereich eine neue Methode erzeugt wird. Diese Methode hat außer in einer Situation immer nur die Person J1 genutzt. Auch wurde beobachtet, dass während J1 diesen Dialog genutzt hat, J2 ihm gesagt, dass er doch bitte eine Methode extrahieren soll. In einer anderen Situation hat J1 diesen Dialog genutzt und J2 hat auf ihn gewartet, ohne zu wissen, was J1 gerade tut. Hier wurden also immer wieder Probleme festgestellt, die darauf hingewiesen haben, dass Action Awareness Informationen gefehlt haben.

Extract Local Variable Diese Funktion ersetzt einen markierten Ausdruck mit einer neuen lokalen Variable. Diese Funktion wurde öfters genutzt und es kam mehrere Male vor, dass eine der beiden Fragen gestellt hat, wie „Was tust du gerade?“ oder „Was machst du?“. Auch wurde beobachtet, wie J2 solch eine Frage stellt und in dem Moment drückt J1 auf Enter, schließt also das Refactoring ab und J2 sieht dann das Ergebnis. J1 hat also nicht geantwortet, da er wusste, dass J2 sein Ergebnis gleich sehen wird.

Rename Method Diese Funktion erlaubt es, eine markierte Methode umzubenennen und alle Referenzen und Aufrufe zu dieser Methode werden angepasst. Diese Funktion wurde nur ein einziges Mal verwendet und beide waren auch im Following-Modus. Das heißt, dass der Zusehende die Veränderung auch sehen wird. Allerdings wäre das anderen, unabhängig arbeitenden Teilnehmern wahrscheinlich nicht aufgefallen.

Pull Up Bei der Methode „Pull up“ wird zum Beispiel eine Methode in seine Superklasse verlagert. Diese Refactoring-Methode wurde auch nur ein einziges Mal verwendet. Während J1 dies ausgeführt hat, hat J2 zugesehen.

Change Methode Signature Die Funktion „Change Methode Signature“ erlaubt es, die Signatur einer Methode sowie sein Rückgabewert und die default-Werte für seine Parameter anzupassen. Diese Funktion wurde nur zwei Mal verwendet und es hat sich nicht herausgestellt, dass es Bedarf gab, mehr über die Aktion zu wissen. Die Änderung einer Methodensignatur sieht man außerdem, sobald man die Methode über die Autovervollständigung aufruft.

Inline Method Mit dieser Funktion wird der Inhalt einer Methode, Konstante oder Variable anstelle des Aufrufs, der Konstanten oder dem Wert der Variable ersetzt. Man macht quasi eine Auslagerung wieder rückgängig. Diese Funktion wurde zwei Mal genutzt, wobei bei einer Beobachtung J1 zu J2 sagte „Mach das mal“und J2 war aber schon dabei. Man sieht also, dass obwohl nur eine Beobachtung gemacht wurde, diese gezeigt hat, dass hier nach etwas gefragt wurde, weil die Action Awareness Information gefehlt hat.

Rename Resource Mit dieser Funktion wird eine Datei im Projekt umbenannt, wobei es sich nicht um eine Java-spezifische Datei, wie Klasse oder Interface, handeln muss. In der 106. Beobachtung (siehe entsprechendes Dokument auf dem beiliegenden Datenträger) wurde zum Beispiel eine Textdatei umbenannt. J2 hat dabei jede seiner Aktionen verbalisiert mit „Dann nenn ich die mal alle ... ehm ... und ...“ hat dabei die Textdatei umbenannt. Diese Funktion wurde nur wenige Male verwendet, aber bei jedem Mal wurde nachgefragt, ob man das Ergebnis sehen würde. Hier wird es stärker deutlich, dass nach Informationen gefragt wurde oder Informationen mündlich mitgeteilt wurden, da der jeweils andere ansonsten nichts über die Aktivität wissen würde.

Rename Compilation Unit Mit dieser Funktion kann man die Klasse umbenennen. In den Aufzeichnungen wurde dieser Dialog selten genutzt und es waren auch keine Probleme festzustellen, da beide Teilnehmer vor der Umbenennung das angesprochen hatten.

Diese Liste zeigt, dass unterschiedlichste Refactoring Dialoge zum einen unterschiedlich oft genutzt wurden und außerdem nicht bei allen größere Probleme festzustellen waren.

Diese unterschiedlichen Ausprägungen der Nutzungs- und Problemhäufigkeit konnte ich auch bei der Verwendung von Views feststellen. Der Package Explorer wurde für die verschiedensten Zwecke verwendet:

- Öffnen von Dateien
- Löschen von Dateien
- Anlegen von Kopien einer Datei

- Anlegen neuer Dateien (Klasse, Interface, Paket, Datei, Ordner)
- Verschieben von Dateien
- Suchen nach dem Aufenthaltsort von Dateien oder Paketen (also in welchem Projekt)
- *Displacement Activity* (deutsch: „Übersprungsbewegung“), das ziellose Markieren von Dateien, während z. B. telefoniert wird

Einige dieser genanntem Operationen konnte ich häufiger, andere weniger beobachten. Beim Erzeugen von Dateien wurde häufiger nach dem Namen gefragt. Auch kam es ein Mal vor, dass J2 einen JUnit-Test erzeugen wollte, aber stattdessen den Erzeugungsdialog für eine normale Java Klasse offen hatte. So hat er eine neue Klasse erzeugt und erst danach festgestellt, dass er den falschen Dialog verwendet hatte. Hätte man diese Information übertragen bekommen, hätte J1 vielleicht frühzeitig Bescheid gesagt, dass J2 den falschen Dialog nutzt und es wäre nicht zu diesem Fehler gekommen. Anders sieht es bei der JUnit View aus: Hier gab es eindeutige Hinweise darauf, dass zu wenig Action Awareness Informationen vorhanden waren. Beide Personen haben mehrmals denselben Test ausgeführt und sich gegenseitig das Ergebnis mitgeteilt. Es gab sogar eine Situation, in der J1 den Test laufen lassen wollte, aber es gestoppt hat als J2 „Der Test ist grün! Yeah!“ gerufen hat. In einer anderen Situation hat J2 einen Test ausgeführt und kurz darauf J1 auch und beide haben sich gleichzeitig das Ergebnis mitgeteilt. Man kann also erkennen, dass solche Situationen gemieden werden könnten, wenn Action Awareness Informationen zur Testausführung gegeben wären.

Die Kategorie Andere Anwendungen habe ich nicht weiter in Betracht gezogen, denn wie in Kapitel 2.2.3 bereits beschrieben wurden, gibt es in Saros die Funktion, dass wenn man über einen längeren Zeitraum Eclipse nicht mehr im Fokus hat, dies bei den anderen im Saros Roster entsprechend mitgeteilt wird. Von daher habe ich hier keine weitere Überlegungen angestellt.

Letztendlich war es schwierig, aus diesen Erkenntnissen ein Konzept zu entwickeln, welches aussagt, zu welchen Dialogen und Views nun Informationen übertragen werden sollen und bei welchen weitere Details nötig sind. Sollen zu allen Dialogen und Views so viele Informationen wie möglich übertragen werden? Oder sollen nur die Dialoge und Views behandelt werden, bei denen ich viele Beobachtungen im Bezug auf fehlende Action Awareness machen konnte? Diese und ähnliche Fragen, die im kommenden Abschnitt zusammengefasst werden, haben es mir an dieser Stelle unmöglich gemacht, ein finales Konzept für die Verbesserung der Action Awareness in Saros zu entwerfen.

3.5 Ergebnis der qualitativen Analyse

Durch die Videoanalyse konnte ich feststellen, dass es einige spezielle Eclipse Views oder Eclipse Dialoge gab, bei denen Bedarf für Action Awareness Informationen ersichtlich wurde. Allerdings wäre es kein gutes Konzept gewesen, wenn ich nur diese speziellen Dialoge und Views behandelt und den ganzen Rest ausgeblendet hätte. Andere Nutzer hätten vielleicht woanders Bedarf für mehr Informationen gehabt. Saros würde inkonsistent wirken, denn die Nutzer würde sich fragen, warum plötzlich zu Dialog X und View Y Action Awareness Informationen übertragen werden und zu View Z oder anderen nichts. Andersherum würde man den Nutzer überfordern, wenn man zu allen Dialogen und Views so viele Informationen wie möglich übertragen würde. Der Nutzer würde das schnell als Spam deklarieren und die Anzeige nicht mehr berücksichtigen. Und selbst wenn er das nicht tun würde, wäre er mit der Extraktion der Informationen aus der Anzeige eventuell überfordert. Außerdem konnten diese Bedenken nicht klar umrissen werden, denn es war nicht klar, wie es wiederum aussehen würde, wenn die Nutzer je nach Arbeitsmodus Informationen übertragen bekommen oder ob sie es vorziehen würden, wenn sie die Anzeige oder die Art (welche Dialoge oder welche Views) und Menge (nur Titel oder mehr Details) der Informationen selbst konfigurieren können.

Abschließend stand neben der Problematik, dass noch kein klares Konzept entworfen werden konnte, die Frage für mich im Raum, was ich alles programmatisch von Eclipse an Informationen bekommen kann. Natürlich sollte die Lösung auch so generisch wie möglich sein. Das bedeutet, dass ich nicht für jeden Dialog oder für jede View eine eigene Behandlung entwickeln möchte, denn das ist schwerer zu warten und würde auch den großen Nachteil haben, dass in der Zukunft bei neuen Dialogen und Views in Eclipse diese von Saros nicht mehr beachtet werden würden. Aber selbst wenn es eine generische Lösung gegeben hätte, war mir weiterhin unklar, wie weit ins Detail diese Informationen gingen, also ob man nur an den Titel der IDE-Elemente gelangen kann oder ob man auch herausfinden kann, was wo in diesen Dialogen oder Views eingegeben wird. Auch in diesem Fall stand ich vor der Frage, ob das Extrahieren der Informationen für diese speziellen Handlungen (Was wird eingetippt? Welcher Button wurde geklickt? Was ist das Ergebnis dieses Dialoges?) nicht bereits eine spezielle Behandlung darstellt, denn immerhin ist jeder Dialog und jede View in Eclipse anders gebaut.

Das Ergebnis der qualitativen Analyse war insofern, dass ich noch kein Konzept für die Verbesserung der Action Awareness in Saros entwerfen konnte, denn es war eine Recherche zur technischen Realisierbarkeit notwendig, um zu sehen, welche Informationen ich von Eclipse zur Verfügung habe.

4 Implementierung des ersten Prototyps

Dieses Kapitel beschreibt die Entwicklung des Prototyps für die ersten Nutzertests. Für die Entwicklung waren einige technische Recherchen notwenig, um herauszufinden, an welche Informationen ich von Eclipse und seinen Ereignissen komme. Die technische Recherche diente vor allem dazu, die Realisierbarkeit der Anzeige von Action Awareness Informationen zu überprüfen. Das Kapitel beschreibt zuerst zwei Mechanismen, mit denen es theoretisch möglich ist, an Informationen über Interaktionen mit dem Arbeitsbereich selbst (Öffnen von Dialogen, Aktivieren von speziellen Views) oder über andere Interaktionen (Tests wurden gestartet/beendet, etwas wurde refactort, etc.) zu kommen. Anschließend wird beschrieben, inwieweit dieser Prototyp auch für andere Programmiersprachen bei der Nutzung von Saros genutzt werden kann. Danach werden die Funktionen des ersten Prototyps vorgestellt und die darauffolgenden zwei Kapitel beschäftigen sich mit der Entwicklung des ersten Prototyps. Denn am Ende der Entwicklung der ersten Version habe ich einige technische Schwierigkeiten festgestellt, die beim Lesen der API-Dokumentation nicht ersichtlich wurden, sodass das Konzept für den ersten Prototyp angepasst werden musste und ich eine zweite Version implementieren musste.

4.1 Technische Realisierbarkeit

Dieses Kapitel beschreibt die Ergebnisse der Recherche zur technischen Realisierbarkeit des ersten Prototyps. Dabei sind zwei Mechanismen interessant: Der *Listener* Mechanismus von Java und der *Extension bzw. Extension Point* Mechanismus von Eclipse. Beim Listener-Mechanismus konnte ich ungefähr einschätzen, was möglich war und was nicht, da ich das bereits aus dem Studium kannte. Allerdings kannte ich den Extension-Mechanismus nicht, sodass ich mich darüber informieren musste, um einschätzen zu können, ob ich mithilfe dieses Mechanismus irgendwelche verwertbaren Informationen erhalten oder extrahieren kann. Außerdem wollte ich untersuchen, ob unabhängig vom genutzten Mechanismus die gegebenen Informationen überhaupt ausreichend waren.

4.1.1 Listener

Typisierte SWT Listener

SWT, das *Standard Widget Toolkit*, ist ein Open-Source-Toolkit für die Entwicklung graphischer Oberflächen in Java und ist die Basis für die Eclipse-IDE. Demnach basieren auch Plugins, wie das Saros Plugin, und ihre einzelnen Bestandteile auf SWT. In SWT kann man über Listener und Events auf Interaktionen mit den Elementen der graphischen Nutzeroberfläche reagieren.

Events beschreiben dabei die Interaktion an sich oder das Ergebnis einer Interaktion. Zum Beispiel ist das Drücken einer Taste oder das Betätigen der rechten Maustaste ein Event mit dem Hinweis darauf, was passiert ist. SWT stellt verschiedene Events zur Verfügung, die je nach ihrem Typ detaillierte Informationen über die Interaktion hergeben. Wenn zum Beispiel etwas geschlossen wird, wird ein `Dispose` Event versendet, welches nicht nur darauf hindeutet, dass etwas geschlossen wurde, sondern auch was wann geschlossen wurde und mehr. Die folgende Tabelle zeigt eine Auswahl verschiedener Event-Typen und wann diese versendet werden:

Event Name	**Wird versendet, wenn ...**
Arm	Ein Menüeintrag wurde hervorgehoben
Dispose	Ein Kontrollobjekt wird geschlossen
Focus	Ein Kontrollobjekt wird fokussiert oder der Fokus wird von einem Kontrollobjekt entfernt
Key	Eine Taste wurde gedrückt oder losgelassen
Modify	Text wurde modifiziert
Mouse	Eine Maustaste wurde gedrückt, losgelassen oder doppelgeklickt
MouseMove	Die Maus wurde über ein Kontrollobjekt bewegt
Selection	Ein Item wurde in einem Kontrollobjekt ausgewählt
Shell	Ein Fenster wurde minimiert, maximiert, aktiviert, deaktiviert oder geschlossen

Tabelle 3: Verschiedene SWT-Eventtypen und wann sie versendet werden

Diese Ereignisse stellen *Mikro-Interaktionen* dar, also elementare Interaktionen, die das Verhalten eines (Kontroll-) Elements beschreiben und losgelöst von ihrem Kontext der Anwendung sind [Web07]

Listener werden eingesetzt, um auf solche Events zu reagieren. Dabei werden sie einem Widget hinzugefügt und können auf Interaktionen mit diesem reagieren. Für jede Art von Event gibt es in der SWT Bibliothek entsprechende Listener, die dann auch entsprechend `<EventName>Listener` heißen. So gibt es z. B. für das Schließen eines Widgets ein `DisposeEvent` und dem zu beobachtenden Widget kann dann ein entsprechender Listener über `addDisposeListener` hinzugefügt und über `removeDisposeListener` entfernt werden. Die folgende Tabelle enthält eine Übersicht über einige Listener für Mikro-Interaktionen und wofür sie verwendet werden können (für eine komplette Übersicht, siehe [Ecl11c]):

Untypisierte SWT Listener

Bei den in Abschnitt 4.1.1 beschriebenen Listenern handelt es sich um *typisierte* Listener. Typisierte Listener benutzen Interfaces, die speziell für bestimmte

Listener	Verwendung
`DisposeListener`	Wird verwendet, wenn ein GUI-Element geschlossen wird
`FocusListener`	Wird verwendet, wenn der Fokus auf ein GUI-Element gesetzt oder davon genommen wird
`KeyListener`	Wird verwendet, wenn eine Taste der Systemtastatur gedrückt wird
`ModifyListener`	Wird verwendet, wenn Text modifiziert ist
`MouseListener`	Wird verwendet, wenn Maus-Klicks ausgeführt werden
`PaintListener`	Wird verwendet, wenn ein GUI-Element (neu) gezeichnet werden muss
`SelectionListener`	Wird verwendet, wenn ein GUI-Element, wie z. B. ein Button oder Text, ausgewählt wird
`ShellListener`	Wird verwendet, wenn ein `Shell`-Objekt aktiviert, deaktiviert, geöffnet oder geschlossen wird

Tabelle 4: Übersicht über SWT-Listener und wann sie verwendet werden

SWT-Events gedacht sind. So kann der `ShellListener` nur `Shell`-Events abfangen. Möchte man jedoch auf eine Menge von unterschiedlichen Events reagieren, so können *untypisierte* Listener verwendet werden. Hier gibt es nur einen Listener, der auch `Listener` heißt und nur eine Methode anbietet, und zwar `handleEvent`, welches als Parameter auch ein einfaches `Event`-Objekt übergeben bekommt. Mit einem untypisierten Listener ist es demnach möglich, eine Menge von Events abzufangen, ohne mehrere unterschiedliche typisierte Listener zu implementieren bzw. jede der Listener-Methoden implementieren zu müssen.

Das folgende Beispiel soll verdeutlichen, wann untypisierte Listener ihren Vorteil verdeutlichen: Man möchte eine GUI-Anwendung bauen, in der man auf die Aktivierung und Deaktivierung von `Shell`-Objekten und zusätzlich auf Mausklicks reagieren möchte. Wenn man nur typisierte Listener verwenden würde, müsste man einen `ShellListener` und einen `MouseListener` verwenden. Jedoch bietet der `ShellListener` auch zwei andere Methoden an (`shellOpened` und `shellClosed`) und der `MouseListener` bietet sogar neben `mouseClicked` noch vier weitere Methoden an[8]. Man müsste also auch die Listener-Methoden implementieren, für die man sich nicht stark interessiert. In diesem Fall müsste man den Methodenrumpf immer leer lassen. In diesem Fall könnte man stattdessen einen untypisierten Listener verwenden, der in der `handleEvent`-Methode die Events `Activate`, `Deactivate` und `Mouse` filtert und entsprechend behandelt. So ist man in der Lage, einen generischen

[8] http://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseListener.html

Listener zu bauen und ihn beliebig durch Eventtypen zu erweitern.

Weitere Listener

Bei den in Abschnitt 4.1.1 und 4.1.1 beschrieben Listenern handelt es sich um SWT-Listener, also Listener, die vor allem auf Events reagieren können, die Interaktionen mit der auf SWT basierenden grafischen Nutzeroberfläche darstellen. Es gibt jedoch noch eine große Menge weiterer Listener. Frank Schlesinger hat im Jahre 2005 eine Diplomarbeit verfasst, in der es darum ging, ein Konzept zu entwickeln, mit dem Programmieraktivitäten in der Eclipse-Umgebung aufgezeichnet werden konnten [Sch05]. Dabei hat er sich des Listener-Mechanismus bedient und neben einige der beschriebenen SWT-Listenern auch weitere Listener eingesetzt. Die Tabelle 5 zeigt eine Menge an verschiedenen Listenern und wann diese verwendet werden können.

Listener	**Verwendung**
`IPartListener2`	Dieser Listener wird dazu verwendet, um Änderungen von Parts, also Views, zu beobachten. So kann z. B. auf das Öffnen, Schließen, Aktivieren und Deaktivieren einer View in Eclipse reagiert werden.
`IDocumentListener`	Dieser Listener kommt aus der JFace Bibliothek und wird verwendet, um auf Events zu reagieren, die beschreiben, dass gerade ein Text in einem Dokument eingegeben wird oder eingegeben wurde. Dies kann z.B. dafür verwendet werden, um anzuzeigen, ob jemand im gemeinsamen Quellcode editiert.
`IElementChanged-Listener`	Dieser Listener fängt Events ab, die versendet werden, wenn Java Elemente hinzugefügt werden. Dabei kann es sich um Klassen, Pakete, Projekte oder sogar Felder und Methoden handeln.
`IElementState-Listener`	Dieser Listener wird verwendet, um Events über Dirty States, Entfernung, Modifikation und Bewegung von Text im Texteditor abzufangen.
`IJobChange-Listener`	Dieser Listener hälft Ereignisse über Hintergrundjobs fest. Dabei kann es sich um Jobs handeln, die Team-Aktivitäten beschreiben, wie z. B. Commits, oder auch wenn etwas synchronisiert wird, wie z. B. der Workspace mit dem Repository.
`IPropertyListener`	Dieser Listener wird verwendet, um darüber informiert zu werden, wenn für ein Objekt die Eigenschaften verändert wurden. Bei dem Objekt kann es sich um jedes Objekt handeln, welches die Klasse `Object` erweitert, also im Grunde alle Klassen in Java.

`IDebugEventSet-Listener`	Dieser Listener fängt Events ab, die beschreiben, ob gerade ein Programm ausgeführt oder gedebuggt wird. Weiterhin kann bei solch einem Event der Status abgefragt werden, wie z. B. ob im Debug-Modus die Ausführung supended ist, ein *step-over* ausgeführt wurde usw. (siehe [Ecl11a])
`TestRunListener`	Dieser Listener horcht auf Testlauf- und Testcase-Events. Wird der Listener benachrichtigt, kann es Informationen darüber erhalten, ob ein Test (und welcher) gestartet oder beendet ist und wie dieser Test abgelaufen ist.
`IRefactoring-ExecutionListener`	Dieser Listener wird benachrichtigt, wenn etwas refactort wurde. Dabei kann der Zeitpunkt des Refactorings, das Projekt, indem es stattfindet, und was für ein Refactoring (Rename, Move, etc.) durchgeführt wurde, ermittelt werden.
`ResourceChange-Listener`	Dieser Listener wird benachrichtigt, wenn irgendwelche Ressourcen im Workspace verändert wurden.
`IQueryListener`	Hierbei handelt es sich um einen Listener, der darüber informiert wird, wenn über den Suchdialog in Eclipse eine Suche abgesetzt wurde, beendet wurde, wenn der Suchbegriff entfernt wurde oder wenn gerade gesucht werden soll (das Suchwort ist also eingegeben, aber die Suche hat noch nicht begonnen).

Tabelle 5: Eine Auflistung von interessanten Listener in Java

4.1.2 Extensions

Noch vor der Recherche zur technischen Realisierbarkeit fiel im Stand-Up Meeting (siehe Kapitel 7.1) öfters der Begriff *Extension* als mögliche Informationsquelle über Aktionen in Eclipse. Eclipse bietet die Möglichkeit, die Erweiterung von Plug-ins zu vereinfachen. Mithilfe eines *Extension Points* kann ein Plug-in die eigene Funktionalität anderen Plug-ins über eine definierte Schnittstelle zur Verfügung stellen. Das bedeutet, dass andere Plug-ins ihre eigene Funktionalität mithilfe des Extension Points erweitern können. Das Besondere an dieser Schnittstelle ist, dass das originale Plug-in nichts darüber weiß, welche anderen Plug-ins den Extension Point nutzen. Es bietet lediglich eine Menge an Schnittstellen und Klassen zur Verfügung, die öffentlich für andere Plug-ins zugänglich sind und für die Erweiterung genutzt werden können. Eine *Extensions* ist dagegen die Erweiterung der Funktionalität eines Plug-ins durch Implementierung dieser Schnittstellen oder Nutzung der

vorgestellten Klassen. Beide Mechanismen werden in der `plugin.xml`-Datei eines Plug-in-Projekts definiert.

Da der erste Prototyp keine Erweiterung anbieten, sondern stattdessen so viele Informationen wie möglich über Interaktionen mit der Benutzeroberfläche ermitteln soll, fiel der Fokus meiner Recherche auf Extensions. Die folgende Referenz [Ecl11c] enthält eine Liste aller Extension Points, die in der Eclipse Plattform zur Verfügung stehen und über die mithilfe einer Extension theoretisch Informationen ermittelbar sind und die auch aufgrund der Ergebnisse der qualitativen Analyse des Aufzeichnungsmaterials in Erwägung gezogen wurden (siehe Kapitel 3.5).

Refactoring

Für das Refactoring in Eclipse stehen einige Extension Points zur Verfügung. Der `Copy Participants` Extension Point [Ecl12a] bietet Informationen darüber, was kopiert werden soll, wohin es kopiert werden soll und eine Liste, in der die Auswirkungen dieses Kopiervorgangs auf das gesamte Projekt gespeichert werden. Der `Move Participants` Extension Point [Ecl12c] bietet entsprechende Informationen über das Bewegen von Ressourcen zur Verfügung. Ein weiterer Extension Point ist `Refactoring Contributions` [Ecl12e] und erlaubt es im Allgemeinen, das Refactoring in Eclipse zu erweitern. Weiterhin liefert es alle übergebenen Argumente, die für das Refactoring übergeben wurden. Ein weiterer interessanter Extension Point ist das `Rename Participants` [Ecl12f], welches eigentlich dafür gedacht ist, das Umbenennen von Ressourcen durch Extensions zu erweitern, jedoch auch Informationen darüber gibt, welche Quellressource zu welcher Zielressource umbenannt werden soll und welche Auswirkungen das auf das Projekt hat. Weiterhin gibt es noch den Extension Point `Refactoring Change Preview Viewers`, welcher die möglichen Änderungen eines Refactorings speichern und darstellen [Ecl12d] kann. Leider ist aus der Referenz der Extension Points in [Ecl11c] nicht ersichtlich, ob auch Extensions Points zu anderen Refactoring-Methoden, wie *Inline Method*, *Extract Local Variable* etc. existieren oder diese Methoden bereits durch die genannten Extension Points abgedeckt werden.

Suche

Für die Suche mithilfe des Suchdialoges in Eclipse existieren ebenfalls einige Extension Points. Der Extension Point `Search Pages` erlaubt es, dem Suchdialog weitere Seiten hinzuzufügen, um den Dialog mit einer speziellen Suche zu erweitern [Ecl12g]. Der Extension Point `Search Result View Pages` [Ecl12h] bietet die Möglichkeit, die View, in der die Suchergebnisse angezeigt werden, zu erweitern.

Erzeugungswizards

Ein weiterer Extension Point ist `Creation Wizards` [Ecl12b]. Mithilfe dieses Extension Points ist es möglich, dem *New*-Dialog in Eclipse weitere Ressourcentypen hinzuzufügen. Um eine Extension für diesen Extension Point zu schreiben, muss die Schnittstelle `org.eclipse.ui.INewWizard` implementiert werden, die eine Methode, namens `IDialogSettings getDialogSettings()`, zur Verfügung stellt, durch die man die Einstellungen auf einer Seite des Erzeugungswizards erhalten kann. Wie diese Informationen im Detail aussehen, ist der API-Dokumentation von `IDialogSettings` nicht zu entnehmen.

4.2 Konzept für den ersten Prototyp

Die technische Recherche hat gezeigt, dass es weitaus mehr Listener für unterschiedlichste Ereignisse gibt als ich am Anfang vermutet habe. Leider hat mir die Recherche auch gezeigt, dass es wirklich keine komplett generische Lösung gibt, mit der ich theoretisch in der Lage wäre, alle Informationen zu allen möglichen Interaktionen mit der Eclipse IDE zu ermitteln. Mithilfe des in Abschnitt 4.1.1 beschriebenen untypsierten Listeners ist es möglich, bestimmte SWT-Events abzufangen und so zumindest an den Titel des GUI-Elements zu kommen, aber für jede weitere Information (z. B. was in den Dialogen eingetippt wurde) habe ich vermutet, dass eine spezielle Behandlung durch entweder spezielle Maus-/Tastatur-Listener oder andere Mechanismen, wie dem Extension-Mechanismus nötig sein werden.

In Kapitel 3.5 habe ich erklärt, dass es nicht möglich war, ein finales Konzept für die Entwicklung des ersten Prototyps zu entwickeln, da noch auf die Ergebnisse der Recherche zur technischen Realisierbarkeit gewartet wurde. Nach der Recherche war mir nun klar, welche Informationen zugänglich sein würden und welche nicht. Von daher habe ich ein Konzept entwickelt, bei dem zu allen offenen Dialogen und aktiven Views der Titel angezeigt wird und in einigen Situationen detailliertere Informationen übertragen werden sollten. Die qualitative Auswertung der Sitzungsaufzeichnungen hat gezeigt, dass die Teilnehmer in den aufgenommenen Saros-Sitzungen Bedarf für mehr Informationen über Refactoring, JUnit-Tests und Erzeugungsdialoge hatten. Außerdem wurde der Search Dialog erheblich oft verwendet, auch wenn dort nie Probleme ersichtlich wurden. Aber allein die Menge an Nutzungen des Dialogs könnte ein Hinweis darauf sein, dass auch hier Action Awareness Informationen interessant sein können.

Abbildung 8[9] zeigt einen Zustandsdiagramm, der die Funktionsweise der ersten Version des ersten Prototyps beschreibt. Sobald eine Sitzung aufgebaut wurde, landet ein Nutzer automatisch in einer View oder ist bereits in einer. Hier wird dann der Titel der aktiven View übertragen. Wird eine andere View aktiviert, wird entsprechend der neue Titel übertragen. Öff-

[9] Erstellt mit OmniGraffle

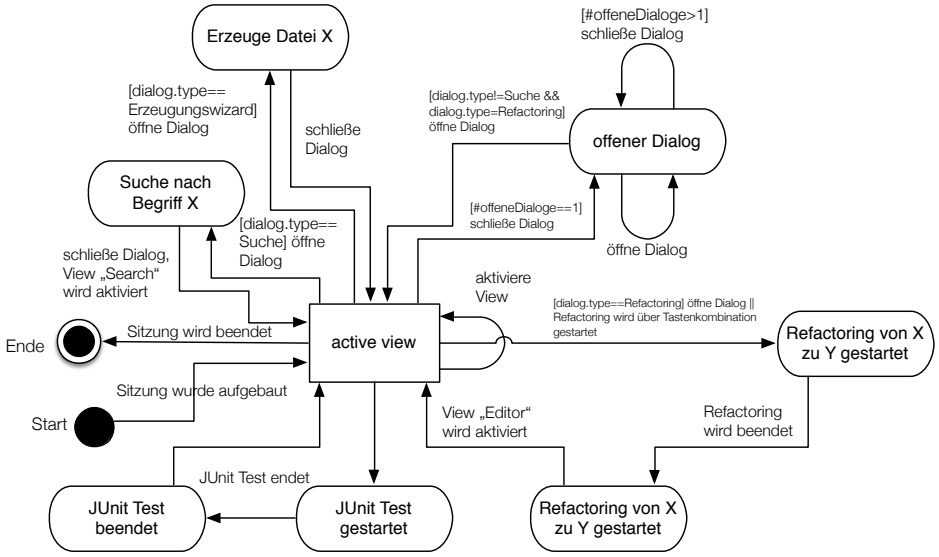


Abbildung 8: Zustandsdiagramm für die erste Version des ersten Prototyps

net der Nutzer einen Dialog, wird der Titel des offenen Dialogs angezeigt. Öffnet der Nutzer einen weiteren Dialog, bei dem der Titel sich je nach Seite ändert(WizardDialoge), wird der Titel entsprechend aktualisiert. Wenn der Nutzer den Suchdialog öffnet, wird erst einmal der Titel des Dialoges angezeigt. Gibt der Nutzer einen Suchbegriff ein, so wird angezeigt, wonach der Benutzer gerade sucht. Schließt der Nutzer den Suchdialog, landet er automatisch in der Search-View, was ebenfalls angezeigt wird. Führt der Nutzer ein Refactoring durch, wird angezeigt, was für ein Refactoring der Nutzer gerade durchführt (Rename, Move, etc.). Dabei ist es nicht wichtig, ob das Refactoring über einen Dialog oder über Tastenkombinationen gestartet wird. Ist das Refactoring beendet, wird diese Information mitgeteilt und wenn verfügbar auch Informationen über mögliche Konflikte, die währenddessen aufgetreten sind. Ist eines der geöffneten Dialoge des Nutzers ein Erzeugungsdialog, soll angezeigt werden, was der Nutzer gerade erzeugen möchte und welchen Namen dieser gerade eintippt. Während die Ressource erzeugt wird, soll etwas in der Art „Created new Java Interface '<vergebenerName>'“ stehen. Schließt der Nutzer einen Dialog, wird der Titel der View angezeigt, in der der Nutzer nach dem Schließvorgang landet. Startet der Nutzer einen JUnit Test, wird angezeigt, welchen Test er gerade gestartet hat. Endet der JUnit Test, so wird angezeigt, welcher Test mit welchem Ergebnis beendet wurde. Sobald der Nutzer nach einem Testdurchlauf wieder eine View oder einen Dialog aktiviert, wird wieder der entsprechende Titel angezeigt.

4.3 Skalierbarkeit

Dieses Kapitel beschäftigt sich mit der Frage, ob die anzuzeigenden Action Awareness Informationen auf die Programmiersprache Java beschränkt sind oder auch im Fall einer anderen Sprache ermittelt werden können. Immerhin ist Eclipse mithilfe von SWT entwickelt und es war mir zu Beginn nicht klar, ob die Informationen nur dann ermittelt werden können, wenn Saros für die Entwicklung von Code in Java genutzt wird. Da mir nun nach der technischen Recherche einiges klarer ist, kann ich diese Frage nun beantworten.

4.3.1 Dialoge und Views

Wie in Kapitel 4.2 bereits beschrieben, werden zur Ermittlung der Titel zu Dialogen und Views SWT Events abgefangen und bearbeitet. Diese werden unabhängig von der verwendeten Programmiersprache versendet. Der `New`-Dialog z. B. kann nämlich egal welche Sprache verwendet wird, genutzt werden.

4.3.2 Erzeugungsdialoge

Abbildung 9 zeigt nach Installation des *PyDev*[10] Plug-ins einen neuen Eintrag für Python-Projekte im New-Dialog. Jedes Plug-in in Eclipse kann einen Eintrag im New Dialog einfügen und somit ist dieser Dialog und somit diese Funktion unabhängig von der Programmiersprache.

4.3.3 Suchdialoge

Hierbei wird ein Listener genutzt, der auf Ereignisse reagiert, die durch die Nutzung des Suchdialoges entstehen. Dieser kann jedoch unabhängig von der verwendeten Sprache genutzt werden, da der Suchdialog auch bei anderen Sprachen verwendet werden kann. Demnach werden auch hier die Informationen unabhängig von der verwendeten Sprache ermittelbar sein. Abbildung 10 zeigt das für ein in Eclipse erzeugtes Python-Projekt. Dem Suchdialog wird ein weiterer Suchtyp hinzugefügt, aber andere Suchtypen, wie der File Search sind auch möglich.

4.3.4 JUnit Testläufe

Diese Funktion ist bereits vom Prinzip her nicht anwendbar auf andere Programmiersprachen, da es JUnit Tests nur in Java gibt.

4.3.5 Refactorings

Die Möglichkeit, Refactorings durchzuführen, ist in Eclipse mittlerweile über den Refactoring-Eintrag im Menü zu erreichen. Dieser ist auch bei anderen

[10] http://pydev.org

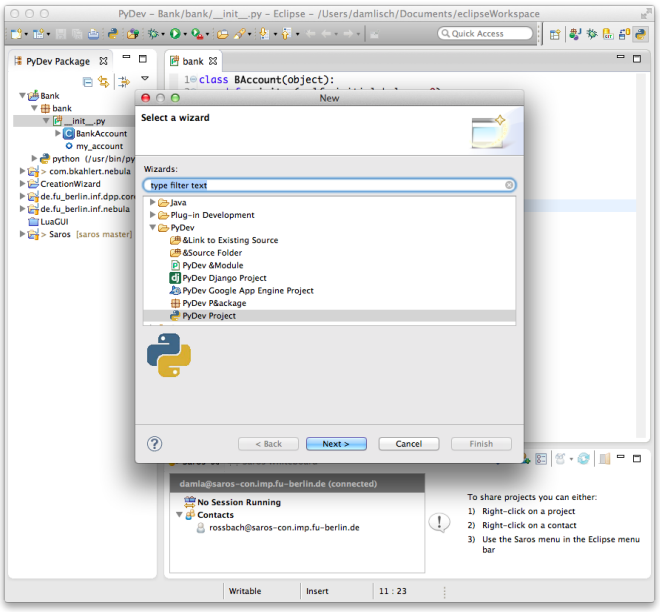


Abbildung 9: Der Erzeugungsdialog ist beliebig durch andere Plugins erweiterbar und somit unbhängig von der Programmiersprache.

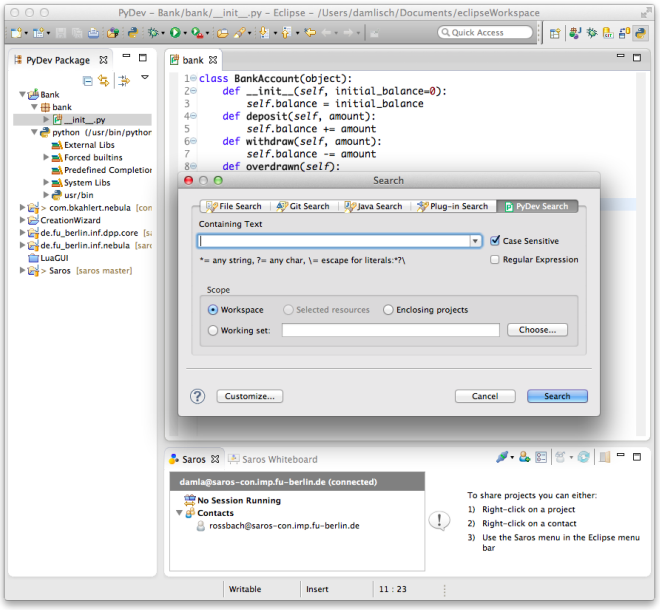


Abbildung 10: Der Suchdialog kann auch in Projekten mit anderen Programmiersprachen genutzt werden.

Programmiersprachen vorhanden und kann dort genutzt werden. Von daher wird auch diese Funktion in Saros unabhängig von der Programmiersprache funktionieren. Abbildung 11a zeigt z. B. ein Refactoring in einem Python-Modul. Abbildung 11b zeigt das Ergebnis des Refactorings.

Außer der Funktion, Informationen über Testläufe anzuzeigen, sind alle anderen Informationen unabhängig von der Programmiersprache ermittelbar. Somit kann Saros außer in einem Fall in den restlichen Fällen unabhängig von der Programmiersprache Action Awareness Informationen anzeigen.

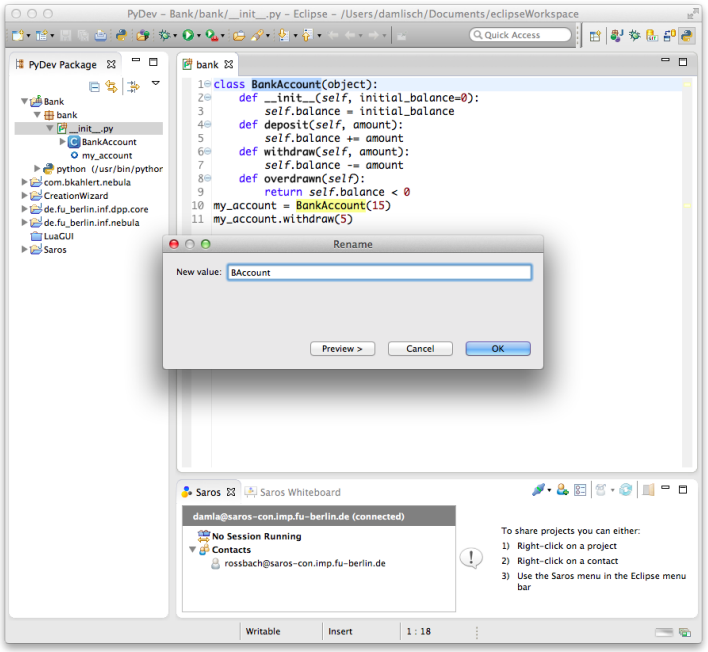
4.4 Erste Version des Prototyps

In diesem Abschnitt möchte ich erläutern, wie ich die erste Version des ersten Prototyps implementiert habe. Der Grund für zwei Versionen des ersten Prototyps war, dass während der Implementierung des ersten Prototyps einige Schwierigkeiten ermittelt wurden, einige Dinge nicht ganz so funktioniert hatten, wie ich es von der API Beschreibung her erwartet habe. Zusammengefasst soll dieses Kapitel schildern, warum das Konzept für den ersten Prototyp nicht komplett umgesetzt werden konnte und ich so eine zweite Version des ersten Prototyps entwickeln musste. Bei der Anzeige von Titeln zu Dialogen und Views konnte alles wie erwartet ermittelt und angezeigt werden. Hierbei gab es also keine Probleme und die Implementierung dieser Funktion blieb auch in der zweiten Version des ersten Prototyps erhalten. Dasselbe gilt auch für die Übertragung von Informationen zu Testläufen. Doch vorher soll beschrieben werden, an welchen Orten die Informationen angezeigt wurden.

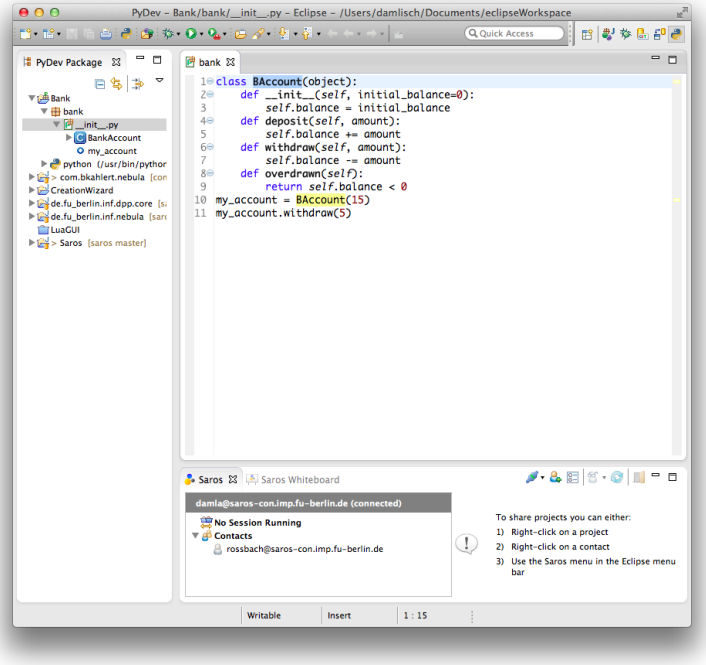
In Kapitel 7 wurde beschrieben, wie der Entwicklungsprozess in Saros abläuft. Quellcodes müssen auf *Gerrit*, einem webbasierten Code-Review System, hochgeladen werden, wo sie durchgesehen werden, bevor sie dann am Ende des Reviewprozesses ins Git-Repository gepusht werden. Zu jeder Datei jeder Änderung gibt es eine eindeutige URL. Im Folgenden wird an entsprechenden Stellen auf eine URL verwiesen, unter der die jeweiligen Quellcodes zu finden sind. Denn es ist nicht das Ziel dieses Kapitels, Codezeile für Codezeile oder seitenweise Codeschnipsel vorzulegen, sondern es soll nur ein grober Überblick über die verwendeten Mechanismen zur Ermittlung der gewünschten Informationen gegeben werden. Es soll angemerkt werden, dass die erste Version des ersten Prototyps unter [Dur14b] zu finden ist. Da die Entwicklung später jedoch auf einen anderen Git-Branch verschoben wurde, wurde diese verworfen und im neuen Branch eingepflegt (siehe [Dur14a]).

4.4.1 Anzeigeorte

Abbildung 12 zeigt alle drei Anzeigeorte. Zum einen hatte man sich überlegt, den Chat zu nutzen, weil der Chat nach Meinungen der meisten Nutzer selten bis gar nicht genutzt wird, da meistens alles nötige über die Audioverbindung besprochen wird. Eine weitere Stelle ist die Session Overview



(a) Refactoring funktioniert auch bei anderen Programmiersprachen in Eclipse. Hier wurde ein Python-Projekt als Beispiel gewählt.



(b) Hier wurde eine Klasse in einem Python-Modul refaktorisiert.

Abbildung 11

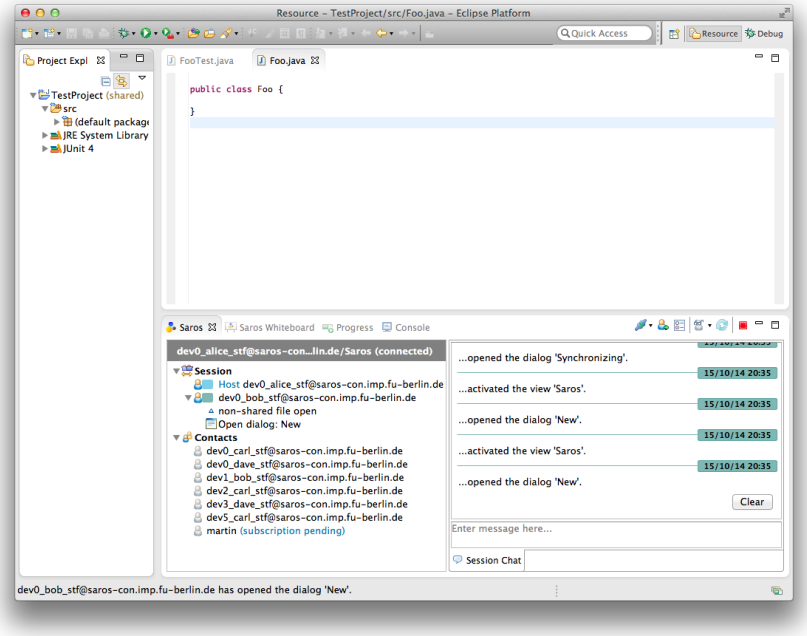


Abbildung 12: Die Action Awareness Informationen werden an drei Orten angezeigt: In der Chat Ansicht sowie in der Session Overview und in der Eclipse Statusleiste unten links.

von Saros. Dort werden bereits Awareness Informationen, wie die aktuell geöffnete Datei, angezeigt (bzw. ein Hinweis, wenn es sich um eine nicht geteilte Datei handelt). Als dritter Ort hatte ich mir die Statusleiste von Eclipse überlegt. Die Idee hinter diesem Ort war, dass bereits Eclipse immer wieder Informationen an diesem Ort anzeigt, wie die Anzahl an angeklickten Elementen oder welche Datei im Package Explorer gerade markiert ist. Da das im Grunde auch Action Awareness Informationen darstellen, wurden auch die ermittelten Informationen hier angezeigt. Über die `saros.properties` Datei ist der Anzeigeort auch konfigurierbar[11]. So kann man dort beliebige Kombinationen der drei Anzeigeorte einstellen und muss bei Umkonfigurierung Saros neustarten. Der Sinn hinter dieser Konfigurationsmöglichkeit ist, in den Nutzertests den Nutzern unterschiedliche Orte vorzulegen, sodass ein Nutzer sie nicht an allen Orten gleichzeitig sieht, sondern nur an einem und diesen dann ausgiebig testen kann. Da sich jedoch später herausgestellt hat, dass ich nicht ausreichend Testnutzer zur Verfügung hatte, konnte ich diese Idee so nicht umsetzen und habe letztendlich allen Nutzern Informationen auf allen Anziegeorten präsentiert.

[11] siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1572/12/de.fu_berlin.inf.dpp/saros.properties

4.4.2 Dialoge und Views

Implementierung

Für die Ermittlung von Titeln für den aktuell offenen Dialog oder aktive View habe ich zwei verschiedene Listener eingesetzt. Zum einen habe ich den in Kapitel 4.1.1 vorgestelltem untypisierte Listener eingesetzt, um den Titel des offenen Dialoges zu ermitteln. Listing 16 zeigt im Groben[12], wie das umgesetzt wurde. Bei der Registrierung des Listeners zum Sitzungsstart wird ein Filter für die drei verschiedenen Eventtypen *SWT.Activate*, *SWT.Deactivate* und *SWT.Dispose* gesetzt. Hier werden diese drei Ereignistypen abgefangen. Zeile 6 zeigt ein Beispiel für das Öffnen eines Dialoges. Wird solch ein Event abgefangen, wird über die `fireActivity` dem darunterliegenden Saros-System mitgeteilt, dass eine neue Aktivität erzeugt wurde, die den anderen Saros-Mitgliedern zugestellt werden soll, also in diesem Fall die Information, dass ein Dialog geöffnet wurde. Handelt es sich um einen Wizard, der mehrere Seiten hat und demnach unterschiedliche Titel besitzt, wird ein Listener verwendet, der speziell darauf ausgelegt ist, Seitenänderungen in einem Wizard abzufangen[13]. In Zeile 12 wird noch ermittelt, welches die letzte View war, in der sich der Nutzer befand, damit diese Information nach dem Schließen eines Dialoges dann entsprechend übermittelt werden kann.

```
1  private final Listener shellListener = new Listener()
       {
2      @Override
3      public void handleEvent(Event event) {
4          Shell shell = (Shell) event.widget;
5          String currentDialogTitle = shell.getText();
6          if (event.type == SWT.Activate) {
7              fireActivity(new IDEInteractionActivity(
                   session.getLocalUser(),
                   currentDialogTitle, Element.DIALOG,
                   Status.FOCUS));
8          } else if (event.type == SWT.Deactivate) {
9              fireActivity(new IDEInteractionActivity(
                   session.getLocalUser(),
                   currentDialogTitle, Element.DIALOG,
                   Status.UNFOCUS));
10         } else if (event.type == SWT.Dispose) {
11             fireActivity(new IDEInteractionActivity(
                   session.getLocalUser(),
                   currentDialogTitle, Element.DIALOG,
                   Status.UNFOCUS));
```

[12] Natürlich gibt es vorher und nachher viele weitere Stellen, wo Abfragen gemacht werden, aber das Listing soll den Kerngedanken der Umsetzung vorstellen

[13] Siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1572/12/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/awareness/IDEInteractionActivitiesManager.java für den gesamten Code

```
12                  handleLandingViewTitle();
13              }
14          }
15  };
```

Listing 1: Ermittlung des Titels für den offenen Dialog

Für die Ermittlung des Titels zur aktiven View wird ein typisierter Listener (siehe 4.1.1), der `IPartListener2` (siehe Tabelle 5), verwendet. Listing 12 zeigt im Groben, wie dies umgesetzt wurde. Der `IPartListener2` besitzt noch weitere Schnittstellen, deren Methodenrümpfe leergelassen wurden, da sie uninteressant waren. In der Methode `partActivated` wird z. B. im richtigen Quellcode überprüft, ob es sich um den Editor oder um eine normale View handelt, denn technisch gesehen werden beide unterschieden bzw. sind Instanzen unterschiedlicher Klassen.

```
1   private final IPartListener2 viewListener = new
        IPartListener2() {
2       @Override
3       public void partActivated(IWorkbenchPartReference
            partReference) {
4           fireActivity(new IDEInteractionActivity(
                session.getLocalUser(), partReference.
                getTitle(), Element.VIEW, Status.FOCUS));
5       }
6
7       @Override
8       public void partActivated(IWorkbenchPartReference
            partReference) {
9           fireActivity(new IDEInteractionActivity(
                session.getLocalUser(), partReference.
                getTitle(), Element.VIEW, Status.UNFOCUS));
10      }
11  };
```

Listing 2: Ermittlung des Titels für die aktive View

Probleme

Während der Bearbeitung dieser Funktion sind massive Probleme entstanden, die viel Zeit beansprucht haben. Zum einen wurde die Entwicklung dieser (und den meisten anderen Funktionen des Prototyps) zeitlich stark durch den Reviewprozess verzögert (siehe Kapitel 7.2). Wenn die Änderungen hochgeladen wurden, dauerte es erst eine Weile, bis der Code vollständig durchgesehen wurde. Da es sich bei den Reviewern um Entwickler handelt, die selbst an eigenem Code entwickeln, trafen die Reviewergebnisse manchmal ein oder zwei Tage später ein, manchmal auch noch später. In den meisten Fällen konnten die Reviewergebnisse schnell eingearbeitet werden und nur in ganz

wenigen Fällen hat das ein oder zwei Tage gedauert, wenn die Änderungen weitreichender waren oder woanders Fehler dadurch entstanden sind, die gedebuggt werden mussten. Was aber das größte Problem darstellte, war, dass ich unter Mac OSX 10.09 für die Entwicklung von Tests gearbeitet habe[14]. Im Test wurde überprüft, ob nach programmatischem Öffnen von Dialogen oder aktivieren von Views diese auch in der Session Overview zu sehen waren. Die Implementierung des Tests an sich war keine große Schwierigkeit, allerdings schlug der Test mal fehl, mal lief er erfolgreich durch. Selbst nachdem ich die Logmeldungen studiert habe, konnte ich keine Hinweise dafür finden, warum dies der Fall war. Als der Fehler selbst nicht gefunden werden konnte, hat sich einer der wissenschaftlichen Mitarbeiter diesem Problem gewidmet und wir haben zusammen mehrere Stunden gedebuggt. Am Ende wurde der Fehler gefunden. Allerdings hat sich mehrere Tage später herausgestellt, dass sich der Fehler immer noch in nicht-deterministischen Zeiten zeigte. Wieder wurden mehrere Tage gedebuggt und nichts wurde gefunden. Irgendwann war die Frustration nach einer Woche so groß, dass ich entschieden habe, diese Funktion des Prototyps von Grund auf neu zu entwickeln. Zeile für Zeile wurde alles neu entwickelt und danach sofort getestet. Aber auch das brachte keinen Erfolg. Irgendwann habe ich selbst Stellen im Code gedebuggt, die kein Teil des eigenen Codes waren, weil ich vermutet hatte, dass es am Test-Framework lag. Das brachte ebenfalls keinen Erfolg. Irgendwann hat sich einer der Open-Source-Entwickler des Saros-Projekts das angesehen und die fehlerhafte Stelle gefunden. Interessanterweise war das nur eine Zeile Code, die zusätzlich geschrieben werden musste, um den Fehler zu beheben[15]. Denn während die Tests mithilfe des Testframeworks ausgeführt werden, werden zwei Saros-Instanzen erzeugt und immer wieder wird eine Aktion auf einer Instanz ausgeführt, danach in der anderen Instanz usw. Und unter Mac OSX 10.09 wurde, wenn in einer neuen Testmethode zur anderen Instanz gesprungen wurde, der jeweils andere Eclipse Workbench deaktiviert. Und wenn eine neue Testmethode begann, die entsprechende Saros-Instanz jedoch noch deaktiviert war, konnten keine Elemente ermittelt werden. Und da die Testmethoden nicht immer in der selben Reihenfolge ausgeführt werden, wurde dieses Verhalten auch nur manchmal sichtbar. Die Lösung war, zu Beginn jeder Testmethode den Eclipse-Workbench programmatisch zu aktivieren, z. B. mittels `BOB.remoteBot().activateWorkbench()`. Natürlich war es ärgerlich, dass die Lösung für das Problem, welches fast drei Wochen Zeit gefressen hat, letztendlich so simpel war. Aber die Erfahrung, dass Codes trotz der Nutzung einer plattformunabhängigen Sprache auf allen Systemen getestet werden

[14] siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1572/12/de.fu_berlin.inf.dpp/test/stf/de/fu_berlin/inf/dpp/stf/test/awareness/IDEInteractionTest.java

[15] siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1572/10/de.fu_berlin.inf.dpp/test/stf/de/fu_berlin/inf/dpp/stf/test/awareness/IDEInteractionTest.java

sollten, konnte ich somit machen und habe etwas wichtiges dazugelernt.

4.4.3 Erzeugungsdialoge

Die Recherche zur technischen Realisierbarkeit in Kapitel 4.1 hatte gezeigt, dass es keinen Listener gibt, der zu Erzeugungsdialogen bzw. zu Ereignissen in diesen Dialogen Ereignisse abfängt. Von daher habe ich hier nicht mehr weiter untersucht und habe mir den Extension Mechanismus angesehen, der in Kapitel 4.1.2 bereits angesprochen wurde. Leider ergab das Ausprobieren sehr schnell die Einsicht, dass dieser Mechanismus für den Prototypen aus den folgenden Gründen nicht angemessen ist:

1. Wenn man die Schnittstelle zum Extension Point des Erzeungswizards implementiert und zum aktuellen Dialog Informationen erhalten möchte, erhält man nur den Titel.

2. Saros ist nicht darin bestrebt, eine Erweiterung für den Erzeugungswizard zu schreiben. Die Nutzung einer Erweiterung als Umweg, um an Informationen zu gelangen ist keine elegante Lösung.

Es war es also nicht möglich, auf weitere Inhalte bei Erzeugungsdialogen direkt über die API zuzugreifen. Allerdings war die Intention, den Namen für die anzulegende Datei während der Erzeugung zu übertragen. Somit kann der entfernte Nutzer sehen, wie die gerade zu erzeugende Datei benannt werden soll und ggf. intervenieren. Da die API das jedoch nicht anbot und die Idee der Nutzung einer Extension aus den oben genannten Gründen verworfen wurde, habe ich eine andere Idee ausprobiert. Und zwar konnte die Funktion zur Anzeige von Dialogen und Views wie bekannt den derzeitig offenen Dialog ermitteln. Mithilfe des *PDE Incubator Spy*[16] für Eclipse kann man sich zu verschiedenen Plug-In's in Eclipse weitere Informationen anzeigen lassen. Abbildung 14 zeigt z. B. die Klasse des aktiven Wizards für den `New Java Class` Dialog an. Hierbei handelt es sich um die Klasse `NewClassCreationWizard`. Theoretisch ist es nun möglich, beim derzeitig offenen Dialog zu überprüfen, ob es eine Instanz dieser Klasse ist. Wenn dem so ist, können alle Kind-Elemente iteriert werden, bis man das Textfeld mit dem Label „Name:“ erreicht. Den Text dieses Feldes könnte man extrahieren oder man fügt einen Listener auf dieses Feld und bei jedem Tastenschlag erhält man somit das eingetippte Wort zur Echtzeit. Um zu testen, ob das alleinige Auslesen des Textes funktioniert, wurde ein Minimalbeispiel implementiert, dessen relevante Stelle in Listing 8 zu sehen ist[17]. Das Ergebnis ist in Abbildung 13 zu sehen.

[16] http://www.eclipse.org/pde/incubator/spy/

[17] Der restliche Code, wo die GUI erzeugt wurde, wurde hier bewusst weggelassen, da nur diese Zeilen die wirklich interessanten sind

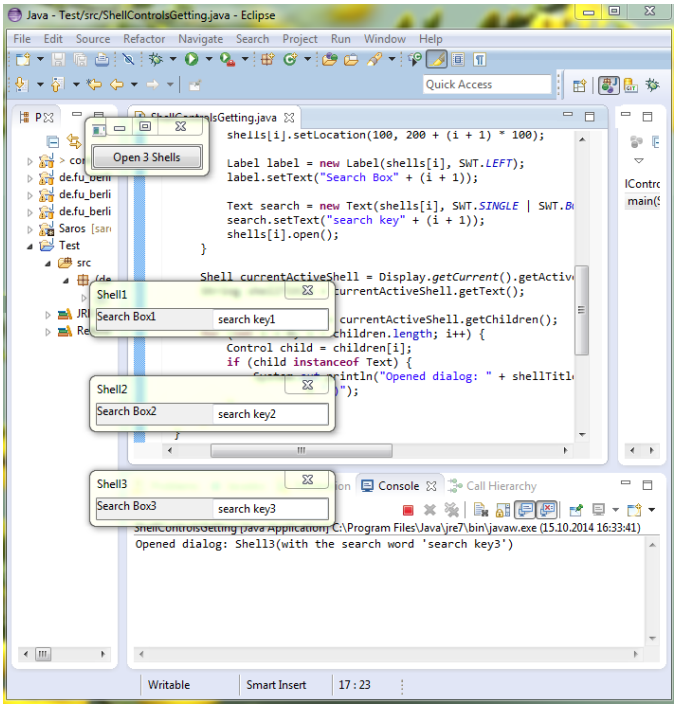


Abbildung 13: Das Textfeld eines Erzeugungsdialoges kann ausgelesen werden.

```
Control[] children = currentActiveShell.getChildren();
for (int i = 0; i < children.length; i++) {
    Control child = children[i];
    if (child instanceof Text) {
        System.out.println("Opened dialog: " +
            shellTitle + "(with the search word '" + ((
            Text)child).getText() + "')");
    }
}
```

Listing 3: Zugriff auf das Textfeld eines DIaloges

Man kann sehen, dass es also theoretisch möglich ist, an den Inhalt dieses Feldes zu kommen. Wenn man dann einen Listener auf dieses Element registriert, könnte man bei jedem Tastenschlag überprüfen, wie der Text sich verändert hat und so in quasi-Echtzeit ermitteln, was der Nutzer dort gerade eingetippt hat. Dieser Ansatz wurde in Saros dann implementiert, aber brachte erhebliche Schwierigkeiten mit sich, die im Folgenden aufgelistet werden:

- Es gibt Dialoge, die haben mehrere Textfelder. Man muss also zuerst das richtige Textfeld finden. Und da jeder Dialog nicht dieselbe Anzahl an Feldern hat, muss man von Fall zu Fall immer unterschieden.

- Nicht immer ist das Text-Element im Eltern-Kind-Baum an oberer Stelle. In einigen Dialogen sind oft um das gewünschte Text-Element noch weitere Elemente, die sich wieder je nach Dialog unterscheiden.

- Jeder Dialog ist eine Instanz einer unterschiedlichen Klasse. So ist der Dialog für ein neues Java-Interface eine Instanz der Klasse `NewInterfaceCreationWizard` (siehe Abbildung 15). Dasselbe gilt z. B. auch für den Wizard für die Erzeugung eines neuen Java-Projekts (siehe Abbildung 16) oder aber auch für Wizards, die Nicht-Java-Dateien erstellen, wie z. B. der Wizrad zur Erstellung eines XML-Dokuments (siehe Abbildung 17) oder zur Erzeugung einer Datei (siehe Abbildung 18). Man muss also bei jedem Dialog überprüfen, ob dieser eine Instanz irgendeiner dieser Klassen ist. Und dann muss man je nach Typ die Kind-Elemente unterschiedlich traversieren und überprüfen, welches Textfeld nun das gesuchte ist. Das ist sehr fehleranfällig und hat auch zu vielen Kopfschmerzen während der Implementierung in Saros geführt.

- Der größte Nachteil ist die fehlende Generezität. Kommen neue Plug-Ins zu Eclipse durch eine neue Version oder den Nutzer selbst hinzu, so werden diese nicht mehr berücksichtigt. Nur eine „Standard“ Eclipse-Version ohne weitere Plug-Ins würde von diesem Feature profitieren.

Diese Nachteile sind schwerwiegend und führten zum Entschluss, die Funktion zur Anzeige von vergebenen Titeln in Erzeugungsdialogen aus dem Konzept zu entfernen. Erzeugungsdialoge werden wie alle anderen Dialoge behandelt und nur der Titel des offenen Dialoges wird angezeigt.

4.4.4 Suchdialog

In Kapitel 4.2 wurde erwähnt, dass für die Ermittlung der Informationen über den Suchdialog der `IQueryListener`[18] verwendet wird. Wenn man sich dieses Interface ansieht, so findet man unter anderem folgende Schnittstelle: `void queryStarting(ISearchQuery query)`. Diese Methode wird aufgerufen, sobald eine Suche gestartet wurde. Diese Suche läuft erst dann los, wenn die Anweisungen innerhalb dieser Methode abgearbeitet sind. Als Parameter kriegt diese Funktion ein Suchobjekt vom Typ `ISearchQuery`. Dieses Objekt bietet eine Funktion `String getLabel()`, welches nach API-Beschreibung „ein benutzer-lesbares Label für die Suche“ zurückgibt. Daher wurde bei der Entwicklung des ersten Konzepts davon ausgegangen, dass es sich dabei um den Suchbegriff handeln muss. Allerdings hat die Implementierung gezeigt, dass es sich dabei nicht um den Suchbegriff handelt, sondern um das Label im entsprechenden Suchtab, also ob es sich um eine Suche im Tab *File Search* oder *Java Search* usw. handelt (siehe Abbildung 19).

[18] http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/search/ui/IQueryListener.html

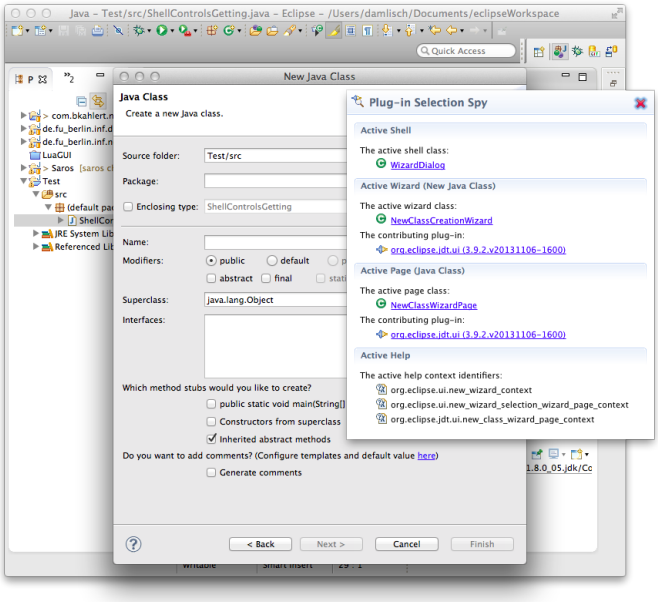


Abbildung 14: Der New Java Class Dialog ist vom Typ `NewClassCreationWizard`

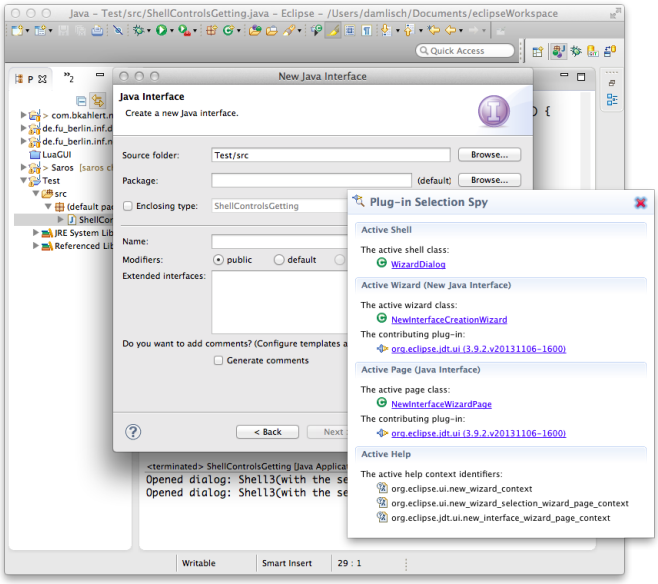


Abbildung 15: Der New Java Interface Dialog ist vom Typ `NewInterfaceCreationWizard`

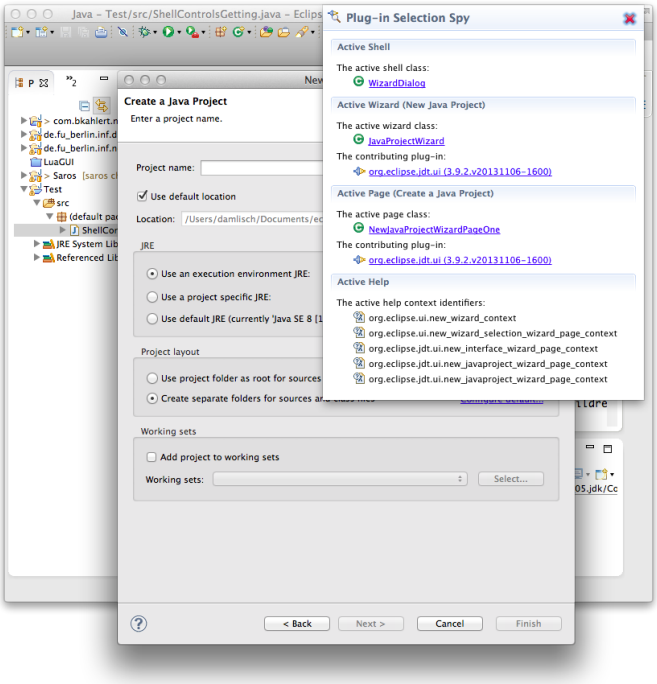


Abbildung 16: Der New Java Project Dialog ist vom Typ `JavaProjectWizard`

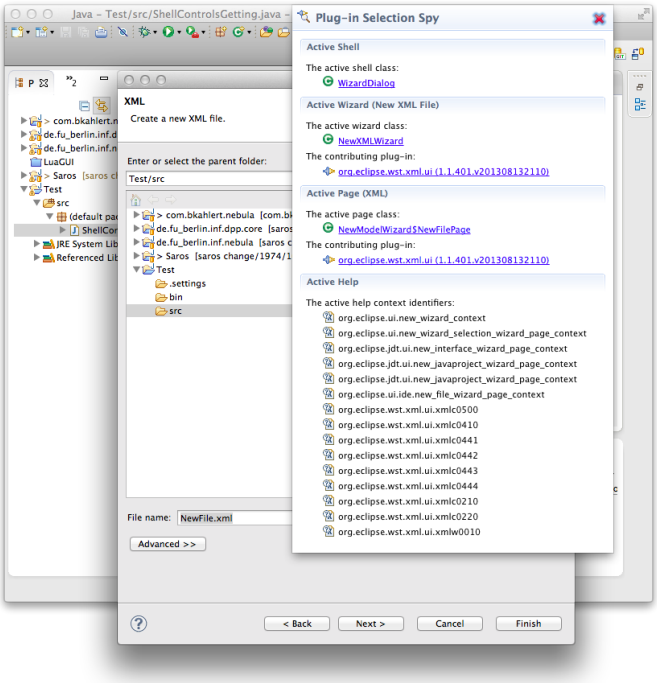


Abbildung 17: Der New XML File Dialog ist vom Typ `NewXMLWizard`

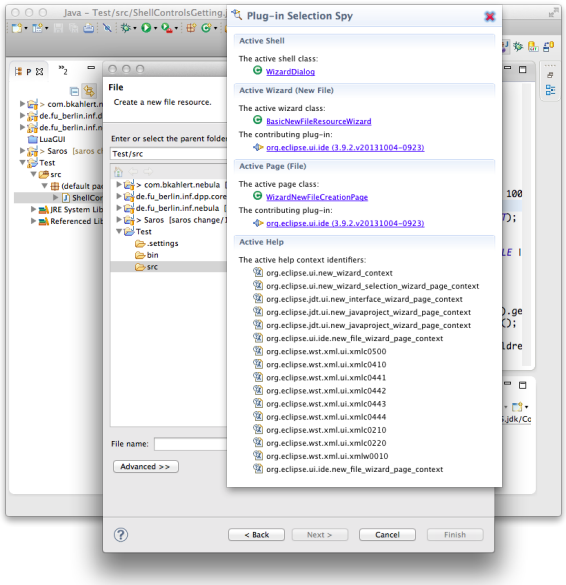


Abbildung 18: Der New File Dialog ist vom Typ `BasicNewFileResourceWizard`

Um an den Suchbegriff heranzukommen, müsste man also so vorgehen wie bereits bei den Erzeugungsdialogen in Kapitel 4.4.3 beschrieben: Man müsste alle Kind-Elemente des aktiven Dialoges traversieren und überprüfen, welches von denen der entsprechende Text für die Sucheingabe ist und dieses Feld dann auslesen. Da das aber dieselben Nachteile mit sich mitbringen würde wie bei den Erzeugunsdialogen, wurde auch diese Funktion aus dem Konzept verworfen und nur der Titel des Suchdialoges wird übertragen, wenn dieser geöffnet wird. Wie das funktioniert, wurde bereits in Kapitel 4.4.2 beschrieben und wurde in der Implementierung der zweiten Version des Prototyps umgesetzt (siehe Kapitel 4.5).

4.4.5 Refactorings

Für die Ermittlung von Informationen zu durchgeführten Refactorings wird der `IRefactoringExecutionListener` verwendet. Dieser erhält bei einem Refactoring ein Ereignis vom Typ `RefactoringExecutionEvent`, welches wiederum folgende Zustände hat:

ABOUT_TO_PERFORM Ein Refactoring wird angestoßen.

ABOUT_TO_REDO Ein Refactoring wird erneut angestoßen.

ABOUT_TO_UNDO Ein Refactoring wird rückgängig gemacht.

PERFORMED Ein Refactoring wurde ausgeführt.

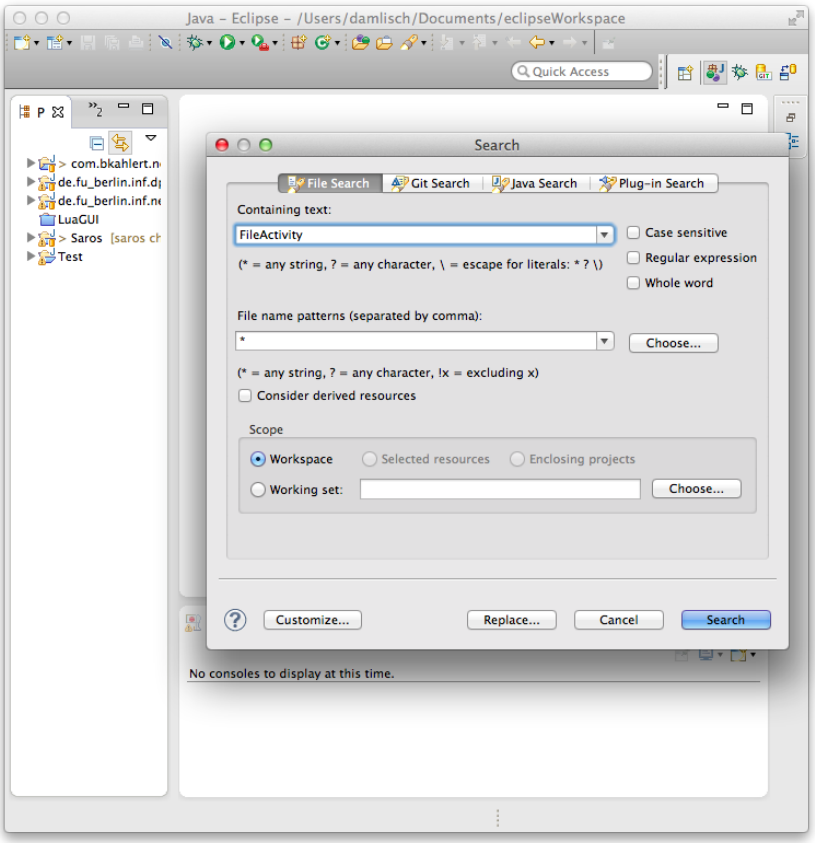


Abbildung 19: Label eines Suchbegriffes entspricht dem Typ der Suche. Der Typ entspricht dem Titel des Tabs im Suchdialog.

REDONE Ein Refactoring wurde erneut ausgeführt.

UNDONE Ein Refactoring wurde rückgängig gemacht.

Für den ersten Prototyp waren vor allem die beiden Zustände **ABOUT_TO_PERFORM** und **PERFORMED** interessant. Vor allem der Zustand **ABOUT_TO_PERFORM** hätte Informationen in Echtzeit geben können, sodass man während jemand den Code refactort, dies sehen kann. Als diese Funktion jedoch implementiert wurde, wurde leider erst dann klar, dass die Zeit zwischen den beiden interessanten Zuständen zu kurz war, denn der Zustand **ABOUT_TO_PERFORM** trat genau dann ein, wenn jemand in einem Refactoring-Dialog auf Ok gedrückt bzw. bei der Verwendung mit Tastenkürzeln die Enter Taste gedrückt hat. Und als alle Änderungen angewendet wurden, trat der Zustand **PERFORMED** ein. So kommt man leider nicht zum einen an Echtzeitinformationen heran und zum anderen macht es keinen Sinn, beide Zustände abzufangen. Von daher wurde entschieden, nur den Zustand `PERFORMED` abzufangen, sodass man nur dann die Information

über ein Refactoring überträgt, wenn der Nutzer dieses bereits durchgeführt hat. Das war im Konzept leider auch nicht so vorgesehen, aber das ist kein großer Nachteil, denn so wird wenigstens ein Refactoring gesehen und liegt auch nicht sehr weit in der Vergangenheit zurück.

4.4.6 JUnit Testläufe

In Kapitel 4.2 wurde beschrieben, welche Informationen zu JUnit Testläufen ermittelt werden. Der Quellcode für diese Funktion ist in [Dam14] vorzufinden. Listing 14 zeigt teilweise in Pseudocode, wie die Informationen zu JUnit Testläufen ermittelt und verschickt werden.

```
1  private final TestRunListener testRunListener = new
       TestRunListener(){
2      @Override
3      public void sessionStarted(ITestRunSession
           testRunSession) {
4          fireActivity(new TestRunActivity(session.
               getLocalUser(), testRunSession.
               getTestRunName(), State.UNDEFINED));
5      }
6
7      @Override
8      public void sessionFinished(ITestRunSession
           testRunSession) {
9          State state = getActivityState(testRunSession.
               getTestResult());
10         fireActivity(new TestRunActivity(session.
               getLocalUser(), testRunSession.
               getTestRunName(), state));
11         }
12     };
13 };
```

Listing 4: Zugriff auf Informationen über JUnit Testläufe

Zuerst wird angezeigt, wenn jemand einen Test gestartet hat, wie es in der Abbildung 20 zu sehen ist. Wird ein JUnit Test gestartet, wird entsprechend diese Information an den Orten angezeigt. Die Information über den Start wird deshalb mitgeteilt, da es auch passieren kann, dass ein Test mal etwas länger dauern kann, wenn jemand eine ganze Testsuite startet mit mehreren Tests. So wird der Sitzungspartner darüber informiert, wenn ein Test gestartet wurde, aber noch nicht beendet ist. Das Ergebnis des Tests wird ebenfalls mitgeteilt. Lief der Test erfolgreich durch, so wird als Ergebnis „SUCCESS“ mitgeteilt. Bei einem fehlgeschlagenen Test wird „FAILURE“ und bei einem fehlerhaften Test wird die Information „ERROR“ übertragen. Im Gegensatz zu den Dialogen und Views konnte bei JUnit Testläufen die Information nicht weiter verkürzt werden, sodass dieselbe Nachricht wie im Chat und in der Statusleiste auch

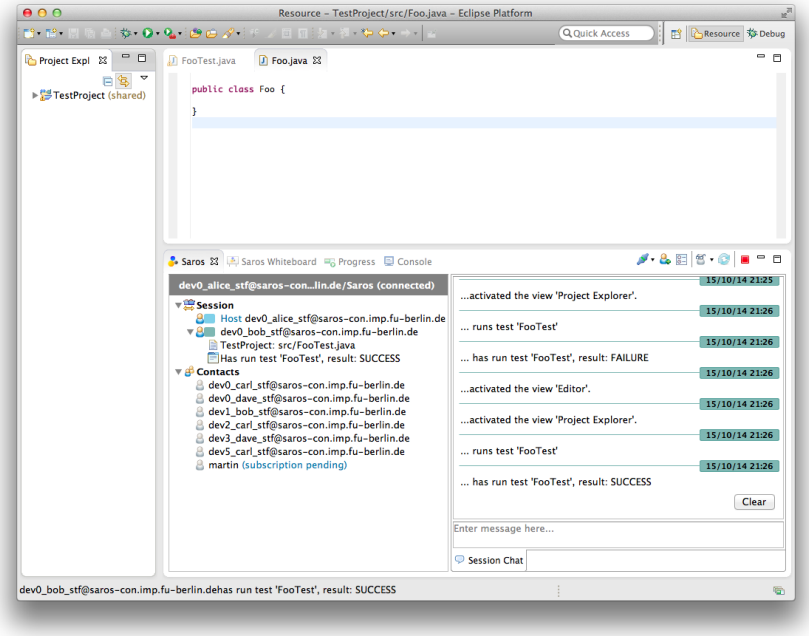


Abbildung 20: Anzeige über Start und Ergebnis eines JUnit Testlaufes

im Session Overview angezeigt wird. Dies hat zwar den Nachteil, dass der String ziemlich lang ist, aber er ist nicht viel länger als die Anzeige der aktuell offenen Datei im darüberliegenden Tree-Element.

4.5 Zweite Version des Prototyps

Dieses Kapitel beschreibt die finale Version der zweiten Version des ersten Prototyps, welcher dann auch in der Form in den Nutzertests verteilt wurde. Die Anzeigeorte sind gleich geblieben (siehe Kapitel 4.4.1). An welchen Stellen sich was geändert hat, wird in den entsprechenden Unterkapiteln im Detail erläutert.

4.5.1 Dialoge und Views

Abbildung 21 zeigt, wie Informationen zu offenen Dialogen übertragen werden. Hierbei wurde darauf geachtet, die Aktivität so zu formulieren, wie sie genau durchgeführt wurde. Der Nutzer öffnet einen Dialog, demnach wird diese Aktivität mit „...opened the dialog 'New'“ beschrieben. Allerdings ist das ein ziemlich langer String und nur im Chat und in der Statusleiste war dafür genügen Platz. In der Session Overview dagegen musste dies auf „Open dialog: New“ verkürzt werden. Ähnlich sieht es aus bei der Übertragung der Informationen zur aktiven View, wie es in Abbildung 21 zu sehen ist.

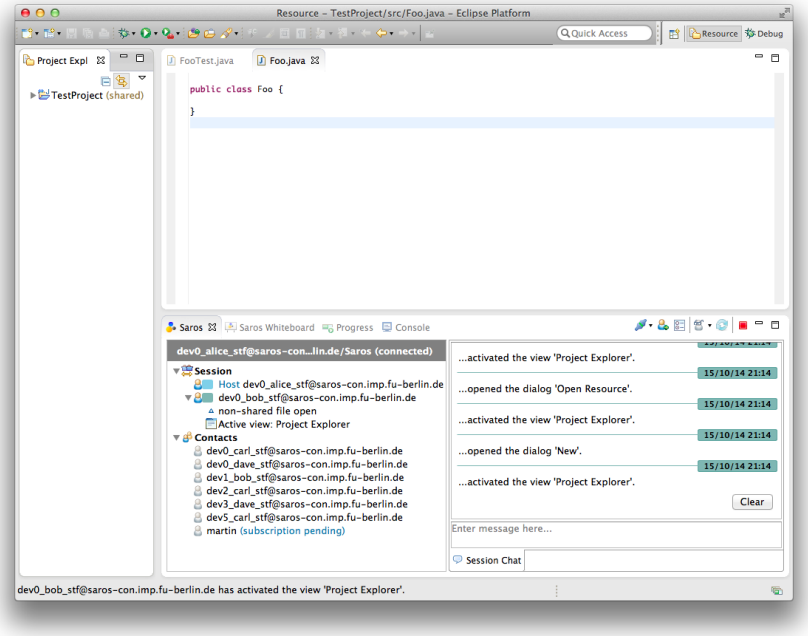


Abbildung 21: Anzeige von offenen Dialogen und aktiven Views

Im Chat und in der Statusleiste wird „...activated the view 'Package Explorer'“ angezeigt, während in der Session Overview nur „Active view: Package Explorer“ angezeigt wird. Die Implementierung unterscheidet sich nicht von der in der ersten Version des ersten Prototyps und kann in Kapitel 4.4.2 nachgelesen werden.

4.5.2 JUnit Testläufe

In Kapitel 4.4.6 wurde beschrieben, wie die Funktion zur Ermittlung und Anzeige von Informationen zu Ausführungen von JUnit Tests umgesetzt wurde. Diese Funktion blieb in dieser Form auch in der zweiten Version des ersten Prototyps erhalten und wird hier nicht mehr weiter besprochen.

4.5.3 Refactorings

Abbildung 23 zeigt, wie Information über durchgeführte Refactorings angezeigt werden. In Kapitel 4.4.5 wurde erläutert, warum es nicht möglich ist, die Information zum Zeitpunkt des Refactoring zu ermitteln, also während der Nutzer den Dialog offen hat und z. B. den neuen Namen für die Compilation Unit eingibt. Daher kann nur diese Information, wie sie in der Abbildung zu sehen ist, ermittelt werden. Diese Information wird vom `IRefactoringHistoryService` in dieser Form als reiner String zur Verfügung gestellt. Es

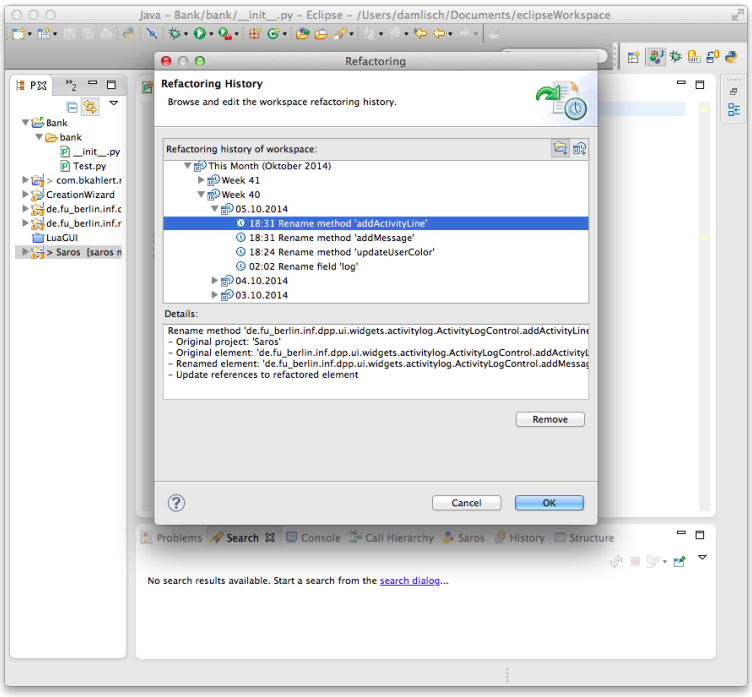


Abbildung 22: Die Refactoring History kann über den Refactoring Menüeintrag in Eclipse eingesehen werden und von dieser werden die Informationen vom Listener auch ermittelt.

handelt sich dabei um denselben Eintrag, der auch in der Refactoring Historie hinterlegt wird, wie es in Abbildung 22 zu sehen ist. Es war demnach nicht möglich, weitere Informationen in den String einzufügen, sodass dieser in der erhaltenen Form angezeigt werden musste.

```
1 private final IRefactoringExecutionListener
      refactoringListener = new
      IRefactoringExecutionListener() {
2      @Override
3      public void executionNotification(
           RefactoringExecutionEvent event) {
4          if (event.getEventType() ==
               RefactoringExecutionEvent.PERFORMED) {
5               String description = event.getDescriptor()
                    .getDescription();
6                fireActivity(new RefactoringActivity(
                     session.getLocalUser(), description
                     ));
7          }
8      }
9 };
```

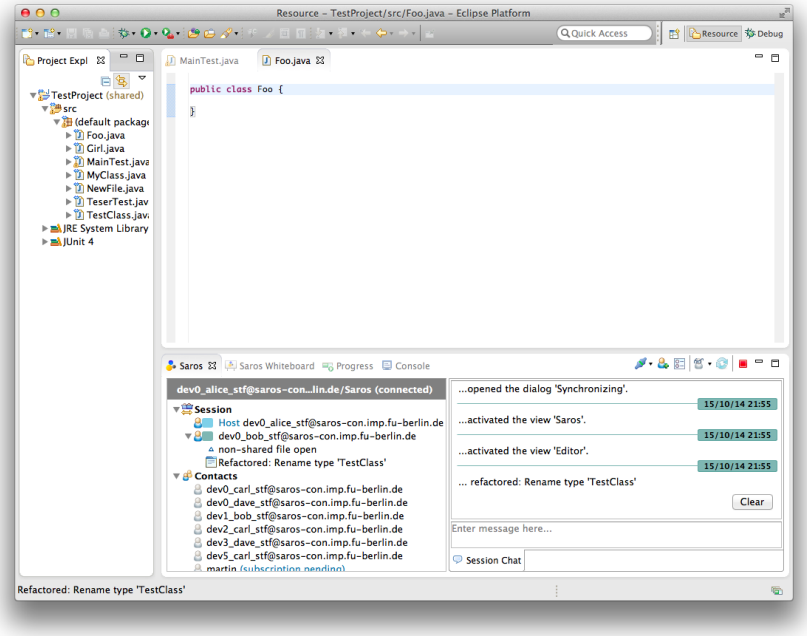


Abbildung 23: Anzeige über durchgeführte Refactorings und erzeugte Dateien

Listing 5: Zugriff auf das letzte Refactoring

Listing 10 zeigt den wichtigsten Teil der Umsetzung, und zwar die Stelle, wo letztendlich die Beschreibung für das letzte Refactoring ausgelesen wird (Zeile 5).

4.5.4 Erzeugte Datei

Die zuletzt entwickelte Funktion des neuen Prototyps ist die Anzeige der neu erzeugten Dateien wie es in Abbildung 24 zu sehen ist. Da die Ermittlung von detaillierteren Informationen aus Erzeugungsdialogen aus den genannten technischen Gründen (siehe Kapitel 4.4.3) nicht umgesetzt werden konnte, war die Idee, einfach die Information kurz nach der Nutzung des Erzeugungsdialoges zu nutzen. Dabei wird z. B. im Chat „...created the file NewFile.java“ angezeigt. Da nach Erzeugung einer Datei sofort in den Editor gesprungen wird, sieht man diese Information in der Statusleiste und in der Session Overview kaum bis gar nicht, weil es sich zu schnell aktualisiert.

Für die Umsetzung wurde in einem früheren Stadium der Entwicklung dieser Funktion der `IElementChangedListener`[19] verwendet. Allerdings kam

[19] Siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1829/3/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/awareness/FileCreationManager.java

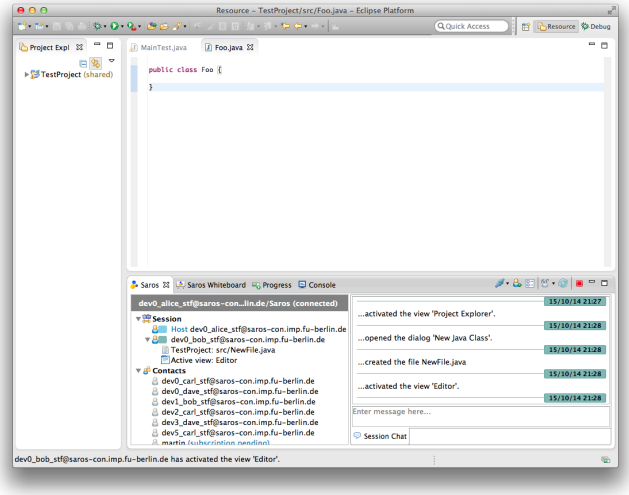


Abbildung 24: Anzeige über die erzeugte Datei

später im Reviewprozess der Hinweis, dass es in der Kernbibliothek von Saros bereits einen Mechanismus gibt, Dateiaktivitäten zu versenden, die u.a. für den Synchronisierungsprozess verwendet wurden. Von daher wurde diese Funktion später refaktorisiert und der bereits vorhandene Mechanismus umgesetzt[20]. Listing 13 zeigt, teilweise in Pseudocode, wie dies letztendlich umgesetzt wurde.

```
1  private IActivityReceiver receiver = new
       AbstractActivityReceiver() {
2          @Override
3          public void receive(FileActivity activity) {
4              User user = activity.getSource();
5              if (activity.getType() == Type.CREATED) {
6                  String file = activity.getPath().
                       getFile().getName();
7                  awarenessInformationCollector.
                       addCreatedFileName(user, activity.
                       getPath().getFile().getName());
8                  notifyListeners();
9              }
10         }
11     };
12 };
```

Listing 6: Zugriff auf den Namen der zuletzt erzeugten Datei

[20] Siehe http://saros-build.imp.fu-berlin.de/gerrit/#/c/1829/6/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/awareness/FileCreationManager.java

5 Evaluation des ersten Prototyps

Nachdem ich den ersten Prototyp entwickelt habe (siehe Abschnitt 4.5), musste dieser mithilfe von realen Nutzern evaluiert werden, um herauszufinden, ob die vom Prototypen angezeigten Informationen in ihrer Art und ihrem Umfang angemessen sind, also ob die Informationen an sich was gebracht haben und ob sie zu grob- oder zu feingranular waren.

Zuerst wird beschrieben, warum es ungemein wichtig war, den Prototypen mit realen Nutzern zu testen. Danach werden die Leitfragen aufgelistet, die zeigen, welche Fragen mithilfe des Nutzertests gelöst werden sollten. Anschließend wird die Aufgabenstellung, die den Nutzern vorgelegt wurden, dargestellt. Dabei wird auch geschildert, mit welchen Schwierigkeiten umgegangen werden mussten, um eine angemessene Aufgabenstellung zu finden. Am Ende eines Nutzertests wurde mit jedem Nutzer ein semi-strukturiertes Interview durchgeführt, dessen Leitfaden im Anschluss vorgestellt wird. Zum Schluss wird geschildert, wie der Nutzertest gestaltet wurde und welche Erkenntnisse aus diesen gewonnen werden konnten.

5.1 Problemstellung

Die Analyse des Videomaterials hat ergeben, dass es keine eindeutige Antwort darauf gibt, welche Action Awareness Informationen in welchem Umfang in Saros übertragen werden sollten. Einige Dialoge oder Views wurden öfter als andere benutzt, bei einigen wenig genutzten haben sich Schwierigkeiten durch fehlende Awareness Informationen ergeben, bei anderen viel genutzten IDE-Elementen haben sich wiederum keine Probleme erwiesen. Es würde also wenig Sinn machen, einfach alle möglichen Informationen über jegliche Interaktion mit der IDE zu übertragen (selbst wenn das technisch realisierbar wäre). Mehr zu dieser Problematik wurde bereits in Kapitel 3.5 erörtert.

Awareness wird subjektiv empfunden und was für einen Nutzer ein Problem darstellt, muss für den anderen nicht zwangsläufig ein Problem sein. Die Videos geben zwar Hinweise für mögliche Stellen in Saros, wo mehr Bedarf für Action Awareness Informationen vorhanden ist, aber das muss nicht grundsätzlich für alle Nutzer gleichermaßen gültig sein. Wenn z.B. über einige Dialoge Informationen übertragen werden, über den anderen jedoch nicht, würde das zwar einen Nutzer freuen, ein anderer könnte sich jedoch wundern, warum an manchen Stellen oder in manchen Situationen plötzlich keine Awareness Informationen übertragen werden. Das Verhalten von Saros würde also inkonsistent wirken. Außerdem geben die Sitzungsaufzeichnungen keine Hinweise zur Art und zum Umfang der erwünschten Awareness Informationen. Es wird also nicht klar, ob allein der Titel des IDE-Elements ausgereicht hätte oder ob mehr Details zur augenblicklichen Interaktion nötig gewesen wären.

Von daher ist es wichtig, dass der erste Prototyp mit realen Nutzern

getestet wird, um zu verstehen, ob die Art und der Umfang an Action Awareness Informationen, die von diesem übertragen werden, ausreichend sind oder ob es noch Verbesserungen gibt. In dieser Arbeit wurde dabei iterativ vorgegangen. Diese Vorgehensweise ermöglicht zum einen, frühzeitig Feedback einzuholen und zum anderen auch zu verstehen, wo Bedarf durch die Implementierung gedeckt wurde und wo nicht. Eine Iteration besteht dabei aus folgenden Schritten:

- Recherche zur *technischen Realisierbarkeit des Vorhabens* (Ist das überhaupt technisch möglich? Was kann ich von Eclipse abfangen? An welche Informationen komme ich nicht ran?).
- *Implementierung*, die auch den Review-Prozess beinhaltet (siehe Kapitel 7.1 zum Entwicklungsprozess in Saros).
- *Planung und Durchführung eines Nutzertests*, wozu auch die Konzipierung einer geeigneten Aufgabenstellung und Überlegung eines Interviewleitfadens gehört.
- *Analyse* der Beobachtungen und Ergebnisse des Nutzertests.

5.2 Leitfragen

Bei der Entwicklung einer geeigneten Aufgabe für die Nutzertests muss klar sein, was das Ziel der Nutzertests ist, was man wissen möchte und was man tun muss, um an die benötigten Informationen zu kommen. Von daher war es für mich ganz hilfreich, Fragen zu formulieren, die als Leitfragen gedient haben, um

a) eine angemessene Aufgabenstellung zu gestalten und

b) das Interview zu gestalten bzw. die Interviewfragen zu formulieren.

Folgende Fragestellungen waren bei der Suche und Entwicklung einer passenden Aufgabenstellung sowie der Ausarbeitung des Interviewleitfadens hilfreich:

- Was ist mein Ziel?
- Was will ich wissen?
- Was will ich sehen?
- Wodurch wird das sichtbar?
- Wie komme ich an die nötigen Informationen ran?
- Was werde ich messen?

- Was will ich an Daten herausschreiben?

Einige dieser Fragen sind sich ziemlich ähnlich und werden von daher bei der Beschreibung der Aufgabenstellung nicht einzeln beantwortet, sondern zu einer Frage zusammengefasst.

5.3 Aufgabenstellung

Dieses Kapitel beschreibt die Entwicklung der Aufgabenstellung. Dieser Prozess hat länger gedauert als ursprünglich geplant, denn die Aufgaben, die gefunden wurden, mussten selbst ausprobiert werden, um einschätzen zu können, ob diese zeitlich und inhaltlich in der geplanten Zeit durchführbar waren. Einige Aufgaben brachten einen zu hohen Einrichtungsaufwand mit, andere Aufgaben konnten erst gar nicht ohne Anpassung von Umgebungsvariablen eingerichtet werden.

5.3.1 Definition des Ziels

Durch die Nutzertests möchte ich herausfinden, welche Action Awareness Informationen erwünscht sind und in welchem Umfang diese angemessen sind. Es stellt sich also die Frage, ob es ausreicht, wenn ich zu allen Dialogen und Views den Titel übertrage, ob für die Dialoge und Views, die durch die Videoanalyse Bedarf für mehr Informationen aufgezeigt haben, mehr als nur den Titel übertragen werden soll und wie detailliert diese Informationen sein sollten. Zum Detailgrad kann ich allein aufgrund des Videomaterials keine Aussage treffen, da dieses keine Hinweise auf mehr oder weniger Details gegeben hat. Der Schwerpunkt dieses Nutzertests ist also: „Sind die Informationen im Umfang und in der Menge angemessen?“.

5.3.2 Benötigte Informationen

Dieser Abschnitt dient dazu, die Fragen *Was will ich sehen?* und *Was will ich wissen?* zu beantworten. Vor allem für die Entwicklung des Interviewleitfadens (siehe Abschnitt xx) waren die folgenden Überlegungen sehr hilfreich:

- Waren die übertragenen Action Awareness Informationen ausreichend?
- Wurde die Anzeige der Awareness Informationen sofort wahrgenommen oder musste danach gesucht werden?
- War die Konfiguration sinnvoll (nur Titel oder Titel plus weitere Informationen für ausgewählte Elemente)?
- In welchen Situationen haben die Informationen am meisten geholfen?
- Gab es Situationen, wo Informationen benötigt wurden, die nicht übertragen wurden?

- Was war störend? Wenn ja, warum? Wie hätte es besser sein können?
- Haben die Informationen im gegebenen Umfang ausgereicht oder wann waren mehr Details erforderlich?
- Zu welchen Views/Dialogen waren die Informationen besonders hilfreich? Zu welchen waren sie überflüssig? Warum? Haben Dialoge/Views gefehlt?
- Wurden sowohl im Following- als auch im Nicht-Following-Modus diese Informationen benötigt?
- Würden sie es je nach Arbeitsmodus konfigurieren?
- Waren es zu viele Informationen?
- Würden die Nutzer gerne mehr/weniger Kontrolle darüber haben, ob und was angezeigt wird?
  - Benutzer konfiguriert eigenes System (volle Kontrolle)
  - Benutzer wählt Profile aus (Kompromiss)
  - Benutzer hat keine Auswahl, Experimente geben vor (keine Kontrolle)
- Haben sie noch spezielle Wünsche?
- War der Anzeigeort hilfreich? Welcher gefällt ihnen mehr?

5.3.3 Vorgehen

Dieser Abschnitt dient dazu, die Fragen *Wodurch wird das sichtbar?* und *Wie komme ich an die nötigen Informationen ran?* zu beantworten. Es ist also wichtig zu wissen, was im Nutzertest getan werden muss, damit genau die Informationen generiert werden, die nötig sind, um die Fragen aus Abschnitt 5.3.2 zu beantworten.

Beim Erfüllen der Aufgaben sollen die Nutzer „laut denken“, da das auch eine Form der Beobachtung bzw. Messung ist. Um zu sehen, ob die Funktionen, die der erste Prototyp umsetzt, auch sinnvoll sind, müssen die Aufgaben so gestaltet sein, dass gerade auch diese Funktionen in direkter oder indirekter Weise bei der Bearbeitung der Aufgaben genutzt werden. Von daher soll die Aufgabe folgendermaßen gestaltet sein, um die Art Information sichtbar zu machen, die man für die Beantwortung der Fragen benötigt:

- Gegeben ist ein kleines, überschaubares Projekt mit zwei bis drei Unterverzeichnissen. Die erste Aufgabe besteht darin, durch das Projekt zu navigieren (z.B. mit der Suche). Diese Aufgabe dient dazu, die Struktur des Projekts besser zu verstehen. Außerdem wird dabei vermutlich der

Suchdialog oder andere Dialoge, wie die Open-Type-Dialoge verwendet. Dabei kann gesehen werden, ob die Nutzer die Action Awareness Informationen wahrnehmen.

- Anschließend sollen die Nutzer JUnit-Tests laufen lassen. Dabei werden sie feststellen, dass einige nicht funktionieren. Nun besteht die Aufgabe darin, dass sie gemeinsam den besagten JUnit-Test korrigieren, unabhängig des Arbeitsmodus (das können sie selbst entscheiden). Am Ende wird der Test nochmals ausgeführt und es wird funktionieren. Selbst wenn es nicht funktioniert, spielt das für die weitere Analyse keine Rolle. Wichtig ist zu sehen, ob sie die dafür übertragenen Informationen über Testläufe benötigt haben und ob diese in ihrem Umfang hilfreich waren. Hierbei soll vor allem herausgefunden werden, ob die Übertragung von Informationen über Testläufe überhaupt sinnvoll ist. Die Videoaufzeichnungen haben zwar Probleme durch mangelnde Awareness-Informationen aufgezeigt und Test-Driven-Development wäre eine gute Argumentation für die Anzeige solcher Informationen, aber vielleicht könnte es den einen oder anderen Nutzer geben, der diese Funktion hinterfragen würde.

- Beim Anlegen neuer Tests wird auch der Erzeugungsdialog verwendet. Eventuell je nach Aufgabentyp kann auch eine neue Klasse oder ein neues Interface erzeugt werden. Mithilfe dieser Aktivitäten kann ersichtlich werden, ob die Funktion des Prototyps, die erzeugte Dateien anzeigt, sinnvoll ist oder ob was an der Art oder dem Umfang geändert werden sollte.

- Weiterhin sollen kleinere Codeabschnitte refaktorisiert werden. Dadurch kann ermittelt werden, ob die Nutzer die angezeigten Awareness Informationen zu Refaktorisierungen in Art und Umfang wahrnehmen und ob diese ausreichend sind.

5.3.4 Entwicklung der Aufgabenstellung

Die Beschreibung des Vorgehens aus dem letzten Abschnitt verdeutlicht, welche Aktivitäten von den Nutzern erwünscht sind und wie der Bedarf geweckt wird, dass die Nutzer gerade diese ausüben. Die Nutzer müssen zuerst etwas verstehen, eventuell dazu im Projekt navigieren und müssen dann den bestehenden Code (egal ob den fremden oder eigenen Code) verbessern. Anschließend müssen sie ihre Lösung testen und dazu neue Ressourcen erzeugen. Die Aufgabe darf dazu nicht zu aufwändig sein, da dann zeitlich nicht alles machbar ist und somit manche Aktivitäten nicht mehr aufgezeichnet werden können. Genauso darf die Aufgabe nicht zu einfach sein, da sonst die Gefahr besteht, dass zu wenig Informationen bei der Beobachtung der Nutzer generiert werden, weil z.B. Awareness Informationen gar nicht mehr

benötigt werden. Zudem muss der Einrichtungsaufwand zu Beginn des Tests so gering wie möglich und so hoch wie nötig sein. Wenn z.B. eine Aufgabe aus dem Bereich der Webapplikationen vorgelegt wird, müssten die Nutzer einen lokalen Webserver und evtl. eine Datenbank einrichten, was einfach nicht Hauptbestandteil der Aufgabenstellung sein sollte.

Ursprünglich habe ich für das Finden und für die Gestaltung einer Aufgabenstellung eine Woche eingeplant, allerdings erwies sich die Aufgabenentwicklung als schwieriger als gedacht. Gründe dafür waren zum einen, dass einige gefundenen Aufgaben zu einfach waren, andere wiederum zu schwer. Auch musste ich die gefundenen Aufgaben erst einmal selbst ausprobieren, um einschätzen zu können, ob sie vom (zeitlichen) Aufwand und von der Komplexität angemessen waren. Mehr dazu ist in Kapitel 7.3 nachzulesen. Im Folgenden soll kurz aufgeführt werden, welche Aufgaben zur Debatte standen, warum sie verworfen wurden und warum die letzte Aufgabe letztendlich ausgewählt wurde.

Apache Commons CLI

Bei *Apache Commons CLI* handelt es sich um eine Bibliothek, die eine API zum Parsen von Argumenten auf die Art und Weise anbietet, wie man das von einer Kommandozeile her kennt [com14]. Im Issuetracker des CLI 1.0 Projekts befinden sich einige interessante Bugmeldungen, die potentiell als Aufgabenstellung in Frage kämen. Der Eintrag 95[21] beschreibt dabei ein Feature-Request, bei dem der Validator des Parsers Kommandozeilenargumente auch gegen einen regulären Ausdruck validieren könnte. Eine andere Bugmeldung Nr. 184[22] beschreibt einen Bug, bei dem ein negatives Argument für einen Kommdanozeilenparameter als eigener Parameter interpretiert wird[23]. Bei einer Aufgabenstellung, in dem solch ein Ticket bearbeitet wird, müsste Quelltext bearbeitet und dieser mithilfe von JUnit-Tests getestet werden. Eine anfängliche Navigation durch das Projekt zur Einarbeitung wäre erforderlich und eine Refaktorisierung von Code hatte man einbauen können.

Beim Selbsttest kamen jedoch Probleme auf. Da es sich um ein Maven-Projekt handelte, musste es nach dem Auschecken gebaut werden und das erwies sich zumindest auf Mac OSX als Problem. Manuell mussten Umgebungsvariablen gesetzt werden (maven war installiert gewesen) und an einigen Stellen im Projekt mussten angepasst werden. Auf diversen Internetseiten gab es zwar Anleitungen, wie das behoben werden konnte, jedoch war somit der Einrichtungsaufwand erheblich höher. Außerdem war die Bearbeitungsdauer nicht ganz gering. Ein eigener Test und ein weiterer Test eines in der Saros-Gruppe tätigen Mitarbeiters hat ergeben, dass die Komplexität dieser

[21] https://issues.apache.org/jira/browse/CLI-95
[22] https://issues.apache.org/jira/browse/CLI-184
[23] Wurde mittlerweile behoben

Aufgabe zu hoch wäre und somit wurde sie nicht mehr berücksichtigt.

Platforms

Eine weitere Idee war die Bearbeitung eines Datenmodells im *Platforms*-Projekt, eine MVC-Anwendung, welches von einem wissenschaftlichen Mitarbeiter der Arbeitsgruppe vorgestellt wurde[24]. Dort wäre eine Aufgabe gewesen, einem Domäne-Modell ein weiteres Feld hinzuzufügen und das dann mithilfe von JUnit-Tests zu testen. Dabei wäre eine Suche im Projekt zur Einarbeitung und die Ergebnisse wären in einem Browser sichtbar. Jedoch war für diese Aufgabe vorausgesetzt, dass eine Datenbank und ein lokaler Webserver eingerichtet werden muss und da das einen zu hohen Einrichtungsaufwand bedeuten würde, wurde diese Idee am Ende verworfen.

Saros Bugs und Core - Verschiebung

Neben Platforms oder Apache Commons CLI gab es noch die Idee, Bugs im Saros-Projekt zu lösen. Der Vorteil wäre, dass wenn die Tester bereits Entwickler im Saros sind, der Einarbeitungsaufwand somit minimiert wird. Gleichzeitig profitieren die Mitglieder des Saros-Teams davon, dass weitere Lücken behoben werden. Für diesen Zweck wurden auch einige Einträge im Issue-Tracker von Saros ausgesucht[25][26][27]. Jedoch stand schon fest, dass mindestens einer der Tester im Nutzertest ein Nicht-Saros-Entwickler sein wird und es daher nicht einzuschätzen war, wie hoch der Einarbeitungsaufwand für dieser Tester sein wird. Und da nicht Aspekte der Paarprogrammierung während einer potentiell langen Einarbeitungszeit beobachtet werden sollte, sondern die intensive Nutzung des Prototyps, kam diese Aufgabe nicht in Frage. Aus denselben Gründen kam es ebenfalls nicht in Frage, eine Aufgabe in dem Bereich Core-Verschiebung zu entwickeln. Bei dieser *Verschiebung* handelt es sich um die Verschiebung von Eclipse-IDE-unabhängigen Klassen aus Saros in ein eigenständiges Kern-Projekt, dem Core, damit in Zukunft die Unterstützung neuer IDEs nicht die fast komplette Neuentwicklung von Saros benötigt wird, sondern nur noch der Core importiert werden muss und alles IDE spezifische darüber aufgesetzt werden kann.

JUnit 4 - Behebung eines gelösten Bugs und Refactoring

Zwischen den oben erwähnten Aufgabenstellungen gab es immer wieder kleinere Aufgaben und Ideen zur Weiterarbeit an öffentlichen Github-Projekten. All diese wurden ebenfalls verworfen, weil sie wiederum zu einfach waren.

[24] Mehr Informationen dazu bestehen leider nicht, da Rechte zum Zugang zum SVN-Repository fehen

[25] http://sourceforge.net/p/dpp/bugs/765/

[26] http://sourceforge.net/p/dpp/bugs/756/

[27] http://sourceforge.net/p/dpp/bugs/462/

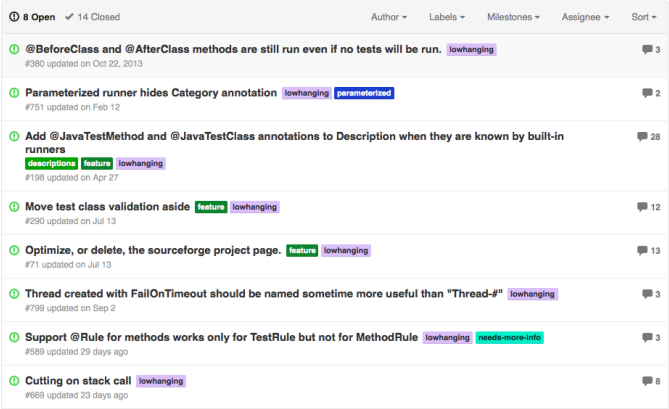


Abbildung 25: Wenn man als Entwickler bei JUnit 4 mitmachen möchte, werden einem die Issues mit dem Label lowhanging empfohlen, die als einfach lösbar eingestuft werden und mit wenig Gedanken über Architektur und Design zu lösen sein sollen.

Meistens konnte man diese innerhalb von 10 bis 15 Minuten lösen und selbst wenn die Tester mehr dafür gebraucht hätten, bestand hier das Risiko, dass die Tester „zu schnell“ mit der Bearbeitung fertig wären und den Prototyp somit nicht sehr intensiv genutzt hätten.

Irgendwann wurde das *JUnit 4* auf Github[28] entdeckt und dort gab es im Issue-Tracker den Vorteil, dass es unter anderem das Laben *lowhanging* gab (siehe Abbildung 25). Einträge mit solchen Labels bedeuten, dass sie keine großen und komplexen Gedanken über Architektur und Design erfordern und quasi als Einstieg in das Projekt dienen. Die Aufmerksamkeit fiel vor allen auf die Issues 380[29] und 848[30]. Beim Ersteren handelt es sich um einen Fehler, bei dem selbst bei einer auf `@Ignore` gesetztes Testklasse die `@BeforeClass`- und `@AfterClass`-Methoden ausgeführt werden. Beim Letzteren handelt es sich lediglich um ein Refactoring von Instanzvariablen, da diese alle mit dem Präfix „f“ beginnen und das die Lesbarkeit des Codes nicht fördert.

Auch diese Aufgabe wurde einmal selbst durchgespielt und zusammen mit der Einrichtung, Einarbeitung und dem Lösen der Aufgaben hat das etwa 45 Minuten gedauert. Ein realer Tester im Nutzertest würde ein Dokument erhalten, wo die Schritte zur Einrichtung erläutert sind. Zusammen mit dem Lesen der Aufgabenstellung und dem Absprechen mit dem Sitzungspartner würde diese zusätzlich Zeit eine ganze Stunde füllen. Für die Bearbeitung dieser Aufgaben müssen die Tester

- das Projekt einrichten,

[28] https://github.com/junit-team/junit
[29] http://github.com/junit-team/junit/issues/380
[30] http://github.com/junit-team/junit/issues/848

- sich einarbeiten und dabei z.B. die Suche verwenden oder im package Explorer navigieren,
- neue Tests erzeugen und schreiben,
- Tests ausführen und
- den Code refaktorisieren.

Indem diese Aufgaben bearbeitet werden, wird dabei der erste Prototyp intensiv genutzt und die Tester generieren dabei die Informationen, die benötigt werden, um die Fragen aus Abschnitt 5.3.2 zu beantworten.

Die erste Version der Aufgabenstellung hatte ausdrückliche Informationen darüber, wer die Aufgaben bearbeitet und wer sich dabei im follow-Modus befindet. Der Grund war, dass so beobachtet werden sollte, ob der Mehrwert des Prototyps je nach Arbeitsmodus unterschiedlich war. Allerdings war damit jede Handlung bis ins kleinste Detail vorgegeben und wenn die Nutzer nur damit beschäftigt wären, die Teilschritte abzuarbeiten, wäre damit kein natürliches Verhalten beobachtbar. Daher wurde das aufgelöst und es wurde nur die Einrichtung, die Aufgabe und das Ziel in einem groben Überblick gegeben, ohne eine feste Reihenfolge für die Bearbeitungsschritte zu geben. Die finale Aufgabenstellung ist im Anhang A vorzufinden. Für den Test wurde ein Repository auf Github angelegt, wo alle nötigen Materialien hinterlegt wurden.

5.4 Interviewleitfaden

In Abschnitt 5.3.2 wurde beschrieben, was mit Hilfe der Nutzertests herausgefunden werden sollen. Die nächste Überlegung ist, wie ein Interview gestaltet werden kann bzw. was für Fragen den Nutzern gestellt werden können, um diese Informationen zu erhalten.

Das Hauptziel des Interview ist herauszufinden, ob die im ersten Prototypen übertragenen Informationen in Art und Umfang sinnvoll bzw. hilfreich sind und was noch verbessert werden kann. Das Interview soll dabei semi-strukturiert gestaltet werden. Das hat den Vorteil, dass die Frageliste nicht starr abgearbeitet werden muss, sondern Fragen auch weitere Fragen auslösen können, auf die reagiert werden kann und je nach Antwort sich auch Fragen ändern können, sodass diese Antwort als Leitfaden für neue Fragen dienen kann [Dre95].

Die Art und Reihenfolge der Fragen in einem Interview kann ausschlaggebend für den Erfolg oder Misserfolg des Interviews sein [Nor11]. Durch nicht durchdachte Fragen, eine zufällig erscheinende Reihenfolge oder ungünstige Arten der Fragen kann das Interview langweilig wirken, die Antworten können wenig Informationen liefern und insgesamt kann die nachfolgende qualitative Analyse des Interviews erschwert oder gar unmöglich gemacht werden.

In [Nor11] wird vorgeschlagen, unterschiedliche Arten von Fragen zu stellen, um das Problem oder das Thema aus unterschiedlichen Perspektiven zu betrachten. In der folgenden Liste sind einige Beispiele, die auch im Leitfaden (siehe **??**) wiederzufinden sind:

- **Quantität** Wie oft hast du die Funktion nutzen müssen?
- **Beispiel** Stell dir vor, du siehst Information X: Was würdest du danach machen?
- **Auflistung** Was für weitere Arten von Informationen fallen dir noch ein, die hier fehlen?

Nicht alle Fragen aus [Nor11] eignen sich für jede Art von Interview, da die Art der Fragen auch abhängig vom Kontext sein kann. Nichtsdestotrotz bieten sie eine Inspiration für eigene Fragen und für die sinnvolle Formulierung und Strukturierung dieser.

Die Reihenfolge der Fragen ist ebenfalls essentiell, da bei einer unstrukturierten Reihenfolge die Fragen zufällig gestellt wirken können, die Interviewten sich vielleicht zu Beginn mit komplizierten Fragen überrumpelt fühlen könnten oder das Gefühl bekommen können, dass ihre Antworten keine Bedeutung haben, da nachfolgende Fragen keinen konkreten Zusammenhang zur vorigen Frage hatten. Von daher eignet es sich, zu Beginn des Interviews einfache Fragen zu stellen, um den Teilnehmer aufzuwärmen. Anschließend können die Fragen in einer inhaltlich sinnvollen Strukturierung gestellt werden und Schritt für Schritt komplexer werden. In Anhang B wird der Interviewleitfaden vorgestellt, wie er im Nutzertest der ersten Iteration benutzt wurde.

5.5 Ablauf des Nutzertests

Schon während der Fertigstellung des ersten Prototyps wurde die Einladung zum Nutzertest (siehe Anhang C) verschickt. Insgesamt haben sich 5 Tester gemeldet. Außer einem Teilnehmer waren die vier anderen Teilnehmer Saros-Entwickler, wobei drei von ihnen Abschlussarbeitsabsolventen waren und einer ein wissenschaftlicher Mitarbeiter der AG Software Engineering war. Da eine Testgruppe jedoch immer aus zwei Teilnehmern bestand und keine sechste Person aufzufinden war, wurden letztlich zwei Nutzertests durchgeführt und geplant war, dass wenn nach diesen beiden Nutzertests und der Durchführung von Interviews mit jedem Teilnehmer keine Tendenzen festzustellen wären, nach weiteren Teilnehmern unter den Studenten des Fachbereichs und aus dem beruflichen Umfeld gesucht werden sollte.

Nielsen et al. haben in [NL93] gezeigt, dass im Schnitt mit 5 Teilnehmern 80% Usability-Probleme in einem Nutzertest abgedeckt werden kann, wobei anzumerken sei, dass bei diesem Wert die Standardabweichung höher ausfiel. Faulkner hat in [Fau03] jedoch gezeigt, dass es nicht immer auf die Anzahl der Teilnehmer ankommt, sondern auch auf die Erfahrungen und Fähigkeiten

der Nutzer ankommt. Die vier Teilnehmer hatten folgende Fähigkeiten und Erfahrungen vorzuweisen:

- Der erste Teilnehmer schreibt gerade seine Bachelorarbeit im Saros-Team und hat geringe Erfahrung mit Paarprogrammierung.
- Der zweite Teilnehmer schrieb ebenfalls seine Bachelorarbeit im Saros-Team und hat mit dem ersten Teilnehmer zusammen einige Male Paarprogrammierung betrieben.
- Der dritte Teilnehmer schreibt seine Masterarbeit im Saros-Team und hat zwar geringe Erfahrung mit Paarprogrammierung, jedoch hat er bereits intensiver an Saros und der Konfiguration der kontinuierlichen Integration von Saros gearbeitet.
- Der vierte Teilnehmer schreibt gerade seine Masterarbeit und hat bisher nur wenige Male Saros zum Kennenlernen verwendet, hat aber bereits öfters Paarprogrammierung betrieben.

Von allen Teilnehmern habe ich das Einverständnis zur Aufzeichnungen ihres Tons, ihres Audios und Videos sowie ihres Bildschirms eigeholt.

Zur Aufzeichnung der Nutzertests wurde ein Rechner der AG Software Engineering zur Verfügung gestellt. Nachdem die Termine für die Nutzertests feststanden, wurden in *Adobe Connect*, eine Softwarelösung zur Durchführung von Webmeetings, zwei Meetingräume erstellt und den entsprechenden Teilnehmern wurde jeweils eine URL zum Meetingraum zugestellt. Als die Teilnehmer den Meetingraum betreten haben, musste sie ihr Audio, ihr Video und ihren Bildschirm freigeben. Gleichzeitig waren die Teilnehmer untereinander per Skype miteinander verbunden. Die Nutzertests wurden mithilfe von *Camtasia Studio 8*, welches auf dem zur Verfügung gestellten Rechner installiert war, aufgezeichnet. Am Ende des einstündigen Nutzertests wurde noch mit jedem Teilnehmer separat ein Interview (siehe Abschnitt 5.4) über Skype geführt und dieses wurde ebenfalls mit denselben Einstellungen und beschriebenem Vorgehen aufgezeichnet. Die Aufzeichnungen der Nutzertests sowie der Skype-Interviews sind auf dem mit der Arbeit abgegebenen Datenträger vorzufinden.

5.6 Auswertung der Nutzertests

Die vorigen Abschnitte haben die Aufgabe, die Struktur des Interviews sowie die Planung und Durchführung der Nutzertests zu beschrieben. Folglich geht es in diesem Kapitel um die Auswertung der Nutzertests und der gesammelten Daten.

5.6.1 Vorgehen

Bei der Auswertung der erhobenen Daten wurde ähnlich wie bei qualitativen Analyse des Videomaterials (siehe Kapitel 3) vorgegangen. Es wurde bewusst auf die Transkribierung des Videomaterials sowie der Interviews aus denselben Gründen wie für die Analyse der Saros-Aufzeichnungen (siehe Kapitel 3.2) verzichtet, da bereits beim Versuch ein enormer Zeitaufwand festzustellen war und der Ansatz vorgezogen wurde, das Material durch zuschauen und zuhören auszuwerten. Da die interessanten Stellen sowohl in den Videos als auch in den Interviews auch durch reine Notizen ebenso gut festgehalten werden konnten, wurde dieser Ansatz letztendlich auch weiterhin befolgt.

Die Vorgehensweise bei der Auswertung der erhobenen Daten bestand aus folgenden Schritten:

1. **Beobachtung** der Aufzeichnungen des Nutzertests und Festhalten aller Situationen bei der Nutzung des Prototyps, die das Verhalten bei der Nutzung beschreiben oder Hinweise auf Wünsche, Anregungen oder Probleme darstellen.

2. **Kategorisierung** der notierten Beobachtungen zusammen mit den Notizen aus den Interviews und Zuordnung der Beobachtungen und Notizen zu den gefundenen Kategorien.

3. **Auswertung** der Ergebnisse für die Entwicklung eines Konzepts für einen zweiten Prototypen.

5.6.2 Beobachtung

Die notierten Beobachtungen sind im Anhang D nachzulesen. Es muss gesagt werden, dass die Beobachtungen keine großen Erkenntnisse gebracht haben. Es gibt aber eindeutige Hinweise darauf, dass die übertragenen Action Awareness Informationen konsumiert wurden, unter anderem:

- **Video 1, Zeit 10:43** zeigt, dass eine Person über die Anzeige mitbekommen hat, dass beim Sitzungspartner der JUnit Test nicht erfolgreich durchlief. Der Nutzer hat dabei auch den Test nicht selbst durchlaufen lassen.

- **Video 1, Zeit 11:03** zeigt, dass ein Nutzer nach dem Starten eines JUnit-Tests in die JUnit-View manuell wechselt. Sein Sitzungspartner registriert das über die übertragenen Informationen und klärt dem Nutzer auf, dass sich nach Start eines JUnit-Tests die entsprechende View automatisch öffnet.

- **Video 1, Zeit 23:09** zeigt, dass beide ihren aktuellen Stand der Implementierung mithilfe der Ausführung der JUnit-Tests validieren möchten. Dabei starten ein Nutzer den Test und dieser läuft erfolgreich durch.

Der andere Nutzer registriert das über die übertragenen Informationen und sagt dann auch „Ah, succeeds“ und markiert dabei die Anzeige.

- **Video 2, Zeit 14:11** zeigt, dass ein Nutzer seine Aufmerksamkeit vor allem der Anzeige in der Chat-Ansicht der Saros View widmet. Er stellte fest, dass dort eine Information zwei Mal übertragen wird, was natürlich ein Fehler im Prototyp ist. Das war kein ernst zu nehmendes Problem, da es sich letztendlich um einen Prototypen handelt, der sowohl von Iteration zu Iteration als auch innerhalb einer Iteration während des Reviewprozesses stetig verbessert wird. Dass dies jedoch aufgefallen ist, zeigt dagegen, dass auch hier die Anzeige der übertragenen Informationen genutzt wurde.
- **Video 2, Zeit 19:27** zeigt, dass der Nicht-Saros-Entwickler während der Nutzung Feedback zur Übertragung von Action Awareness Informationen gibt. Dieser empfindet die neue Funktion als hilfreich und nimmt die Anzeige in der Chat-Ansicht auch als eine Art Chronik wahr.

Diese Liste ist nicht vollständig und es gibt weitere Beobachtungen, die darauf hinweisen, dass die Informationen in der Anzeige ausgelesen werden. Nichtsdestotrotz zeigen die Beobachtungen keine wirkliche Tendenz in Bezug auf Art und Umfang der angezeigten Informationen. Es hat sich lediglich eine leichte Tendenz im Bezug auf den Anzeigeort gezeigt, da an einigen Stellen deutlich wurde, dass der Chat als Chronik verwendet wurde und zum anderen die Nutzer manchmal nicht mehr nach dem Ergebnis der Tests gefragt haben, weil sie es bereits mitgeteilt bekommen haben. Allerdings wurden die Beobachtungen nicht mit der Erwartung festgehalten, dass diese bereits eine Tendenz zeigen sollten. Dafür waren es dann auch zu wenige Nutzertests und demnach zu wenig Beobachtungen, dass daraus bereits feste Kategorien gebildet werden konnten. Da dazu Awareness auch ein subjektiv erlebtes Phänomen ist, waren für die weitere Auswertung der Nutzertests die Analyse der geführten Interviews notwenig.

5.6.3 Kategorisierung

In [You08] wird für diesen Schritt empfohlen, die Interviews zu transkribieren. Allerdings war schon nach kurzer Zeit klar, dass das einen zu hohen Zeitaufwand erfordert. Von daher wurden die Interviews gezielt angehört und auf Stellen untersucht, an denen bemerkbar wurde, dass Feedback zur Nutzung und zur Erfahrung mit dem Prototyp erwähnt wird. Diese Stellen wurden zusammen mit den Beobachtungen aus dem vorigen Schritt in Verbindung gebracht und in Kategorien zusammengepackt. Bei jedem weiteren gefundenen Hinweis wurde dies entweder in eine bestehende Kategorie eingefügt oder es wurde eine neue Kategorie erzeugt und Punkte aus anderen Kategorien wurden entweder mit der neuen Kategorie in Beziehung gesetzt oder wurden

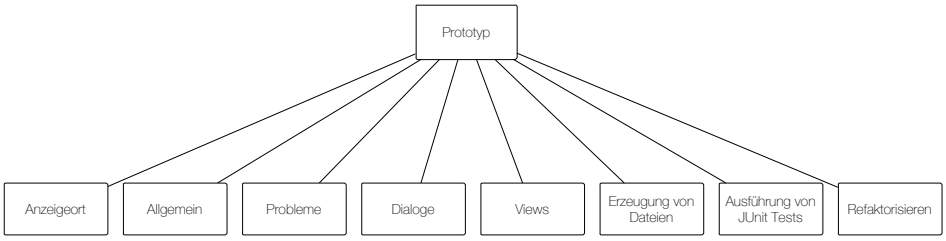


Abbildung 26: Bei der Auswertung der Aufzeichnungen der Nutzertests und den Interviews sind acht Kategorien entstanden.

komplett in die neue Kategorie übertragen. Hier wurden also die Techniken der Grounded Theory, wie sie in Kapitel 2.3.2 beschrieben wurden, angewendet. Insofern wurde in jedem Schritt jede Kategorie erneut reflektiert und eventuell angepasst.

Abbildung **??** zeigt die bei der Auswertung der Aufzeichnungen der Nutzertests und der Interviews entstandenen Kategorien:

Anzeigeort Hier wurden alle Anmerkungen, Beobachtungen und Hinweise zugeordnet, die sich auf den Ort der Anzeige bezogen haben. Hier wurden zum einen das direkte Feedback der Nutzer einsortiert. Weiterhin wurden in dieser Kategorie auch Beobachtungen eingeordnet, die auf die Wahrnehmung und Erfahrungen mit dem Anzeigeort hingewiesen haben.

Allgemein Diese Kategorie beschreibt Aussagen und Beobachten, die zeigen, dass die Unterscheidung zwischen Following- und Nicht-Following-Modus nicht erwünscht ist oder dass die Nutzer keine Konfiguration oder Filterung der Informationen haben möchten.

Probleme Hier wurde nur eine Beobachtung festgehalten, und zwar ist jedes mal beim Refactoring Eclipse bei den Nutzern eingefroren. Der damalige Verdacht war, dass beim Zugriff auf die Refactoring History von Eclipse (die vom Prototypen genutzt wurde, siehe Kapitel 4.5.3) ein Problem entstanden sein muss, was Eclipse zum Einfrieren brachte. Zu einem viel späteren Zeitpunkt stellte sich jedoch heraus, dass es sich um einen Bug in Saros handelt, der zum Vorschein kommt, wenn ein Klassenfeld über die Tastenkombination „Alt+Schift+R“ refaktorisiert wird[31]. Da das damals allerdings nicht bewusst war und als ein ernst zu nehmendes Problem betrachtet wurde, wurde das in eine eigene Kategorie zugeordnet.

Dialoge Hier sind alle Anmerkungen, Beobachtungen und Feedback enthalten, die etwas mit der Anzeige von Dialogtiteln zu tun hatten. Inter-

[31] http://sourceforge.net/p/dpp/bugs/836/

essant war vor allem, dass die Anzeige der Titel nicht immer benutzt wurde oder manchmal nur unterbewusst wahrgenommen wurde, aber wenn sie benötigt wurden, haben sich die meisten Nutzer sogar einige zusätzliche Informationen gewünscht.

View Alle Beobachtungen und Berichte, die etwas mit der Anzeige der derzeitig aktiven View zu tun hatten, wurden in diese Kategorie eingeteilt. Bei dieser Kategorie gab es kein großes Feedback. Viel mehr haben die Beobachtungen in den Aufzeichnungen gezeigt, dass die Information über die aktive View einige Male konsumiert wurde (siehe Abschnitt 5.6.2). Außerdem wurde in den Interviews angemerkt, dass es einige Male genervt hat, wenn der Sitzungspartner schnell zwischen mehreren Views gewechselt ist. Allerdings konnte diese Person nicht klarstellen, ob es an dem Notification-Sound lag, der jedes Mal abgespielt wurde (und später auch ausgeschaltet wurde) oder schlich an der hochfrequent aktualisierten Chat-View.

Erzeugung von Dateien Diese Kategorie ist auch ziemlich klein, da die Nutzer diese Funktion als gut empfanden und nur ein Nutzer dazu gesagt hat, dass es in solchen Situationen erst interessant sein könnte, wenn die Erzeugung einer Datei im eigenen Workspace irgendwelche Konflikte auslösen könnte. Der Nutzer hat aber auch gleichzeitig angemerkt, dass das technisch nicht möglich sei, da außerhalb des geteilten Bereiches dem anderen Nutzer nicht zur Verfügung gestellt wird und somit diese Bedingung dann keinen Mehrwert bringt.

Ausführen von JUnit Tests Hier handelt es sich um Hinweise, Berichte und Beobachtungen im Bezug auf die Übertragung von Informationen zum Starten und dem Ergebnis von JUnit-Tests. Auch hier wurde das Feature als hilfreich wahrgenommen und auch negatives Feedback gab es keines. Außerdem ist mehrmals der Begriff „Test-driven Development“ gefallen, was gezeigt hat, dass die Nutzer auch über den Mehrwert dieser Funktion in der Zukunft nachgedacht haben. Lediglich der Wunsch wurde geäußert, dass wenn ein Test fehlschlägt, der Stacktrace entsprechend dem Sitzungspartner ebenfalls vorgezeigt wird, damit dieser sofort sehen kann, was die Fehlermeldung aussagt, ohne dabei den Test selbst auszuführen.

Refaktorisieren Diese Kategorie beschreibt Beobachtungen und Berichte, die sich auf die Anzeige von durchgeführten Refactorings beziehen. Das Feature wurde als sinnvoll empfunden und es wurde nur der Wunsch geäußert, dass wenn ein Refactoring noch nicht komplett durchgeführt wird, weil es einen Konflikt geben könnte, der Text aus dem Konfliktdialog dem Sitzungspartner auf irgendeine sinnvolle Art und Weise vermittelt wird. Ein Tester hat es etwas bemängelt, dass nicht zur Echt-

zeit, also während des Refactorings Informationen angezeigt werden. In Kapitel 4.4.5 wurde jedoch beschrieben, dass das nicht technisch realisierbar ist. Von daher wurde auf diese Bemerkung auch nicht weiter eingegangen.

5.6.4 Auswertung

Die Einordnungen der Berichte, Hinweise und Beobachtungen in Kategorien ist in einer MindMap im Anhang E vorzufinden. Insgesamt konnte ich bereits eine Tendenz feststellen, und zwar haben die Nutzer den Chat bevorzugt. Das lag zum einen daran, dass dort Änderungen in der Anzeige von Action Awareness Informationen durch das ständige Scrollen der Ansicht bemerkbar wurden. Zum anderen lag es daran, dass es als eine Art Historie verwendet wurde. Da der Chat von allen Nutzern nie verwendet wurde, wurde die Anzeige von Awareness Informationen im Chat nicht als störend empfunden. Viel mehr empfanden die Nutzer es als sinnvoll, den Chat für die Anzeige zu verwenden, da diese selbst den Chat nicht nutzen würden. Jedoch kam nicht klar heraus, ob die Nutzer die Informationen auch in der Zukunft im Chat haben möchten oder ob die Informationen an diesem Ort, also relativ zur Saros-View als besonders angenehm empfunden wurde. Von einem Nutzer kam auch der Wunsch, eine eigene View für die Action Awareness Informationen anzulegen, damit man diese flexibel im Eclipse-Arbeitsbereich anordnen kann. Auch im Bezug auf die angezeigten Informationen in ihrem Detailgrad und ihrer Auswahl wurde überwiegend positives Feedback geäußert und nur wenige Anregungen und Wünsche wurden mitgeteilt (siehe Abschnitt 5.6.3). Da sich also bereits Tendenzen gezeigt haben, wurden keine weiteren Nutzertests mehr eingeplant und da es sich außerdem als äußerst schwierig erwies, einen sechsten Tester zu finden, war dieser Umstand auch keineswegs Besorgnis erregend.

Im Ganzen bedeutet diese Auswertung für den ersten Prototypen Folgendes:

- Es soll nicht zwischen Following- und Nicht-Following-Modus unterschieden werden. Die Informationen sollen in beiden Arbeitsmodi angezeigt werden.
- Die Informationen sollten nicht gefiltert werden oder konfigurierbar sein. Die Nutzer erwarten, dass eine angemessene Konfiguration vorhanden ist und man selbst nichts weiter einstellen muss.
- Wenn beim Refactoring Konflikte entstehen, sollten diese dem entfernten Partner sichtbar gemacht werden.
- Der Chat wurde favorisiert und auch als Historie in Anspruch genommen. Ob der Chat an sich oder nur der relative Ort favorisiert wurde, konnte nicht klar herausgestellt werden. Auch konnten die Nutzer nicht

klar entscheiden, ob sie nach wie vor den Chat favorisieren würden, wenn im Chat auch Chat-Nachrichten gemischt mit Action Awareness Informationen gemischt einfließen würden.

- Die bereits angezeigten Informationen über die Ausführung und Ergebnis von JUnit-Tests ist angemessen. Lediglich die Fehlermeldung bei fehlgeschlagenen Tests wäre hilfreich.
- Generell wäre die textuelle Übertragung von Fehlermeldungen aus Dialogen und Views erwünscht.
- Die Übertragung von Informationen über die erzeugten Dateien soll auch weiterhin so bleiben.

Die Auswertung der Aufzeichnungen der Nutzertests sowie der Interviews haben interessante Rückschlüsse auf die Erfahrungen und den Mehrwert des neuen Features gegeben. Jedoch war es an dieser Stelle noch nicht möglich, ein Konzept für den zweiten Prototyp zu entwickeln, da für dieses eine Recherche zur technischen Realisierbarkeit erforderlich war. Das folgende Kapitel 6.2 geht auf diese technische Recherche sowie die Planung und Entwicklung des zweiten Prototyps ein.

6 Implementierung des zweiten Prototyps

Die Analyse der ersten Nutzertests haben Tendenzen gezeigt: Die Nutzer favorisieren den Chat als Anzeigeort. Es konnte zwar nicht gezeigt werden, ob es der Chat an sich war oder ob einfach der relative Ort in der Saros-View favorisiert wurde, aber nichtsdestotrotz wurde Feedback gegeben und einige Verbesserungswünsche wurden geschildert. Allerdings schien es mir nicht als sinnvoll, aus diesen Verbesserungswünschen ein Konzept zu erstellen, ohne zu wissen, was von denen technisch realisierbar ist.

Dieses Kapitel widmet sich der Recherche zur technischen Realisierbarkeit des zweiten Prototyps. Es schildert außerdem, warum am Ende entschieden wurde, die Action Awareness Informationen nicht mehr im Chat, sondern in einem eigenen Reiter in der Saros-View anzuzeigen. Weiterhin geht es auf die technische Umsetzung ein, denn für die Implementierung des Reiters habe ich einen neuen Ansatz ausprobiert, der sich jedoch schon nach kurzer Zeit als problematischer erwies als ursprünglich geplant, sodass schlussendlich der Reiter nach einem bekannten Ansatz umgesetzt wurde. Die Gründe für die Umplanung werden aufgelistet, sodass nachzuvollziehen ist, was an dem neuen Ansatz kritisch verlief, sodass in der Zukunft daraus gelernt werden kann. Zusätzlich, also am Ende der Arbeit noch Zeit blieb, habe ich den Ansatz auf eine andere Weise ein zweites Mal ausprobiert und war erfolgreich. Dieser zweite Versuch wird ebenfalls beschrieben und am Ende wird die Implementierung des Reiters mittels SWT und nach dem neuen Ansatz miteinander verglichen.

6.1 Technische Realisierbarkeit

Dieses Kapitel beschreibt, inwieweit die gewünschten Verbesserungen realisierbar sind und ob es möglich ist, noch weitere Informationen zu extrahieren. Von daher wurde in diesem Schritt jede Funktion des Prototyps untersucht und versucht herauszufinden, was an zusätzlichen Informationen über die Aktionen aus der API gezogen werden können. Zusätzlich wurde überprüft, welche weiteren visuellen Hilfsmittel dafür verwendet werden können, denn die eintreffenden Nachrichten zur Action Awareness haben sich visuell voneinander außer textuell nicht voneinander unterschieden, sodass auch hier die Idee aufkam, diese visuell voneinander unterscheidbar zu machen.

6.1.1 Dialoge und Views

Das Java Objekt `Shell`[32] repräsentiert ein klassisches Fenster. Solch ein Fenster hat kein Elternobjekt und wird somit als *top-level shell* bezeichnet. Ein Objekt, welches ein Elternobjekt besitzt, ist normalerweise ein Dialog.

[32] http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Shell.html

Abbildung 27 zeigt, welche Informationen aus einem Shell-Objekt extrahiert werden können. Insgesamt ist also mehr als die Ermittlung des Titels des offenen Dialoges über die gegebene API nicht möglich. Wie in Kapitel 4.4.3 beschrieben, ist es theoretisch auch möglich, auf die Kind-Objekte eines Dialoges zuzugreifen. Wenn ein Dialog z. B. ein Textfeld besitzt, kann das Textfeld ausgelesen werden. So wäre es demnach möglich, auch auf den Inhalt eines Dialoges zuzugreifen. Jedoch ist das extrem fehleranfällig und wenn in der Zukunft Dialoge angepasst werde oder neue hinzukommen, greift diese Lösung diese Fälle nicht auf.

Die Klasse `IWorkbenchPartReference`[33] repräsentiert eine Eclipse-View, wobei es sich um einen Editor (`IEditorPart`) oder eine normale View (`IViewPart`) handeln kann. Die Abbildung 28 zeigt einen Ausschnitt der getter-Methoden dieser Klasse. Neben der Methode `getTitle()`, mit der man den Titel einer View erhalten kann, gibt es unter anderem noch die Methode `getContentDescription()`, mit der man die Beschreibung einer View ermitteln kann. Wie genau das aussieht, kann man z. B. der Console-View in Abbildung 29 sehen. Allerdings stellt sich hier die Frage, ob die Übertragung der Beschreibung einer View für den Nutzer wirklich einen großen Mehrwert bringt. Außerdem liefert diese Methode einen leeren String zurück, wenn keine Beschreibung vorhanden ist. Demnach gibt es sicherlich auch Views, die keine Beschreibung besitzen. Letztendlich wurde hier dagegen entschieden, den Beschreibungstext einer View (wenn dieser überhaupt gegeben ist), anzuzeigen.

6.1.2 JUnit Testläufe

Ein Testnutzer hatte vorgeschlagen, dass wenn ein JUnit-Test fehlschlägt, die Fehlermeldung übertragen werden könnte, damit die Sitzungspartner den Grund untersuchen können, ohne den Test selbst ausführen zu müssen. Wie in Kapitel beschrieben, wird für die Ermittlung von Informationen zu Testläufen der `TestRunListener` aus dem Paket `org.eclipse.jdt.junit` verwendet. Der folgende Code zeigt, wie es möglich ist, auch weitere Informationen zu JUnit Testläufen zu ermitteln:

```
1 private final TestRunListener = new TestRunListener()
    {
2     @Override
3     public void testCaseFinished(final
          ITestCaseElement testCaseElement) {
4         double elapsedTime = testCaseElement.
              getElapsedTimeInSeconds();
5         String failureTrace = testCaseElement.
              getFailureTrace().getTrace();
```

[33] http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/IWorkbenchPartReference.html

```
getAccessible() : Accessible - Control
getAlpha() : int - Shell
getBackground() : Color - Control
getBackgroundImage() : Image - Control
getBackgroundMode() : int - Composite
getBorderWidth() : int - Control
getBounds() : Rectangle - Shell
getCaret() : Caret - Canvas
getChildren() : Control[] - Composite
getClass() : Class<?> - Object
getClientArea() : Rectangle - Shell
getCursor() : Cursor - Control
getData() : Object - Widget
getData(String key) : Object - Widget
getDefaultButton() : Button - Decorations
getDisplay() : Display - Widget
getDragDetect() : boolean - Control
getEnabled() : boolean - Control
getFont() : Font - Control
getForeground() : Color - Control
getFullScreen() : boolean - Shell
getHorizontalBar() : ScrollBar - Scrollable
getImage() : Image - Decorations
getImages() : Image[] - Decorations
getIME() : IME - Canvas
getImeInputMode() : int - Shell
getLayout() : Layout - Composite
getLayoutData() : Object - Control
getLayoutDeferred() : boolean - Composite
getListeners(int eventType) : Listener[] - Widget
getLocation() : Point - Shell
getMaximized() : boolean - Shell
getMenu() : Menu - Control
getMenuBar() : Menu - Decorations
getMinimized() : boolean - Shell
getMinimumSize() : Point - Shell
getModified() : boolean - Shell
getMonitor() : Monitor - Control
getOrientation() : int - Control
getParent() : Composite - Control
getRegion() : Region - Shell
getScrollbarsMode() : int - Scrollable
getShell() : Shell - Shell
getShells() : Shell[] - Shell
getSize() : Point - Shell
getStyle() : int - Widget
getTabList() : Control[] - Composite
getText() : String - Decorations
getTextDirection() : int - Control
getToolBar() : ToolBar - Shell
getToolTipText() : String - Control
getTouchEnabled() : boolean - Control
```

Abbildung 27: Ein Ausschnitt der getter-Methoden der Java Klasse `Shell`. Außerdem die Methode `getText()` sieht es bei den anderen Methoden nicht danach aus, dass Informationen mit Mehrwert ermittelt werden können.

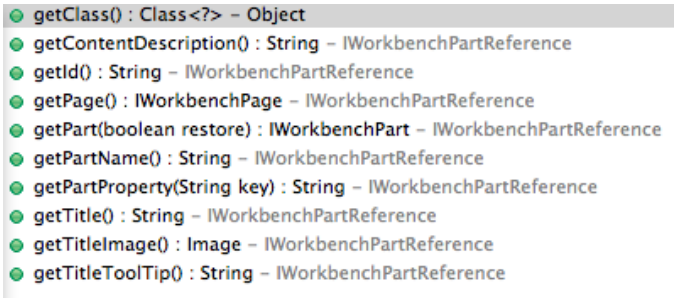


Abbildung 28: Ein Ausschnitt der getter-Methoden der Java Klasse `IWorkbenchPartReference`.

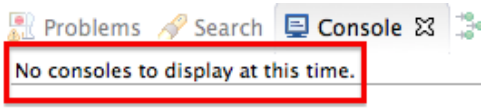


Abbildung 29: Mithilfe der `getContentDescription()` Methode ist es möglich, eine genauere Beschreibung einer View zu extrahieren. In der Abbildung ist dies beispielhaft für die Console-View gezeigt.

```
6            int nrChildren = testCaseElement.
                 getParentContainer().getChildren().length;
7        }
8    };
```

Listing 7: Ermittlung der verlaufenen Zeit, der Fehlermeldung und der Anzahl an Tests für eine Testsuite.

Zeile 4 zeigt die Ermittlung der verlaufenen Zeit für einen Testlauf. Zeile 5 zeigt, wie es möglich ist, an die Fehlermeldung eines nicht erfolgreich durchgelaufenen JUnit-Tests heranzukommen. Das entspricht im Grunde dem Wunsch eines Testnutzers. Jedoch besteht hier das Problem, dass es sich um einen String handelt, der den kompletten Stracktrace beinhaltet, wie es die Abbildung 30 auch zeigt. Zeile 6 zeigt, wie die Anzahl an Tests einer Testsuite ermittelt werden kann. Diese drei beschrieben Zeilen verdeutlichen, welche weiteren Informationen theoretisch zusätzlich übertragen werden können. Allerdings waren die Dauer eines JUnit-Testlaufes und die Anzahl an Tests in einer Suite weniger interessant. Ein Nutzer hat vor allem den Wunsch geäußert, bei Fehlschlägen die entsprechende Meldung beim Sitzungspartner einsehen zu können oder übertragen zu bekommen. Von daher bestand die Idee, den String an der Stelle, an der die Beschreibung aufhört und der Ort des Fehlers angezeigt wird (also alles vor dem Wort „at“) zu schneiden und diese Beschreibung anzuzeigen, wenn ein Test fehlschlägt. Da jedoch in der

```
1 java.lang.AssertionError
2     at org.junit.Assert.fail(Assert.java:86)
3     at org.junit.Assert.assertTrue(Assert.java:41)
4     at org.junit.Assert.assertTrue(Assert.java:52)
5     at ThisSucceeds.test3(ThisSucceeds.java:16)
6     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
7     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
8     at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
9     at java.lang.reflect.Method.invoke(Method.java:597)
10    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
11    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
12    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
13    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
14    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
15    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
16    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
17    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
18    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)
19    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)
20    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:53)
21    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:229)
22    at org.junit.runners.ParentRunner.run(ParentRunner.java:309)
23    at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:50)
24    at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
25    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:467)
26    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:683)
27    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:390)
28    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:197)
```

Abbildung 30: Der Stacktrace eines fehlgeschlagenen JUnit-Tests kann nur in Form eines Strings ermittelt werden.

Chat-View nicht mehr viel Platz vorhanden ist, war die Überlegung, diesen als Tooltip über dem Text oder in einer anderen Form zu übertragen. Wie genau das aussehen sollte, wurde an dieser Stelle noch nicht festgelegt und darauf soll im Folgenden noch eingegangen werden.

6.1.3 Refactorings

Um herauszufinden, welche weiteren Informationen aus dem `IRefactoringHistoryService` (siehe Kapitel 4.5.3) ermittelt werden konnten, habe ich diese und das entsprechende Ereignis `RefactoringExecutionEvent` näher untersucht. Leider konnte ich hier nicht mehr herausfinden als das, was in Kapitel 4.5.3 bereits beschrieben wurde. Der einzige Punkt, welcher vielleicht einer Überlegung wert gewesen wäre, wäre die Möglichkeit anzuzeigen, wenn ein Refactoring rückgängig gemacht wird. Dieser Gedanke ist nicht ganz abwegig, da bei Konflikten nach einem abgeschlossenen Refactoring dieser im schlimmsten Fall auch rückgängig gemacht werden kann. Am wahrscheinlichsten würde sich ein Entwickler jedoch darum kümmern, die Konflikte durch Codeanpassungen aufzulösen, satt das ganze Refactoring rückgängig zu machen. Von daher habe ich auf diese Erweiterung an dieser Stelle verzichtet. Auch der Wunsch eines Nutzers, die Meldung zum Konflikt dem Sitzungspartner anzuzeigen, habe ich nicht umgesetzt aus demselben Grund wie in Abschnitt 6.1.1 beschrieben: Es müssten die Kind-Elemente des Dialoges iteriert werden, welches die Konflikte darstellt, und diese Texte müssten manuell ausgelesen werden, was keine elegante und dazu auch eine fehleranfällige Lösung darstellt.

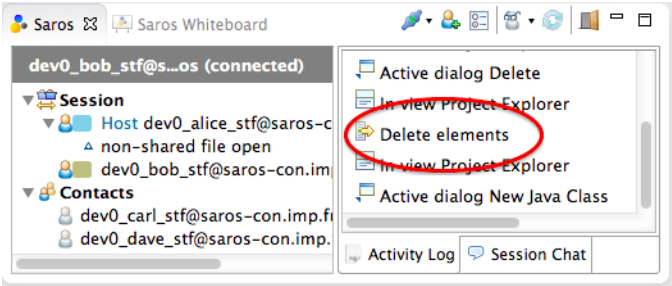


Abbildung 31: Der `IRefactoringHistoryService` zeigt ebenfalls die Information an, dass etwas gelöscht wurde. Jedoch werden keine weiteren Informationen darüber gegeben, was genau gelöscht wurde.

6.1.4 Erzeugte Dateien

Für diese Funktion des Prototyps gab es nur positives Feedback und es wurden auch keine weiteren Verbesserungswünsche ausgesprochen. Eine Idee war die Übertragung der Information, dass eine Datei gelöscht wurde. Allerdings wird diese Information in ähnlicher Weise bereits durch den `IRefactoringHistoryService` übertragen, auch wenn diese Information nicht gerade viele Details preis gibt, wie in Abbildung 31 zu sehen ist: Außer der Information, dass etwas gelöscht wurde, wird nichts weiteres gespeichert und angezeigt. Da es aber keine saubere Lösung darstellen würde, diese eine Information vom `IRefactoringHistoryService` manuell zu löschen und stattdessen selbst die gelöschte Datei mit dem Namen zu ermitteln und anzuzeigen, wurde hier darauf verzichtet, den Prototypen um diese zusätzliche Funktion zu erweitern.

6.1.5 Visuelle Hilfsmittel

Da es bei den oben angeführten Funktionen nur wenig Erweiterungsmöglichkeiten gab, wurde überlegt, die angezeigten Informationen visuell voneinander zu unterscheiden. Die Begründung war, dass wenn viele Nachrichten hochfrequent eintreffen, diese sich optisch kaum voneinander unterscheiden. Wenn diese jedoch durch z. B. Formatierung, Farben und unterschiedlichen Darstellungen hervorgehoben werden, kann bei einem schnellen Blick in die Anzeige schneller ermittelt werden, um was für eine Art von Action Awareness Information es sich handelt.

Eine Idee ist hierbei, den variablen Teil einer Awareness Information fett hervorzuheben. In [MN97] wird eine Studie beschrieben, in der 15 Teilnehmer einen Text lesen sollen. Diese Teilnehmer haben zuerst den gesamten Text überflogen, statt es Wort für Wort sofort zu lesen. Dabei haben sie sich an Überschriften, Listen, Bildern, hervorgehobenen Texten oder fett formatierten Texten orientiert. Diese Studie wurde zwar für Texte durchgeführt, die im

Web veröffentlich werden, aber die Anzeige der Awareness Informationen wird ebenfalls am Bildschirm wahrgenommen, sodass diese Beobachtung auch bei Saros in ähnlicher Weise mit hoher Wahrscheinlichkeit ebenfalls festzustellen wäre. Beim variablen Teil handelt es sich um den Text, der z. B. bei jedem Öffnen eines Dialoges in der Anzeige ändern würde, und zwar der Titel. Auch der Titel einer aktiven View, der Name der erzeugten Datei sowie der Names des JUnit-Tests beim Start soll fettgedruckt werden. Weiterhin war die Idee, beim Beenden eines JUnit-Tests das Ergebnis zusätzlich zur fettgedruckten Hervorhebung diese zusätzlich farblich zu unterscheiden. Bei einem erfolgreich durchgeführten JUnit-Test würde demnach das Wort „OK“ in grüner Farbe dargestellt sein. Darüber hinaus war die nächste Idee, die angezeigten Informationen auch mithilfe von Icons voneinander stärker zu unterscheiden. In [GK13] wird eigentlich erklärt, dass Icons nicht gleich eine Verbesserung der Usability bedeuten. Allerdings betont Gocza et al. in [GK13], dass Icons zusammen mit einem Text benutzt werden sollten und in [Kis09] wird ebenfalls darauf aufmerksam gemacht, dass nicht nur Icons oder Text, sondern idealerweise beides zusammen benutzt werden sollen.

6.1.6 Funktionen des zweiten Prototyps

Insgesamt hat die Recherche zur technischen Realisierbarkeit ergeben, dass es keine sinnvollen Erweiterungsmöglichkeiten im Bezug auf die Funktionen gibt:

Dialoge und Views Bei Dialogen ist außer dem Titel keine weitere Information ermittelbar. Bei Views könnte zusätzlich der Beschreibungstext übertragen werden, wenn dieser zur Verfügung steht. Da das nicht immer der Fall ist und allein der Titel der View für den Nutzer eigentlich ausreicht, werden auch hier außer dem Titel keine weiteren Funktionen wie bisher übertragen.

JUnit Testläufe Hier könnte der Beschreibungstext (also z. B. `java.lang.-AssertionError`) beim Halten des Mauszeigers über dem Text `FAILURE` angezeigt werden. Jedoch reicht diese Beschreibung möglicherweise nicht aus, um daraus schlussfolgern zu können, was genau der Fehler ist. Der Nutzer müsse am Ende trotzdem in den Code schauen oder nachfragen. Von daher wird auf diese Erweiterung verzichtet.

Refactorings Es gäbe die Möglichkeit, die Information, dass ein Refactoring rückgängig gemacht wurde, zu übertragen. Allerdings wurde sowohl bei der Analyse des Videomaterials als auch bei den Nutzertests sichtbar, dass im Falle eines Konflikts bei Refactorings dieser nicht rückgängig gemacht wurde, sondern dass die Konflikte manuell behoben wurden. Von daher wird auf diese Erweiterung verzichtet.

Abbildung 32: Mittels HTML und CSS wurde ein Entwurf erstellt, der die Umsetzung der visuellen Darstellungsmöglichkeiten zeigen soll. Die variablen Texte werden fett hervorgehoben sowie die Ergebnisse der Testläufe farblich unterschieden. Zusätzlich dazu werden passende Icons verwendet, die das Unterscheiden der angezeigten Informationen verbessern.

Erzeugte Dateien Neben der Anzeige von erzeugten Dateien könnten auch gelöschte Dateien angezeigt werden. Jedoch wird dieser Fall bereits dank des `IRefactoringHistoryService` bereits zur Verfügung gestellt, sodass auch hier nichts angepasst wurde.

Konfiguration und Arbeitsmodus Die Tester haben betont, dass sie sich um keine Konfiguration kümmern möchten und von dem Tool Saros erwarten, dass diese dem Nutzer die angemessenen Informationen anzeigt, ohne selbst etwas konfigurieren oder filtern zu müssen. Auch zwischen dem Following- und Nicht-Following-Modus soll nach Aussagen der Tester nicht unterschieden werden.

Infolgedessen wurde sich dafür entschieden, die Funktionen des Prototyps nicht weiter zu verändern, sondern nur die visuelle Darstellung wie in Kapitel 6.1.5 beschrieben anzupassen. In Abbildung 32 ist ein Entwurf zu sehen, wie die angepasste visuelle Darstellung letztendlich aussehen könnte.

6.2 Gründe für einen neuen Reiter: Activity Log

Im vorigen Kapitel 6.1 wurde untersucht, inwieweit die in den Nutzertests genannten Berichte umgesetzt werden können. Am Ende in Kapitel 6.4.2 wurde dann das Aussehen und die Funktionsweise des zweiten Prototyps vorgestellt.

Die Auswertung der Nutzertests in Kapitel 5.6.4 hatten gezeigt, dass die Tester den Chat als Anzeigeort favorisiert haben. Abbildung 33 zeigt, wie es

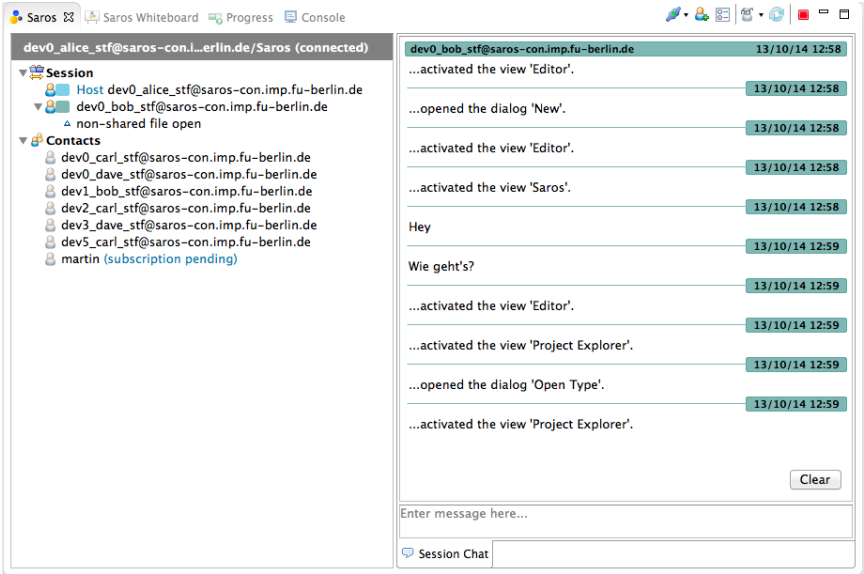


Abbildung 33: Der erste Prototyp konnte die Action Awareness Informationen im Chat anzeigen. Das Bild zeigt, wie es aussieht, wenn sowohl Action Awareness Informationen als auch Chat-Nachrichten übertragen wurden.

ausschaut, wenn sowohl Chat-Nachrichten als auch Awareness Informationen empfangen werden. Auch wenn der Chat weniger genutzt wird und das eigentlich ein Grund war, warum es als weiterer Anzeigeort genutzt wurde (siehe Kapitel 4.4.1), werden hier Aktivitäteninformationen mit Chat-Nachrichten logisch gemischt. Das war ein Umstand, der nicht zu vernachlässigen war. Allerdings gab es neben diesem noch weitere Gründe, warum es notwendig wurde, über die Umsetzung in einem neuen Reiter nachzudenken.

Chat und Awareness gemischt

Wie bereits beschrieben und in Abbildung 33 verdeutlicht, werden im ersten Prototypen Chat Nachrichten mit Awareness Informationen gemischt. Wenn viele Nachrichten eintreffen, wird es schwerer, die Chat Nachrichten von Awareness Informationen schnell zu unterscheiden.

Keine Gmail-Unterstützung

Nutzer, die Saros über einen Gmail-Account nutzen, haben keinen Multi-User-Chat, da das von Gmail nicht unterstützt wird. Sie müssten in den Saros-Einstellungen einen eigenen Chat-Server einrichten, wie es in Abbildung 34 zu sehen ist. Und da nicht zu erwarten ist, dass alle Gmail-Nutzer das machen, würden diese die Action Awareness Informationen nicht übertragen bekommen.

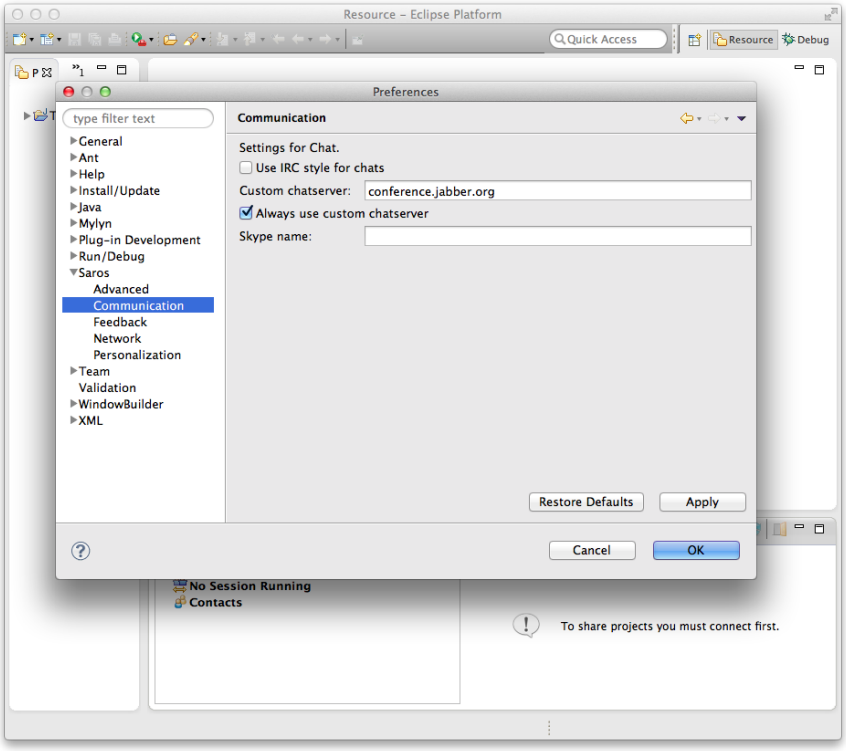


Abbildung 34: Gmail-Nutzer haben standardmäßig keinen Multi-User-Chat zur Verfügung und müssen manuell einen eigenen Chat-Server über die Saros-Einstellungen eintragen.

Kaum Verwendung für Chat

Während der Mitarbeit im Saros-Projekt wurde einige Male erwähnt, dass der Chat eher selten bis gar nicht genutzt wird was auch der Grund war, warum der Chat für die Anzeige von Awareness Informationen quasi „zweckentfremdet“ werden sollte. Zwar müsste diese Beobachtung durch Nutzertests überprüft werden, aber da die Nutzer für die Audioverbinduung höchstwahrscheinlich Skype nutzen, nutzen sie entweder den Chat oder benötigen keine, da alles nötige auch ausgesprochen wird. Wenn also in Zukunft der Chat entfernt wird, wird auch damit automatisch dieses Feature entfernt.

Abhängigkeit vom Chat

Bei den Nachrichten im Chat handelt es sich um XMPP-Nachrichten, die über das Netzwerk übertragen werden. Die Awareness Informationen werden in die Chat Historie direkt eingespeist. Das bricht eigentlich die Architektur und macht das Feature technisch strak abhängig vom Chat-Modell. Wird am Chat eine Änderung vorgenommen, wird das Feature eventuell nicht mehr

funktionieren. Auch wenn sich der Autor der Veränderung darum kümmern müsste, stellt diese unsaubere Abhängigkeit sicherlich Mehrarbeit dar.

Umsetzung mittels Hack

Der Hauptgrund, warum die Anzeige im Chat ein Problem darstellt, ist die Umsetzung: Der einzige Weg, Nicht-Chat-Nachrichten in die Chat-Historie einzuschleusen ging über eine Methode, die auf `public` gesetzt war, aber nicht sein sollte. Listing 17

```
1  private IActivityReceiver receiverChat = new
       AbstractActivityReceiver() {
2      @Override
3      public void receive(IDEInteractionActivity activity
           ) {
4          String message;
5          if (activity.getStatus() == Status.FOCUS) {
6              if (activity.getElement() == Element.DIALOG)
                   {
7                  message = "...opened the dialog '" +
                       activity.getTitle() + "'.";
8              } else {
9                  message = "...activated the view '" +
                       activity.getTitle() + "'.";
10             }
11
12             sessionChat.addHistoryEntry(new ChatElement(
                   message, activity.getSource().getJID(),
                   new Date()));
13             ((AbstractChat) sessionChat).
                   notifyJIDMessageReceived(activity.
                   getSource().getJID(), message);
14         }
15     }
16 };
```

Listing 8: Einfügen von Nicht-Chat-Nachrichten in die Chat Historie, um so die Awareness Informationen in der Chat Ansicht darzustellen.

Zeile 12 zeigt, dass dort ein neues `ChatElement` erzeugt wird und in die Chat Historie hinzugefügt wird. Anschließend werden die Listener in Zeile 13 über das neue Element benachrichtigt und z. B. die Klasse `ChatControl`, welches sich um die Anzeige des Inhaltes der Chat Ansicht kümmert, wird benachrichtigt. Hier wurde also direkt eine Nicht-Chat-Nachricht in das Chat-Moodel geschrieben, weil glücklicherweise die beiden Methoden öffentlich waren. Insofern ist die Umsetzung also ein Hack.

Die aufgeführten Gründe haben mich letztendlich überzeugt, statt des Chats einen neuen Reiter für die Awareness Informationen einzuführen. Ein neuer Reiter hat auch aus Sicht der Nutzerfreundlichkeit Vorteile:

- Ein neuer Reiter teilt den Action Awareness Informationen in Saros einen eigenen Abschnitt zu [Mif11]. Somit werden die Informationen nicht wie im Chat nur nebenbei wahrgenommen, sondern als eine eigenständig und wichtige Funktionalität von Saros hingestellt.
- Der Nutzer kann sich frei aussuchen, ob er den Chat oder die Action AwarenessInformationen sehen möchte.
- Wenn beides gebraucht wird, kann der Nutzer leicht zwischen dem Chat und der Awareness Anzeige wechseln, da zwei Tabs das Wechseln erleichtern. Wäre alles in einem Tab, müsste der Nutzer beim Hinsehen nach der entsprechenden Information suchen, also „Ist das jetzt eine Chat-Nachricht?“ oder“Wo ist jetzt die Anzeige zwischen all den Chat-Nachrichten, die mir zeigt, was Person X zuletzt gemacht hat?“.
- Würde sich das Chat-Design ändern, müssten die Awareness Informationen visuell ebenfalls an das Design der Chat-Nachrichten angepasst werden, da es sonst inkonsistent wirken könnte. Wenn Chat und Awareness Informationen jedoch getrennt in Reitern sind, können sie je nach Bedarf unterschiedlich gestaltet werden.

6.3 HTML-basierte Implementierung

Nachdem ich die die Nutzertests ausgewertet habe (siehe Kapitel 5) und die Recherche zur technischen Realisierbarkeit (siehe Kapitel 6.1) ergeben hatten, dass der erste Prototyp nicht um die gewünschten Funktionen erweitert werden kann, wurde wie bereits erläutert entschieden, die Gestaltung der angezeigten Informationen zu verbessern (siehe Kapitel 6.4.2). Zu diesem Zeitpunkt gab es ein Treffen mit dem Betreuer und dieser hatte die Idee, eine HTML-basierte Implementierungsweise auszuprobieren. Das bedeutet, dass zwar der Reiter an sich ein normales GUI-Element ist, der Inhalt jedoch ein eigenes Browser-Widget ist, welches mit den bekannten Webtechnologien HTML, CSS und JQuery gestaltet werden kann. Bei der neuen Activity Log ist es auch einfacher, dies umzusetzen, da der Reiter keine Eingaben benötigt, sprich es gibt keine Interaktionsmöglichkeiten und somit müssen nur eingetroffene Ereignisse, die neue Awareness Informationen bereitstellen, verarbeitet und angezeigt werden.

Der Grund für die Überlegung war die Entwicklung eines Plug-ins eines wissenschaftlichen Mitarbeiters, namens Björn Kahlert[34]. Björn Kahlert führt im Rahmen seiner Promotionsforschung qualitative Datenanalysen durch (QDA) und hat dafür ein Eclipse Plug-in, in dem eine Galerie mit einer Menge von Widgets zur Verfügung steht. Dabei sind es einfache Widgets, wie Labels und Composites, aber auch teilweise sehr komplexe Widgets, wie ein Chat (der Saros-Chat) und andere komplexe Objekte. Unter anderem sind dabei

[34] http://www.mi.fu-berlin.de/w/Main/BjoernKahlert

Widgets, deren Inhalt eine Browser-Komponente ist, die mittels bekannten Webtechnologien gestaltet wurde. Die Idee hinter der Nutzung dieses Plug-ins für Saros ist, den Inhalt des Reiters mithilfe dieses Plug-ins zu gestalten, um zu sehen, wie aufwändig und vorteilhaft diese Implementierungsweise das ist. Wenn nämlich herauskommt, dass das ein vorteilhafter Ansatz ist, können in Zukunft Inhalte mittels HTML, CSS und Javascript dargestellt werden und diese müssten dann bei der Erweiterung von Saros für weitere IDE's, wie IntelliJ oder NetBeans nicht neu programmiert werden, sondern nur den Container, der im Fall von SWT ein normales `Composite` ist, müsste neu entwickelt werden.

Dieses und die folgenden Kapitel beschreiben die Implementierung des zweiten Prototyps. Denn zu Beginn wurde dieser Ansatz erprobt und aufgrund einiger technischer Schwierigkeiten wurde den Reiterinhalt mittels SWT implementiert (siehe Kapitel 6.4). Anschließend wurde der HTML-basierte Ansatz auf eine andere Herangehensweise ausprobiert und es brachte Erfolg. Diese Ergebnisse haben nun den Vorteil, dass es zum einen einen neuen Reiter in der Saros View gibt, der Action Awareness Informationen zur Verfügung stellt und zum anderen wurde in einem Versuch ermittelt, dass es theoretisch möglich ist, eine HTML-basierte Implementierungsweise durchzuführen. Beide Vorgehensweisen können somit miteinander verglichen werden (siehe Kapitel 6.6) und in der Zukunft können weitere Überlegungen in dieser Richtung gemacht werden.

Dieses Kapitel soll zuerst erklären, wie solch ein HTML-basierter Ansatz im Prinzip funktioniert. Daraufhin wird beschrieben, auf welche Art und Weise der erste Versuch erfolgte und warum dieser verworfen wurde. In Kapitel 6.5 wird beschrieben, auf welche Weise dieser Ansatz erneut ausprobiert wurde und welche Ergebnisse dabei erzielt wurden. In Kapitel X**??**werden beide Herangehensweisen, also SWT und HTML-basiert, miteinander verglichen.

6.3.1 Vorgehensweise

Im Allgemeinen handelt es sich bei diesem Ansatz um den Versuch, den Inhalt einer GUI-Komponente in SWT also Browser zu deklarieren und dem Browser den anzuzeigenden Inhalt vom Java-Code aus zu senden. In der SWT-Bibliothek gibt es bereits eine Klasse `Browser` [Ecl11b], deren Verwendung in beispielhaft in [Cor04] gezeigt wird. Björn Kahlert hat in seinem Plug-in jedoch einen Wrapper um diese Funktionalität geschrieben, um so verschiedene Funktionen zusammenzufassen, weitere Funktionen anzubieten und so insgesamt die Arbeit mit der Browser-Komponente von SWT zu vereinfachen. SWT greift dabei auf einen installierten Browser zu, sodass man nicht gezwungen ist, eine Browser-Engine mitzuliefern oder einzurichten. Allerdings soll es auf Linux-Systemen nach [Phi11] zu Problemen führen, was jedoch für diese Arbeit kein Hindernis war, es auszuprobieren, da ich in den wenigsten Fällen unter Windows und überwiegend unter Mac OSX entwickelt

und getestet habe. Hat man solch eine Browser-Komonente in die SWT-GUI, also in ein SWT-Composite, eingebaut, so kann man den Inhalt beliebig mithilfe von HTML, JavaScript und CSS gestalten. Da es mittlerweile viele CSS-Frameworks, wie *Twitter Bootstrap*[35] gibt, die nicht nur weit verbreitet sind, sondern auch ein einheitliches und ansehnliches Design anbieten, ist die Entwicklung von Inhalten als Prototyp ein schnelles Vorhaben. Außerdem hat diese Vorgehensweise den Vorteil, dass wenn die Webseite cross-browser kompatibel entwickelt wurde, die Darstellung nicht nur plattformübergreifend ist, sondern auch leicht in andere GUI-Elemente eingefügt werden kann, wie es ja die Überlegung für Saros-I bereits der Fall ist.

Es gibt dabei zwei Vorgehensweisen:

1. Man entwickelt eine Webseite, bestehend aus einer Seite, die vollständig ohne die Java-Anbindung im Browser lauffähig ist. Man könnte also theoretisch die Webseite komplett nur im Browser anzeigen. Diese Webseite wird dann in die Browser-Komponente innerhalb des GUI-Widgets geladen und zwischen dem Java-Code und der Browser-Komponente befindet sich eine *Fassade*, die vom Java-Code angesprochen werden kann, und die JavaScript-Funktionen aufruft, um die Webseite dynamisch gestalten zu können. JavaScript, CSS und HTML müssen hier also komplett fertiggestaltet sein.

2. Man programmiert direkt den Inhalt des Browsers vom Java-Code aus. Dafür ruft man eine leere HTML-Seite auf und belegt den Inhalt der Webseite mit Methoden, wie u.a. `setBodyHTML()`, `runJS()` und`injectCSS()`. Das Einbinden von *JQuery*[36] und ähnlichen JavaScript-Frameworks ist auch möglich, sodass man sich sogar die Arbeit einfacher machen kann. Das von Björn Kahlert entwickelte Plug-in besitzt bereits Browser-Komponenten, die mit JQuery oder Twitter Bootstrap ausgestattet sind.

Beide Vorgehensweisen sind je nach Szenario oder Anforderungen besser geeignet. Die einzige Voraussetzung ist, dass die Webseite cross-browser kompatibel sein sollte, da je nach installiertem Browser sonst der Inhalt des Plug-ins anders aussehen würde.

Der große Vorteil dieser Vorangehensweisen ist, dass man so die Gemeinsamkeiten zwischen Saros für Eclipse (*Saros-E*) und Saros für IntelliJ (*Saros-I*) vergrößern könnte und somit die Entwicklung weiterer IDE's vereinfachen könnte, denn der Inhalt mancher GUI-Komponenten wäre auf diese Weise schneller nachzubauen (bzw. nur der Wrapper um die Browser-Komponente müsste sichergestellt sein). Es stellt sich auch die Frage, ob somit weniger Bereiche der GUI plattformspezifisch sein würden und man somit auch weniger plattformabhängig testen müsste. Wenn dem so wäre,

[35] http://getbootstrap.com
[36] http://jquery.com

wäre das ein Segen für die Testautomatisierung. Zudem könnte man so auch die GUI-Logik von der Logik trennen, die sich um die letztendliche Anzeige kümmert, was erhebliche Vorteile hat, wenn man z. B. Usabilitytests macht und dabei herauskommt, dass Teile der GUI Anpassungen benötigen. Da es sich um reines HTML, CSS und JavaScript handeln würde, würde man die Design-Entwürfe viel schneller umsetzen können. Außerdem verbraucht dieser Ansatz keine großartige CPU-Leistung, weil alles synchron abläuft:

- Erzeugen einer Browser-Komponente aus dem Java-Code
- JavaScript-Code wird hineingeladen (synchron)
- CSS-Code wird hineingeladen (synchron)
- Java-Code ruft über die Fassade den JavaScript-Code auf (synchron), der sich wiederum um das Rendering der Seite kümmert
- Browser-Komponente wird aktualisiert
- Es bedarf keinem Thread-Handling, denn die Browser-Komponente kümmert sich schon darum, dass alles im richtigen Thread abläuft

Der Versuch, den ich im Rahmen dieser Arbeit unternommen habe, soll zeigen, dass dieser Ansatz für Saros funktioniert, wie er funktioniert und wie dieser mit der Implementierung mit SWT zu vergleichen ist. Denn es bedarf noch nach weiteren Überlegungen, wie viele Vorteile dieser Ansatz mit sich bringt, was alles in Saros nach diesem Ansatz implementiert werden soll, was im Bezug auf das oben erwähnte Linux-Problem unternommen werden kann und ob es überhaupt in anderen IDE's, wie IntelliJ, dieser Ansatz auch so komfortable umgesetzt werden kann wie für Saros für Eclipse.

6.3.2 Erster Versuch ohne Plug-in Anbindung

Nachdem nun feststand, dass ich diesen Ansatz ausprobieren soll, hatte ich zwei Möglichkeiten: Entweder ich kopiere die `Browser`-Klasse aus Björns Plug-in und alle fehlenden Klassen, die ich noch dazu benötige, in das Saros-Projekt oder ich binde das gesamte Plug-in als Abhängigkeit an das Saros-Projekt ein und kann es an entsprechenden Stellen nach einem Import einfach nutzen. Damals war die Frage, welches die bessere Vorgehensweise ist, nicht zu beantworten, weil bei beiden Vorgehensweisen noch keine Erfahrung im Saros-Team vorhanden war und man so nicht einschätzen konnte, was weniger aufwändig ist. Einer der Entwickler hatte in der Diskussion, die über die Mailingliste stattfand, seine Bedenken bzgl. dieses Vorgehens ausgesprochen. Einer der Gründe war, neben der Frage, ob dieser HTML-basierte Ansatz überhaupt Vorteile bringt, die Sorge, dass das Plug-in so groß sei und Saros quasi eine „schwere Last am Fuß“ haben würde. Das gesamte Plug-in ist, wenn es frisch aus dem Github-Repository ausgecheckt wird, um die 40MB

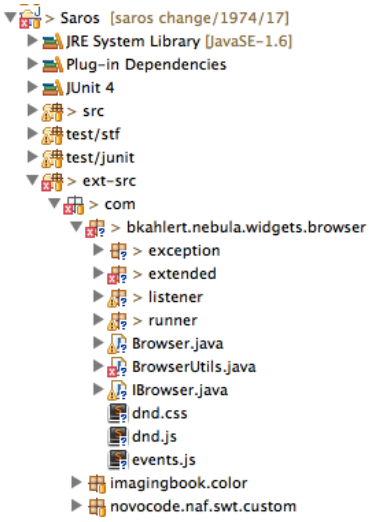


Abbildung 35: Eine Möglichkeit für die Integration der Browser-Wrapper von Björn Kahlert ist das Herüberkopieren aller benötigter Klassen aus seinem Plug-in das Saros Projekt. Das gesamter `browser` Package wurde kopiert.

groß. Dieser Entwickler hat sich später an mich gewendet und mir den Tipp gegeben, alle benötigten Klassen in Saros zu kopieren. Das würde bedeuten, neben `Browser.java` aus [Kah14] alle weiteren abhängigen Klassen aus dem entsprechenden Package in Saros zu kopieren. Abbildung 35 zeigt, wie das aussehen kann. In Saros gibt es ein Verzeichnis, namens `ext-src`, in das fremde Quellcodes eingefügt und somit in das Projekt integriert werden können. Das ist vor allem dann sinnvoll, wenn es sich nicht um eigenständige Plug-ins handelt oder wenn man nur einen bestimmten Teil des Plug-ins benötigt.

Allerdings wurde hier schon eine Schwierigkeit bemerkbar: Es sind weitaus mehr Klassen, die benötigt werden, als ursprünglich geschätzt. Je mehr fehlender Klassen hinzugefügt wurden, desto mehr abhängige Klassen wiederum mussten ebenfalls kopiert werden. Leider wurden in den vielen Klassen auch meistens nur `RuntimeException`'s geworfen, sodass eine spezielle Behandlung nötig war. Ich musste also an vielen Stellen den Code manuell anpassen und das ExceptionHandling verbessern. Björn Kahlert hat in seinem Plug-in auch die Möglichkeit gegeben, die Konsolenausgaben des Browsers auszugeben. So war es beim erstmaligen Ausprobieren des Plug-ins möglich, mit dem JavaScript Elemente dynamisch zu verändern und anhand der Konsolenausgaben zu sehen, wo z. B. der DOM-Baum verändert wurde, wie er danach aussah und was für JavaScript-Fehler gemeldet werden. Beim Kopieren der benötigten Klassen wurde dies ausgelassen, da ohnehin schon viele Klassen rüberkopiert wurden. Kurz gefasst gab es folgende Probleme:

- Es mussten immer mehr Klassen kopiert werden, was alles auf der

Saros-Seite immer mehr aufgebläht hat.

- Viele nützliche Funktionalitäten, wie die Konsolenausgabe für JavaScript-Ausgaben konnten nicht einfach genutzt werden.
- Es mussten viele Anpassungen im ExceptionHandling gemacht werden.

All diese Gründe haben dazu geführt, dass der Quellcode immer unübersichtlicher wurde, bis ich irgendwann mehr damit beschäftigt war, das vernünftig zum Laufen zu bringen, anstatt den Inhalt des Reiters zu gestalten. Das erhöht auch den Zeitdruck, denn am Ende der Arbeit sollte logischerweise ein fertiger zweiter Prototyp stehen und kein Haufen an Quellcode, über den jeglicher Überblick fehlt. Von daher musste ich eine Entscheidung treffen, die ich im folgenden Kapitel näher erörtern werde.

6.4 Implementierung mithilfe von SWT

Nachdem nun die technischen Schwierigkeiten bei der HTML-basierten Umsetzung ohne Anbidnung des Plug-ins dazu geführt haben, dass die Zeit immer knapper wurde und ich noch keinen fertigen zweiten Prototyp entwickelt hatte, musste ich mich entscheiden: Entweder ich versuche den HTML-basierten Ansatz weiter oder entwickle den Inhalt des Reiters mit SWT. Für mich war damals ausschlaggebend, dass ich zum einen grundlegende Erfahrung in SWT hatte und zum anderen wollte ich nicht das Risiko eingehen, den HTML-basierten Ansatz weiter zu verfolgen und wenn das auch erfolglos bleiben würde, keine Zeit mehr für die Entwicklung eines zweiten Prototyps mithilfe von SWT bleiben würde. Insofern habe ich mich nach Absprache entschieden, den zweiten Prototyp mithilfe von SWT umzusetzen und wenn nach der Fertigstellung und dem Schreiben noch Zeit bleibt, den HTML-basierten Ansatz durch Anbindung des Plug-ins auszuprobieren, was letztendlich auch der Fall war.

Dieses Kapitel beschreibt die Implementierung des zweiten Prototyps mithilfe von SWT. Es hat nicht die Absicht, die Programmierung bis ins kleinste Detail mit Quelltext zu darzulegen, sondern soll nur die ungefähre Idee der Umsetzung beschreiben. Der komplette Quelltext ist in [D14] vorzufinden[37].

6.4.1 Technische Umsetzung

Bei der Chat-Ansicht in Java handelt es sich um ein SWT Composite, namens `ChatRoomsComposite`, und verwaltet den Inhalt dieser Ansicht, also sowohl die korrekte Anzeige der Hinweistexte als auch die Reiter für den Multi-User-Chat und die User-User-Chats. Die Reiter werden dabei innerhalb des

[37] Da der Quellcode teilweise sehr lang ist, wurde darauf verzichtet, hier wichtige Codeabschnitte vorzustellen, denn es würden am Ende zu viele sein, sodass hier nur die Grundidee vorgestellt werden soll.

`ChatRoomsComposite`'s in einem `CTabFolder`[38] organisiert. Jeder Tab stellt dabei ein Objekt der Klasse `CTabItem`[39] dar. Auch die Tabs für die Chats sind dabei Instanzen dieser Klasse. Demnach muss ein neues `CTabItem` erzeugt und im `ChatRoomsComposite` hinzugefügt werden.

Die Klasse `ChatRoomsComposite` war natürlich an die Verwaltung von Tabs für die Anzeige von Chats angepasst. Ein Tab, der einen Chat darstellt, implementiert das `IChat` Interface. Demnach wurden an vielen Stellen im `ChatRoomsComposite` überprüft, ob die Tabs dieses Interface implementieren. Gleichzeitig wurden Listener erzeugt, die bestimmte Sachen überprüfen, wie ob etwas geschlossen wurde, ob die Sitzung noch besteht, ob etwas in den Saros-Einstellungen angepasst wurde und vieles mehr. Infolgedessen musste an vielen Stellen der Code angepasst bzw. Überprüfungen auf Implementierung einer Schnittstelle mussten erweitert werden. Natürlich ist somit der Namen `ChatRoomsComposite` nicht mehr ganz zutreffend und im Quellcode gibt es ebenfalls mehrere Stellen oder Variablen, die umbenannt werden müssten. Zum Beispiel heißt der `CTabFolder`, der die Reiter für die Chats verwaltet, `chatRooms`, obwohl nun ein weiteres `CTabItem` enthalten ist, welches keinen Chat-Raum darstellt. Allerdings wurde mit den Hauptentwicklern von Saros abgesprochen, dass die korrekte Benennung und das Refactoring nicht durchgeführt werden muss, da in der Zukunft am Chat noch gearbeitet wird und diese Aufgabe dann mit eingebunden wird. Alle übrigen Anpassungen am `ChatRoomsComposite`, die im Rahmen dieser Aufgaben durchgeführt werden mussten, wurden ebenfalls von den Hauptentwicklern akzeptiert.

Der Inhalt eines `CTabItem`'s wiederum ist ein weiteres Composite, namenns `ActivityLogControl`[40], der sich um viele verschiedene Aufgaben kümmert:

- Erzeugung der Aktivitätenzeilen
- Erzeugung der Trennlinien
- Erhalt der neuen Awareness Informationen
- Weitere Aufgaben (Sitzungsinformationen, Einstellungen, ...)

Das eigentliche Anzeigen der Inhalte des Reiters wird durch diese Klasse gesteuert. Die Aktivitätenzeilen und Trennlinien werden in der Klasse `ActivityLogDisplay`[41] technisch umgesetzt.

[38] http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/custom/CTabFolder.html

[39] http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/custom/CTabItem.html

[40] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1974/17/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/ui/widgets/activitylog/ActivityLogControl.java

[41] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1974/17/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/ui/widgets/activitylog/ActivityLogDisplay.java

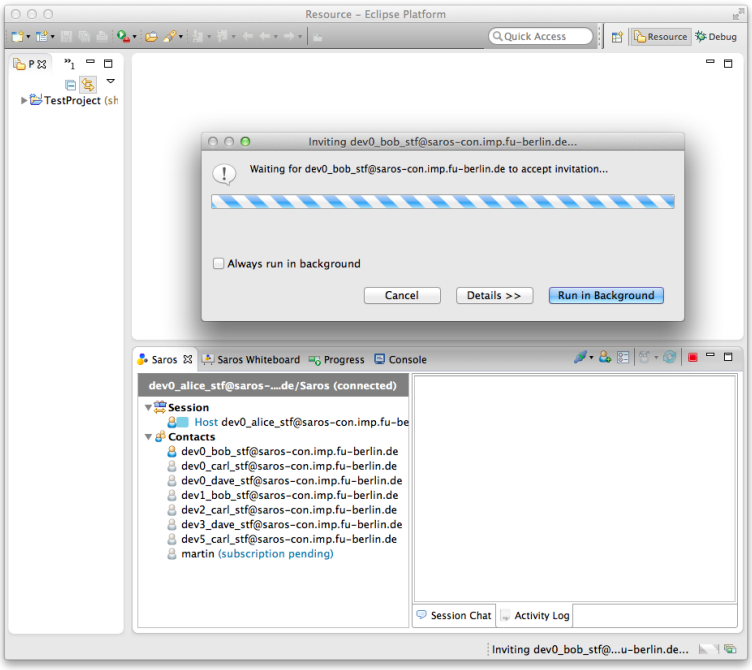


Abbildung 36: Einladeprozess: Beim Starten einer Sitzung wird eine leere Activity Log geöffnet.

6.4.2 Funktionen

Dieser Abschnitt soll das Ergebnis der Implementierung des zweiten Prototyps mittels SWT vorstellen. Startet man eine Sitzung, so werden ein Tab für den Multi-User-Chat und die neue Activity Log erstellt, sind jedoch während des Einladeprozesses noch leer, wie in Abbildung 36 zu sehen ist. Wurde die Einladung angenommen, so wird das zu teilende Projekt mit dem Arbeitsbereich des Sitzungsteilnehmers synchronisiert. Sobald die Sitzung aufgebaut wurde, werden bereits Action Awareness Informationen ermittelt und übertragen. Das sieht man gut in Abbildung 37: Beim entfernten Sitzungsteilnehmer wurde vom Editor zum Package Explorer gewechselt, was passiert ist, als dieser das Projekt bei der Synchronisierung neu anlegen ließ. In Abbildung 38 ist zu sehen, welche Informationen übertragen werden, wenn ein JUnit-Test gestartet wurde. Nachdem in Eclipse ein JUnit-Test gestartet wird und man sich nicht in der JUnit-View befindet, so wird nach dem Start diese aktiviert. Genau dieses Verhalten ist auch in der Abbildung zu sehen. Ist der Test durchgelaufen, so wird dies mit dem Ergebnis übertragen. Fehlgeschlagene Tests werden Rot angezeigt. Tests mit Fehlern werden blau und erfolgreiche Tests sind grün hervorgehoben. Bei Refactorings werden die beschreibenden Texte wie in Kapitel 4.5.3 beschrieben dem `IRefactoringHistoryService` entnommen, wie in Abbildung 39 zu sehen ist. Da dieser Text fest vorgegeben

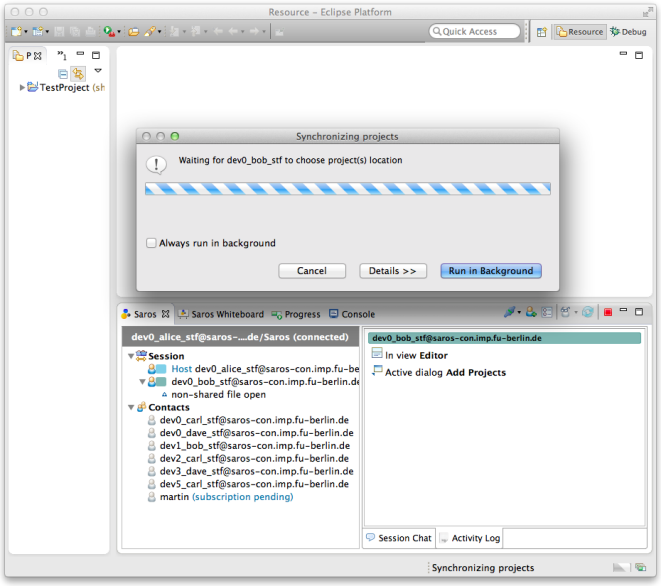


Abbildung 37: Synchronisierung: Während der Synchronisierung wurde die Sitzung bereits aufgebaut. Beim Start einer Sitzung werden bereits Action Awareness Informationen ermittelt und übertragen.

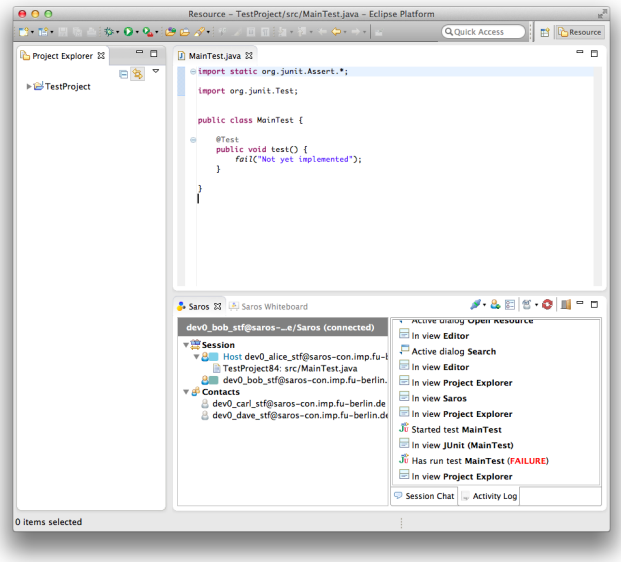


Abbildung 38: Testen: Die Activity Log zeigt an, zwischen welchen Views gewechselt wurde, welcher Dialog offen ist und auch, welcher Test gestartet und mit welchem Ergebnis dieser beendet wurde. Fehlgeschlagene Tests werden Rot angezeigt. Tests mit Fehlern werden blau und erfolgreiche Tests sind grün hervorgehoben.

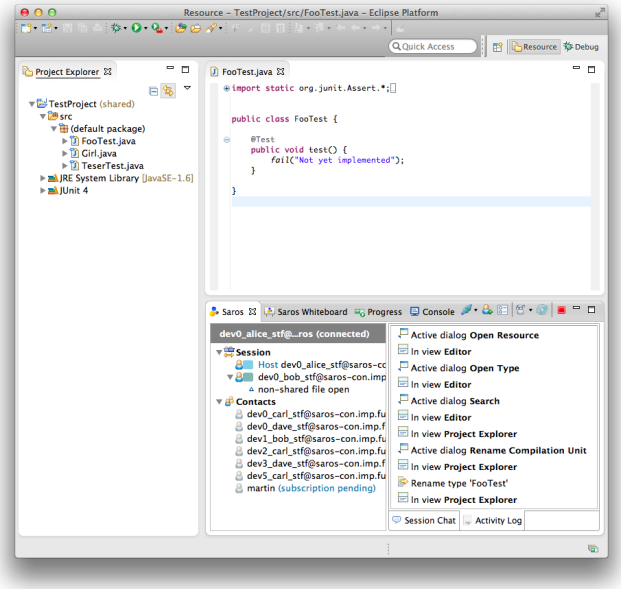


Abbildung 39: Refactoring: Hier wurde ein Test umbenannt. Die angezeigte Beschreibung wurde dem `IRefactoringHistoryService` entnommen.

ist und nicht immer derselbe Teil des Strings variable ist, kann hier nicht generisch ein Teil des Strings formatiert werden. Insofern wird dieser in der ermittelten Form direkt angezeigt. Werden Dateien erzeugt, so wird wie in Abbildung 40 zu sehen ist, diese Information ebenfalls übertragen. Dabei ist die Umsetzung unabhängig davon, was für eine Datei erzeugt wurde, also ob es sich um ein Java Interface, eine Java Klasse oder eine komplett andere Datei handelt.

6.5 HTML-basierte Implementierung mit Plug-in

Nachdem nun der zweite Prototyp mittels SWT implementiert wurde und einige Tage vor der Abgabe die zweite Fassung dieser schriftlichen Ausarbeitung fertiggestellt war, blieb noch Zeit für einen zweiten Anlauf für die HTML-basierte Implementierung, und zwar wurde dieses Mal das gesamte Plug-in von Björn (siehe Kapitel 6.3) in das Saros-Projekt eingebunden. Dieses Kapitel soll beschreiben, wie dabei vorgegangen wurde und was das Resultat dieses Vorgehens war.

6.5.1 Versuch unter Windows

Dieses Kapitel soll beschreiben, wie ich versucht habe, das Plug-in von Björn Kahlert unter Windows einzurichten und warum es vermutlich gescheitert ist. Für diesen Zweck habe ich eine frische Eclipse 4.3 Version heruntergeladen,

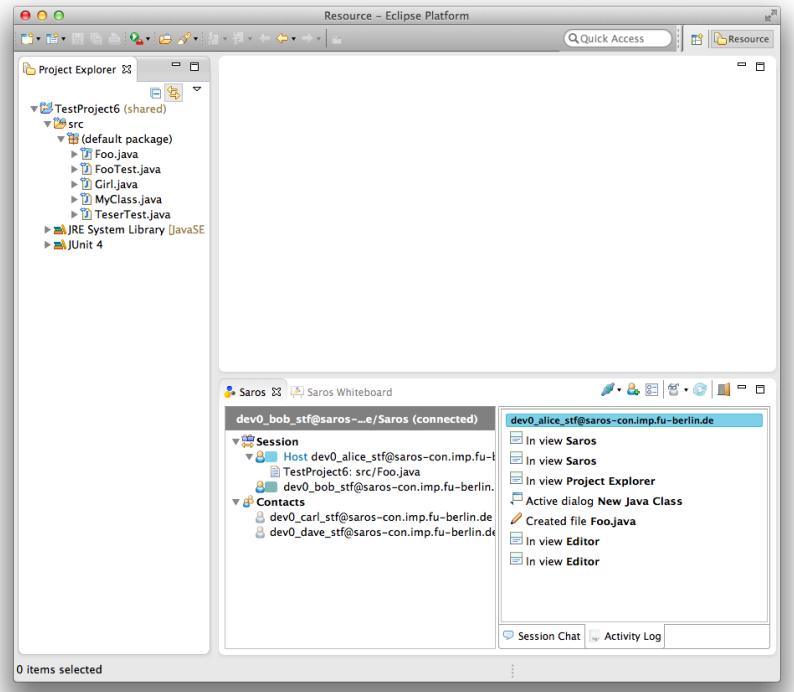


Abbildung 40: Erzeugung von Dateien: Hier wurde eine neue Datei erzeugt und die Information übertragen. Dabei spielt es keine Rolle, ob es sich um ein Interface, eine Klasse oder andere Arten von Dateien handelt.

um herauszufinden, was die Minimalanforderungen für die Arbeit mit diesem Plug-in sind.

Nachdem das Plug-in ausgecheckt[42] wurde, wurde es in der frisch installierten Version von Eclipse importiert. Die vom Plug-in erforderlichen Bundles sind in einer Standard-Eclipse Version nicht vorinstalliert, sodass man sie herunterladen muss. Dabei handelt es sich um folgende Bundles[43]:

- Apache Commons Collections
- Apache Commons IO
- Apache Commons Lang
- Apache Jakarta Log4j Plug-in
- Jackson JSON processor
- Data mapper for Jackson JSON processor

Als alle Bundles installiert waren und Eclipse neugestartet wurde, wurden auch keine Kompilierfehler mehr angezeigt. Das Plug-in wurde wie Björn Kahlert es empfohlen hatte, gestartet, indem es im Debug-Modus als Eclipse-Applikation ausgeführt werden sollte. Allerdings wurde dabei eine Exception geworfen und nur Teile des Plug-ins wurden angezeigt, wie es in Abbildung 41 zu sehen ist. Der Stacktrace zeigt folgende Meldung:

```
java.lang.RuntimeException: java.net.URISyntaxException:
Illegal character in opaque part at index 15:
bundleresource:\67.f...
```

Die genaue Stelle, an der die Exception geworfen wurde, konnte erst durch Debuggen ermittelt werden, da hier eine `RuntimeException` geworfen wurde. Abbildung 42 zeigt die Klasse und die entsprechende Stelle im Quellcode, an der dieser Fehler auftritt. An dieser Stelle wurde nicht mehr weiterversucht, das Plug-in auf Windows auszuführen. Björn Kahlert hat das Plug-in auf Mac OSX entwickelt und da ich selbst auf demselben System entwickle, habe ich einen zweiten Anlauf unter Mac OSX gestartet, der im folgenden Kapitel beschrieben wird.

6.5.2 Versuch unter Mac OSX

Dieses Kapitel beschreibt den zweiten Anlauf der HTML-basierten Implementierung des Inhalts des neuen Reiters durch Einbindung des Plug-ins von Björn. Die Einrichtung des Plug-ins erfolgte genauso wie es in Kapitel 6.5.1 beschrieben wurde. Wenn es als Eclipse Application gestartet wurde und alles korrekt angezeigt wird, so ist das Plug-in einsatzfähig.

[42] https://github.com/bkahlert/com.bkahlert.nebula

[43] Alle benöigten Bundles sind auf dieser Update Site zu finden: http://download.eclipse.org/tools/orbit/downloads/drops/R20140114142710/repository/

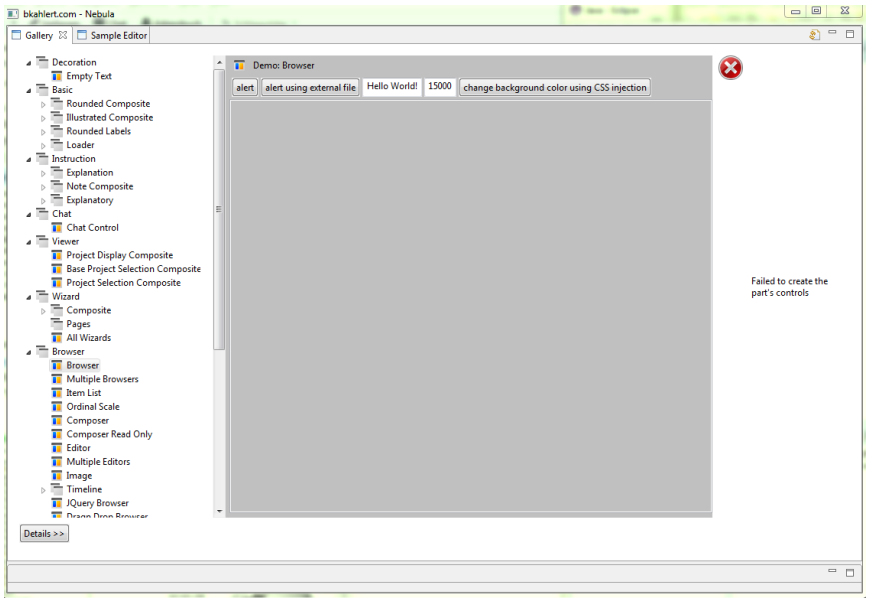


Abbildung 41: Bei der Ausführung des Plug-ins unter Windows tritt ein Fehler auf und das Plug-in wird fehlerhaft angezeigt.

Abbildung 42: Ein Pfad kann aufgrund eines Doppelpunktes nicht geparst werden.

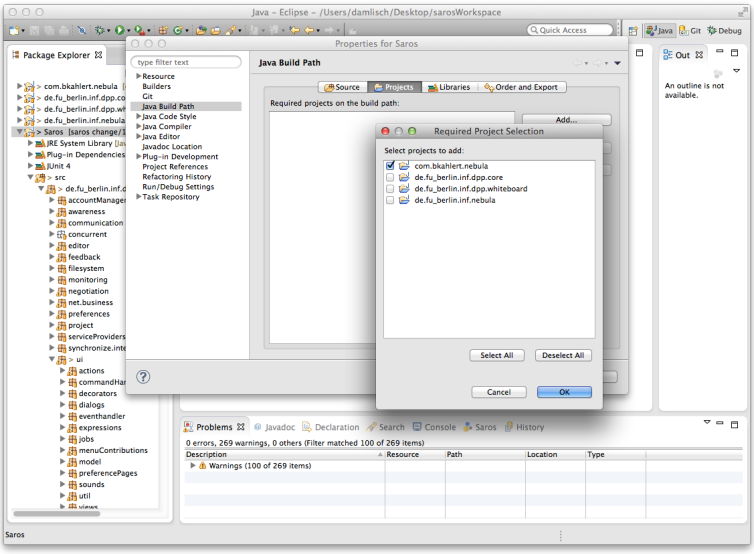


Abbildung 43: Einbindung des Plug-ins von Björn in das Saros Projekt

Einbindung in das Saros-Projekt

Um das Plug-in in Saros einzubinden, muss dieses im Saros-Projekt als abhängiges Projekt eingerichtet werden. Dazu geht man auf das Saros-Projekt und klickt nach einem Rechtsklick auf „Build Path“ und dort auf “Configure Build Path“. Dort geht man in den Bereich „Projects“ und kann im neuen Dialog das Plug-in von Björn Kahlert auswählen (siehe Abbildung 43). Nach diesem Schritt können die Klassen des Plug-ins nach einem Import in den entsprechenden Saros-Klassen wie gewohnt verwenden.

Einbinden der Browser-Komponente

In Kapitel 6.4 wurde vorgestellt, wie der neue Reiter als neues `CTabItem` in die Chat-View von Saros eingefügt werden konnte. Und zwar wurde es dabei in die Klasse `ChatRoomsComposite` eingefügt. Die Klasse `ActivityLogControl`[44] kümmerte sich dabei um die Anzeige des Reiterinhaltes mithilfe von SWT. Anstelle eines neuen Composites, welchen mithilfe von SWT-Funktionen den Inhalt darstellt, muss nun stattdessen die Browser-Komponente von Björn's Plug-in eingebunden werden. Listing 35 zeigt, wie es letztendlich umgesetzt wird. Dabei wurden Methoden und Objekte, die bereits in der SWT-Implementierung vorhanden waren und benötigt werden (z. B. die Behandlung der Sitzung oder verschiedene Listener-Implementierungen), hier aus Übersichtlichkeit weggelassen bzw. durch ein Kommentar markiert. Der

[44] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1974/17/de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/ui/widgets/activitylog/ActivityLogControl.java

vollständige Quellcode für diese Datei ist im SVN[45]

```
1  public class ActivityLogControl extends Composite {
2
3      // ...
4
5      @Inject
6      private AwarenessInformationCollector collector;
7      private ActivityLogDisplay activityLogDisplay;
8
9      public ActivityLogControl(final ChatRoomsComposite
           chatRooms,
10         final Composite parent, final int style,
11         final Color displayBackgroundColor) {
12
13         super(parent, style & ~SWT.BORDER);
14         SarosPluginContext.initComponent(this);
15         setLayout(new FillLayout());
16         activityLogDisplay = new ActivityLogDisplay(this,
               SWT.BORDER);
17
18         // ...
19
20         collector.addAwarenessUpdateListener(
               awarenessUpdateListener);
21     }
22
23     // ...
24     private final AwarenessUpdateListener
           awarenessUpdateListener = new
           AwarenessUpdateListener() {
25
26         @Override
27         public void update(final ActionTypeDataHolder data)
               {
28             assert data != null;
29             final User user = data.getUser();
30             Color color = getColorForUser(user);
31             activityLogDisplay.addMessage(user, data, color)
                   ;
32         }
33     };
34 }
```

Listing 9: Einbinden der Browser-Komponente in den neuen Activity Log Reiter

Die Klassendefinition bleibt gleich. Es handelt sich also immernoch um ein Composite. Im Konstruktor muss nun statt eines neuen Composites, welches den Inhalt des Reiters mithilfe von SWT darstellen soll, nun die Browser-Komponente aus Björn Kahlert's Plug-in einfügen. In Zeile 13 wird

[45] https://svn.imp.fu-berlin.de/agse/students/durmaz/Browser/MacOSX/activitylog/ActivityLogControl.java

wie gewohnt der Superkonstruktoraufruf durchgeführt. Zeile 15 ist wichtig, denn nur wenn das Layout dieses Composites ein `FullLayout` ist, wird der Inhalt, also die Browser-Komponente vollständig angezeigt. Anschließend wird das Objekt `ActivityLogDisplay` erzeugt, welches die Browser-Komponente ist. Die genaue Umsetzung dieser Klasse wird im folgenden Abschnitt genauer beschrieben. Die restlichen Zeilen im Konstruktor kümmern sich um die Verwaltung der Sitzung und um die Registrierung von Listenern. Zeile 20 zeigt, wie der `AwarenessUpdateListener` hinzugefügt wird, der in Zeile 24 erzeugt wird. Dort sieht alles genauso aus wie in der Implementierung mittels SWT: Das Objekt zum Anzeigen des Inhaltes, nämlich `ActivityLogDisplay`, ruft die Methode `addMessage(user, data, color)` auf, in dem letztendlich dem Inhalt des Reiters eine neue Aktivitätenzeile eingefügt wird. Damit ist die Browser-Komponente eingebunden.

Gestaltung der Browser-Komponente

In diesem Kapitel wird beschrieben, wie nun die Browser-Komponente mithilfe von Java-Code gestaltet werden kann. Listing 67 zeigt hierbei die Umsetzung, wobei erneut irrelevante Stellen aus Übersichtlichkeit weggelassen wurden. Der Originalcode ist ebenfalls im SVN[46] vorzufinden.

```
1  public class ActivityLogDisplay extends BootstrapBrowser {
2
3      // ...
4      private Object lastUser;
5      private int timeout = 60000;
6
7      public ActivityLogDisplay(Composite parent, int style) {
8          super(parent, style | SWT.INHERIT_FORCE);
9          this.open(BrowserUtils.getFileUrl(ActivityLogDisplay
               .class,
10             "html/index.html", "?internal=true"), timeout);
11     }
12
13     public void addMessage(User user, ActionTypeDataHolder
           data, Color color) {
14
15         // sender's line
16         if (lastUser == null || !lastUser.equals(user)) {
17             // new / different user
18             RGB rgbColor = new RGB(color.getRGB().red, color
                   .getRGB().green,
19                 color.getRGB().blue);
20             addOriginatorChangeLine(user, rgbColor);
21         }
22
23         // activity line
```

[46] https://svn.imp.fu-berlin.de/agse/students/durmaz/Browser/MacOSX/activitylog/ActivityLogDisplay.java

```
24          addActivityLine(data);
25          lastUser = user;
26      }
27
28      private void addActivityLine(ActionTypeDataHolder data)
            {
29          StringBuilder sb = new StringBuilder();
30          ActionType type = data.getType();
31
32          // ...
33          if (type == ActionType.ADD_TESTRUN) {
34
35              String css = "b.junit-green { color: rgb(0, 255,
                    0); }"
36                  + "b.junit-red { color: rgb(255, 0, 0); }"
37                  + "b.junit-blue { color: rgb(0, 0, 255); }";
38              this.injectCss(css);
39
40              sb.append("<p><img src=\"img/junitstart.gif\"
                    class=\"img-rounded\" style=\"display:
                    inline;\">");
41              if (data.getTestRunState() == State.UNDEFINED)
42                  sb.append(" Started test <b>" + data.
                        getTestRunName()
43                      + "</b>");
44              else {
45                  if (data.getTestRunState() == State.OK)
46                      sb.append(" Has run <b>" + data.
                            getTestRunName()
47                          + "</b> (<b class=\"junit-green\">"
48                          + data.getTestRunState() + "</b>)");
49                  else if (data.getTestRunState() == State.
                        ERROR)
50                      sb.append(" Has run <b>" + data.
                            getTestRunName()
51                          + "</b> (<b class=\"junit-blue\">"
52                          + data.getTestRunState() + "</b>)");
53                  else if (data.getTestRunState() == State.
                        FAILURE)
54                      sb.append(" Has run <b>" + data.
                            getTestRunName()
55                          + "</b> (<b class=\"junit-red\">"
56                          + data.getTestRunState() + "</b>)");
57              }
58              sb.append("</p>");
59          }
60          this.run("\$(" + JSONUtils.enquote(sb.toString())
61              + ").appendTo('body');");
62          this.scrollTo(0, 9999);
63      }
64
65      // ...
66  }
```

Listing 10: Gestaltung der Browser-Komponente

Bei dieser Klasse handelt es sich also um den eigentlichen Browser. Je nachdem, für welches Vorgehen aus Kapitel 6.3.1 man sich entscheidet, sieht der Konstruktor dieser Klasse anders aus. Ich habe mich aus diversen Gründen, die später noch aufgeführt werden, für die zweite Vorgehensweise entschieden, also baue ich den Browser direkt aus dem Java-Code heraus. In Zeile 9 öffne ich eine leere HTML-Datei, namens `index.html`, die nur aus einem leeren HEAD- und BODY-Tag besteht. Den zweiten Parameter `?internal=true` übergibt man, da das Plug-in von Björn Kahlert so eingestellt ist, dass die JavaScript-Datei `facade.js`, die für die Übermittlung von Java-Code-Aufrufen an JavaScript-Aufrufe zuständig ist, diesen Parameter überprüft und wenn dieser fehlt, Dummy-Daten erzeugt werden, sodass man den Inhalt des Browsers auch so testen kann, ohne sich um die Ermittlung von Daten zu kümmern. Wie man am Klassenkopf erkennen kann, handelt es sich bei diesem Browser um einen `BootstrapBrowser`. Björn Kahlert hat in seinem Plug-in unter anderem eine normale Browser-Komponente, eine mit Twitter Bootstrap eingebunden und eine mit nur JQuery eingebunden zur Verfügung gestellt. Bei den beiden letzten Varianten hat man so den Vorteil, dass man bereits bekannte CSS-Frameworks und JQuery zur Verfügung hat, was den Vorteil hat, dass man nicht komplett mit reinem HTML, CSS und JavaScript arbeiten muss.

In der `addMessage(User user, ...)` Methode werden nun neue Zeilen hinzugefügt. Zuerst wird überprüft, ob es sich um denselben Teilnehmer handelt. Wenn nicht, wird eine neue Trennlinie für einen Nutzer erzeugt, wie sie auch schon im SWT-basierten Reiter zu sehen ist. Die genaue Implementierung der Methode aus Zeile ist im oben genannten Quellcode zu finden. Am Ende wird eine neue Aktivitätenzeile hinzugefügt, was in der Methode `addActivityLine(...)` in Zeile 28 geschieht. Zuerst baut man sich einen `StringBuilder`[47], mit dem man sein HTML-Gerüst Stück für Stück zusammenbaut. In Zeile 35 wird eine CSS-Eigenschaft konstruiert. Um diese anwenden zu können, muss man die Methode `injectCSS(css)` aufrufen, der man die konstruierte CSS-Eigenschaft in Form eines Strings übergibt. Man könnte auch das ganze CSS in eine Datei auslagern und dem Browser diese Datei mithilfe der Methode `injectCssFile(uri)` übergeben, sodass man Darstellung und Design noch stärker trennen kann, was zur Einfachheit hier weggelassen wurde. In Zeile 40 kann man sehen, wie der HTML-Tag aufgebaut wird: Zuerst wird ein P-Tag eingefügt und danach ein IMG-Tag, welches das Icon darstellt. Die Klasse dieses Tags entspricht gerade einer Eigenschaftklasse aus Twitter-Bootstrap, die durch die Nutzung dieser Browser-Klasse bereits eingebunden ist. Zeile 47 zeigt z. B. , wie eine

[47] http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

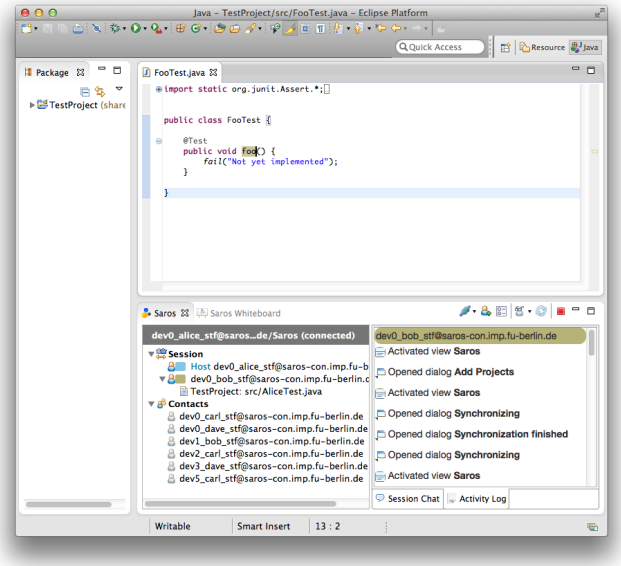


Abbildung 44: Beim Wechsel eines Nutzers oder Sitzungsstart wird eine Trennline dargestellt.

CSS-Eigenschaft, die vorher in Zeile 38 injiziert wurde, nun angewendet wird. Und zwar wird der Text zwischen den B-Tags hier grün dargestellt. Am Ende wird in Zeile 58 der HTML-Tag abgeschlossen und in Zeile 60 wird dieser über die JQuery-Methode `$(...).appendTod('body')` dem BODY-Tag des HTML-Dokuments hinzugefügt. Zeile 62 dient dazu, dass der Browser nach unten scrollt, um die aktualisierte Information anzuzeigen.

Ergebnis

Das letzte Kapitel hat gezeigt, wie die Gestaltung der Browser-Komponente umgesetzt wurde. Man kann sehen, dass man dadurch, dass man die Gestaltung mithilfe von HTML, CSS und JavaScript durchführen kann, wirklich schnell Änderungen einarbeiten kann. Wenn man z. B. die Farben oder die Gestaltung der Tags ändern möchte, kann man das ganz schnell anpassen und könnte Saros sogar im Debug-Modus starten, sodass man Änderungen am CSS, HTML und JavaScript durch ein einfaches Refresh des Reiterinhaltes sofort sehen kann. Abbildung 44 zeigt, wie der Reiterinhalt nun in der HTML-basierten Implementierungsweise aussieht. Abbildung 45 zeigt, wie es während einer Sitzung aussieht. Man sieht, wie die Zeilen für das Öffnen von Dialogen, Aktivieren von Views, Ausführen von Tests, Erzeugen von Dateien und Durchführen von Refactorings aussehen. Wenn man diese Anzeige mit der SWT-basierten Implementierung vergleicht, wie sie z. B. in Abbildung 38 zu sehen ist, merkt man, dass sie sich kaum unterscheiden.

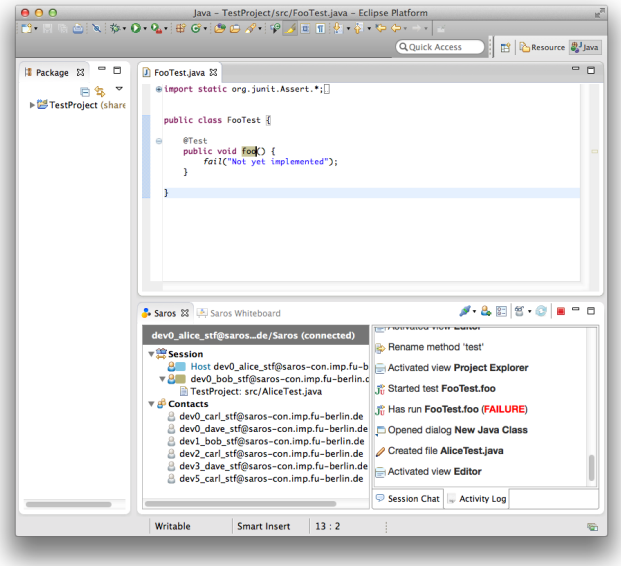


Abbildung 45: Der Inhalt sieht fast genauso aus wie in der SWT-Lösung und würde bei Anpassungen an Rändern und einigen Schriftoptimierungen genauso aussehen.

Natürlich kann man kleine Abstände oder die Trennlinie etwas anpassen, damit es noch mehr aussieht wie in der SWT-basierten Implementierung. Aber Ziel dieses Versuches war es nicht, den Inhalt des Browsers 1:1 wie den Inhalt der SWT-basierten Implementierung zu gestalten, sondern mit so wenig Aufwand wie möglich den Inhalt so ähnlich wie möglich zu gestalten, damit man sehen kann, wie viel Aufwand man insgesamt betreiben muss, um den Inhalt von Widgets mithilfe von HTML und weiteren Webtechnologien zu gestalten.

6.6 Vergleich beider Ansätze

Insgesamt haben alle Ansätze gezeigt, dass es drei Wege gibt, den Inhalt eines Widgets zu gestalten:

1. HTML-basierter Ansatz ohne Einbindung des Plug-ins
2. SWT-basierter Ansatz
3. HTML-basierter Ansatz mit Einbindung des Plug-ins

Der erste Ansatz hat sich als keine gute Vorgehensweise herausgestellt, weil es irgendwann einfach unübersichtlich wurde. Beim SWT-basierten Ansatz konnte man sehen, dass der Inhalt des Reiters wiederum ein Composite

war, welches wiederum aus vielen kleinen Composites bestand. Auf der anderen Seite konnte man beim HTML-basierten Ansatz mit Plug-in Einbindung sehen, dass der Inhalt des Reiters eine Browser-Komponente war, die dank der vielen Funktionen des Plug-ins mithilfe von bekannten Webtechnologien gestaltet werden konnte. Im folgenden sollen die letzten beiden Vorgehensweisen miteinander verglichen werden, um besser einschätzen zu können, welche Vorgehensweise auch in der Zukunft eine gute zu sein scheint. Solch ein Vergleich gelingt nur, wenn sie beiden Ansätze nach gleichen Kriterien bewertet werden. Ein Kriterium ist hierbei ein Szenario, in dem untersucht wird, wie die Umsetzung bei beiden Ansätzen aussehen würde und welche Vor- oder Nachteile es gibt.

6.6.1 Erzeugung der Gesamtstruktur

Hierbei geht es darum zu untersuchen, wie es bei beiden Ansätzen aussehen würde, wenn man die Grundstruktur des Reiterinhaltes komplett neu bauen müsste. Beim SWT-basierten Ansatz muss man für jedes Element ein neues Composite bauen, was natürlich auf Dauer schnell unübersichtlich werden kann. Beim HTML-basierten Ansatz muss man auch eine Grundstruktur innerhalb des HTML-Dokuments bauen. Entweder baut man es auf der Java-Code-Seite, wie ich es umgesetzt habe, oder man baut eine komplette Webseite und sendet dieser die Daten per JavaScript-Aufrufe und der Browser kümmert sich um das Rendering. Der HTML-basierte Ansatz hat hier einen klaren Vorteil, denn man könnte vor der Umsetzung in einem SWT-Widget die komplette Webseite einmal im browser bauen und das dann am Ende in den Reiter einbauen, egal ob man die komplette Seite im Code baut oder per JavaScript-Aufrufe durch den Browser aufbauen lässt. Zwar gibt es reichlich Tools, wie den *WindowBuilder*[48], mit dem man GUI-Anwendungen per Drag&Drop gestalten kann, aber die Gestaltung von Webseiten lassen sich in der Regel schneller durchführen als wenn man das gleiche Layout mittels ineinander verschachtelten Composite-Objekten umsetzen wollen würde.

6.6.2 Gestaltung des Browser-Inhaltes

Kapitel 6.5.2 und Kapitel 6.5.2 haben gezeigt, wie die Klassen `ActivityLogControl` und `ActivityLogDisplay` in beiden Vorgehnsweisen aussehen. Man kann sehen, dass die Klassen, die nach dem SWT-basierten Ansatz aufgebaut sind, wesentlich länger sind und viel mehr Aufwand dahinter steckt, die Anzeige zu gestalten, während beim HTML-basierten Ansatz die meiste Arbeit darin liegt, das CSS und das JavaScript zu gestalten (bzw. den HTML-Code, wenn man diesen komplett neu auf der Java-Code Seite baut). Insgesamt konnte ich auch den HTML-basierten Ansatz schneller umsetzen als den SWT-basierten Ansatz.

[48] http://www.eclipse.org/windowbuilder/

6.6.3 Aufwand bei Änderungen

Kommt es mal im Rahmen von Nutzertests oder Feedback zu Änderungswünschen des Aussehens, so zeigt sich ganz klar der HTML-basierte Ansatz als Gewinner. Als ich z. B. die Farbgebung der Ergebnisse der JUnit-Tests einbauen wollte, musste ich im SWT-basierten Ansatz den Substring, der das Ergebnis darstellt, extrahieren, ein Objekt erzeugen, welches eine Formatierung darstellt (`StyledString`[49]) und dieses Objekt dem Composite, welches den Text darstellt, über eine bestimmte Methode zuweisen. Das war eine sehr fehleranfällige Aufgabe und hat mich bei der SWT-basierten Implementierung auch am meisten Zeit gekostet. Wenn ich ein Beispiel für die HTML-basierte Vorgehensweise ausführe, sieht man sofort den Vorteil: Kurz bevor ich die HTML-basierte Implementierung den wissenschaftlichen Mitarbeitern vorstellen wollte, habe ich festgestellt, dass die Farbgebung der JUnit-Test Ergebnisse fehlt, musste ich nur im CSS die Farbe hinzufügen, diese CSS-Eigenschaft hinzufügen und damit war die Arbeit getan. Vorher habe ich es noch einmal lokal im Browser getestet, um zu schauen, ob die Farbe am Bildschirm auch angenehm rüberkommt. Das Anpassen der Farbe konnte sehr schnell durchgeführt werden. So ähnlich würde das in Zukunft auch in anderen Szenarien ablaufen, wo das Aussehen eines Widgets angepasst werden muss.

6.6.4 Testen

Das Schreiben von Tests würde in allen Fällen funktionieren. Nur stellt sich die Frage, wie aufwändig das je nach Ansatz ist und wie gut die Tests plattformunabhängig sind. Beim SWT-basierten Ansatz testet man z. B. nach vorkommen bestimmter Widgets im Reiter, beim HTML-basierten Ansatz testet man z. B. , ob der erstellte P-Tag dem entspricht, was man erwartet. Hat man eine Browser-Komponente gebaut, die nicht vom Java-Code aus gebaut wird, sondern quasi nur noch per JavaScript-Aufrufe mit Daten „gefüttert“wird, kann man immernoch die zu versenden Daten testen. Allerdings kann man Tests mithilfe der HTML-basierten Weise leichter plattformunabhängig gestalten. Während man bei SWT für andere IDE's komplett neue Tests schreiben müsste, müsste man beim HTML-basierten Ansatz nur den Container plattformspezifisch testen und das Testen des Browser-Inhaltes würde bei allen IDE's ziemlich identisch sein.

6.6.5 Komplexität der Umsetzung und Threadhandling

Das ist ein Kriterium, was ziemlich subjektiv bewertet werden kann. Ich habe vor der Arbeit an Saros geringfügige SWT-Erfahrung gehabt und

[49] http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/viewers/StyledString.html

konnte im Zuge der Mitarbeit im Saros-Projekt meine Kenntnisse in der SWT-Programmierung steigern. Trotzdem hat die Umsetzung mittels SWT mehrere Tage und aufgrund des Reviewprozesses sogar noch länger gedauert. Die Logik ist zwar in wenigen Stunden fertig gewesen, jedoch treten die meisten Probleme beim Threadhandling auf. Werden in der SWT-basierten Technik Widgets aus falschen Threads heraus aktualisiert, führt das zu Exceptions. Gerade diese Eigenschaft hat bei der SWT-basierten Umsetzung viel Zeit gekostet. Selbst im fortlaufenden Reviewprozess sind den Reviewern zwischendurch noch weitere Stellen aufgefallen, die in der Zukunft Probleme verursachen könnten. Beim HTML-basierten Ansatz dagegen muss man sich so gut wie gar nicht um Threadhandling kümmern. Die einzige Stelle, wo so etwas nötig wird, ist wenn man auf Ergebnisse des Browsers, z. B. nach einem JavaScript-Aufruf, wartet. So geben die Methoden, wie

```
this.run("\$(" + JSONUtils.enquote(sb.toString())
            + ").appendTo('body');");
```

Objekte vom Typ `Future<Object>` zurück, auf die man die Methode `get()` aufrufen kann. Dies sollte man nur außerhalb des Main-Threads machen, da es sonst zu Deadlocks in der Anwendung kommen und dieser einfrieren kann. Für diesen Fall gibt es im Plug-in die Klasse `ExecUtils`, die entsprechende Methoden, wie

```
public static <V> Future<V> nonUIAsyncExec
    (final Class<?> clazz, final String purpose,
    final Callable<V> callable, final int delay)
```

zur Verfügung stellt, um genau das zu erreichen. Insgesamt war die SWT-basierte Umsetzung durch das Threadhandling komplexer und fehleranfälliger, während ich mich bei der HTML-basierten Variante um wenige Funktionen kümmern musste, da sie von der Browser-Komponente bereits erledigt wurden.

6.6.6 Fazit

Insgesamt bietet der HTML-basierte Ansatz wesentlich mehr Vorteile als der SWT-basierte Ansatz. Die Umsetzung im Rahmen dieser Arbeit hat auch gezeigt, dass es praktisch mit angemessenem Aufwand möglich ist. Außerdem ermöglicht die HTML-basierte Implementierungsweise eine schnelle Anpassung des Inhaltes bei Änderungswünschen und hat den großen Vorteil, dass wenn andere IDE's in der Lage sind, das Plug-in einzubinden, die Browser-Komponente nutzen können um den Inhalt plattformunabhängig zu gestalten. Selbst wenn das Plug-in von Björn Kahlert in anderen IDE's nicht genutzt werden kann, konnte ich mit diesem Versuch zeigen, dass es prinzipiell möglich ist, solch einen HTML-basierten Ansatz zu verfolgen. Natürlich muss in der Zukunft noch weiter untersucht werden, wie die reale Umsetzung in anderen IDE's erfolgt und vor allem, wie das in Kapitel 6.3.1 erwähnte Linux-Problem zu lösen.

7 Mitarbeit im Saros-Projekt

Saros ist ein Open-Soucre-Projekt und lebt von der Mitarbeit jedes seiner Mitglieder. Während der Anfertigung der Abschlussarbeit ist jedes Saros-Mitglied automatisch Teil dieses Open-Source-Teams und ist verpflichtet, nicht nur eigenen Code zu produzieren, sondern auch an der Gesamtentwicklung mitzuwirken oder gefundene Fehler und Verbesserungsmöglichkeiten im Bug-Tracker von Saros einzutragen[50]. Das bedeutet, dass nicht nur die Codequalität des eigenen Codes vom Team überprüft wird, sondern dass die Studentin oder der Student selbstständig auch den Code anderer Saros-Entwickler auf die Qualität hin untersucht. Dieses Vorgehen führt nicht nur zu einer stetigen Verbesserung oder Anpassung des Saros-Codes, sondern dient auch dem Wissensaustausch unter den Mitwirkenden.

Dieses Kapitel soll diesen Prozess näher beschreiben, denn dieser Prozess begleitet einen Abschlussabsolventen nicht nur während der Arbeit, sondern kann die eigene Arbeit auch beeinflussen. Zuerst wird der Entwicklungsprozess im Allgemeinen erläutert. Anschließend wird der Reviewprozess im Detail beschrieben, wobei es sich um den Prozess der Überprüfung der Codequalität handelt. Zum Schluss gibt es ein Kapitel über den persönlichen Gewinn dieses Arbeitsprozesses, aber auch über die Hürden und Probleme, die dieser Prozess mit sich bringt.

7.1 Entwicklungsprozess im Saros-Projekt

Wöchentlich finden jeden Dienstag und Freitag Stand-Up Meetings statt[51]. Diese Meetings dienten dazu, nicht nur den aktuellen Stand seit dem letzten Meeting und das Vorhaben bis zum nächsten Meeting zu kommunizieren, sondern auch Wissensaustausch zu betreiben. Dies gelang durch den ständigen Austausch über den Stand, über neue Erkenntnisse und Ergebnisse und über die gemeinsame Überlegung von Lösungsansätzen, wenn ein Teilnehmer z. B. an einem Problem festhing. Handelte es sich um größere Probleme mit mehr Redebedarf oder mit einer Einsicht in den Quellcode, so wurde das Gespräch auf die Zeit nach dem Meeting verlagert und da mussten die restlichen Mitglieder nicht mehr anwesend sein. Diese Vorgehensweise hatte große Vorteile, denn auch bei der Bearbeitung dieser Arbeit hatte ich Probleme, die innerhalb dieser Meetings schnell erkannt und bei der Behebung geholfen wurde. Es gab sogar Situationen, wo der Leiter des Stand-Up Meetings und ich noch nach dem Meeting den Code gedebuggt und die Logmeldungen intensiv auf den Grund des Problems hin untersucht haben. Auch habe ich während ich den Stand meiner Arbeit beschrieben habe, immer wieder konstruktive Kritik und neue Ideen für mein Vorgehen vorgeschlagen bekommen.

[50] Ich habe zufällig einen Bug gefunden und ihn unter https://sourceforge.net/p/dpp/bugs/839/ eingetragen

[51] Die Termine wurden währenddessen auf montags und donnerstags verlegt

7.2 Reviewprozess

Dieses Kapitel beschreibt den genauen Entwicklungsverlauf innerhalb des Saros-Teams. Dabei wird darauf eingegangen, der Prozess der Codequalitätskontrolle, den sogenannten *Reviews*, funktioniert und welchen Gewinn, aber auch welche Hürden dieser Prozess für den einzelnen Studenten mit sich bringt.

7.2.1 Reviews von eigenen Änderungen

Die Abbildung 46 beschreibt den Reviewprozess und Arbeitsprozess bei Saros[52]. Nachdem man das Git Repository ausgechekt hat, fügt man dem Saros Code seine Änderungen hinzu und pusht diese dann an *Gerrit*[53], einem webbasierten Code Review system, wo der Code dann auf die Codequalität hin untersucht wird. Solch ein Push wird auch als *Change* bezeichnet. Dieser Change kann nun von anderen Saros-Entwicklern untersucht werden. Wenn dieser Review negativ verläuft, sprich es gibt noch Anmerkungen, Fragen, Verbesserungen oder Fehler, muss der untersuchende Entwickler diese Stellen mit Kommentaren beschreiben und ein *-1* vergeben. Somit muss dieser Change verbessert und erneut hochgeladen werden. Solch eine Verbesserung wird auch als *Patch-Set* bezeichnet[54]. Dieser Reviewprozess wird mit so vielen Patch-Sets fortgesetzt, bis mindestens zwei Entwickler ein *+1* vergeben. Ist dies der Fall, wird der Change abgeschlossen und in den Master oder entsprechenden Branch zusammengefügt.

[52] http://www.saros-project.org/our-dev-workflow
[53] https://code.google.com/p/gerrit/
[54] Insofern besteht ein neu hochgeladener Change aus einem ersten Patch-Set

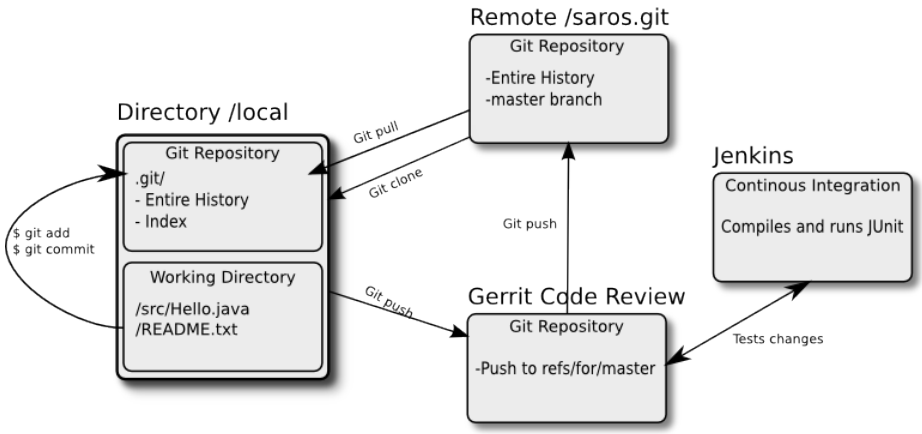


Abbildung 46: Eigen produzierter Code wird auf das Code Review System *Gerrit* hochgeladen, wo es von anderen Saros-Mitgliedern auf die Codequalität und semantische Probleme hin untersucht wird. Gleichzeitig sorgt *Jenkins* für die kontinuierliche Integration und die Durchführung der JUnit-Tests.

7.2.2 Reviews von fremden Änderungen

Der in 7.2.1 beschriebener Reviewprozess läuft in der selben Weise ab, wenn man die Änderungen anderer begutachten möchte. Man lädt sich die Änderung herunter und spielt das in seinen Arbeitsbereich ein. Dann begutachtet man den Code, führt ihn gegebenenfalls aus und spielt damit herum (falls es sich z. B. um eine Änderung oder neues Feature in der GUI handelt). In manchen Fällen musste man den Change nicht in seinen lokalen Arbeitsbereich einspielen, sondern konnte die Änderungen auch auf der Webseite des Systems betrachten, begutachten und dort auch gleich die Kommentare mit Anmerkungen, Problemen und Fragen verfassen.

7.2.3 Hürden und Gewinn

Die oben beschriebenen Prozesse stellen einen großen Gewinn, aber auch einige Hürden für den Studenten dar, der unter diesen Umständen seine Abschlussarbeit schreibt.

Es ist z. B. nicht festgelegt, wie lange ein Reviewprozess dauert. Es kann sein, dass die Reviewkommentare noch am selben Tag eintreffen, es kann aber auch sein, dass es mehrere Tage dauert. Das hängt unter anderem von zwei Faktoren ab:

- Wenn man zu große Patches, also mit zu vielen Dateien oder zu vielen Änderungen auf einmal hochlädt, kann das andere Entwickler vor dem Review abschrecken oder sie benötigen einfach viel mehr Zeit.

- Die meisten Entwickler sind selbst Studenten, die an ihren Abschlussarbeiten sitzen. Sie können sich nicht sofort, sobald man einen Patch hochgeladen hat, daransetzen, da sie gegebenenfalls selbst zu tun haben. Dasselbe gilt auch für nicht-studentische Saros-Entwickler, wie die wissenschaftlichen Mitarbeiter in der AG Software Engineering oder andere Open-Source-Entwickler.

Neben der zeitlichen Verzögerungen gibt es noch den weitere Nachteile

- Werden sehr kleine Patches hochgeladen, werden diese teilweise derart detailliert begutachtet, dass manchmal sogar die Javadoc-Dokumentation auf höchstem Niveau kritisiert wird und man für diese winzigen Änderungen ein weiteres Patch-Set hochladen und wieder auf die Reviewergebnisse warten muss.

- Natürlich kann es vorkommen, dass einige Stellen im Code in einem Patch-Set nicht kritisiert werden, jedoch im nächsten Patch-Set. Das kann daran liegen, dass es den Reviewern einfach erst in späteren Patch-Sets aufgefallen ist oder erst durch die neuen Änderungen entstanden sind. In solchen Fällen waren wieder neue Patch-Sets nötig und es musste wieder auf Reviews gewartet werden.

- Wenn einige Entwickler im Urlaub sind oder die Menge an studentischen Entwicklern zu dem Zeitpunkt klein ist, dauert es entsprechend, bis Reviews kommen.

Durch die genannten Nachteile entstehen manchmal merkliche Zeitverzögerungen. Diese kann man zwar mit anderer Arbeit, wie Schreiben oder Entwerfen von Konzepten füllen, aber irgendwann kommt man an den Punkt, wo wieder am Code gearbeitet werden muss. Diese Umstände haben es manchmal erschwert, die kommenden Tage und Wochen genauer zu planen, da man nie genau einschätzen konnte, wann neue Reviewkommentare eintreffen werden.

Nichtsdestotrotz ist dieser Reviewprozess auch ein großer Gewinn sowohl für den Studenten selbst als auch für Saros. Wenn ich darüber nachdenke, was für Stellen in meinem Code am Anfang kritisiert wurden und was ich alles gelernt habe, erstaunt mich immer wieder. Ich habe gemerkt, dass ich mittlerweile beim Implementieren viel intensiver über die Einhaltung der Architektur, über gute Dokumentation und über die sinnvolle Auslagerung in sinnvolle Klassen nachdenke als früher. Diese neuen Erkenntnisse habe ich auch neben der Masterarbeit im Beruf immer wieder eingesetzt. Auch für Saros hat dieser Reviewprozess nur Vorteile, denn der Code wird immer wieder intensiv begutachtet und dadurch erhöht sich nicht nur die Codequalität, sondern so wird auch nebenbei viel Wissen ausgetauscht.

7.3 Zeitlicher Verlauf

In diesem Kapitel soll der zeitliche Verlauf und der Lernprozess während der Bearbeitung dieser Arbeit präsentiert werden. Während der Arbeit hatte ich immer wieder mit dem Umstand umzugehen, dass ich nicht immer alles genau planen konnte, denn ich war manchmal abhängig vom Reviewprozess. Dieser hat, wie im Kapitel 7.2.3 beschrieben, neben den genannten Vorteil den großen Nachteil, dass es manchmal dauern kann, bis die ersten reviews eintreffen und man in der Zeit nicht immer programmieren konnte, da man von den Ergebnissen abhängig war. Abbildung 47 zeigt eine grafische Darstellung des zeitlichen Verlaufes der Arbeit. Jeder Monat ist dabei in vier Blöcke unterteilt, die jeweils eine Woche darstellen. Im Folgenden möchte ich die Blöcke grob beschreiben und auf Probleme eingehen, die ich während dieser Zeit hatte. So kann nachvollzogen werden, an welchen Stellen ich welche Entscheidungen treffen musste und mit welchen Schwierigkeiten ich umgehen musste. Diese Erörterung dient vor allem, um vorzustellen, wie in dieser Arbeit vorgegangen wurde, aber auch um darzustellen, wie mein Lernprozess letztendlich aussieht. Zudem habe ich Tabellen erstellt, die all meine Changes und zu jedem Change die entsprechenden Patches mit dem Hochladedatum festhalten. So können die Programmierphasen in Abbildung 47 genaueren Changes zugeordnet werden. Die Tabellen sind zu finden unter 6, 7, 8, 9 und 10.

Showing open dialogs or active views of users (#1515)	
Patchnr.	**Datum**
1	24.04.2014
2	27.04.0214
3	27.04.2014
4	27.04.2014
5	27.04.2014
6	28.04.2014
7	03.05.2014

Tabelle 6: Patch-Histroy für Patch 1515

7.3.1 Erster Monat

In der ersten Woche (also im ersten Block) habe ich die erste Version des ersten Prototyps entwickelt (siehe Kapitel 4.4), in dem ich herausfinden wollte, welche Informationen ich von Eclipse abgreifen kann bzw. welche Information alle Ereignisse, die ich in Eclipse abfangen kann, gemeinsam haben. Währenddessen habe ich die entsprechende Literatur studiert und habe mich auch mit Abschlussarbeiten, unter anderem von Frank Schlesinger [Sch05], beschäftigt, in denen ähnliche Ansätze überlegt und vorgestellt

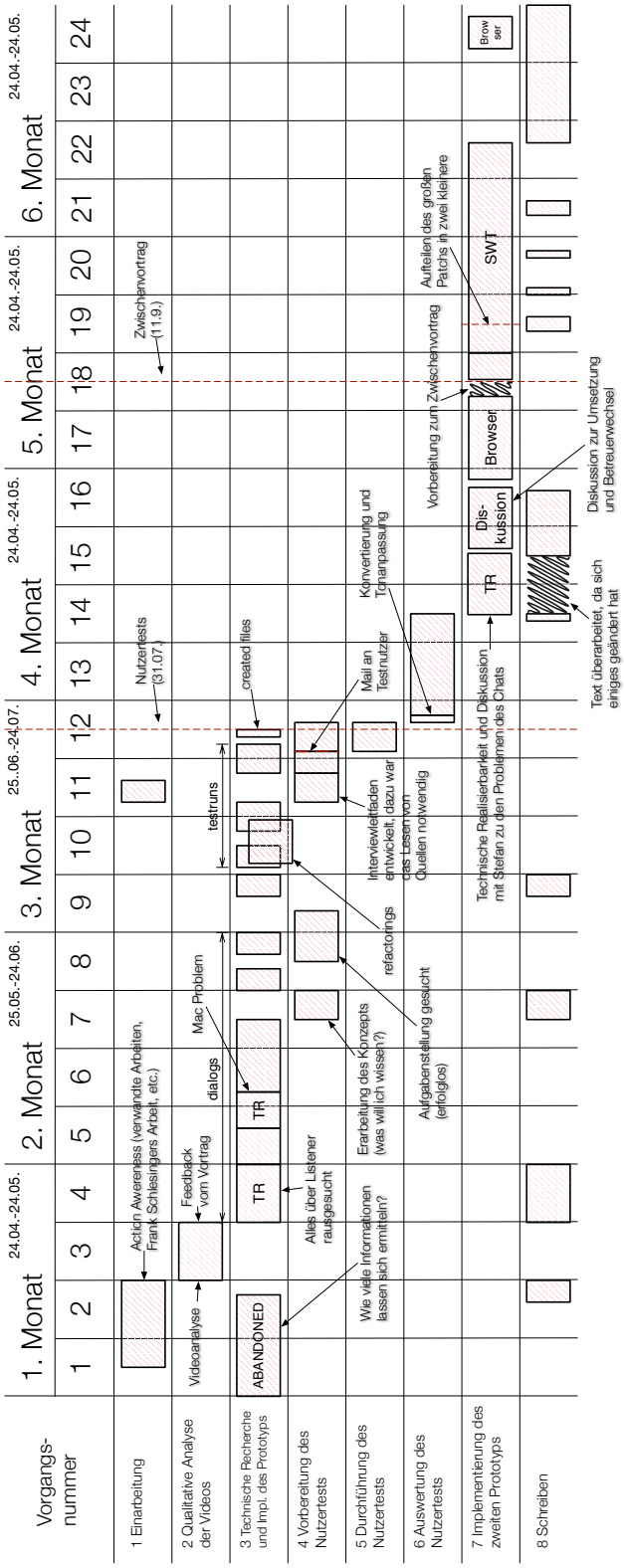


Abbildung 47: Zeitlicher Verlauf während der Arbeit

Show open dialogs/active views of others (#1572)	
Patchnr.	**Datum**
1	24.05.2014
2	24.05.2014
3	25.05.2014
4	02.06.2014
5	09.06.2014
6	13.06.2014
7	14.06.2014
8	14.06.2014
9	16.06.2014
10	24.06.2014
11	25.06.2014
12	01.07.2014

Tabelle 7: Patch-Histroy für Patch 1572

Merge master into development/actionAwareness (#1957)	
Patchnr.	**Datum**
1	16.09.2014

Tabelle 8: Patch-Histroy für Patch 1957

wurden. Während ich die Literatur studiert und gegen Ende der zweiten Woche auch bereits erste Texte geschrieben habe, habe ich in der zweiten Woche weiter an meinem Patch #1515 gearbeitet. Dieser diente für mich auch als Einstieg in die Programmierung in Saros. Dementsprechend hat der Reviewprozess etwas länger gedauert, da ich noch die typischen Fehler gemacht habe, die die meisten neuen Entwickler bei der Berührung mit Saros machen, wie fehlende Dokumentation oder kein korrektes Threadhandling. In der dritten Woche habe ich mir die Aufzeichnungen der Saros-Sitzungen, die mir von der AG Software Engineering zur qualitativen Datenanalyse zur Verfügung gestellt wurden, erneut angesehen und analysiert (siehe Kapitel 3). In der vierten Woche habe ich mich mit der Recherche zur technischen Realisierbarkeit des ersten Prototyps (siehe Kapitel 4.1) beschäftigt. Dies hat eine Woche gedauert. Das Problem war, dass im Stand-Up immer wieder der Begriff *Extension* gefallen ist und mir das damals nicht bekannt war. Ich habe mich damit beschäftigt, um herauszufinden, ob das für mich einen Mehrwert hat. Am Ende der Recherche habe ich festgestellt, dass der Listener-Mechanismus ausreichend ist und außerdem ich es nicht für sinnvoll hielt, einen Erweiterungsmechanismus zu entwickeln, nur um an Informationen zu gelangen, an die ich dank des Listener-Mechanismus ebenfalls gelange. Die Ergebnisse dieser Recherche habe ich währenddessen niedergeschrieben. Im

Show test runs with their result (#1788)	
Patchnr.	**Datum**
1	04.07.2014
2	04.07.2014
3	12.07.2014
4	12.07.2014
5	19.07.2014
6	19.07.2014
7	21.07.2014
8	16.09.2014
9	17.09.2014

Tabelle 9: Patch-Histroy für Patch 1788

Show performed refactorings (#1826)	
Patchnr.	**Datum**
1	26.07.2014
2	27.07.2014
3	28.07.2014
4	28.07.2014
5	28.07.2014
6	17.09.2014
7	17.09.2014

Tabelle 10: Patch-Histroy für Patch 1826

ersten Monat verlief alles nach Plan. Bloß die Implementierungsphase hat sich zeitlich gestreckt, da der Reviewprozess alles immer in die Länge gezogen hat.

7.3.2 Zweiter Monat

Währenddessen wurde die Entwicklung des ersten Prototyps auf einen eigenen Branch verlagert, sodass der Patch #1515 abgelehnt wurde und der neue Patch #1572 erstellt wurde, der bereits Verbesserungen des Patches #1515 enthielt. Zwischendurch trat bei mir das in Kapitel 4.4.2 beschrieben Problem bei der Entwicklung von Tests unter Mac OSX auf. Allein das Debuggen hat mich eine ganze Woche Zeit gekostet und trotzdem war ich nicht in der Lage, den Grund dafür herauszufinden. Ich habe angefangen, Literatur über das richtige Debuggen zu studieren, weil ich anfangs vermutet hatte, dass ich darin noch nicht gut bin. Allerdings hat sich ein anderer Entwickler rangesetzt und wie im erwähnten Kapitel beschrieben das Problem sehr schnell behoben. Ich kam einfach nicht darauf, den Patch einmal unter Windows zu testen. Denn

Show created file names (#1829)	
Patchnr.	**Datum**
1	28.08.2014
2	30.08.2014
3	17.09.2014
4	17.09.2014
5	25.09.2014
6	01.10.2014

Tabelle 11: Patch-Histroy für Patch 1829

Refactored code (#1975)	
Patchnr.	**Datum**
1	01.10.2014
2	01.10.2014

Tabelle 12: Patch-Histroy für Patch 1975

hätte ich das getan, hätte ich festgestellt, dass der Fehler überhaupt nicht auftritt und das hätte bei mir ebenfalls den Verdacht geweckt, dass es am genutzten Betriebssystem liegt bzw. wie das Betriebssystem Fensterwechsel handhabt. Da irgendwann in der siebten Woche der Punkt erreicht war, wo ich nicht mehr auf Reviews warten konnte, habe ich begonnen, ein Konzept für die Nutzertests zu entwickeln (siehe Kapitel 5). Am Ende der 8. Woche und Beginn der 9. Woche habe ich neben der Entwicklung die Aufgabenstellung gestalten. Wie ich in Kapitel 5.3 bereits beschrieben habe, war das keine triviale Aufgabe. Es hat über eine Woche gedauert, eine passende Aufgabe zu finden, obwohl ich ursprünglich dafür höchsten 5 Tage vorgesehen hatte.

7.3.3 Dritter Monat

Im dritten Monat musste ich viel parallel arbeiten, da ich im zweiten Monat bereits vieles ungeplant immer abwechselnd erledigen musste und das natürlich die Pläne im aktuellen Monat beeinflussen kann. Das war eine der größten Hürden während dieser Arbeit, denn aufgrund des Reviewprozesses hat sich vieles verlangsamt. Oft konnte ich nebenbei noch schreiben oder Konzepte entwickeln, aber ab bestimmten Zeitpunkten brauchte man den Quellcode, den man weiterentwickeln wollte. In diesem Monat habe ich parallel sowohl am Patch #1788 (Anzeige von Testergebnissen) als auch am Patch #1826 (Anzeige von Refactorings) gearbeitet. Während ich auf Reviewergebnisse gewartet habe, habe ich den Interviewleitfaden entwickelt und mich um die Vorbereitung der Nutzertests gekümmert. Nach der Entwicklung des Leitfadens und der Vorbereitung der Nutzertests habe ich noch den Patch #1829 (Anzeige von erstellten Dateien) erstellt. Dieser ging zum einen recht

Adds a new item to the tab folder (#1973)	
Patchnr.	**Datum**
1	26.09.2014
2	28.09.2014
3	29.09.2014
4	01.10.2014
5	02.10.2014
6	02.10.2014
7	02.10.2014

Tabelle 13: Patch-Histroy für Patch 1973

schnell, da ich an den meisten Stellen mittlerweile wusste, was die üblichen Fehler sind, die man in Saros machen kann und zum anderen habe ich auch ausdrücklich um mehr Reviews gebeten, da ich sie für die Nutzertests benötigt habe.

7.3.4 Vierter Monat

Ungefähr zwei Wochen lang habe ich mich dann mit der Auswertung der Aufzeichnungen der Nutzertests beschäftigt. Die ersten Tage habe ich mit der Anpassung des Tons beschäftigt. Da ich mich mit sowas nicht auskannte, fiel mir das natürlich etwas schwerer. Und da ich außerdem keine Komprimierung ausgewählt hatte, hat das Konvertieren der Aufzeichnungen ins .avi-Format sehr lange gedauert. In der Mitte des Monats war auch meine Recherche zur technischen Realisierbarkeit abgeschlossen und es hat sich herausgestellt, dass das Konzept so von den Nutzern als passend angenommen wurde bzw. die Verbesserungswünsche, die ausgesprochen wurden, technisch unmöglich waren oder keinen Mehrwert für Saros hatten (siehe Kapitel 5.6.4). Leider kam es dann zu einer Diskussion, in der es darum ging, dass die Umsetzung des ersten Prototyps im Chat softwaretechnisch keine gute Umsetzung darstellte und im Mailaustausch wurde darüber diskutiert, wie es nun weitergehen soll. Diese Zeit war nicht gerade angenehm, da ich nicht wusste wie es nun weitergehen soll. Die Zeit habe ich letztendlich zum Schreiben ausgenutzt.

7.3.5 Fünfter Monat

Zu Beginn des fünften Monats haben mein neuer Betreuer und ich, wie in Kapitel 6.3 bereits geschildert, über die Gestaltung des Inhalts des neuen Reiters mithilfe einer HTML-basierten Implementierungsweise nachgedacht und entschieden, dass ich den Ansatz ausprobieren soll. Währenddessen bekam ich den Tipp, nur die benötigten Klassen in das Saros-Projekt zu kopieren, was sich am Ende zu einer sehr frustrierende Angelegenheit entwickelt hat (siehe Kapitel 6.3). Immer wieder habe ich alles gelöscht und von vorne

Adds a new item to the tab folder (#1974)	
Patchnr.	**Datum**
1	30.09.2014
2	01.10.2014
3	01.10.2014
4	01.10.2014
5	03.10.2014
6	03.10.2014
7	03.10.2014
8	04.10.2014
9	04.10.2014
10	04.10.2014
11	04.10.2014
12	04.10.2014
13	04.10.2014
14	04.10.2014
15	04.10.2014
16	06.10.2014
17	08.10.2014

Tabelle 14: Patch-Histroy für Patch 1974

begonnen mit der Begründung, dass ich wohlmöglich etwas übersehen haben muss oder einfach nur den Überblick verloren habe. Irgendann musste ich jedoch pragmatisch denken, da ich bei meiner Abgabe auch einen fertigen zweiten Prototyp vorzeigen wollte. Also entschied ich mich in der Mitte des Monats, den Reiter mithilfe von SWT umzusetzen und falls mir Zeit bleibt am Ende auch noch den HTML-basierten Ansatz mit Einbindung des Plug-ins zu testen (siehe Kapitel 6.4).

7.3.6 Sechster Monat

Vor der Implementierung mittels SWT habe ich meinen alten Quellcode überarbeitet, verbessert und veraltete Sachen herausgeworfen (Patch #1975). Wie man der Abbildung entnehmen kann, hat die Umsetzung des Reiters mithilfe von SWT lange gedauert (Patch #1973 und #1974). Die Logik war innerhalb von einem Tag implementiert. Allerdings hat sich der Reviewprozess in die Länge gezogen, da in der Zeit außer ein Entwickler keine anderen Entwickler Zeit zum Reviewen hatten oder nicht vor Ort waren. Als dann jedoch die Implementierung des zweiten Prototyps mithilfe von SWT abgeschlossen war und die zweite Fassung der schriftlichen Ausarbeitung beendet wurde, blieb noch Zeit für den Versuch, den Inhalt des Reiters mithilfe der HTML-basierten Implementierweise mit Einbindung des Plug-ins von Björn Kahlert

umzusetzen (siehe Kapitel 6.4).

7.3.7 Rückblick

Wie man sehen kann, lief in meiner Arbeit vieles nicht komplett linear ab, was natürlich auch nicht erwartet wurde und ich auch nicht erwartet habe. Allerdings war es für mich nicht immer einfach, die kommenden Wochen zu planen, da immer unerwartete Ereignisse eintrafen oder der Reviewprozess länger als geplant gedauert hat. Das lag manchmal mehr daran, dass einfach nicht zu jeder Zeit jeder Entwickler eine Codedurchsicht betreiben konnte. Und wenn dann einmal Reviewergebnisse eintrafen, musste ich die aktuelle Arbeit niederlegen und mich darum kümmern, weil die Bearbeitung des Codes eine höhere Priorität hatte, da ich neben einer schriftlichen Ausarbeitung auch ein Ergebnis präsentieren wollte.

Außerdem gab es einige unvorhergesehene Ereignisse, so z. B. die Probleme auf dem Mac, die unerwartet lange Zeit für die Entwicklung einer angemessenen Aufgabe für die Nutzertests mit sich brachten sowie die Phase, in der über den weiteren Fortgang des Activity Logs innerhalb des Chats oder in Form eines neuen Reiters diskutiert wurde. Obwohl ich diese Zeit genutzt habe, um meine bisherigen Ergebnisse niederzuschreiben, ist es für mich nicht leicht gewesen, kommende Tage und Wochen zu planen.

Aus diesen Gründen fiel es mir auch jederzeit schwer zu sagen, ob ich als Leiterin meines eigenen Projekts im Projektplan liege. Allerdings hatte ich vor Beginn der Arbeit mit meiner damaligen Betreuerin über das Ziel der Arbeit gesprochen und sie hat verdeutlicht, dass ich in dieser Arbeit vieles neu ausprobieren werde und sich somit jederzeit etwas ändern kann. Insofern wusste ich ungefähr, was mich erwartet und bin auch zufrieden damit, dass ich mit solchen Situationen umgehen musste. Denn ich habe durch diese Umstände lernen müssen, verschiedene Aufgaben parallel zu bearbeiten. Kurz gefasst kann ich daher rückblickend behaupten, dass ich insgesamt meine Arbeitsweise im Rahmen dieser Arbeit verbessert habe.

8 Fazit und Ausblick

Das Ziel der Arbeit war die Verbesserung der Action Awareness in Saros. Zu diesem Zweck habe ich mithilfe der Grounded Theory Methode Aufzeichnungen zu Saros-Sitzungen analysiert, um Stellen zu finden, die Bedarf für mehr Action Awareness zeigen. Aus diesen Beobachtungen habe ich ein Konzept für die Verbesserung der Action Awareness entwickelt und dieses im Rahmen der Implementierung eines ersten Prototyps umgesetzt. Dieser zeigt an, welchen Dialog ein Nutzer offen hat, welche View er aktiviert hat, welche Datei er zuletzt erzeugt hat, was für ein Refactoring durchgeführt wurde und welcher Test mit welchem Ergebnis ausgeführt wurde. Anschließend habe ich diesen Prototypen mithilfe von Nutzertests evaluiert. Dazu habe ich eine angemessene Aufgabenstellung entwickelt, die so gestaltet war, dass die Nutzer Bedarf für die durch meinen Prototyp zur Verfügung gestellten Informationen hatten und mir so genau sagen können, was an dem Prototypen bereits gut umgesetzt ist und was eventuell noch verbessert werden kann. Nach der Durchführung der Nutzertests habe ich zudem semi-strukturierte Interviews durchgeführt, um persönliches Feedback von den Nutzern einzuholen. Die Auswertung der Nutzertests hat ergeben, dass die Nutzer mit den Funktionen des ersten Prototyps sehr zufrieden waren und das die entwickelte Lösung sehr stabil lief. Einige Wünsche konnten technisch nicht umgesetzt werden. Andere Wünsche wiederum schienen nicht sinnvoll für die Umsetzung in Saros.

Von daher habe ich mich entschieden, statt neuer Funktionen die Gestaltung der Informationen zu verbessern. Aufgrund technischer Schwierigkeiten wurde entschieden, einen neuen Reiter in der Saros-View, die Activity Log, zu erzeugen. Im Rahmen der Umsetzung dieses Reiters im zweiten Prototyp wurde eine neue HTML-basierte Implementierungsweise ausprobiert, die zu Beginn zu Problemen geführt hat und erst mal nicht weiter verfolgt wurde. Dann habe ich die Activity Log mthilfe von SWT umgesetzt. Da am Ende noch Zeit übrig blieb, habe ich die HTML-basierte Implementierweise auf eine andere Herangehensweise ausprobiert und das führte letztendlich zum Erfolg. Der Vorteil dieses Arbeitsverlaufs ist, dass ich letztendlich drei verschiedene Implementierweisen ausprobiert habe und vor allem zwischen den letzten beiden, der SWT-basierten und der zweiten HTML-basierten Vorgehensweise, einen Vergleich ziehen konnte, der für zukünftige Arbeiten am Activity Log oder anderen Saros-Elementen von Bedeutung ist. Insgesamt habe ich mit dieser Arbeit nicht nur die Action Awareness in Saros verbessert, sondern habe auch verschiedene technische Umsetzungen ausprobiert und konnte dort ebenfalls Optimierungsmöglichkeiten erfolgreich feststellen.

Da die Nutzer mit dem ersten Prototyp zufrieden waren, wurde entschieden, dass ich keinen zweiten Nutzertest benötige und mich stattdessen auf die Gestaltung und die Optimierung der technischen Umsetzung im Rahmen eines HTML-basierten Versuchs konzentrieren kann. In der Zukunft kann man diese HTML-basierte Vorgehensweise weiterhin verfolgen, denn diese

Arbeit hat gezeigt, dass es im Prinzip möglich ist, mit wenig Aufwand eine funktionierende und ansehnliche Anzeige mit einfachen Techniken wie HTML, JavaScript und CSS zu gestalten. Die zukünftige Untersuchung richtet sich darauf, wie man nun diesen HTML-basierten Ansatz nutzen kann, um die Gemeinsamkeiten zwischen Saros für Eclipse und Saros für IntelliJ zu vergrößern. Weiterhin kann man parallel dazu in weiterführenden Abschlussarbeiten weitere Funktionen zur Verbesserung der Action Awareness überlegen und mithilfe von Nutzertests evaluieren. Da die Umsetzung eines Prototyps mithilfe von HTML, CSS und JavaScript nun möglich ist, kann man das neu entwickelte Konzept schnell umsetzen und frühzeitig in Form von Nutzertests evaluieren und anpassen.

A Aufgabenstellung für die ersten Nutzertests

Freie Universität Berlin

Fachbereich
Mathematik und Informatik

Masterarbeit

Institut für Informatik

Verbesserung der
Action Awareness in Saros

Damla Durmaz

28.07.2014

Nutzertest #1

Einleitung

"Action-Awareness" bezeichnet das Verstehen der Aktivitäten und Absichten der anderen Teammitglieder. Dieses Verständnis kann auf unterschiedlichen Ebenen bestehen, entweder ganz allgemein "der andere ist in einer anderen Anwendung" oder sehr detailliert "der andere macht gerade ein Refactoring in Datei X in Zeile Y". Saros überträgt Action-Awareness-Informationen aus dem Code-Editor (wer tippt gerade was und wo). Aktivitäten, die jedoch in anderen Anwendungen, anderen Eclipse Views als dem Editor oder Dialogen (Suche, Refactoring etc.) stattfinden, sind für die anderen Teammitglieder nicht transparent.

Ich habe daher im Rahmen meiner Masterarbeit einen Prototypen entwickelt, der die Action-Awareness von Saros verbessern soll. Dieser Nutzertest dient der qualitativen Bewertung des entworfenen Prototyps. Sie erhalten eine Reihe von Aufgaben, die nicht dazu dienen, ihre Programmierfähigkeiten, sondern den Prototypen zu testen.

Ihnen wurde ein Zugang zu einem Adobe Connect Server zur Verfügung gestellt, bei dem sie sich mit den gegebenen Zugangsdaten anmelden müssen. Dieser Server wird im Laufe des Nutzertests ihren Bildschirm und Ihren Ton aufnehmen. **Die Aufnahme des Tons dient für die Think Aloud-Methode. Dabei bitte ich Sie, bei der Erfüllung Ihrer Aufgaben auszusprechen, was Sie denken, was sie gerade tun, was die gerade betrachten, was sie eventuell fühlen usw., während Sie die Aufgaben bearbeiten.**

Am Ende des Nutzertests möchte ich mit jedem Teilnehmer anschließend bei einem persönlichen Skype-Gespräch ein Interview durchführen. Dieses Interview wird nicht länger als 30 Minuten dauern und dient der Erfassung der qualitativen Bewertung, der Eindrücke, Anregungen und Wünsche bzgl. des Prototyps.

Der Nutzertest findet unter folgenden Bedingungen statt:

- Die Bearbeitung muss in getrennten Räumen stattfinden, sodass Sie sich gegenseitig nicht hören.
- Mindestens Eclipse 3.7 ist lokal vorhanden
- Kommunikation über Skype
- Verbindung zum Adobe Connect Server
- Kein Zeitlimit

Aufgabe 1

In dieser Aufgabe werden Sie den Prototypen in Ihrer lokalen Eclipse-Installation einrichten und ein JUnit4-Projetk importieren, in dem sie zwei zusätzliche Tests erzeugen werden.

1. Einrichtung des Prototyps

> Für diesen Schritt ist es wichtig, dass Sie eine Eclipse-Version installiert haben, in der Saros nicht bereits installiert ist. Befolgen Sie einfach die folgenden Anweisungsschritte.

Anweisungsschritte:

1. Navigieren Sie in Ihrem lokalen Webbrowser auf die Seite https://github.com/dildik/saros-first-usertests
2. Laden sie das dort vorhandene Repository **saros-first-usertests** in ein von Ihnen gewünschtes Verzeichnis runter, indem sie auf der Seite rechts einfach auf den Button **Download ZIP** klicken.
3. Begeben Sie sich nun zum Verzeichnis, wo die .zip-Datei nach dem Download gesichert wurde. Die Datei heißt dabei **saros-first-usertests-master.zip**.
4. Entpacken Sie dieses Archiv.
5. Im entpackten Archiv befindet sich ein Verzeichnis, namens **plugins**. In diesem Verzeichnis befinden sich wiederum zwei .jar-Dateien.
6. Kopieren Sie alle vier Dateien in den **dropins**-Ordner ihrer lokalen Eclipse-Installation. Dafür müssen Sie ins Verzeichnis, in dem Eclipse bei Ihnen installiert ist. Dort befindet sich ein Ordner, namens **dropins**. Hier müssen alle vier Dateien hineinkopiert werden.
7. Starten Sie ihre Version von Eclipse.
8. Ist die Saros-View nicht zu sehen, können Sie dies unter **Window** --> **Show View** --> **Other** und dort unter **Saros** --> **Saros** vornehmen.

2. Einrichtung des Junit4-Projekts, Konfiguration von Nutzeraccounts und Starten einer Saros-Sitzung

> In diesem Schritt muss das JUnit4-Projekt eingerichtet, ein neuer Account konfiguriert und eine Saros-Sitzung gestartet werden.
>
> Die Konfiguration eines neuen Accounts ist deshalb wichtig, da dieser Prototyp unter anderem im Multichat-Fenster Action-Awareness Informationen überträgt und dieser bekanntlich bei einigen Anbieter, wie Gmail, nicht korrekt funktioniert.

Anweisungsschritte:

1. Gehen Sie in den Package Explorer, führen Sie einen Rechtsklick aus und wählen Sie dann die Option **Import**.
2. Im erscheinenden **Import**-Dialog wählen Sie **General** die Option **Existing Projects into Workspace** aus und klicken auf **Weiter**.
3. Im nächsten Dialog klicken Sie auf den Button **Browse** und navigieren dorthin, wo **saros-first-usertests** entpackt wurde.
4. Dort wählen Sie das Verzeichnis **junit** aus und bestätigen die Auswahl mit **Ok**.
5. Klicken Sie im letzten Dialog auf **Finish**.
6. Konfigurieren Sie einen neuen Nutzeraccount. Gehen Sie dafür im Menü auf **Saros --> Add Account** und konfigurieren Sie einen Account aus Folgenden, die nach diesem

2

Schema für Tests zur Verfügung stehen: `dev[0-9]_[alice|bob|carl|dave]_stf` (Passwort: bei allen `dev`)

7. Starten Sie eine Saros-Sitzung mit dem importierten JUnit4-Projekt.

3. Erzeugung von zwei Testklassen

In diesem Schritt werden beide Teilnehmer jeweils eine Testklasse erzeugen und diese anschließend ausführen. Das Ergebnis der Ausführung wird beiden deutlich machen, was das zu lösende Problem dieses Nutzertests sein wird.

Anweisungsschritte:

1. Erzeugen Sie im Verzeichnis `src/main/java` ein Package mit der Bezeichnung **usertest**.
2. Im Verzeichnis **saros-first-usertests** liegen zwei JUnit Tests vor, die im oben genannten Package erzeugt werden sollen. Diese sind:
   - `ThisShouldSucceedTest.java`
   - `ThisSucceedsTest.java`
3. Führen Sie die Tests aus und überlegen Sie, warum es zum angezeigten Ergebnis gekommen ist.

Aufgabe 2

Die erste Aufgabe des Nutzertests ist nun beendet. Beim vorliegenden Projekt handelt es sich um einen früheren Stand des JUnit4-Projekts und das Problem wurde durch die Ausführung der ersten Aufgabe deutlich:

Wird eine Klasse mit der `@Ignore`-Annotation markiert, so werden keine Testmethoden oder andere Methoden ausgeführt, die die `@AfterClass` oder `@BeforeClass`-Annotation besitzen. Wird jedoch nur die einzige Methode der Testklasse mit der `@Ignore`-Annotation markiert, werden trotzdem die Methoden ausgeführt, die die Annotation `@BeforeClass` oder `@AfterClass` besitzen.

Dieses Verhalten ist nicht erwünscht. Stattdessen soll es möglich sein, dass wenn eine Testklasse nur eine Methode besitzt und diese ignoriert werden soll, die Klasse behandelt werden soll als wäre die gesamte Klasse auf `@Ignore` gesetzt: Die Methoden mit der Annotation `@AfterClass` und `@BeforeClass` sollen also nicht ausgeführt werden. Zusammengefasst bedeutet das: Der Test `ThisShouldSucceedTest.java` sollte, wie der Name schon sagt, eigentlich durchlaufen. Dies soll in dieser zweiten Aufgabe umgesetzt werden.

In diesem Schritt soll dieses unerwünschte Verhalten nun behoben werden. Das Ziel ist, dass wenn eine Testklasse nur eine Methode enthält, die jedoch auf `@Ignore` gesetzt wurde, die Testmethoden, die auf `@BeforeClass` oder `@AfterClass` gesetzt wurden, nicht ausgeführt werden und der Test ohne ein `FAILURE` durchgeführt wird. Ihre Ideen können Sie natürlich überprüfen, indem Sie die von Ihnen erzeugten Testklassen ausführen.

Im Gegensatz zur ersten Aufgabe gibt es hier keine klaren Anweisungen mehr, sondern Hinweise, die beim Einstieg in die Aufgabe helfen sollen.

3

B Interviewleitfaden

Dieser Anhang repräsentiert den Interviewleitfaden. Dieser wurde nicht in genau dieser Reihenfolge befolgt, da einige Tester beim Beantworten einiger

Fragen bereits andere beantwortet haben oder aber auch durch manche Ihrer Antworten Fragen gestellt wurden, die nicht im Leitfaden geplant waren, sondern spontan eingefallen sind. Die Passagen zwischen den Stichpunkten, die in der Ich-Perspektive geschrieben wurden, sind Leitsätze, die von der Organisatorin des Nutzertests in ähnlicher Form auch gesagt wurden, um einen Übergang zwischen den Themen des Interviews zu schaffen.

- Wie verlief für dich die Sitzung?
- Gab es etwas Störendes?
- Wie hat dir insgesamt das neue Feature gefallen?
- Konntest du dich gut orientieren oder warst du eher orientierungslos?
- Haben dir die angezeigten Informationen im Allgemein etwas gebracht?
- Glaubst du, dass du mithilfe der angezeigten Informationen in der Zukunft besser mit anderen zusammenarbeiten kannst?
- Wie oft hast du die angezeigten Informationen wirklich benötigt?
  - Wo war es häufiger?
  - Wo war es seltener?
- Kannst du mir aufzählen, an welchen Stellen
  - die Infos deiner Meinung nach sinnvoll waren?
  - dir die Infos nichts gebracht haben?
- Wo gab es Momente, wo du etwas wissen wolltest, aber es nicht mitgeteilt wurde?

Die letzten Fragen habe ich gestellt, um zu sehen, was dir im Groben und Ganzen gefallen hat und was nicht. Bei den folgenden Fragen möchte ich auf bestimmte Funktionen des Prototyps näher eingehen und wissen, ob diese dir geholfen haben oder was du verbessern würdest. Wie dir sicher aufgefallen ist, wurde stets der Titel eines Dialoges angezeigt oder der Titel der View, in der sich dein Gegenüber derzeitig befand.

- In welchen Situationen hast du es als hilfreich empfunden, dass der Titel des Dialoges oder der View angezeigt wurde?
- Wo gab es Situationen, wo dir der Titel allein nicht ausgereicht hat?
- Es gab ja drei Orte, an denen diese Informationen angezeigt wurden: Welchen favorisierst du?

Vielen Dank, das sind wirklich tolle Einsichten. Machen wir weiter mit einer weiteren Dialog-Kategorie: dem Refactoring. Zu Refactorings wurde angezeigt, welche Refactoring-Methode der Gegenüber verwendet hat.

- Haben Sie diese Funktion als sinnvoll empfunden?
  - ‘Wenn Befragter sagt, dass es ihm sehr geholfen hat ...‘
    - Wie oft hatten Sie das Gefühl, dass Ihnen die Information echt weitergebracht hat?
    - Inwieweit hat es Ihnen geholfen?
  - ‘Ansonsten‘
    - Woran lag es, dass diese Funktion Ihnen nichts gebracht hat?
    - Was würden Sie zusätzlich anzeigen, wenn es möglich wäre?

Das wars erst einmal zu den Fragen zu Dialogen. Ich danke Ihnen. Eine Ihrer Aufgaben bestand ja darin, Tests zu schreiben und auszuführen. Dies habe ich so entworfen, da ich sehen wollte, ob die Anzeige über Testläufe hilfreich sein kann.

- Erinnern Sie sich doch bitte einmal an ihre zurückliegenden Paarprogrammier-Sitzungen:
  - Wie häufig kam es vor, dass Sie Tests schreiben mussten?
  - Wie häufig haben Sie in diesen Situationen mit dem anderen diese schreiben müssen?
  - In welchen Situationen hätten Ihnen diese Informationen etwas gebracht?
- Stellen Sie sich nun bitte vor, Sie sitzen jetzt in einer Paarprogrammier-Sitzung. Würden Ihnen diese Informationen weiterhelfen?
  - ‘Wenn ja‘
    - Würden Sie irgendwas daran verbessern?
    - An welchen Stellen waren die Informationen nicht ausreichend?
  - ‘Wenn nein‘
    - Woran läge es denn?
    - Empfanden Sie die Informationen als überflüssig?
- Während der Entwicklung des Prototyps habe ich von anderen Feedback bekommen, dass Anzeige von Testläufen und ihrem Ergebnis wenig bringt. Was denkst du dazu?

Das sind interessante Einsichten, die Sie mir da geben. Ich habe um mir ein Gesamtbild auszumalen, noch weitere Fragen überlegt, die ich Ihnen gerne stellen würde:

- Finden Sie das sinnvoll, dass zu allen Dialogen und Views der Titel angezeigt wird und in einigen Situationen mehr?
- Gäbe es noch weitere Stellen, wo Sie sich vorstellen könnten, mehr Informationen zu erhalten?
- Stellen Sie sich vor, sie haben die Möglichkeit sowas zu konfigurieren: Was könnte man Ihrer Meinung nach konfigurabel machen?
- Glauben Sie, dass Ihnen die Informationen auch was bringen würden, wenn Sie nciht im following-Modus arbeiten?

Zuletzt habe ich noch Fragen zur Art und Weise der Darstellung. Die Informationen wurden an unterschiedlichen Orten angezeigt, da ich herausfinden möchte, was wohl der beste Ort sein kann:

- Welcher Ort hat Ihnen am meisten gefallen?
- Sind sie mit Ihrem Favoriten zufrieden oder hätten Sie sich noch bei der Darstellung etwas dazu gewünscht?

Zum Abschluss möchte ich noch einige allgemeine Fragen stellen, die rein statistischen Zwecken dienen:

- Wie oft im Monat hast du Paarprogrammierung durchgeführt?
- Wie vertraut bist du mit der Nutzung von Saros?

Ich bedanke mich sehr für dieses Interview. Ihre Antworten werden mir auf jeden Fall dabei helfen zu verstehen, inweiweit die übertragenen Informationen geholfen haben und was man noch verbessern könnte. Abschließend würde ich Sie noch fragen, ob Sie noch spezielle Wünsche haben?

C Einladung zum Nutzertest

Betreff:
Your feedback needed on the new action awareness features of the actual Saros prototype.

Inhalt:
Dear Saros developers,

Next week, I want to conduct a first usability test on the first prototype of Saros with a enhanced action awareness features. Action awareness refers to the understanding of the activities and intentions of other team members in collaborative work. Your feedback helps me to understand which new features are helpful for you and to identify other issues, problems or wishes regarding the prototype and its functions. Everyone is invited to attend.

Only with your help I can identify where more action awareness information is needed, where the transferred information are too much or too less, what place is appropriate to display the action awareness information and, in general, what can be improved else.

The test will take less than an hour and each test user can attend from everywhere from his own machine, only an audio connection (e.g. Skype) an a webcam should be. For the test each participant simply needs to attend an Adobe Connect web-meeting via his web browser. That's all ;-)

To get individual feedback, I would like to do a short Skype-interview with each participant after the test.

Ideally, the tests shold be conducted during the next week (CW 31) from Tuetsday. Please just tell me what times would be good for you.

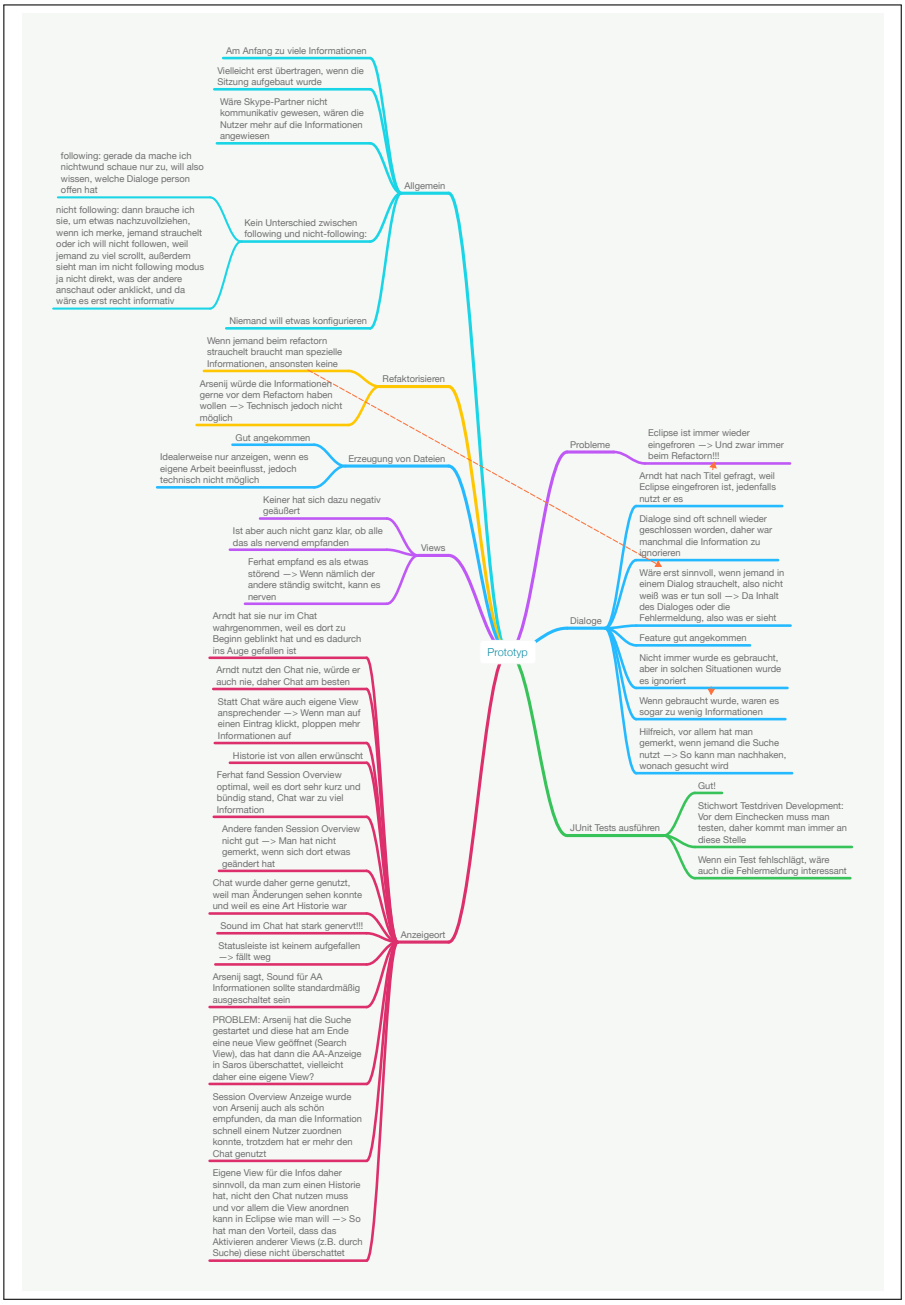
If you are interested or have questions, please don't hesitate to contact me. I am happy to help and to give further information.

Best regards,
Damla Durmaz

D Notierte Beobachtungen

Video	Zeit	Beobachtung
1	05:45	Beide haben eine Sitzung begonnen. Zu Beginn kommen viele Zwischendialoge, wie Synchronizing usw. Dabei werden schon viele Awareness-Infos angezeigt und im Chat wird dementsprechend ständig der Sound wiedergegeben. Martin sagte sogar „Sehr viele Geräusche.“ Ich habe als Veranstalterin des Web-Meetings den Hinweis gegeben, dass man es abschalten kann.
1	07:00	Arndt hat in der Saros-View auf die Awareness-Anzeige geklickt. Ob er versucht hat, sie doppelzuklicken, ist nicht klar. Man könnte sich aber überlegen, beim Doppelklick weitere Infos oder andersartiges anzuzeigen oder zu triggern.
1	07:08	Arndt hat gemerkt, dass Infos im MUC-Chat in der Saros-View angezeigt werden. Er hat dann die Saros-View vergrößert.
1	10:06	Arndt schaut sich die beiden Testklassen an und versucht, herauszufinden, was der Unterschied zwischen den Klassen ist. Dabei schaut Martin einfach nur zu. Er schaut sich selbst dabei die Dateien an.
1	10:43	Arndt hat gesagt „Nimm’s mal weg“ und daraufhin hat Martin ein @Ignore-Annotation entfernt. Daraufhin hat Arndt den Test erneut ausgeführt und Martin hat sich gewundert mit „Oh, das ist auch fehlgeschlagen“. Er hat also dabei die von Arndt übertragenen Informationen wahrgenommen.
1	10:49	Martin hat gesagt „Ich führ’s mal auch aus, damit ich was sehe“. Vermutlich möchte er dabei die genaue Fehlerinformation sehen, also was die Meldung, dass es nicht erfolgreich lief, genau sagt.
1	11:03	Martin versucht, die JUnit-View zu öffnen und fragt währenddessen dabei „Gibt’s hier keine…“. Dabei sieht Arndt anhand der übertragenen Informationen, dass Martin sich im „Show view“-Dialog befindet und sagt, dass sich diese View automatisch öffnet, wenn er den Test ausführen lässt.
1	23:09	Martin und Arndt haben versucht, die Aufgabe zu lösen. Anschließend hat Arndt den Test ausgeführt und Martin hat das Ergebnis über die AAAnzeige mitbekommen und gesagt „Ah, succeeds“. Dabei hat er mit dem Mauszeiger auf die entsprechende Anzeige im Chat gezeigt. Das ist ein Hinweis darauf, dass er zurzeit die Chat-Anzeige nutzt.
1	29:50	Arndt hat Martin mitgeteilt, dass er gerade debuggt und Martin hat mit „Hm.“ mitgeteilt, dass er das mitbekommen hat. Eventuell wäre es hilfreich gewesen, dass Informationen darüber übertragen werden, dass generell gedebuggt wird.
1	44:33	Martin hat refactort. Selbst wenn er das über die Tastenkürzel macht, erscheinen dann die Informationen „Rename blablabla“. Wenn man stattdessen die Refactoring-Dialoge verwendet und dann auf Enter drückt, so wird zuerst die Anzeige für einen offenen Dialog angezeigt und anschliießend, wenn der Dialog geschlossen wird, wird die Information gezeigt, dass etwas refactort wurde. Da aber Martin die Tastenkürzel verwendet hat, wurden keine Dialog-Informationen angezeigt, und er war verwirrt, weil er nicht wusste, was Martin gerade tut. Hier wären Echtzeitinformationen über das Refactoring eigentlich interessant gewesen.
2	14:11	Arsenij hat festgestellt, dass die Anzeige „Active View: Package Explorer“ zwei mal angezeigt wurde. Das ist nur ein Anzeigefehler. Aber das zeigt trotzdem, dass er die Anzeige wahrnimmt.
2	19:27	Ferhat sieht die Chat-View und redet mit Arsenij allgemein darüber. Er sagt, dass er diese ganz gut findet, da er sieht, was Arsenij macht und weil es auch wie eine Chronik ist.
2	19:56	Arsenij führt einen Test aus und Ferhat führt ihn nicht aus, sieht jedoch anhand der Anzeige, dass der Test geklappt hat. Ein weiterer Hinweis darauf, dass er die Anzeige wahrnimmt und nutzt.
2	20:35	Ferhat hat im Gegensatz zu allen anderen Teilnehmern nichts vom Prototyp und seinen Funktionen gewusst. Er hat gesagt, dass es toll ist, dass unten die Testergebnisse angezeigt werden. Im vorigen Schritt hat er es vermutlich aus dem Chat erfahren. Hier hat er gesehen, dass es in der Session Overview angezeigt wurde, denn nachdem er das gesagt hat, markiert er das entsprechende Tree-Element.
2	26:54	Ferhat sagt, dass er es sinnvoll findet, dass angezeigt wird, welche Dateien vom PP-Partner erzeugt wurden. Dabei markiert er die entsprechende Stelle im Chat.
2	51:59	Arsenij hat mithilfe des Suchdialoges etwas gesucht und es Ferhat mitgeteilt, indem er gesagt hat „Jetzt bin ich in der Suche“. Ferhat bekommt das natürlich schon über die Anzeige mit.
2	58:07	Arsenij hat vor kurzem die Saros-View vergrößert. Ich nehme mal an, das liegt daran, dass eine Menge Nachrichten nun in der Chat-View sind und er das hochscrollt, um auch ältere zu sehen. Wenn es ihm nur nach den neusten Nachrichten gehen würde, würde er vermutlich einfach nur die zuletzt eingetroffene Nachricht ansehen. Aber das ist schwer zu beruteilen, da es nur zwei Personen sind. Hier stellt sich daher die Frage, wie es mit mehreren Leuten gewesen wäre.
2	01:03:44	Arsenij hat sich über die ständigen Chat-Notifications aufgeregt. Kurze Zeit später schaltet er den Sound ab.
2	01:11:07	Ferhat beschwert sich darüber, dass sein Eclipse sehr langsam ist. Komischerweise passiert das auch zu der Zeit, wo refactort wird. Es kann wirklich am Refactorn liegen. Das muss ich unbedingt einmal testen.
2	01:11:29	Ferhat hat sich nach dem langsamen Eclipse dazu entschieden, Arsenij zu followen. Da er kurz davor in der Session Overview die Awareness-Anzeige angeklickt hat, ist es sehr wahrscheinlich, dass er die Awareness-Informationen aus der Session Overview nimmt bzw. in dieser Situation das auch getan hat.
2	01:13:03	Arsenij nutzt gerade die „Find/Replace“-View. Allerdings gibt es hier im Prototypen einen Bug und daher wird nur bei Ferhat der Editor als derzeitige Aktion angezeigt. Gerade in dem Moment fragt Ferhat Arsenij auch, was er tut und da es ihm sofort einfällt, beantwortet er sich selbst die Frage. Aber auf jeden Fall konnte er, dadurch dass es fälschlicherweise nicht übertragen wurde, nicht sehen, was Arsenij gerade tut.

E Entwickelte Kategorien



Literatur

[AS95] Mark S. Ackerman and Brian Starr. Social activity indicators: Interface components for cscw systems. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 159–168, New York, NY, USA, 1995. ACM.

[ATP95] Marilyn J. Adams, Yvette J. Tenney, and Richard W. Pew. Situation Awareness and the Cognitive Management of Complex Systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, pages 85–104, March 1995.

[Car07] Jeffrey Carver. The use of grounded theory in empirical software engineering. In VictorR. Basili, Dieter Rombach, Kurt Schneider, Barbara Kitchenham, Dietmar Pfahl, and RichardW. Selby, editors, *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, volume 4336 of *Lecture Notes in Computer Science*, pages 42–42. Springer Berlin Heidelberg, 2007.

[CH07] Jan Chong and Tom Hurlbutt. The social dynamics of pair programming. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.

[com14] commons-dev. Commons CLI, Februar 2014. http://commons.apache.org/proper/commons-cli/, Letzter Zugriff am 10.10.2014.

[Cor04] Christophe Cornu. Viewing HTML pages with SWT Browser widget, August 2004. http://www.eclipse.org/articles/Article-SWT-browser-widget/browser.html, Letzter Zugriff am 23.10.2014.

[D14] Damla D. [FEATURE] Activity log tab has a new design, Oktober 2014. http://saros-build.imp.fu-berlin.de/gerrit/#/c/1974/, Letzter Zugriff am 20.10.2014.

[Dam14] Damla. [FEATURE] Action Awareness: Show test runs with their result, September 2014. http://saros-build.imp.fu-berlin.de/gerrit/#/c/1788/, Letzter Zugriff am 20.10.2014.

[DB92] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[Dje06] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung. Master's thesis, Freie Universität Berlin, Institut für Informatik, 2006.

[DOS07] Riad Djemili, Christopher Oezbek, and Stephan Salinger. Saros: Eine Eclipse-Erweiterung zur verteilten Paarprogrammierung. In Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven, editors, *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 317–320. GI, 2007.

[Dre95] Eric Drever. *Using Semi-Structured Interviews in Small-Scale Research. A Teacher's Guide*. Scottish Council for Research in Education, Edinburgh, 1995.

[Dur14a] Damla Durmaz. [feature] action awareness: Show open dialogs/active views of others, Mai 2014. http://saros-build.imp.fu-berlin.de/gerrit/#/c/1572, Letzter Zugriff am 20.10.2014.

[Dur14b] Damla Durmaz. [feature] action awareness: Showing open dialogs or active views of users, Mai 2014. http://saros-build.imp.fu-berlin.de/gerrit/#/c/1515, Letzter Zugriff am 20.10.2014.

[Ecl11a] Eclipse contributors and others. Class DebugEvent, 2011. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fdebug%2Fcore%2FIDebugEventSetListener.html, Letzter Zugriff am 09.10.2014.

[Ecl11b] Eclipse contributors and others. Package org.eclipse.swt.browser, 2011. http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/browser/package-summary.html, Letzter Zugriff am 23.10.2014.

[Ecl11c] Eclipse contributors and others. Package org.eclipse.swt.events. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fevents%2Fpackage-summary.html, 2011.

[Ecl12a] Eclipse contributors and others. Copy participants. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ltk_core_refactoring_copyParticipants.html&cp=2_1_1_98, 2012.

[Ecl12b] Eclipse contributors and others. Creation Wizards. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-

points%2Forg_eclipse_ui_newWizards.html&cp=2_1_1_174, 2012.

[Ecl12c] Eclipse contributors and others. Move Participants. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ltk_core_refactoring_moveParticipants.html&cp=2_1_1_101, 2012.

[Ecl12d] Eclipse contributors and others. Refactoring change preview viewers. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ltk_ui_refactoring_changePreviewViewers.html&cp=2_1_1_104, 2012.

[Ecl12e] Eclipse contributors and others. Refactoring Contributions. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ltk_core_refactoring_refactoringContributions.html&cp=2_1_1_102, 2012.

[Ecl12f] Eclipse contributors and others. Rename Participants. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ltk_core_refactoring_renameParticipants.html&cp=2_1_1_103, 2012.

[Ecl12g] Eclipse contributors and others. Search Pages. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_search_searchPages.html&cp=2_1_1_106, 2012.

[Ecl12h] Eclipse contributors and others. Search Result View Pages. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_search_textSearchQueryProvider.html&cp=2_1_1_110, 2012.

[End95] Mica R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):32–64, 1995.

[Fau03] Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3):379–383, 2003.

[GG96] Carl Gutwin and Saul Greenberg. Workspace awareness for groupware. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 208–209, New York, NY, USA, 1996. ACM.

[GG98] Carl Gutwin and Saul Greenberg. Effects of awareness support on groupware usability. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '98, pages 511–518, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co.

[GG02] Carl Gutwin and Saul Greenberg. A descriptive framework of workspace awareness for real-time groupware. *Comput. Supported Coop. Work*, 11(3):411–446, November 2002.

[GGGG95] Carl Gutwin, Carl Gutwin, Saul Greenberg, and Saul Greenberg. Support for group awareness in real-time desktop conferences. In *University of Waikato*, pages 18–21, 1995.

[GGR96a] Carl Gutwin, Saul Greenberg, and Mark Roseman. Supporting awareness of others in groupware. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 205–, New York, NY, USA, 1996. ACM.

[GGR96b] Carl Gutwin, Saul Greenberg, and Mark Roseman. Workspace awareness in real-time distributed groupware: Framework, widgets, and evaluation. In *Proceedings of HCI on People and Computers XI*, HCI '96, pages 281–298, London, UK, UK, 1996. Springer-Verlag.

[GK13] Zoltan Gocza and Zoltan Kollin. Myth #13: Icons enhance usability, 2013. http://uxmyths.com/post/715009009/myth-icons-enhance-usability, Letzter Zugriff am 12.10.2014.

[Gre91] Ralph Greenberg, Saul und Bohnet. Groupsketch: A multi-user sketchpad for geographically-distributed small groups. In *Proc. Graphics Interface'91*, 1991.

[Gut96] Mark Gutwin, Carl unnd Roseman. A usability study of workspace awareness widgets. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 214–215, New York, NY, USA, 1996. ACM.

[HG04] Jason Hill and Carl Gutwin. The maui toolkit: Groupware widgets for group awareness. *Comput. Supported Coop. Work*, 13(5-6):539–571, December 2004.

[Kah14] Bjoern Kahlert. com.bkahlert.nebula, Oktober 2014. https://github.com/bkahlert/com.bkahlert.nebula/tree/master/src/com/bkahlert/nebula/widgets/browser, Letzter Zugriff am 23.10.2014.

[Kis09] Craig Kistler. The Problem With Icons, Dezember 2009. http://www.smallfarmdesign.com/blog/2009/12/28/the-problem-with-icons/, Letzter Zugriff am 11.10.2014.

[Mif11] Justin Mifsud. 14 Guidelines For Web Site Tabs Usability, Oktober 2011. http://usabilitygeek.com/14-guidelines-for-web-site-tabs-usability/, Letzter Zugriff am 13.10.2014.

[MN97] John Morkes and Jakob Nielsen. Concise, SCANNABLE, and Objective: How to Write for the Web, Januar 1997. http://www.nngroup.com/articles/concise-scannable-and-objective-how-to-write-for-the-web/, Letzter Zugriff am 12.10.2014.

[NL93] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 206–213, New York, NY, USA, 1993. ACM.

[Nor11] Mia Northrop. Developing Your Interviewing Skills, Part I: Preparing for an Interview. http://www.uxmatters.com/mt/archives/2011/01/developing-your-interviewing-skills-part-i-preparing-for-an-interview.php, January 2011.

[PBH04] Denise Polit, Cheryl Tatano Beck, and Bernadette Hungler. *Lehrbuch Pflegeforschung: Methodik, Beurteilung und Anwendung*. Verlag Hans Huber, 2004.

[Phi11] Phillipus. How to get Eclipse SWT Browser component running on Ubuntu 11.04 (Natty Narwhal) with Webkit, April 2011. http://stackoverflow.com/questions/5817263/how-to-get-eclipse-swt-browser-component-running-on-ubuntu-11-04-natty-narwhal, Letzter Zugriff am 23.10.2014.

[RG96] Mark Roseman and Saul Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Trans. Comput.-Hum. Interact.*, 3(1):66–106, March 1996.

[Rug09] Wolfgang Ruge. Grounded Theory. Technical report, Fakultät fr Geistes-, Sozial- und Erziehungswissenschaften, 2009.

[SC96] A.L. Strauss and J.M. Corbin. *Grounded Theory: Grundlagen Qualitativer Sozialforschung*. Beltz, Psychologie-Verlag-Union, 1996.

[Sch05] Frank Schlesinger. Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung. Diplomarbeit, AG Software Enginerring, Freie Universität Berlin, 2005.

[Sch07] Stephan Schümmer, Till und Lukosch. *Patterns for computer-mediated interaction*. John Wiley and Sons, 2007.

[Sch12] J. Schenk. Evaluating awareness information in distributed collaborative editing by software-engineers. In *User Evaluation for Software Engineering Researchers (USER), 2012*, pages 35–38, June 2012.

[SL07] T. Schummer and S. Lukosch. *Patterns for Computer-Mediated Interaction*. Wiley Software Patterns Series. Wiley, 2007.

[SOBS10] Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. Saros: An eclipse plug-in for distributed party programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 48–55, New York, NY, USA, 2010. ACM.

[SPP07] Stephan Salinger, Laura Plonka, and Lutz Prechelt. A Coding Scheme Development Methodology Using Grounded Theoryfor Qualitative Analysis of Pair Programming. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group*, 2007.

[SPS13] Julia Schenk, Lutz Prechelt, and Stephan Salinger. Distributed-pair programming can work well and is not just distributed pair-programming. *arXiv preprint arXiv:1311.6249*, 2013.

[SS00] Till Schümmer and Jan Schümmer. Support for distributed teams in extreme programming. In *Proceedings of eXtreme Programming and Flexible Processes Software Engineering - XP2000*, pages 355–377. Addison Wesley, 2000.

[SSB+02] David Stotts, Jason McC. Smith, Prashant Baheti, Prashant Baheti, Laurie Williams, Laurie Williams, Edward Gehringer, and Edward Gehringer. Distributed pair programming: Empirical studies and supporting environments. Technical report, 2002.

[Web07] Markus Weber. Mikro-Interaktionen vs. Makro-Interaktionen, Juli 2007. http://www.centigrade.de/de/blog/article/mikro-

interaktionen-vs-makro-interaktionen/, Letzter Zugriff am 09.10.2014.

[Wik14] the free encyclopedia Wikipedia. Computer-assisted qualitative data analysis software, September 2014. http://en.wikipedia.org/wiki/Computer-assisted_qualitative_data_analysis_software, Letzter Zugriff am 19.10.2014.

[WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, July 2000.

[You08] I. Young. *Mental Models: Aligning Design Strategy with Human Behavior*. Rosenfeld Media, 2008.