Freie Universität Berlin, Fachbereich Mathematik und Informatik

Institut für Informatik, Studiengang Informatik (Bsc)

Berlin, 23. Februar 2010

Bachelorarbeit

zum Thema

„Unterstützung unterschiedlicher Programmiersprachen und Plugins in einem Werkzeug für kollaborative Softwareentwicklung“

von

Alena Kiwitt

Matr.-Nr.: 4055744

kiwitt@inf.fu-berlin.de

Betreuer: Karl Beecher und Stephan Salinger

Eingereicht bei: Prof. Dr. Lutz Prechelt

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die vorliegende Bachelorarbeit von niemand anderem als meiner Person verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Alena Kiwitt

Berlin, den 23. Februar 2010

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung von Software egal welcher Art findet zunehmend nicht mehr zwangsweise an ein- und demselben Ort statt. Gerade im Rahmen von Open Source Projekten kommt es häufig vor, dass die beteiligten Entwickler gar in unterschiedlichen Ländern arbeiten und ausschließlich über das Internet kommunizieren.

Um eine Zusammenarbeit in Echtzeit trotz der Verteilung der Entwickler über verschiedene Standorte zu ermöglichen, hat die Arbeitsgruppe Software-Engineering ein Werkzeug zur verteilten Programmierung namens Saros entwickelt, das als Plugin für die Eclipse-Plattform verfügbar ist.

Da standardmäßig zunächst nur die Entwicklung mit Java unterstützt wird, soll diese Arbeit die Verwendbarkeit von Saros mit anderen Programmiersprachen untersuchen und auswerten.

Die Abschnitte 2.1 und 2.2 erläutern zunächst die Grundlagen zu Eclipse und Saros, bevor in Abschnitt 3 neben der Motivation auf die bisherige Situation und die genaue Zielsetzung beleuchtet wird.

Abschnitt 4 beschreibt mögliche Auswahlkriterien für zu testende Sprach-Plugins und gibt im Anschluss eine Übersicht über den konkreten Auswahlprozess und die ausgewählten Testkandidaten.

In Abschnitt 5 werden die verschiedenen Funktionen des Saros-Plugins einerseits und der zu testenden Plugins andererseits identifiziert und in einer umfangreichen Anforderungsanalyse zu konkreten Anforderungen zusammengefasst.

Diese ermittelten Anforderungen dienen nun in Abschnitt 6 zunächst als Grundlage für die Erstellung von Testfällen, mit deren Hilfe im Anschluss die Testkandidaten auf ihre Kompatibilität mit Saros hin getestet werden.

Nach der Auswertung der Testergebnisse in Abschnitt 7 zieht Abschnitt 8 ein Fazit über die gewonnenen Erkenntnisse. Ein Ausblick auf weitere Aspekte, die sich aus den Ergebnissen der Untersuchungen ergeben, schließt die Arbeit ab.

2 Grundlagen

Zum besseren Verständnis erläutern die folgenden zwei Abschnitte zunächst einige Grundlagen und Konzepte in Bezug auf Eclipse und Saros.

Neben der Definition stehen hier vor allem die Architektur der beiden Komponenten sowie die Einordnung und Abgrenzung im Vordergrund.

2.1 Eclipse

Eclipse ist eine integrierte Entwicklungsumgebung (kurz: IDE von engl. „Integrated Development Environment“), die basierend auf der Java-Technologie[1] ursprünglich für die Programmiersprache Java entwickelt wurde. Die IDE steht als quelloffene Software plattformunabhängig zur Verfügung und ist durch die Unterstützung der unterschiedlichsten Plugins mittlerweile für fast alle gängigen Programmiersprachen nutzbar.

Eclipse ist neben der Basis-Ausstattung „Eclipse Classic“, die auf die Entwicklung in Java zugeschnitten ist, auch noch in zahlreichen anderen vorkonfigurierten Paketen verfügbar, wie beispielsweise in einer Version speziell für PHP-Entwickler, für die Entwicklung mit Java EE oder in einer Version, die sich an C/C++ Entwickler richtet.[2]

All diese Pakete enthalten als Kern die eigentliche Eclipse Plattform, die aktuell in der Version Eclipse Galileo 3.5.1 vorliegt. Dieser Kern wird je nach Paket durch passende Plugins erweitert, sodass die Arbeitsumgebung an die speziellen Bedürfnisse der verschiedenen Entwickler angepasst werden kann.

Diese Erweiterbarkeit der Eclipse IDE stellt ihre größte Stärke dar, denn der relativ schlanke Kern der Entwicklungsumgebung (in Abb. 1 als „Eclipse Platform“ bezeichnet) ist durch die Kombination mit den verschiedensten Plugins individuell anpassbar und so kann ein breites Spektrum an Anforderungen durch das richtige Plugin oder auch die gleichzeitige Verwendung mehrerer Plugins erfüllt werden.

Abbildung 1 stellt die wesentliche Architektur der Eclipse IDE schematisch dar. Der eigentliche Kern („Eclipse Platform“) besteht vor allem aus der Eclipse Laufzeitumgebung („Runtime“), dem *Workspace* und dem *Workbench*.

Im Workspace verwaltet Eclipse die vom Benutzer angelegten Projekte. Der Workbench stellt die grafische Benutzeroberfläche der Eclipse IDE dar, also letztendlich das, was der Anwender von der Eclipse IDE sieht.

Alle weiteren Elemente sind als Plugins realisiert, indem sie sogenannte *Extension Points* erweitern, die von Eclipse zur Verfügung gestellt werden. Jedes Plugin kann zudem selbst Extension Points anbieten, die wiederum von anderen Plugins erweitert werden können – so können verschiedene Plugins aufeinander aufbauen und es wird eine hohe Flexibilität der verwendeten Komponenten erreicht. Weiterführende Informationen zur Eclipse-Architektur und Extension Points bietet die Ausarbeitung [TW].

1 Zusammenfassung der Programmiersprache Java und verschiedener Laufzeitumgebungen zum Einsatz von Java-basierter Software. Details siehe http://www.sun.com/java/about/

2 Eine Übersicht der verfügbaren Pakete ist auf der Eclipse Projektseite unter http://www.eclipse.org/downloads/ erhältlich.

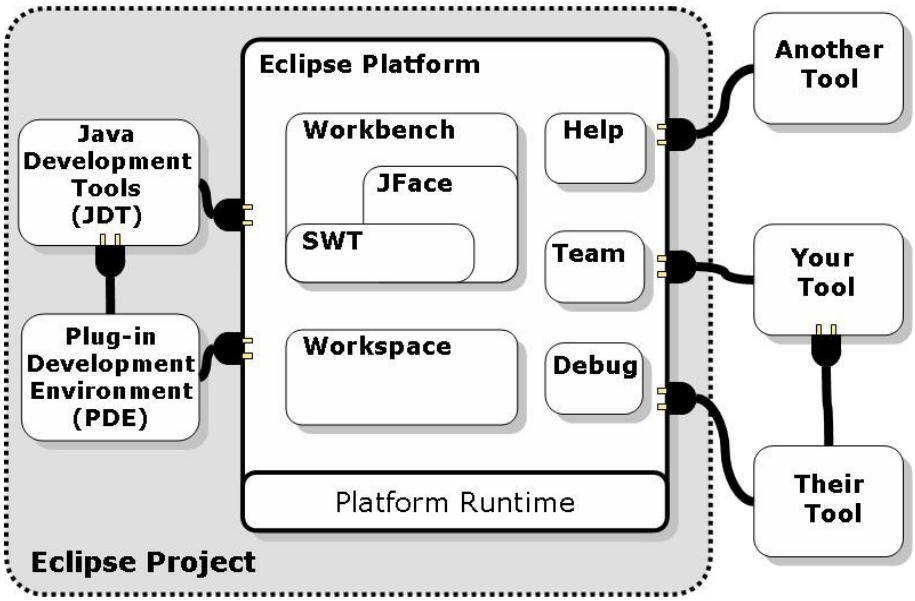

Abbildung 1: Schematische Darstellung der Architektur der Eclipse IDE

Die Benutzeroberfläche der Eclipse IDE hält verschiedene Konzepte bereit, um das Arbeiten mit der Software so übersichtlich wie möglich zu gestalten:

Sichten (engl. „Views“) sind Komponenten, die bestimmte Funktionen bereitstellen und in verschiedenen Formen in die Benutzeroberfläche integriert oder auch per Drag & Drop frei positioniert werden können. Eclipse hält standardmäßig z.B. eine Sicht bereit, die das Navigieren durch geöffnete Projekte ermöglicht (Ordner und Dateien werden hierbei in einer Baumstruktur dargestellt) oder eine andere Sicht, die Elemente innerhalb einer Datei (z.B. Klassen, Funktionen oder Variablen) in der gleichen Weise darstellt. Abbildung 2 zeigt Beispiele für Sichten grün umrandet.

Editoren nehmen meist den Großteil der Benutzeroberfläche ein, da sie als Hauptwerkzeug zur Entwicklung dienen. Sie bieten die Möglichkeit zum Editieren des Quelltextes und je nach Plugin weitere Funktionalität wie Syntax-Hervorhebung, automatische Code-Vervollständigung oder Code-Formatierung. Neben den reinen Quelltext-Editoren, die meist gesondert für jede einzelne unterstütze Programmiersprache zur Verfügung stehen, gibt es auch eine Reihe visueller Editoren, wie z.B. einen UML[3]-Editor, der die Erstellung von UML-Diagrammen in einer grafischen Oberfläche ermöglicht, einen XML[4]-Editor, der Dateiinhalte in einer Baumstruktur darstellt, oder auch einen „Drag & Drop“-Editor zum Erstellen von grafischen Benutzeroberflächen. In Abbildung 2 ist der Editor durch eine rote Umrandung gekennzeichnet.

3 Unified Modeling Language zur Erstellung von Implementierungs-Modellen mithilfe von Strukturdiagrammen [UML]

4 Extensible Markup Language [XML]

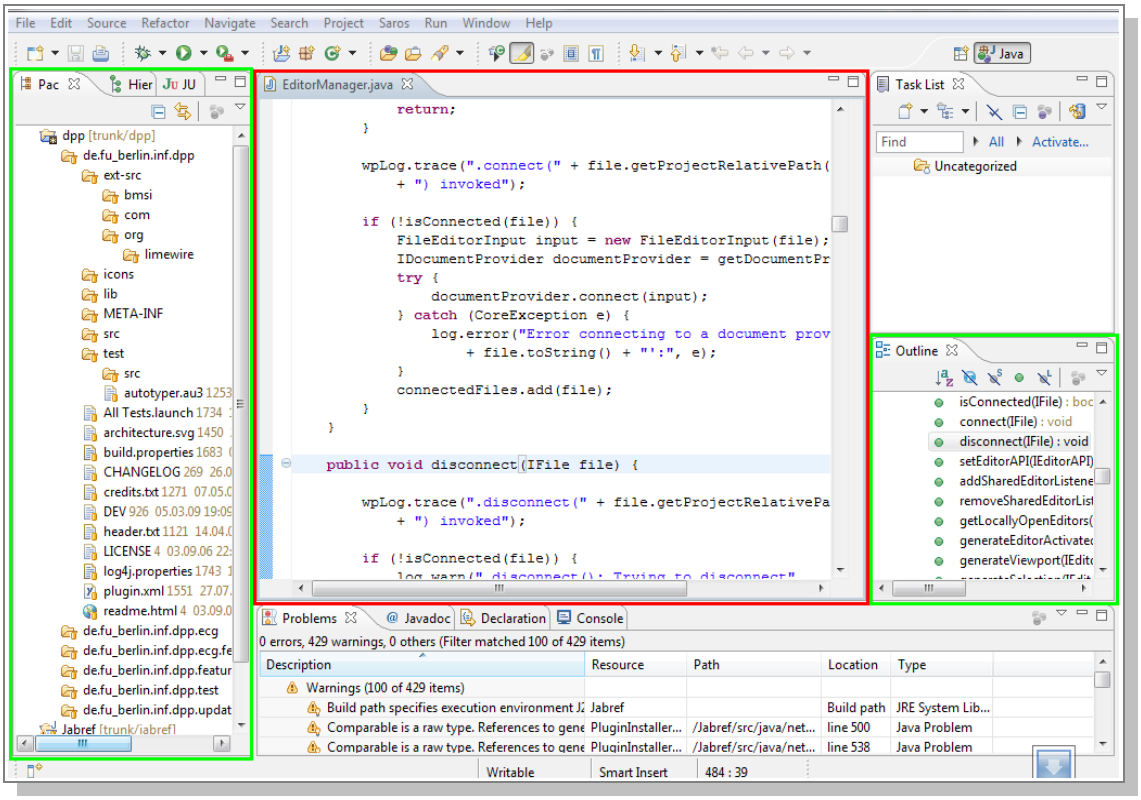

Abbildung 2: Benutzeroberfläche der Eclipse IDE.

Perspektiven schließlich fassen ganze Kombinationen von Editoren, Sichten und Menü- bzw. Symbolleisten zusammen, die auch in der jeweiligen Benutzereinstellung gespeichert und später wiederverwendet werden können. Eclipse-Erweiterungen beinhalten meist wenigstens eine Perspektive, die standardmäßig eine voreingestellte Kombination der benötigten Komponenten lädt. Diese Perspektiven kann der Benutzer nachträglich an seine Bedürfnisse anpassen.

2.2 Saros

Saros wird als Eclipse-Plugin von der Arbeitsgruppe Software-Engineering der Freien Universität Berlin entwickelt und dient als Werkzeug für verteilte kollaborative Softwareentwicklung[5]. Im Vordergrund steht hier vor allem die verteilte Paarprogrammierung.

Das Konzept der Paarprogrammierung wird vor allem im Rahmen der agilen Softwareentwicklung angewendet. Zwei Programmierer bearbeiten hier gemeinsam dieselbe Aufgabe. Während der Programmierer in der Rolle des sogenannten *Drivers* aktiv Programmcode produziert und dabei sein Vorgehen erläutert, überwacht sein Partner, der sogenannte *Observer* (oder Navigator), diesen Prozess und greift ggf. mit Verbesserungsvorschlägen, Korrekturen von Flüchtigkeitsfehlern o.ä. ein. Eine detaillierte Beschreibung des Konzepts sowie eine Abwägung der Vor- und Nachteile ist unter [PP] verfügbar.

Saros beschränkt sich jedoch nicht auf die Synchronisation von nur zwei Teilnehmern. Auch mehrere Teilnehmer an jeweils unterschiedlichen Orten können an einer Saros-Sitzung teilnehmen.

5 https://www.inf.fu-berlin.de/w/SE/DPP

Alle Teilnehmer einer Saros-Sitzung besitzen eine identische Kopie des Projektes, das sie miteinander bearbeiten wollen. Während des Sitzungsverlaufs sorgt Saros nun dafür, dass all diese Kopien stets synchronisiert werden, sodass alle Teilnehmer den Programmierprozess mitverfolgen können.

Zur Initialisierung der Sitzung verwendet Saros einen XMPP- bzw. Jabber-Server (siehe [XJ] und weiterführende Links). Dieser wird auch für die interne Kommunikation genutzt, sollten einzelne Sitzungsteilnehmer nicht direkt via IP erreichbar sein (z.B. weil sie sich hinter einer Firewall befinden). Um eine geringe Latenz und hohe Bandbreite bei der Datenübertragung zwischen den einzelnen Sitzungsteilnehmern zu erreichen, benutzt Saros STUN[6] und Jingle[7].

Angelehnt an den Ansatz der Paarprogrammierung teilt Saros die Benutzer in einen Driver und ein oder mehrere Observer auf. Der Driver übernimmt den aktiven Part des Programmierens, besitzt Schreibrechte und dient als Ausgangspunkt der Synchronisation durch Saros. Ein Observer dagegen besitzt keine Schreibrechte und kann entweder unabhängig im Projekt navigieren oder die Aktivitäten des Drivers unmittelbar im sogenannten *Follow Mode* unmittelbar mitverfolgen.

Die Rolle des Drivers kann prinzipiell jedem Teilnehmer zugewiesen werden, bei Sitzungsbeginn befindet sich allerdings stets der *Host*, also der Initiator der Sitzung, in der Driver-Rolle. Experimentell unterstützt Saros auch mehrere Driver gleichzeitig.

Saros integriert sich in die Eclipse-Benutzeroberfläche mit zwei speziellen Sichten, dem *Roster* und der *Saros Session.* Beide Sichten erscheinen standardmäßig im Bereich unterhalb des Editors.

Der Roster stellt die Funktionalität für die Verbindungsverwaltung der teilnehmenden Saros-Nutzer bereit. Hier kann der Nutzer sich mit seinen hinterlegten Zugangsdaten bei einem Instant-Messaging-Dienst wie Jabber[8] anmelden, neue Kontakte zu seiner Freunde-Liste hinzufügen oder während einer bestehenden Saros-Sitzung Kontakte zu dieser Sitzung einladen.

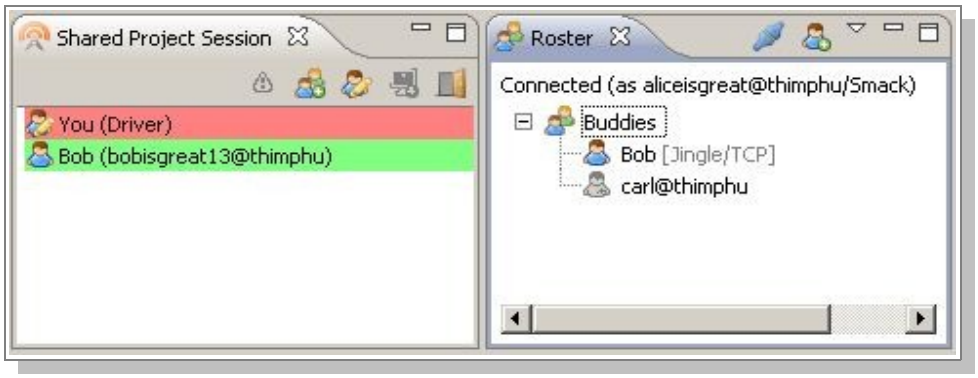

Abbildung 3: Ansicht von Roster und Shared Session während einer Saros-Sitzung.

Die Saros Session-Sicht dient zur Koordination der Teilnehmer einer Saros-Sitzung. Sie

6 Session Traversal Utilities for NAT, Definition siehe [STU]
7 http://xmpp.org/tech/jingle.shtml
8 Auf dem offenen Standard XMPP basierender Instant-Messaging Dienst, Näheres siehe http://www.jabber.org/about/

wird erst dann aktiviert, wenn die Benutzer eine Saros-Sitzung miteinander gestartet haben und ein Projekt miteinander teilen. Die Teilnehmer werden der Übersichtlichkeit wegen in verschiedenen Farben markiert. Die Farben entsprechen denen, die zur Markierung der Aktivitäten eines Benutzers im Editor benutzt werden (vgl. Seite 20, Abschnitt 5.1.3 Awareness-Informationen). Zudem lässt sich ablesen, welche Benutzerrolle einem Teilnehmer aktuell zugewiesen ist. Die Verteilung der Benutzerrollen kann hier vom Host der Saros-Sitzung ebenfalls geändert werden. Jeder Teilnehmer kann über die Saros Session-Sicht den Follow Mode aktivieren oder auch wieder deaktivieren. Sollten während einer Saros-Sitzung Inkonsistenzen zwischen den Projekt-Versionen der Teilnehmer auftreten, können diese hier mittels Klick auf das erscheinende Warndreieck behoben werden.

Saros dient eindeutig nicht als Werkzeug für Screensharing. Zwar ist für eine der kommenden Saros-Versionen die Integration einer zusätzlichen Screensharing-Funktionalität vorgesehen [Lau10], doch diese muss als Extra betrachtet werden, das unabhängig von der eigentlichen Synchronisation der Eclipse-Projekte zur Verfügung gestellt wird, um den Entwicklern die Möglichkeit zu geben, auch Aktivitäten außerhalb der Eclipse IDE verfolgen zu können, ohne weitere externe Programme einsetzen zu müssen.

3 Motivation

Da Saros als Eclipse Plugin mithilfe der PDE (von engl. „Plug-in Development Environment" – Plugin Entwicklungsumgebung) unter Eclipse Classic entwickelt und getestet wird, unterstützt es standardmäßig zuerst einmal die Entwicklung mit Java im Rahmen des JDT (von engl. „Java Development Tools" – Java Entwicklungswerkzeuge), welches im Eclipse Classic Paket bereits enthalten ist. Zwar baut Saros an keiner Stelle auf besondere Eigenschaften des JDT oder auf Eigenschaften der Sprache Java auf, sondern bedient sich zur Synchronisation der vordefinierten Eclipse Plugin-Schnittstellen für Editoren, doch kann ohne ausreichende Tests zunächst nichts über das Zusammenspiel mit anderen Eclipse-Plugins zur Unterstützung anderer Programmiersprachen gesagt werden.

Dass es jedoch reizvoll ist, Saros auch für die Entwicklung mit anderen Programmiersprachen als Java einsetzen zu können, liegt nahe. Denn auch wenn die Sprache Java aktuell in einer Vielzahl von Projekten eingesetzt wird und sowohl in der freien als auch in der kommerziellen Softwareentwicklung weit verbreitet ist, macht sie laut Tiobe[9] gerade einmal ein gutes Sechstel der beliebtesten Programmiersprachen aus. Abbildung 4 zeigt die Verteilung der nach dem Tiobe-Index[10] 10 beliebtesten Programmiersprachen.

9 Seit Oktober 2000 bestehende Gesellschaft, die sich mit der Messung von Software-Qualität befasst; nähere Informationen siehe [TC]

10 Aus den Treffer-Listen der verbreitetsten Suchmaschinen monatlich errechneter Programmiersprachen-Index. Genaue Informationen zur Auswahl der Suchmaschinen, verwendeter Definition von Programmiersprache, etc. siehe [TID]

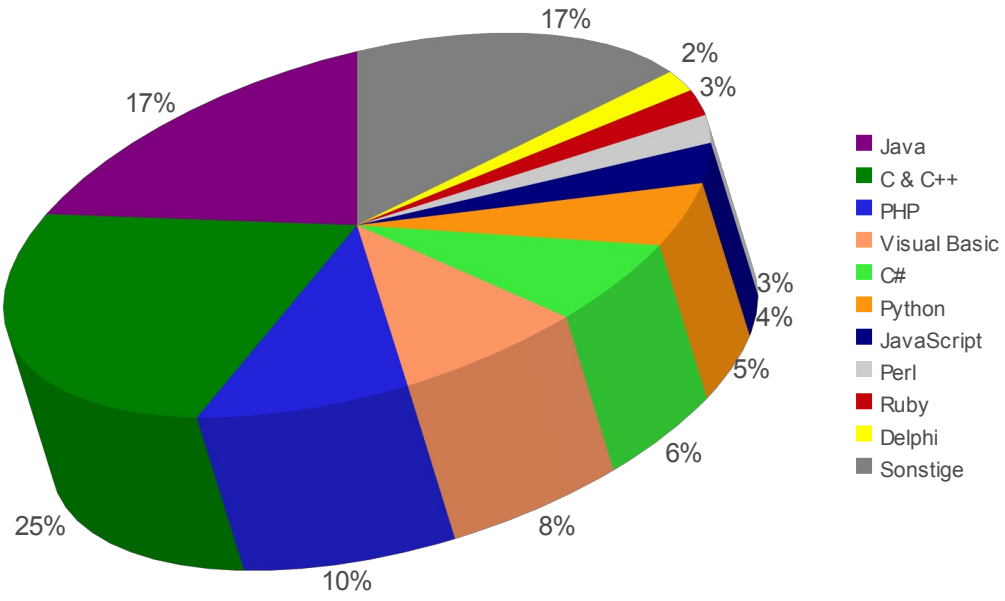

Abbildung 4: Verteilung der beliebtesten Programmiersprachen nach dem Tiobe-Index.

Je mehr dieser Sprachen also durch Saros unterstützt werden können, desto mehr Entwickler können potentiell erreicht werden und desto mehr kann sich auch der Verbreitungsgrad des Saros-Plugins steigern.

Besonders für das Saros-Team ist die umfangreiche Verbreitung des Saros-Plugins und dessen Benutzung durch möglichst viele Anwender von großem Interesse. Denn nur auf diese Weise kann nützliches und ausreichendes Feedback gewonnen werden (vgl. [Doh09], insbesondere Kapitel 2.2), um die Entwicklung von Saros in die aus Sicht der Anwender richtige Richtung voranzutreiben sowie Fehler zügig entdecken und entfernen zu können.

Auch im Hinblick auf die Arbeit von Eike Starkmann, der sich im Rahmen seiner Diplomarbeit [Sta10] unter anderem mit der Bekanntmachung von Saros in verschiedenen Open Source Projekten beschäftigt, ist eine Ausweitung der unterstützen Programmiersprachen reizvoll, da auf diese Weise die Anzahl der möglichen Kandidaten unter den Projekten steigt, also mehr verschiedene Projekte angesprochen und untersucht werden können.

3.1 Bisherige Situation

Die Situation von Saros in Bezug auf die Unterstützung von anderen Programmiersprachen als Java ist bisher vor allem durch ein hohes Maß an Unsicherheit gekennzeichnet. So wird aufgrund der Architektur von Saros und Eclipse (vgl. Seite 6, Abschnitt 2.1) zwar angenommen, dass die Verwendung von Saros auch in Kombination mit allen anderen Eclipse-Erweiterungen möglich sein sollte, soweit diese die von Eclipse definierten Schnittstellen korrekt implementieren – genau hier liegt aber auch schon ein Schwachpunkt, denn auf welche Plugins diese Annahme zutrifft, ist vorerst nicht bekannt.

Um die Unsicherheit zu beseitigen oder wenigstens zu verringern wurden in der Vergangenheit vereinzelte Tests durchgeführt, die die Arbeit mit Saros und einigen bestimmten

Plugins untersuchen sollten. Die Ergebnisse dieser Tests führten zu einer Tabelle, die eine Auflistung aller bisher getesteten Plugins inklusive eines Vermerks über die Kompatibilität zu Saros enthält.[11] Diese Auflistung hat jedoch gleich mehrere Schwachstellen:

Sprache	Plugin	Version	Eclipse	Saros	Unterstützt
Java	JDT	3.5	3.5	9.6.18	✓
C/C++	CDT	6.0.1	3.4	9.6.18	✓
PHP	PDT	2.0	3.4	9.7.10	✓
Ruby	Aptana Studio	1.5	3.4	9.6.18	✓
Python	PyDev	1.4.8	3.4	9.7.10	✓
TypoScript	Dev3	1.0.3	3.4	9.7.10	✗

Tabelle 1: Exemplarische Darstellung der früheren Kompatibilitätsliste.

Das auffälligste Problem besteht zunächst darin, dass die Tests meist gleich in dreierlei Hinsicht veraltet sind – sowohl die zugrunde liegende Version der Eclipse IDE, als auch die zum Test verwendete Version von Saros sowie schließlich die Versionen der getesteten Plugins sind nicht aktuell. Da sich besonders das Saros-Plugin jedoch in einer Entwicklungsphase befindet, in der oft größere und grundlegende Änderungen stattfinden und auch die meisten Plugins einer ständigen Weiterentwicklung unterliegen, ist die Aktualität der Tests von entscheidender Bedeutung.

Zudem ist die Anzahl der getesteten Plugins angesichts der Menge an für Eclipse verfügbaren Sprach-Plugins relativ gering. So wurden beispielsweise nicht einmal alle der ersten 10 beliebtesten Programmiersprachen berücksichtigt.

Das weitaus schwerwiegendste Problem stellt allerdings die fehlende Dokumentation der Tests dar. So ist aus der Tabelle weder ersichtlich, welche Eigenschaften von Saros auf der einen und vom getesteten Plugin auf der anderen Seite untersucht wurden, noch ist nachvollziehbar, auf welche Weise die Untersuchungen durchgeführt wurden. Dies ist in erster Linie darauf zurückzuführen, dass keine klaren Anforderungen formuliert wurden, um den Begriff der Kompatibilität von Saros mit einem Plugin in diesem Zusammenhang zu definieren. Als Folge daraus sind die Tests in keiner Weise vergleichbar, denn im Zweifel ist es sogar möglich, dass bei den verschiedenen Plugins auch unterschiedliche Eigenschaften getestet wurden.

3.2 Zielsetzung

Das übergeordnete Ziel dieser Arbeit besteht darin, die Voraussetzungen für eine möglichst weite Verbreitung von Saros zu schaffen. Zu diesem Zweck sollen viel genutzte Programmiersprachen identifiziert und deren Unterstützung durch Saros untersucht werden.

11 https://www.inf.fu-berlin.de/w/SE/DPPCompatiblePlugins

Es soll daher eine Reihe von Plugins ausgewählt werden, die im weiteren Verlauf der Arbeit auf ihre Kompatibilität mit Saros hin getestet werden soll. Dabei liegt ein besonderes Augenmerk auf den Auswahlkriterien, d.h. der Art und Weise, wie aus den zahlreichen verfügbaren Eclipse-Erweiterungen diejenigen ausgesucht werden können, deren Unterstützung durch Saros sinnvoll und nützlich ist.

Um den Begriff der Kompatibilität von Saros mit anderen Plugins genau zu fassen, sollen dann im Rahmen einer umfassenden Anforderungsanalyse Funktionalitäten identifiziert werden, die die Arbeit mit Saros einerseits sowie mit dem jeweiligen Plugin andererseits charakterisieren. Während der Funktionsumfang des Saros-Plugins sehr detailliert bekannt ist, besteht die Herausforderung vor allem darin, bei den verschiedenen Sprach-Plugins, die sich möglicherweise sehr stark in der Benutzung unterscheiden, die jeweils wichtigsten Features sowie typische Arbeitsabläufe zu erkennen. Um diesen Schritt der Anforderungserhebung zukünftig zu vereinfachen, soll ein Konzept entworfen werden, das zur Identifizierung von wichtigen bzw. unterstützenswerten Features möglichst unabhängig vom betrachteten Plugin herangezogen werden kann.

Aus den ermittelten Anforderungen sollen im Anschluss Testfälle abgeleitet werden, die Saros und das jeweilige Sprach-Plugin auf die Erfüllung der Anforderungen hin überprüfen. Die Testfälle sollen so ausgewählt werden, dass sie in der Regel ohne größere Anstrengung auf jedes beliebige Plugin anwendbar sind, d.h. dass sie insbesondere nicht von der verwendeten Programmiersprache abhängig sind.

Mithilfe der erstellten Testfälle sollen nun die vorher ausgewählten Sprach-Plugins getestet werden. Die Vorgehensweise bei der Durchführung der Tests wird hierbei genau beschrieben, sodass eine Vergleichbarkeit zwischen den verschiedenen Plugins hergestellt werden kann.

Bei der anschließenden Auswertung der Testergebnisse gilt es schließlich, evtl. gefundene Inkompatibilitäten bzw. fehlerhaftes Verhalten bei der Zusammenarbeit von Saros und den untersuchten Plugins zu bewerten und daraus nach Möglichkeit typische Konfliktpunkte abzuleiten. Außerdem sollen mögliche Lösungsansätze für die gefundenen Konflikte vorgestellt werden und die Testergebnisse abschließend in einer Plugin-Kompatibilitätsliste zusammengefasst werden.

4 Auswahl der zu testenden Plugins

Der Grundgedanke bei der Auswahl der zu testenden Plugins besteht darin, Saros für so viele Entwickler wie möglich nutzbar zu machen, damit sowohl für die Programmierer als auch für das Saros-Team der größtmögliche Nutzen entsteht.

Im Wesentlichen folgt daraus, dass nach der Auswahl der Plugins die am weitesten verbreiteten Programmiersprachen (vgl. Seite 10, Abbildung 4) jeweils durch wenigstens ein Plugin vertreten sein sollen. Stehen zur Unterstützung einer bestimmten Programmiersprache mehrere Plugins zur Verfügung, so sollen diese wiederum absteigend nach dem Grad der Verbreitung geordnet werden, sodass das am weitesten verbreitete Plugin zuerst getestet wird.

Wie genau der Grad der Verbreitung eines Plugins bestimmt wird, liegt dabei natürlich zunächst nicht auf der Hand und auch durch das Heranziehen verschiedener Ansätze wird in diesem Punkt keine exakte Genauigkeit möglich sein. Dies ist allerdings auch gar nicht notwendig, da eine grobe Sortierung für die Auswahl ausreichend ist.

Die folgenden Abschnitte stellen mögliche Vorgehensweisen bei der Auswahl geeigneter Testkandidaten aus den verfügbaren Eclipse-Erweiterungen vor und wählen dementsprechend schließlich eine Reihe von Plugins aus, die im weiteren Verlauf dieser Arbeit auf Kompatibilität mit Saros getestet werden sollen.

4.1 Auswahlkriterien

Der Eclipse Marketplace[12] bietet eine Auflistung der meisten für Eclipse verfügbaren Plugins und dient daher als Informationsquelle und weitere Grundlage zur Auswahl geeigneter Plugins. Die Seite stellt zum Einen eine Kategorisierung der gelisteten Plugins nach Art bzw. Einsatzgebiet bereit. Dies ermöglicht eine erste Einschränkung der rund 1000 dort geführten Erweiterungen. Da Saros ausschließlich auf die Synchronisation von Textinhalten ausgelegt ist, kommen nur Plugins in Frage, deren Hauptfunktionalität ebenfalls in der Bearbeitung von Quelltexten besteht. Zudem liegt der Fokus dieser Arbeit auf der Unterstützung zusätzlicher Programmiersprachen, was die Kategorisierung nach Sprach-Plugins nahe legt.

Zum Anderen bietet der Eclipse Marketplace die Möglichkeit, die Auflistung der Plugins in der jeweils gewählten Kategorie nach verschiedenen Eigenschaften zu sortieren. Von Interesse sind hier besonders zwei Eigenschaften: die Beliebtheit eines Plugins und der Grad der Aktivität bzw. der Zeitpunkt der letzten Aktualisierung. Diese Eigenschaften sind Indizien dafür, dass ein Plugin einerseits aktiv genutzt und eingesetzt und andererseits auch aktiv weiterentwickelt wird. Beide Aspekte sind deshalb wichtig für die Aufnahme eines Plugins in die Auswahlliste, da es keinen Nutzen für das Saros-Projekt hätte, ein Plugin zu unterstützen, das in der Praxis gar nicht oder nur sehr vereinzelt eingesetzt wird. Vor allem das für die produktive Weiterentwicklung von Saros unabdingbare Feedback möglicher Nutzer wäre hier gar nicht zu erwarten.

Auch die Aktivität der Entwickler-Community spielt eine nicht zu vernachlässigende Rolle. Sollten bei einem Plugin Probleme bei der gemeinsamen Verwendung mit Saros festgestellt werden, die ihren Ursprung nicht auf Seiten von Saros haben, so wird eine Auflösung des Konflikts sehr erschwert, sollte das Plugin nicht stetig weiterentwickelt oder wenigs-

12 http://marketplace.eclipse.org

tens gewartet werden. Zwar wäre es in diesem Fall möglich, den Konflikt selbst aufzulösen, da es sich in den meisten Fällen um Open Source Projekte handelt. Doch hierzu wäre vermutlich eine umfangreiche Einarbeitung in das jeweilige Plugin notwendig, um im Zweifel vielleicht ein recht simples Problem zu lösen. Ist die Entwickler-Community dagegen aktiv, so kann die Konfliktlösung von Entwicklern übernommen werden, die mit dem jeweiligen Plugin vertraut sind und für die diese Aufgabe vermutlich mit deutlich weniger Aufwand verbunden ist.

Neben der Suche über den Eclipse Marketplace hat sich zudem die Verwendung einer Suchmaschine als nützlich erwiesen. Ähnlich dem Ansatz des Tiobe-Indexes zur Ermittlung der beliebtesten Programmiersprachen kann angenommen werden, dass besonders beliebte oder aktive bzw. weit verbreitete Plugins auch eine entsprechend hohe Positionierung innerhalb der Suchergebnisse erzielen. Um eine Vergleichbarkeit herzustellen, sollte bei der Suche ein einheitliches Suchmuster verwendet werden, wie beispielsweise *Eclipse Plugin <Programmiersprache>* oder schlicht *Eclipse <Programmiersprache>*.

Ein wichtiger Aspekt ist außerdem die Art der Lizenzierung, der ein Plugin unterliegt. So gibt es nicht wenige Plugins, die unter einer kommerziellen Lizenz vertrieben werden, welche sogar eine kostenlose nicht-kommerzielle Nutzung des Plugins ausschließt. Die kostenfreie Nutzung ist jedoch zwingende Voraussetzung für die Eignung eines Plugins, sodass diese Kandidaten umgehend ausscheiden. Eine Ausnahme hierzu stellen ggf. solche Plugins dar, die in vollem Umfang für begrenzte (aber ausreichende) Zeit als kostenlose (Test-)Version zur Verfügung stehen, sodass ein Test der angebotenen Funktionen trotzdem möglich ist, bzw. solche Plugins, für die abweichende Lizenz-Bedingungen für den akademischen Bereich gelten.

4.2 Der konkrete Auswahlprozess

Auf dem Tiobe-Index basierend werden zunächst die 10 am weitesten verbreiteten Programmiersprachen identifiziert. Dieses sind in absteigender Reihenfolge C/C++[13], Java, PHP, Visual Basic, C#, Python, JavaScript, Perl, Ruby und Delphi. Für jede dieser Sprachen (mit Ausnahme von Java, da diese wie ausgeführt bereits nativ unterstützt wird) wird nun zum einen der Eclipse Marketplace benutzt, um nach geeigneten Plugins zu suchen. Dabei wird stets der Name der Sprache für die Suche verwendet. Aus den Ergebnissen werden diejenigen Plugins ausgewählt, die den im letzten Abschnitt erläuterten Kriterien entsprechen.

Zum anderen wird die Suchmaschine Google[14] verwendet, um weit verbreitete bzw. beliebte Eclipse-Plugins zu jeder der oben genannten Programmiersprachen zu finden. Hierbei wird für die Suche stets das gleiche Suchmuster *Eclipse <Programmiersprache>* verwendet, also beispielsweise *Eclipse Python* für die Suche nach Eclipse-Plugins für die Unterstützung von Python. Berücksichtigt werden jeweils die Einträge der ersten zwei Ergebnisseiten, also insgesamt 20 Einträge.

Nach erfolgter Auswertung der Suchergebnisse ergibt sich folgende Tabelle der je Programmiersprache verfügbaren Eclipse-Erweiterungen:

13 Die beiden Sprachen C und C++ werden hier gemeinsam betrachtet, da sich die verfügbaren Eclipse-Plugins jeweils überschneiden. Es werden also stets beide Sprachen unterstützt.

14 Suchmaschine des US-amerikanischen Unternehmens Google Inc., zu erreichen unter http://www.google.de/

Sprache	Verfügbare Plugins
C/C++	C/C++ Development Tooling (CDT), EasyEclipse for C/C++
PHP	PHP Development Tools (PDT), Zend Studio, PHPEclipse, EasyEclipse for PHP
Visual Basic	*- keine Erweiterung für Eclipse verfügbar -*
C#	Improve (veraltet)
Python	PyDev, EasyEclipse for Python
JavaScript	Aptana Studio
Perl	Eclipse Perl Integration (EPIC), EasyEclipse for Perl
Ruby	Aptana RadRails, Ruby Development Tools (RDT)
Delphi	*- keine Erweiterung für Eclipse verfügbar -*

Tabelle 2: Verfügbare Eclipse-Erweiterungen je Programmiersprache.

Mit Ausnahme von Delphi und Visual Basic wurden zu jeder Programmiersprache ein oder mehrere Plugins gefunden, die die Entwicklung unter Eclipse möglich machen. Das Plugin „Improve“, das laut Suchergebnis offenbar als einzige Alternative für die Entwicklung unter Eclipse mit C# zur Verfügung steht, wird laut Projektseite[15] seit 2004 nicht mehr weiterentwickelt und deshalb auch im Folgenden nicht weiter berücksichtigt, da die Aktualität eines Plugins als zwingende Voraussetzung festgelegt wurde. Alle übrigen Plugins aus der Tabelle werden für die folgenden Kompatibilitätstests herangezogen.

Eine Besonderheit unter den ausgewählten Plugins stellt Aptana Studio dar. Dieses Plugin ist speziell auf die Entwicklung von Webanwendungen ausgelegt und neben der Version als Eclipse-Plugin auch als eigenständige Entwicklungsumgebung verfügbar. In der Basisausführung bietet es lediglich Unterstützung für die im Web allgemein eingesetzten Sprachen Javascript und HTML sowie für *Cascading Style Sheets* (CSS). Eine Ausrichtung auf eine bestimmte Programmiersprache erfolgt erst durch die Integration weiterer Plugins, nämlich dem PDT zur PHP-Unterstützung, PyDev für die Unterstützung von Python oder RadRails zur Integration von Ruby einschließlich des Frameworks Rails. Zusätzlich ist auch noch die Integration eines Adobe AIR Plugins möglich, das sich in diesem Kontext eher an Entwickler von HTML und Javascript richtet. Die eigenständigen Laufzeitumgebung unterstützt im Gegensatz dazu auch die Entwicklung mit Flash oder Flex.

Bei den verschiedenen Versionen von EasyEclipse handelt es sich genau genommen nicht um Plugins im eigentlichen Sinne. Vielmehr stellt EasyEclipse verschiedene vorkonfigurierte Pakete bereit, jeweils bestehend aus der Eclipse Plattform und je nach Einsatzzweck ausgewählten weiteren Plugins. Auf diese Weise soll den Entwicklern der Einstieg erleichtert werden, da sie sich das passende Paket aussuchen und nach der Installation umgehend mit der Entwicklung beginnen können, ohne noch weitere Komponenten hinzufügen und konfigurieren zu müssen. Da dieses Projekt in nahezu jedem Suchergebnis der ausgewählten Programmiersprachen zu finden war, was eine recht hohe Verbreitung vermuten lässt, wird es in die folgenden Tests miteinbezogen.

15 Projektseite des Plugins Improve: http://www.improve-technologies.com/alpha/esharp/

Außer den in Tabelle 2 genannten Plugins werden zudem zwei weitere Plugins getestet, da das Saros-Team entsprechende Anfragen über die Saros-Mailingliste erhalten hat.

Zum Einen soll auch Dev3 auf Kompatibilität getestet werden, das die Entwicklung mit TypoScript unterstützt. Hier waren bereits in einem früheren Test Probleme bei der Bearbeitung von TypoScript-Dateien festgestellt worden, die sich auf eine fehlerhafte Schnittstellennutzung der Dev3-Entwickler zurückführen ließen.

Zum Anderen wird auch das Adobe® Flex Builder Plugin, das zur Entwicklung in Flash eingesetzt wird, in die Tests aufgenommen. Dieses Plugin unterliegt zwar einer kommerziellen Lizenz und eine kostenfreie Nutzung ist daher nicht möglich. Für den akademischen Bereich gelten aber abweichende Bedingungen, sodass nach Vorlage eines entsprechenden Nachweises eine kostenfreie Version bezogen werden kann. Außerdem steht eine kostenfreie Testversion zur Verfügung, die sechs Wochen lang in vollem Funktionsumfang getestet werden kann und auch für die folgenden Tests verwendet wurde.

5 Anforderungsanalyse

In diesem Abschnitt sollen die verschiedenen Funktionalitäten des Saros-Plugins auf der einen und des jeweils zu testenden Sprach-Plugins auf der anderen Seite identifiziert und zu Anforderungsbereichen zusammengefasst werden. Jeder Anforderungsbereich definiert anhand der beschriebenen Funktionalitäten konkrete Anforderungen, die ein Plugin erfüllen muss, um als kompatibel mit Saros eingestuft zu werden.

Zu jedem Anforderungsbereich wird zudem vermerkt, welchen Stellenwert dieser einnimmt. Die Funktionalitäten werden im Folgenden in drei Kategorien eingeteilt:

- ***Kritisch***: Diese Funktionalität muss unbedingt gegeben sein, damit ein Plugin kompatibel mit Saros ist. Wird eine kritische Anforderung nicht erfüllt, führt dies zur Einstufung des getesteten Plugins als inkompatibel.
- ***Wesentlich***: Ohne diese Funktionalität ist das Arbeiten mit Saros zwar möglich, doch sie stellt einen wesentlichen Teil des Funktionsumfangs dar. Wird eine wesentliche Anforderung nicht erfüllt, führt dies zur Einstufung des getesteten Plugins als eingeschränkt kompatibel.
- ***Unkritisch***: Ohne diese Funktionalität ist die Arbeit mit Saros problemlos möglich. Sie stellt lediglich einen Zusatz dar. Unkritische Funktionalitäten werden daher nicht zur Formulierung von Anforderungen herangezogen und haben so bei Nichterfüllung keinen Einfluss auf die Einstufung des Plugins hinsichtlich der Kompatibilität.

Eine vollständige Liste der in den folgenden Abschnitten ermittelten Anforderungen kann in Anhang B nachgelesen werden.

5.1 Funktionalitäten des Saros-Plugins

Um die wesentlichen Funktionalitäten von Saros identifizieren zu können, muss zunächst eingegrenzt werden, welche Aufgaben dieses Plugin erfüllen bzw. welchem Zweck es dienen soll.

Wie in Abschnitt 2.2 dargestellt, dient Saros zur Unterstützung der verteilten Programmierung, und insbesondere der verteilten Paarprogrammierung. Hierbei unterstützt Saros

das Konzept der Paarprogrammierung durch die Zuweisung der Benutzerrollen Driver und Observer, synchronisiert die Projekte beteiligter Programmierer und bietet den Benutzern die Möglichkeit, den Aktivitäten anderer Benutzer im sogenannten Follow Mode automatisch zu folgen.

Verschiedene Awareness-Informationen helfen dabei, die Aktivitäten der Nutzer und besonders des Drivers im Überblick zu behalten.

Die folgenden Abschnitte beschreiben verschiedene Funktionalitäten, die die Arbeit mit dem Saros-Plugin charakterisieren und die aus Sicht von Saros unterstützt werden sollen, um die Kompatibilität mit anderen Sprach-Plugins herzustellen.

Hier werden zunächst die Funktionalitäten betrachtet, die Saros zur Verfügung stellt. Weitere Funktionalitäten, die von den verschiedenen Plugins bereitgestellt werden und die bei der Synchronisation von Saros berücksichtigt werden müssen, werden in Abschnitt 5.2 vorgestellt.

5.1.1 Installation und Aufbau einer Saros-Sitzung

Damit das Saros-Plugin zusammen mit einem anderen Sprach-Plugin erfolgreich verwendet werden kann, ist es unbedingt notwendig, dass beide Plugins gleichzeitig, d.h. in ein und der selben Eclipse IDE, installiert werden können. Da verschiedene Plugins eventuell bestimmte Versionen der gesamten Eclipse IDE oder Teilen davon voraussetzen, kann dieser Umstand jedoch nicht ohne weiteres angenommen werden.

Um die verschiedenen Funktionalitäten des Saros-Plugins nutzen zu können, muss als weitere Grundvoraussetzung eine gültige Saros-Sitzung gestartet werden, in der zwei oder mehr Teilnehmer gemeinsam an einem ausgewählten Projekt arbeiten können.

1. Die gleichzeitige Installation von Saros und dem zu testenden Plugin muss möglich sein.
2. Eine Saros-Sitzung muss zwischen mindestens zwei verschiedenen Teilnehmern hergestellt werden können.

Beide genannten Anforderungen sind als kritisch einzustufen, da das erfolgreiche Arbeiten mit Saros bei Nichterfüllung ausgeschlossen ist.

Es muss allerdings darauf hingewiesen werden, dass Anforderung 2 bei genauer Betrachtung nicht vom verwendeten Plugin abhängig ist, da der Sitzungsaufbau direkt über den Eclipse-Kern abgewickelt wird und sich deshalb nicht unterscheidet, egal welches Plugin gerade genutzt wird. Aus Gründen der Vollständigkeit wird diese Anforderung allerdings trotzdem aufgenommen, da sie Grundvoraussetzung für die Erfüllung aller weiteren Anforderungen ist.

5.1.2 Benutzerrollen

Im Verlauf einer Saros-Sitzung kann ein Teilnehmer zwei verschiedene Benutzerrollen einnehmen. Als Driver übernimmt der Teilnehmer den aktiven Part des Programmierens und hat dementsprechend Schreibrechte, sodass er das Projekt modifizieren kann. Die Rolle des Observers hingegen stellt den passiven Part dar. Der Teilnehmer hat hier keine Schreibrechte und verfolgt im Sitzungsverlauf die Aktivitäten des Drivers.

Während der Arbeit an einem Projekt tauschen die Programmierer entsprechend dem

Konzept der Paarprogrammierung nach gewissen Zeitintervallen die Rollen, d.h. der bisherige Observer übernimmt die Rolle des Drivers und umgekehrt. Diesen Prozess unterstützt Saros und modifiziert mit der neuen Rollenverteilung auch die Rechteverteilung der Benutzer, sodass wieder der Driver die Schreib- und der Observer die Leserechte erhält.

Im Hinblick auf mögliche Anforderungen, die sich aus den geschilderten Funktionalitäten ergeben, kann man im Wesentlichen zwei Äquivalenzklassen identifizieren. Im ersten Fall hat der Initiator der Saros-Sitzung, der sogenannte Host, die Rolle des Drivers inne und ein oder mehrere andere Teilnehmer befinden sich in der Rolle des Observers. Im zweiten Fall dagegen befindet sich der Host (eventuell zusammen mit weiteren Teilnehmern) in der Observer-Rolle und ein anderer Teilnehmer befindet sich in der Rolle des Drivers.

Die Betrachtung dieser verschiedenen Äquivalenzklassen ist deshalb notwendig, weil der Host im internen Programmablauf von Saros eine besondere Rolle einnimmt, denn die Synchronisation geht ursprünglich vom Host aus und synchronisiert die Projekte der übrigen Teilnehmer entsprechend – diese Vorgehensweise muss entsprechend angepasst werden, sobald sich der Host nicht mehr in der Rolle des Drivers befindet (wie zu Beginn einer jeden Saros-Sitzung), denn die Modifikationen am geteilten Projekt werden nun von einem anderen Teilnehmer durchgeführt und die Änderungen müssen von dort aus mit den übrigen Teilnehmern synchronisiert werden.

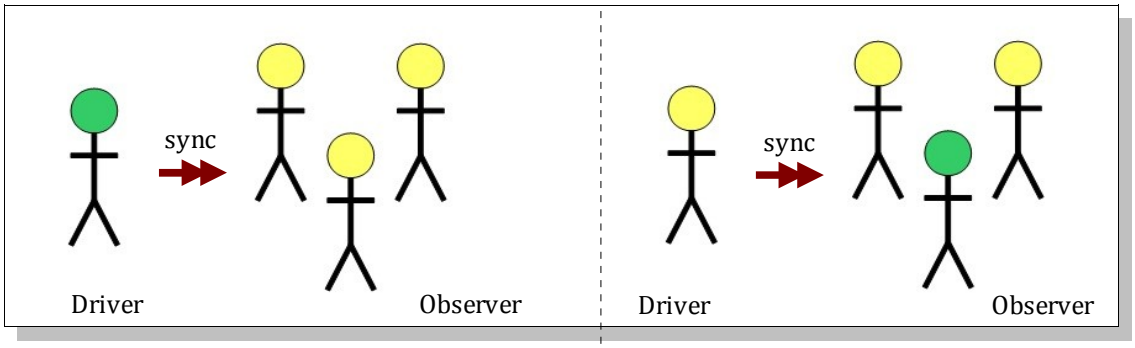

Abbildung 5: Verteilung der Benutzerrollen von Host (grün) und übrigen Teilnehmern (gelb).

Weitere Äquivalenzklassen ergeben sich durch die Einbeziehung des bislang experimentellen Saros-Features „Multi-Driver-Support“, welches gleichzeitig mehrere Driver innerhalb einer Saros-Sitzung erlaubt, sodass mehr als nur ein Teilnehmer Schreibrechte besitzt. Hier kann der Host mit mindestens einem anderen Teilnehmer Driver sein, während alle übrigen Teilnehmer Observer sind, oder der Host besitzt die Observer-Rolle (allein oder mit anderen Teilnehmern), während ein oder mehrere andere Teilnehmer die Driver-Rolle inne haben. Da das Feature bislang experimentell ist, werden die beiden zuletzt geschilderten Äquivalenzklassen im Rahmen dieser Arbeit nicht berücksichtigt und zur Formulierung von Anforderungen herangezogen.

Als Anforderungen ergeben sich also für diesen Abschnitt:

1. Der Driver einer Saros-Sitzung besitzt Schreibrechte, d.h. insbesondere, dass er den Inhalt einer sich im Projekt befindlichen Datei editieren kann.
2. Jeder Observer einer Saros-Sitzung besitzt ausschließlich Leserechte, d.h. insbesondere, dass er den Inhalt einer sich im Projekt befindlichen Datei nicht editieren kann.
3. Findet ein Rollentausch statt, d.h. der bisherige Driver übernimmt die Observer-

Rolle und ein bisheriger Observer wird Driver, bleibt die Rechteverteilung erhalten.

Anforderung 1 muss als kritische Anforderung gesehen werden, da der Besitz von Schreibrechten eine essentielle Voraussetzung für die Programmierarbeit des Drivers darstellt. Die Anforderungen 2 und 3 dagegen sind hier als wesentliche Anforderungen einzustufen, denn bei aktuellem Stand des Saros-Plugins sind verschiedene Fälle bekannt, in denen es einem Observer über bestimmte Vorgehensweisen möglich ist, die Sperrung der Schreibrechte in einigen Punkten zu umgehen (vgl. Seite 22, Abschnitt 5.1.6 Vermeidung und Auflösung von Inkonsistenzen). Befolgen die Benutzer selbstständig die Vorgabe, in der Observer-Rolle nicht aktiv in das Programmieren einzugreifen, kann jedoch problemlos mit Saros gearbeitet werden.

5.1.3 Awareness-Informationen

Um den Teilnehmern einer Saros-Sitzung die gemeinsame Arbeit zu erleichtern, bietet Saros verschiedene Extras, die auf die Aktivitäten der einzelnen Teilnehmer aufmerksam machen. Diese Extras werden unter dem Begriff „Awareness“ zusammengefasst, was direkt übersetzt „Bewusstsein“ bedeutet.

Ein Wesentlicher Bestandteil der Awareness-Informationen ist die Unterscheidung der Teilnehmer durch die Zuweisung verschiedener Farben. Diese sind zunächst, wie in Abbildung 6 dargestellt, in der Shared Session Sicht zu erkennen, werden aber im Sitzungsverlauf konsequent genutzt, um Aktivitäten den einzelnen Teilnehmern zuzuordnen.

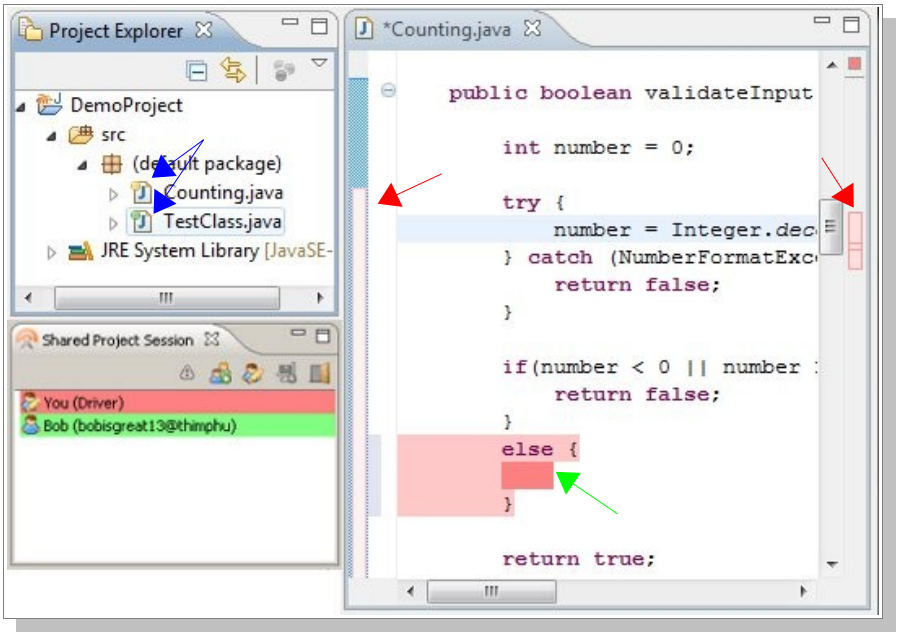

Abbildung 6: Verschiedene Awareness-Informationen während einer Saros-Sitzung.

So werden Text-Markierungen, die ein Teilnehmer in einer Datei vornimmt, bei allen übrigen Sitzungsteilnehmern in der ihm zugewiesen Farbe dargestellt.

Textänderungen, die der Driver vorgenommen hat, werden auf Seiten der übrigen Teilnehmer mit der zum Driver gehörigen Hintergrundfarbe versehen (in Abb. 6 durch einen grünen Pfeil markiert), sodass leicht erkennbar ist, was verändert wurde. Auch die aktuel-

le Cursor-Position des Drivers sowie ggf. die Position seines Scrollbalkens (siehe roter Pfeil in Abb. 6) wird bei den übrigen Sitzungsteilnehmern hervorgehoben.

Die Observer werden außerdem darüber informiert, welche Dateien der Driver aktuell geöffnet hat und welche Datei gerade aktiv ist. Letztere wird mit einem grünen Punkt am Dateisymbol versehen, alle geöffneten Dateien erhalten einen gelben Punkt (in Abb. 6 durch blaue Pfeile gekennzeichnet).

Aus diesem Abschnitt lassen sich also folgende Anforderungen ableiten, die unabhängig davon gelten, ob sich einzelne Teilnehmer im Follow Mode befinden:

1. Hat der Driver Datei A geöffnet und ist diese Datei gerade aktiv, so ist Datei A bei allen anderen Teilnehmern mit einem grünen Punkt markiert. Ist sie geöffnet, aber nicht aktiv, ist sie mit einem gelben Punkt markiert.
2. Markiert Sitzungsteilnehmer A eine Textstelle innerhalb einer Datei, so wird diese Textstelle bei allen übrigen Teilnehmern mit der zu Teilnehmer A gehörigen Farbe hinterlegt.
3. Scrollt der Driver in einer Datei, so ist die Position seines Scrollbalkens bei allen übrigen Sitzungsteilnehmern in dieser Datei durch die zum Driver gehörige Farbe markiert.
4. Fügt der Driver Text zu einer Datei hinzu, so wird dieser neu hinzugefügte Text bei allen übrigen Sitzungsteilnehmern in der zum Driver gehörigen Farbe hinterlegt.
5. Die aktuelle Cursor-Position des Drivers ist bei allen übrigen Sitzungsteilnehmern in der zum Driver gehörigen Farbe markiert.

Die Anforderungen 1 bis 5 stellen durchweg wesentliche Anforderungen dar, denn das Arbeiten mit Saros wäre ohne sämtliche Awareness-Funktionen zwar möglich, doch gerade diese Funktionen zählen zu den besonders nützlichen Eigenschaften des Saros-Plugins, die die Zusammenarbeit während des Programmierens deutlich erleichtern.

5.1.4 Der Follow Mode

Der Follow Mode dient in erster Linie dazu, den Observern die Verfolgung der Aktivitäten anderer Sitzungsteilnehmer, insbesondere die des Drivers, zu erleichtern.

Bei aktiviertem Follow Mode folgt die Ansicht des Observers allen Aktionen des „verfolgten“ Sitzungsteilnehmers. Schließt dieser also eine Datei oder öffnet eine neue, so passiert dies auch auf der Seite des Observers. Scrollt der Driver innerhalb einer Datei, so folgt der Observer und scrollt ebenfalls. In der Konsequenz gleichen sich die Ansichten des Verfolgers und des Verfolgten zu jeder Zeit.

Folgende Anforderungen lassen sich also aus diesem Abschnitt ableiten:

1. Wechselt Teilnehmer A in den Follow Mode zu Teilnehmer B, so springt seine aktuelle Ansicht in die gleiche Position, in der sich die von Teilnehmer B befindet.
2. Befindet sich Teilnehmer A im Follow Mode zu Teilnehmer B...
    1. ...entspricht die aktive Datei von Teilnehmer A immer der von Teilnehmer B.
    2. ...scrollt die Ansicht von Teilnehmer A in der aktuellen Datei immer dann, wenn Teilnehmer B in der Datei scrollt.

3. Der Follow Mode wird verlassen, wenn...
    1. ...Teilnehmer A die aktive Datei verlässt.
    2. ...Teilnehmer A den Follow Mode über den dafür vorgesehenen Befehl beendet.

Die oben genannten Anforderungen sind durchweg als wesentliche Anforderungen einzustufen, da sie den Follow Mode charakterisieren und dieser eine Schlüssel-Funktionalität des Saros-Plugins darstellt. Zwar ist das gemeinsame Arbeiten auch ohne die Verwendung des Follow Modes möglich, doch ein wesentlicher Teil von Saros stünde den Benutzern nicht zur Verfügung.

5.1.5 Dateioperationen und Refactoring

Während der Arbeit an einem Softwareprojekt, ganz gleich in welcher Programmiersprache es umgesetzt wird, werden laufend neue Dateien erstellt, editiert und von Zeit zu Zeit auch wieder gelöscht. Gleiches gilt für Dateiordner. Saros synchronisiert die Projekte der Teilnehmer nicht nur in Bezug auf die Änderung bereits vorhandener Dateien, sondern unterstützt auch das Hinzufügen und Löschen von Dateien und Dateiordnern während einer laufenden Saros-Sitzung.

Ebenso häufig tritt der Fall ein, dass erstellte Ordner oder Dateien umbenannt oder innerhalb der Projektstruktur verschoben werden müssen. Für diesen Fall bietet Eclipse die Möglichkeit zum Refactoring, sodass z.B. Namensänderungen automatisiert durch das ganze Projekt hindurch greifen und der Programmierer nicht mühsam per Hand aktualisieren muss. Saros unterstützt diese Funktionalität und synchronisiert auch über das Refactoring geänderte Projektstrukturen bzw. -Inhalte.

Als Anforderungen lassen sich aus diesem Abschnitt also folgende Punkte ableiten:

1. Nach dem Erstellen sowie Löschen von Dateien (bzw. Ordnern) müssen die Projekt-Kopien aller Sitzungsteilnehmer identisch sein.
2. Nach dem Umbenennen sowie Verschieben von Dateien (bzw. Ordnern) müssen die Projekt-Kopien aller Sitzungsteilnehmer identisch sein.

Die Anforderungen 1 und 2 sind der Kategorie kritisch zuzuordnen, da die Nichterfüllung in beiden Fällen zur Folge hat, dass die Arbeitskopien des gemeinsam bearbeiteten Projektes der Teilnehmer nicht mehr identisch sind.

5.1.6 Vermeidung und Auflösung von Inkonsistenzen

Der wesentliche Teil zur Vermeidung von Inkonsistenzen, d.h. unterschiedlicher Inhalte zwischen den Teilnehmern einer Saros-Sitzung, wird durch die Rechteverteilung erreicht. Wäre diese Rechteverteilung konsequent durchgeführt, so könnten per Definition keine Inkonsistenzen entstehen, da ausschließlich der Driver Änderungen am Projekt vornehmen kann und diese dann an die Observer übertragen werden.

Da Eclipse allerdings eine sehr komplexe Benutzeroberfläche hat und sehr viele Funktionen zur Manipulation des Programmcodes bietet, ist es sehr schwer, dem Observer konsequent alle möglichen Operationen zu untersagen. Zwar werden die gängigsten Möglichkeiten von Saros berücksichtigt, sodass der Observer z.B. in der Tat nichts im Editor schreiben kann – doch über das Kontextmenü aufgerufene Befehle wie beispielsweise „Undo typing“, „Reverse File“ oder oft auch „Copy & Paste“ werden nicht gefiltert und Änderungen, die so

vorgenommen werden, werden auf Seiten des Observers trotzdem wirksam. Da jedoch nicht vorgesehen ist, dass der Observer etwas am Projekt verändert, findet erwartungsgemäß auch keine Synchronisation vom Observer zum Driver statt (nur in umgekehrter Richtung), sodass Driver und Observer nunmehr unterschiedliche Versionen des Projektes besitzen.

Saros erkennt diesen Unterschied jedoch und informiert den Observer mithilfe eines kleinen Warndreiecks in der Shared-Session-Sicht. Nun kann der Observer durch Klick auf dieses Symbol die Inkonsistenz beseitigen und erhält nach erfolgreicher Ausführung die Projekt-Version des Drivers.

Dieser Abschnitt führt also zu folgenden Anforderungen:

1. Gelingt es dem Observer, die Sperrung seiner Schreibrechte zu umgehen und das Projekt zu modifizieren, so muss Saros die entstandene Inkonsistenz als solche erkennen.
2. Hat Saros eine Inkonsistenz erkannt und den Benutzer darauf hingewiesen, so muss die Inkonsistenz durch den Aufruf des entsprechenden Befehls aufgelöst werden können.

Beide Anforderungen sind als wesentlich einzustufen. Das gemeinsame Arbeiten ist bei Nichterfüllung zwar möglich, denn Inkonsistenzen können allein dadurch vermieden werden, dass sich Benutzer in der Observer-Rolle strikt daran halten, keine Modifikationen am Projekt vorzunehmen. Da solch eine Modifikation aber auch unbeabsichtigt erfolgen kann und es Aufgabe von Saros ist, diesen Fall abzufangen, muss eine Nichterfüllung zur Einstufung des Plugins als eingeschränkt kompatibel führen.

5.2 Funktionalitäten der verschiedenen Plugins

Im vorigen Abschnitt wurden die Funktionalitäten betrachtet, die aus Sicht des Saros-Plugins eine erfolgreiche Zusammenarbeit mit einem anderen Eclipse-Sprach-Plugin ausmachen. Dieser Abschnitt nimmt nun die entgegengesetzte Perspektive ein und soll Möglichkeiten aufzeigen, wie bei der Identifizierung von wichtigen Funktionalitäten typischerweise, d.h. unabhängig vom jeweils zu testenden Plugin, vorgegangen werden kann.

Dieser Vorgang stellt deshalb eine große Herausforderung dar, weil zunächst nicht klar ist, welche Funktionalitäten das jeweils untersuchte Plugin überhaupt zur Verfügung stellt und demzufolge auch nicht bekannt ist, an welchen Stellen mögliche Konflikte in Bezug auf die Synchronisation durch Saros drohen.

Die Annahme, dass die verschiedenen Plugins sich mitunter sehr stark in Bezug auf die bereitgestellte Funktionalität unterscheiden, kann nach ihrer umfangreichen Analyse nur teilweise bestätigt werden. So gibt es zwar in der Tat eine Reihe von Unterschieden, die sich aus der jeweils verwendeten Programmiersprache bzw. der eingesetzten Technologie ergeben. Diese Unterschiede betreffen jedoch nur sehr eingeschränkt die Funktionalität des Saros-Plugins, sondern eher über die Bearbeitung von Quelltext hinausgehende Aspekte.

Viel mehr lassen sich eine Reihe von Funktionalitäten identifizieren, die in ähnlicher Form von allen untersuchten Plugins bereitgestellt werden. Die folgenden Abschnitte charakterisieren diese Funktionalitäten und fassen sie zu allgemeineren Bereichen zusammen. Außerdem wird erläutert, welche Funktionalitäten mögliches Konfliktpotenzial bie-

ten und wie genau sie im Zusammenhang mit den Funktionen des Saros-Plugins stehen.

5.2.1 Ausführung des Programmcodes

Ein wesentliches Element einer IDE stellt die Möglichkeit zur Ausführung des geschriebenen Programmcodes direkt aus der IDE heraus dar. Dazu liefern die untersuchten Plugins meist eine sprachspezifische Laufzeitumgebung mit, die oft durch die Installation weiterer Komponenten ergänzt werden muss. Um beispielsweise Python-Code ausführen zu können, ist die Installation einer aktuellen Python-Version notwendig, möchte man Ruby-Code ausführen, muss eine entsprechende Ruby-Installation auf dem System vorhanden sein. Die Ausführung des jeweiligen Programmcodes wird in der Regel über den Eclipse-Befehl „Run...“ gestartet.

Neben der standardmäßigen Ausführung des Programmcodes bieten die untersuchten Plugins meist auch Unterstützung für die Ausführung im sogenannten Debug-Modus, der der dynamischen Fehlersuche dient.

Einige Plugins ergänzen außerdem eine Unterstützung für die Ausführung von automatisierten Testfällen. In Java geschieht dies z.B. durch die Unterstützung von Junit, in PHP analog durch die Integration von PHPUnit.

Die in diesem Abschnitt vorgestellten Funktionalitäten spielen bei der Betrachtung der Kompatibilität mit Saros nur eine untergeordnete Rolle, da die Ausführung des Programmcodes von Saros nicht synchronisiert wird.

Als Anforderung kann daher lediglich gefolgert werden, dass das Ausführen des Programmcodes in den verschiedenen Modi keine Konflikte in einer Saros-Sitzung auslösen darf und diese nach Beendigung der Ausführung problemlos fortgesetzt werden kann.

5.2.2 Werkzeuge zur Quelltext-Analyse

Bei der Untersuchung der verschiedenen Plugins findet man verschiedene Features, die dem Entwickler die Analyse des geschriebenen Quelltextes erleichtern sollen.

Allen voran ist hier das je nach Plugin mehr oder weniger konfigurierbare *Syntax-Highlighting* zu nennen. Dabei werden Sprachelemente bzw. Schlüsselwörter im Quelltext mit einer bestimmten Formatierung versehen, um die Lesbarkeit zu erhöhen. Dieses Feature findet sich ohne Ausnahme in allen untersuchten Plugins, bietet jedoch keine Grundlage für die Entstehung möglicher Synchronisationskonflikte des Saros-Plugins.

Weiterhin bieten die untersuchten Plugins die Möglichkeit, verschiedene zusammengehörige Elemente innerhalb des Quelltextes hervorzuheben, etwa durch die Hinterlegung des jeweiligen Elements mit einer bestimmten Hintergrundfarbe. Der Umfang dieses Features variiert je nach Plugin. Häufig können jedoch gleiche Variablen- oder Funktionsnamen sowie zusammengehörige Klammern durch die Markierung eines Vorkommens hervorgehoben werden. Auch die Ergebnisse einer Suche oder vom Plugin erkannte syntaktische Fehler werden oft farblich markiert. An dieser Stelle ist ein Konflikt mit den Awareness-Informationen von Saros denkbar, da auch hier verschiedene Hintergrundfarben zur Markierung der Nutzeraktivitäten verwendet werden (vgl. Seite 20, Abschnitt 5.1.3 Awareness-Informationen). Solch ein Konflikt ist allerdings als unkritisch zu betrachten, da die Arbeit der Entwickler nicht wesentlich beeinflusst wird und die Wahrscheinlichkeit des Auftretens zudem sehr gering ist. So könnte im schlimmsten Fall evtl. Unklarheit darüber beste-

hen, ob ein Hervorgehobenes Element durch das Plugin oder durch eine Aktivität eines anderen Sitzungsteilnehmers markiert wurde, was durch eine kurze Absprache über die ja ebenfalls bestehende Audioverbindung schnell aufzulösen wäre.

Viele der untersuchten Plugins bieten zudem die Möglichkeit zum sogenannten *Code Folding*, wobei Teile des geschriebenen Quelltextes wie beispielsweise Funktionsdefinitionen verborgen werden können, indem der entsprechende Abschnitt des Quelltextes eingeklappt wird. Diese Funktionalität ist im Hinblick auf die Synchronisation der Benutzeransicht im Follow Mode von Interesse, da es hier eventuell zu Problemen kommen kann, falls ein Teilnehmer Teile des Quelltextes eingeklappt hat, der andere aber nicht. Bei der Bewertung eines solchen Konfliktes muss die Schwere der Auswirkungen berücksichtigt werden: kann der Verfolger die Aktivitäten des Verfolgten im Wesentlichen sehen, so ist der Konflikt als unkritisch einzustufen, auch wenn sich die Ansichten beider Sitzungsteilnehmer möglicherweise nicht vollkommen decken. Befindet sich die Ansicht des Verfolgers aber beispielsweise in einem völlig anderen Teil der bearbeiteten oder in einer anderen Datei oder wird gar der Follow Mode unvorhergesehen beendet, wird die Arbeit der Entwickler stark beeinflusst und der Konflikt ist daher als kritisch anzusehen.

Darüber hinaus wird dem Entwickler häufig die Möglichkeit geboten, sich per Tastenkombination (oft Strg + linke Maustaste) oder über das Kontextmenü die Definition einer bestimmten Variable oder Funktion anzeigen zu lassen. Dabei springt die Ansicht entweder in der bereits geöffneten Datei an die Stelle der jeweiligen Definition oder es wird ggf. auch in eine andere Datei gewechselt, die oft erst hierbei geöffnet wird. Auch dieser Fall ist bei der Synchronisation der Benutzeransicht im Follow Mode interessant, da es passieren könnte, dass sich die Ansicht des Verfolgers nicht automatisch anpasst und der Follow Mode unbeabsichtigt verlassen wird. Solch ein Konflikt ist als kritisch zu betrachten, da der reibungslose Ablauf der Saros-Sitzung hierbei unterbrochen und so die Arbeit der Entwickler massiv beeinflusst würde.

5.2.3 Werkzeuge zur automatischen Quelltext-Modifikation

Um dem Programmierer die Entwicklung zu erleichtern, bieten die untersuchten Plugins verschiedene Funktionen und Werkzeuge an, die automatisch Teile des Quellcodes generieren.

Alle untersuchten Plugins unterstützen die automatische Vervollständigung des geschriebenen Programmcodes. Hierbei stellt das Plugin anhand der bisher vom Programmierer geschriebenen Buchstaben Vorschläge für die Vervollständigung des Codes bereit, angefangen bei simplen Sprachelementen wie beispielsweise Variablen, die im bereits geschriebenen Code vom Programmierer selbst definiert wurden, über verschiedene sprachspezifische Schlüsselwörter bis hin zu komplexeren Sprachelementen wie Schleifen, Verzweigungen oder Blöcken zur Ausnahmebehandlung. Code-Vervollständigung stellt eine allgemein beliebte und viel genutzte Funktionalität dar, die deshalb auch von anderen verbreiteten integrierten Entwicklungsumgebungen unterstützt wird, denn sie kann dem Entwickler Zeit sparen und so seine Effizienz steigern.

Ein weiteres nützliches Feature, das sich bei den meisten der untersuchten Plugins findet, ist die Möglichkeit zur automatischen Formatierung des geschriebenen Quelltextes. Dabei werden vor allem Zeilenumbrüche sowie Leerzeichen angepasst und der Quelltext entsprechend der jeweiligen Einstellung eingerückt. Besonders in größeren Projekten mit mehreren Programmierern trägt dies zur Vereinheitlichung und somit zu einer leichteren

Verständlichkeit und besseren Übersichtlichkeit des Programmcodes bei.

Unabhängig von der verwendeten Programmiersprache besteht neben der einfachen Formulierung von Quelltext auch immer die Möglichkeit, direkt in den Quelltext Kommentare zu integrieren, meist zu Dokumentationszwecken. Alle getesteten Plugins verfügen in diesem Kontext über ein Feature, das Teile des Quelltextes automatisch über das Kontextmenü oder eine Tastenkombination als Kommentar markiert oder eine solche Markierung aufhebt.

Da Saros Änderungen am Quelltext erkennt und synchronisiert, ganz gleich, wodurch sie ausgelöst werden, sollten weder bei der automatischen Code-Vervollständigung noch bei der Code-Formatierung oder der Kommentar-Funktion Konflikte auftreten. Trotzdem sollten diese Funktionen getestet werden, um sicherzustellen, dass keine unerwarteten Verhaltensweisen auftreten.

5.2.4 Identifizierung weiterer Features und deren Einordnung

Zur Einordnung weiterer Funktionalitäten können je nach Bereich die folgenden Anhaltspunkte zur Hilfe genommen werden:

Fall 1: Der Quelltext wird durch die Anwendung des Features nicht modifiziert, d.h. es wird also weder Quelltext hinzugefügt, noch entfernt oder bestehender Quelltext verändert.

Einordnung: Unkritisch. Direkte Konflikte mit Saros sind nicht möglich.

Fall 2: Der Quelltext wird in irgendeiner Form durch das Feature markiert.

Einordnung: Unkritisch. Ein Konflikt mit Saros ist möglich, die Auswirkung allerdings nicht gravierend. Die Arbeit der Entwickler wird kaum beeinträchtigt.

Fall 3: Die Anwendung des Features beinhaltet eine Form der Ausführung des Programmcodes.

Einordnung: Sollte getestet werden. Auch wenn die eigentliche Ausführung des Programmcodes nicht synchronisiert wird, muss das Weiterarbeiten in der Saros-Sitzung nach erfolgter Ausführung problemlos möglich sein.

Fall 4: Der Quelltext wird durch die Anwendung des Features modifiziert.

Einordnung: Muss unbedingt getestet werden. Jegliche Änderung des Quelltextes muss durch Saros synchronisiert werden.

Eine allgemeine Übersicht der je Plugin zur Verfügung gestellten Features kann den jeweiligen Projektseiten entnommen werden. Features, die keinem der aufgeführten Fälle entsprechen, betreffen die Synchronisation durch Saros mit hoher Wahrscheinlichkeit nicht und müssen daher in den Kompatibilitätstests berücksichtigt werden.

6 Das Testen auf Kompatibilität

Die besondere Herausforderung beim Testen der verschiedenen Plugins auf Kompatibilität mit Saros besteht darin, dass das Erstellen von wirklich aussagekräftigen Testfällen eine sehr komplexe Aufgabe darstellt. Im Wesentlichen müssen hier drei Systeme betrachtet werden: Die Eclipse IDE, das Saros-Plugin und das jeweils zu testende Eclipse-(Sprach-)Plugin. Jedes dieser Systeme wird in unterschiedlichen Projekten entwickelt – eine erfolgreiche Interoperabilität hängt also in hohem Maße davon ab, wie gut und gewissenhaft bereitgestellte Schnittstellen benutzt und bedient werden.

6.1 Erstellung von Testfällen

Entsprechend der Unterteilung der Anforderungen werden die Testfälle in einen allgemeinen und einen Plugin-spezifischen Bereich aufgeteilt. Der allgemeine Teil deckt diejenigen Anforderungen ab, die aus den Funktionalitäten des Saros-Plugins abgeleitet wurden. Der Plugin-spezifische Teil dagegen beschreibt die Vorgehensweise bei der Erstellung von Testfällen aus den verschiedenen Anforderungsbereichen, die sich bei der Analyse der Sprach-Plugins ergeben haben und stellt einige Beispiele für konkrete Testfälle vor.

Eine detaillierte Beschreibung aller erstellten Testfälle findet sich in Anhang C.

6.1.1 Allgemeine Testfälle

Unabhängig von den speziellen Eigenschaften und Funktionalitäten eines jeden Plugins gibt es eine Reihe von Arbeitsabläufen, die typischerweise in jeder Saros-Sitzung auftreten. In Abschnitt 5.1 wurden diejenigen Funktionalitäten identifiziert, die aus Sicht von Saros für die reibungslose Zusammenarbeit mit einem anderen Eclipse-Plugin unterstützt werden müssen. Diese Funktionalitäten wurden zu Anforderungen zusammengefasst, die nun als Grundlage zur Erstellung von allgemeinen Testfällen dienen sollen. Als „allgemein“ werden die Testfälle deshalb bezeichnet, weil sie solche Funktionalitäten testen, die unabhängig vom jeweils getesteten Eclipse-Plugin unterstützt werden sollen.

Ziel ist es, jede Anforderung in mindestens einem Testfall zu berücksichtigen, sodass alle geforderten Funktionalitäten durch entsprechende Tests abgedeckt werden. Auf diese Weise kann nach erfolgreicher Durchführung aller Tests sichergestellt werden, dass alle Anforderungen erfüllt werden und das getestete Plugin tatsächlich kompatibel mit Saros ist.

Die Vorgehensweise bei der Ableitung von Testfällen aus den allgemeinen Anforderungen verläuft relativ geradlinig. So werden die geforderten Verhaltensweisen jeweils in Form von konkreten Aktionsschritten in einen Testfall aufgenommen und überprüft. Da die Anforderungen oft nur auf sehr kleine Details abzielen, werden thematisch passende Anforderungen zu einem Testfall zusammengefasst, um ein Mindestmaß an Übersichtlichkeit zu wahren (vgl. Testfall *Awareness-Informationen über den Driver der Sitzung*, Anhang C).

Wichtig ist in diesem Zusammenhang, dass für alle Anforderungen, die zusammengefasst werden sollen, die gleichen Voraussetzungen gelten. Im Allgemeinen ist dies allerdings unproblematisch, da stets die Installation von Eclipse, Saros und dem zu testenden Plugin vorausgesetzt wird, sowie gültige IM-Accounts der jeweiligen Sitzungsteilnehmer, die untereinander bekannt sind[16], und eine bereits initialisierte Saros-Sitzung inklusive einem

16 Alle Teilnehmer müssen sich gegenseitig ihrer Freunde-Liste hinzugefügt haben.

Beispielprojekt, über das alle Teilnehmer verfügen. Ferner ist außerdem eine Internetverbindung notwendig.

Lediglich die Voraussetzungen für das Testen der Installation und des Sitzungsaufbaus unterscheiden sich vom allgemeinen Fall. Hier kann im ersten Fall lediglich eine Eclipse-Installation und das zu testenden Plugin vorausgesetzt werden, alle weiteren Schritte müssen bereits als Teile des Testfalls aufgefasst werden. Als erwartetes Testergebnis kann ein erfolgreich installiertes Saros-Plugin angenommen werden, wobei erfolgreich bedeutet, dass die verschiedenen Saros-Komponenten (Sichten, Menüeintrag, etc.) nun in Eclipse vorhanden sind.

Im zweiten Fall, also beim Test des Sitzungsaufbaus, muss wiederum die Installation vorausgesetzt werden. Da dieser Test wie in Abschnitt 5.1.1 erläutert vor allem aus Vollständigkeitsgründen durchgeführt wird, wird auch die Verbindung der Benutzer mit einem IM-Dienst als ein Schritt mit in diesen Testfall aufgenommen. Genau genommen sollte dieser Vorgang in einem eigenen Testfall abgehandelt werden, da er eigentlich selbst aus einer Vielzahl von Einzelschritten besteht; da aber auch hier kein Zusammenhang mit dem jeweils verwendeten Plugin besteht, wird aus Gründen der Einfachheit darauf verzichtet.

6.1.2 Plugin-spezifische Testfälle

Im Gegensatz zu den sehr eindeutig definierten allgemeinen Anforderungen, aus denen im letzten Abschnitt Testfälle abgeleitet wurden, sind die Anforderungen an die Plugin-spezifischen Funktionalitäten etwas schwerer zu handhaben. In Abschnitt 5.2 Funktionalitäten der verschiedenen Plugins wurden einige Anforderungsbereiche vorgestellt und typische Funktionalitäten beschrieben, die häufig von den untersuchten Plugins unterstützt werden.

Bei der Ableitung von Testfällen aus diesen Anforderungsbereichen soll nun vom Test eines konkreten Features abstrahiert werden. Ziel ist es viel mehr, Testfälle für verschiedene Klassen von Features zu entwerfen; Diese Klassen zeichnen sich dadurch aus, dass ihnen zugeordnete Funktionalitäten sich bzgl. der Art der Auswirkung auf die Zusammenarbeit mit Saros gleichen. Der große Vorteil dieser Methode liegt darin, dass es bei der späteren Ausführung der Testfälle nicht auf die konkrete Umsetzung eines Features durch das jeweils getestete Plugin ankommt, sondern der jeweilige Testfall so generisch gestaltet ist, dass er auf alle Features der entsprechenden Klasse angewendet werden kann.

Als Ausgangspunkt für die Ermittlung der Feature-Klassen, für die jeweils Testfälle erstellt werden sollen, dienen die vorher beschriebenen Anforderungsbereiche.

Aus dem ersten Bereich „Ausführung des Programmcodes“ können drei verschiedene Klassen von Funktionalitäten abgeleitet werden, nämlich zum einen das einfache Ausführen, zum anderen das Debugging und schließlich das automatisierte Testen.

Die Auswertung des Anforderungsbereichs „Werkzeuge zur Quelltext-Analyse“ liefert ebenfalls drei Feature-Klasse.

Die Klasse „Markierung des Quelltextes“ soll alle Features abdecken, die durch eine besondere Markierung Teile des Quelltextes hervorheben, also beispielsweise Syntax-Highlighting oder das Hervorheben zusammengehöriger Elemente.

Unter „Anzeige von Zusatzinformationen“ werden alle Features zusammengefasst, die per Tastenkombination oder über das Kontextmenü aufgerufen werden können und bei An-

wendung das Springen im Projekt zur Folge haben können. Das am häufigsten zu findende Beispiel hierfür ist das Springen zur Definition eines Sprachelementes.

Die Klasse „Quelltext ausblenden“ beinhaltet aktuell nur das Code Folding. Kennzeichnend für diese Klasse ist jedoch, dass sich die aktuelle Ansicht des Benutzers durch die Anwendung des Features verändert, ohne dass jedoch der Quelltext selbst modifiziert wird.

Der Anforderungsbereich „Werkzeuge zur automatischen Quelltext-Modifikation“ beinhaltet die meisten Features und vor allem auch die meisten Unterschiede in der Umsetzung der Features durch die verschiedenen Plugins. Hier kommt die Abstraktion daher am deutlichsten zum Tragen.

Zunächst sind diejenigen Features zusammenzufassen, die durch eine Tastenkombination oder über das Kontextmenü automatisch den bereits geschriebenen Quelltext vervollständigen. Dies betrifft insbesondere die kontextsensitive Quelltextvervollständigung, das automatische Einfügen von Importen oder auch das Hinzufügen von besonderen Sprachelementen wie Konstrukten zur Ausnahmebehandlung.

Davon zu unterscheiden sind Features, die im weitesten Sinn unter „Refactoring“ zusammengefasst werden können. Hier werden z.B. alle Vorkommen eines bestimmten Elementes in einer Datei oder sogar dateiübergreifend auf die selbe Art modifiziert (z.B. Umbenennung einer Variablen) oder ganze Quelltext-Abschnitte als Methoden extrahiert.

Eine relativ eng eingegrenzte Klasse stellt die Quelltext-Formatierung dar, bei der ganze Dateien oder nur Teile des Quelltextes automatisch entsprechend den jeweiligen Voreinstellungen formatiert werden. Hierbei ist hervorzuheben, dass sich lediglich die Struktur des Quelltextes verändert, nicht aber der Inhalt selbst. Auch das Hinzufügen oder Entfernen von Kommentaren fällt in diese Klasse.

Alle Plugin-Features, die potentiell die Interaktion mit Saros beeinträchtigen könnten, sollten einer der vorgestellten Feature-Klassen zugeordnet werden können. Andersherum ausgedrückt haben Features, die sich keiner der Klassen zuordnen lassen, höchstwahrscheinlich keine Auswirkung auf die Projekt-Synchronisation während einer Saros-Sitzung

Eine Übersicht der zu den jeweiligen Klassen erstellten Testfällen ist in Anhang C zu finden.

6.2 Durchführung der Tests

Alle hier beschriebenen Tests werden in der Eclipse IDE, Version 3.5.1 „Galileo“, durchgeführt. Dabei dient im Regelfall eine Eclipse-Instanz als Basis, d.h. hier wird sowohl Saros als auch das zu testende Plugin installiert (siehe Abschnitt 6.2.1 Installation von Saros bzw. 6.2.2 Installation des zu testenden Plugins). Aus dieser Basis heraus wird dann für jeden Teilnehmer eines Tests eine eigene Eclipse-Instanz gestartet.

Zwar ist es auch möglich, die Tests mit verschiedenen eigenständigen Eclipse-Anwendungen nachzuvollziehen, doch um im Anschluss eine detaillierte Auswertung vornehmen zu können, ist es unverzichtbar, die Eclipse Testanwendungen aus einer laufenden Eclipse-Anwendung heraus zu starten, um auf die Fehlerprotokolle zugreifen zu können.

Da das automatische Testen von Saros aus verschiedenen Gründen momentan nur sehr umständlich realisierbar ist, werden die Testfälle hier manuell ausgeführt. Abschnitt 9.1 Automatisierung der Tests beschreibt, wie zukünftig eine automatisierte Ausführung der Testfälle ermöglicht werden könnte.

6.2.1 Installation von Saros

Um die Tests auf Kompatibilität durchführen zu können, muss zunächst eine Version des Saros-Plugins installiert werden. Dabei sind zwei Vorgehensweisen zu unterscheiden:

Generell sollte Saros zu Testzwecken als Projekt aus dem SVN[17] ausgecheckt werden. Da Eclipse in der verwendeten Version noch keine SVN-Unterstützung beinhaltet, muss diese Funktionalität nachgerüstet werden. Hierzu eignet sich das Eclipse-Plugin Subclipse, welches analog zu der im nächsten Abschnitt beschriebenen Vorgehensweise installiert werden kann[18]. Nach der Installation kann Saros als neues Projekt im Workspace aus dem SVN ausgecheckt werden.

Die Installation über die Eclipse-Update-Seite bzw. als Dropin muss dagegen durchgeführt werden, um gewährleisten zu können, dass spätere Benutzer Saros entsprechend der Installationsanleitung einrichten können. Zu diesem Zweck ist darauf zu achten, dass die Installation des zu testenden Plugins (siehe nächster Abschnitt) vor der Installation von Saros erfolgt. Dies entspricht eher der Vorgehensweise späterer Benutzer, denn es kann davon ausgegangen werden, dass diese bereits eine bestimmte Eclipse-Konfiguration verwenden und in ihrem Entwicklungsprozess einsetzen, zu der Saros erst nachträglich als zusätzliche Komponente hinzugefügt wird.

6.2.2 Installation des zu testenden Plugins

Zusätzlich zum Saros-Plugin muss auch das Plugin installiert werden, das auf Kompatibilität mit Saros überprüft werden soll. Die genaue Vorgehensweise bei der Installation sollte dabei der Projektseite des jeweiligen Plugins entnommen werden. Dort finden sich zumeist detaillierte Installationsanleitungen und Hinweise auf Besonderheiten (soweit vorhanden).

Im Wesentlichen ist der Ablauf allerdings immer identisch: In der gestarteten Eclipse-Anwendung wählt man im Menü unter „Help“ den Eintrag „Install new Software...“ aus und trägt durch Klick auf „Add...“ die Adresse der zum Plugin gehörigen Update-Seite[19] ein (vgl. Abb. 7). Im Fenster darunter wählt man dann das gewünschte Plugin aus und klickt sich mithilfe des „Next >“-Buttons durch den folgenden Installationsdialog. Im Laufe des Dialogs wird man die jeweiligen Nutzungsbedingungen bestätigen und zum Abschluss die Eclipse-Anwendung neu starten müssen. Nach dem Neustart ist das zu testende Plugin nun einsatzbereit.

17 Subversion Repository des Saros-Projekts. Erreichbar unter https://svn.sourceforge.net/svnroot/dpp. Informationen zu Subversion unter http://svnbook.red-bean.com/nightly/de/svn-book.html

18 Die entsprechende Update-Seite findet sich unter http://subclipse.tigris.org/update_1.6.x

19 Die jeweils verwendete Update-Seite kann in Anhang A zu jedem verwendeten Plugin nachgelesen werden.

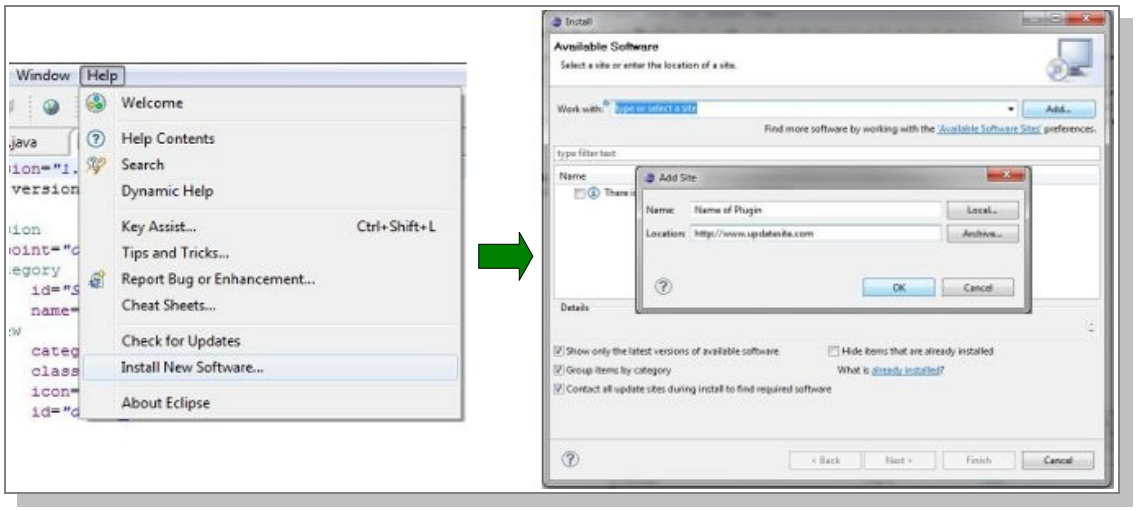

Abbildung 7: Hinzufügen der Update-Seite für ein Plugin

Eine Abweichung von der beschriebenen Vorgehensweise stellt die Installation verschiedener Plugins zur Ergänzung von Aptana Studio dar. Dieses Plugin richtet sich an Entwickler von Internet-Applikation und bietet deshalb zunächst lediglich eine Unterstützung für die im Web allgemein gebräuchlichen Sprachen HTML, Javascript und CSS. Weitere Sprachen werden durch die Installation weiterer Plugins unterstützt, die teilweise auch unabhängig von Aptana Studio in Eclipse verwendet werden können, z.B. Aptana PyDev zur Unterstützung von Python. Hierfür stellt Aptana Studio eine eigene Plugin-Verwaltung zur Verfügung, die über den entsprechenden Link auf der MyAptana-Seite aufgerufen werden kann.[20] Dort kann das gewünschte Plugin ausgewählt werden, indem man es ebenfalls anklickt. Der folgende Installationsdialog entspricht dann dem zuvor vorgestellten und auch hier ist abschließend ein Neustart von Eclipse erforderlich.

Auch die Installation des Adobe® Flex Builder verläuft anders. Die hier verwendete sechswöchige Testversion des Plugins kann erst nach einer Registrierung auf der Adobe Webseite heruntergeladen werden. Es handelt sich hierbei um ein automatisches Installationsprogramm, einen sogenannten *Installer*, dessen Ausführung durch einen Installationsdialog führt.[21] In diesem Dialog muss unter anderem der Pfad zur installierten Eclipse-Version angegeben und den Lizenzbedingungen zugestimmt werden. Alle übrigen Konfigurationsmöglichkeiten sollten die vordefinierten Standardwerte beibehalten. Nach Beendigung des Installers sollte Flex Builder in der während des Installationsdialogs angegebenen Eclipse IDE als Plugin zur Verfügung stehen.

6.2.3 Die Testumgebung

Um eine „saubere“ Arbeitsumgebung zu schaffen, d.h. den unbeabsichtigten Einfluss von Eclipse-Komponenten, die nicht Teil des Tests sind, nach Möglichkeit zu vermeiden, ist es wichtig, für jeden Test eine neue Ausführungskonfiguration („run configuration“) einzurichten. Dazu wählt man per Rechtsklick auf das Saros-Projekt den Befehl „Run As“ und im Anschluss „Run Configurations...“ aus. Nun wählt man als Konfigurationsgrundlage „Eclipse

20 Die MyAptana Seite wird nach der Installation von Aptana Studio standardmäßig nach jedem Start von Eclipse im Editor angezeigt.

21 Die hier beschriebene Vorgehensweise bezieht sich auf die Installation auf einem Microsoft Windows Betriebssystem.

Application“ aus und erstellt durch Klick auf das entsprechende Symbol eine neue Konfiguration.

Grundsätzlich muss für jede Eclipse-Instanz, die in einem Testszenario verwendet wird, ein eigener Workspace eingerichtet werden, d.h. ein Workspace, auf den ausschließlich diese Eclipse-Instanz zugreift und der in den meisten Fällen leer ist. Alternativ kann der Workspace auch ein für den Test notwendiges, genau definiertes Projekt enthalten, wie z.B. bei der Ausführung des Testfalls „Awareness-Informationen über den Driver der Sitzung“.

Jetzt müssen einige Standardwerte angepasst werden, damit die parallele Ausführung mehrerer Eclipse-Instanzen auf einem System reibungslos funktioniert.

Der Parameter `-XX:MaxPermSize` , der die Größe des zur Verfügung stehenden Speichers reguliert, sollte auf mindestens 256 gesetzt werden, da sonst gerade beim Testen von umfangreicheren Plugins wie beispielsweise Aptana Studio Fehler durch Speicherüberläufe auftreten und die Test-Instanzen von Eclipse nicht erfolgreich gestartet werden können. Für mehr Speicher während der Ausführung sorgt der zusätzliche Parameter `-Xmx512m`.

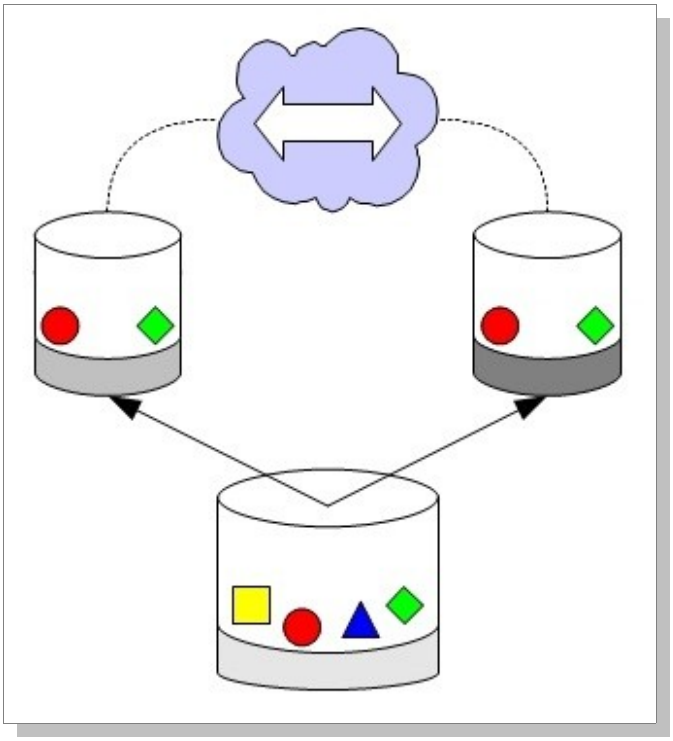

Abbildung 8: Basis-Eclipse und daraus gestartete Instanzen mit jeweils eigenem Workspace (grau) und ausgewählten Plugins (Formen).

Weiterhin muss sichergestellt werden, dass möglicherweise zusätzlich installierte Plugins, die im Rahmen des Tests keine Rolle spielen, in einer Test-Instanz von Eclipse nicht mit geladen werden. Hierzu werden alle Komponenten, die nicht zu Saros oder dem zu testenden Plugin gehören, in der Ausführungskonfiguration deaktiviert. Als Nebeneffekt wirkt sich diese Deaktivierung außerdem positiv auf die Ladezeit beim Start von Eclipse aus.

6.2.4 Ausführung der Testfälle

Damit das jeweilige Testszenario einem realen Arbeitsablauf während der Entwicklung so nahe wie möglich kommt, wird bei der Erstellung von Dateien oder Ordnern stets der entsprechende vom Plugin bereitgestellte Wizard verwendet, soweit ein solcher vorhanden ist.

Diese Vorgehensweise bietet besonders bei der Initialisierung der Tests einen großen Vorteil, denn um den Großteil der Testfälle ausführen zu können, ist es erforderlich, dass der Initiator der Saros-Sitzung bereits ein Projekt in seinem Workspace vorhält, das er mit anderen Benutzern teilen kann. Dieses Projekt muss wenigstens einmal für jedes zu testende Plugin erstellt werden. Ein einheitliches Beispielprojekt ist deshalb nicht sinnvoll, weil sich die Projektstrukturen je nach eingesetzter Programmiersprache bzw. Technologie sehr stark unterscheiden. Verwendet man den Plugin-spezifischen Projekt-Wizard, so wird meist eine typische Projektstruktur angelegt sowie oft auch einige Konfigurations- oder Beispieldateien.

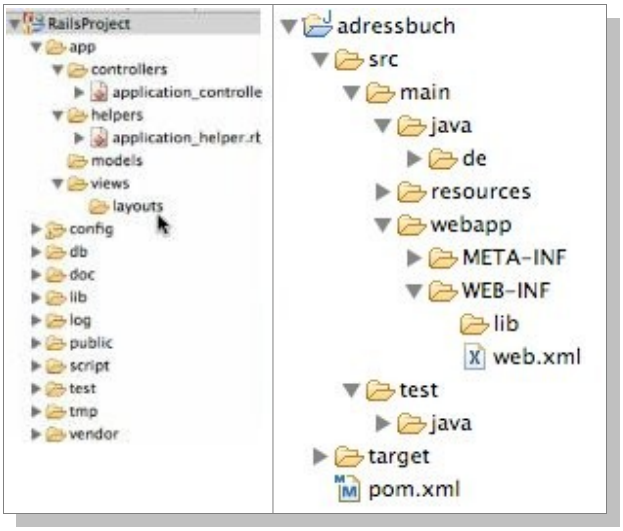

Abbildung 9: Exemplarische Projektstruktur eines Ruby On Rails- (links) bzw. eines Java-Projekts (rechts).

Die in den Testfällen beschriebenen Dateinamen sind durchgängig ohne Dateiendung angegeben, um eine Unabhängigkeit von der jeweils eingesetzten Programmiersprache zu gewährleisten. Im konkreten Testfall werden die Dateinamen nun jeweils um die entsprechende Dateiendung ergänzt, also z.B. .rb für eine Ruby-Datei, .py für eine Python-Datei usw.. Welche Dateiendung im Einzelfall zu wählen ist, muss der jeweiligen Projektseite entnommen werden. Dort finden sich meist Dokumentationen oder Tutorials, die einen kurzen Überblick über die Eigenschaften ermöglichen. Auch die per Wizard generierten Beispielprojekte bieten hier erste Ansatzmöglichkeiten. Schließlich kann auch die jeweilige Plugin-Konfigurationsseite in den Eclipse-Präferenzen („Window“ → „Preferences...“) Hinweise liefern, da dort ersichtlich ist, welche Dateitypen standardmäßig mit welchem Editor verknüpft sind.

Die Testfälle werden nun Schritt für Schritt ausgeführt. Die Reihenfolge spielt dabei im Wesentlichen keine Rolle. Lediglich der Installationstest sowie das Testen der Sitzungsinitialisierung finden sinnvollerweise ganz zu Anfang statt.

Bei der Ausführung der Plugin-spezifischen Testfälle ist zu beachten, dass hier jeweils nicht das Testen eines konkreten Features des Plugins beschrieben ist, sondern eine Klasse von Features, in die mit hoher Wahrscheinlichkeit mehr als ein Feature des betrachteten Plugins hinein fällt. In diesem Fall ist der Testfall für jedes der betroffenen Features auszuführen und jeweils zu vermerken, welches Feature konkret getestet wurde.

7 Auswertung der Testergebnisse

Bei der Auswertung der Testergebnisse konnte erfreulicherweise festgestellt werden, dass die Zusammenarbeit von Saros mit den verschiedenen Eclipse-Erweiterungen im Allgemeinen keine Probleme verursacht.

7.1 Kompatibilität der getesteten Plugins

Beim Test der verschiedenen Plugins zeigte sich, dass die meisten der getesteten Erweiterungen kompatibel mit Saros sind. Lediglich in drei Fällen schlugen die Tests derart fehl, dass die jeweiligen Plugins als nicht kompatibel eingestuft werden mussten (Dev3, EasyEclipse sowie Flex Builder), doch scheint es hier in allen Fällen so zu sein, dass diese Inkompatibilität entweder auf einen Versionskonflikt bzgl. einzelner Eclipse-Komponenten oder auf eine fehlerhafte Implementierung einzelner Eclipse-Schnittstellen durch das Sprach-Plugin zurückzuführen ist. Das Plugin Aptana Studio wurde als einziges als eingeschränkt kompatibel eingestuft, da es hier im Bezug auf die von Saros zur Verfügung gestellten Awareness-Informationen zu fehlerhaftem Verhalten kam.

Einzelheiten zu den aufgetreten Konflikten und zu möglichen Lösungsansätzen finden sich im nächsten Abschnitt.

Eine Übersicht über die getesteten Plugins und deren Kompatibilität zu Saros ist in Tabelle 3 zu sehen.

Getestetes Plugin	Unterstützte Sprache	Kompatibel?	Konflikte / Anmerkungen
Aptana Studio	PHP, Ruby, Python, Javascript	✓	
Aptana Studio	HTML, CSS, ERB	✗	Awareness-Informationen
CDT	C/C++	✓	
Dev3	TypoScript, PHP	✗	Keine Synchronisation
EasyEclipse	C/C++, PHP, Ruby & Rails, Python, Perl	✗	Installation von Saros nicht möglich.
EPIC	Perl	✓	
Flex Builder	Flash	✗	Gleichzeitige Installation mit Saros nicht möglich (Versionskonflikt).
JDT	Java	✓	
PDT	PHP	✓	
PHPeclipse	PHP	✓	
PyDev	Python, Jython, IronPython	✓	

Tabelle 3: Übersicht der untersuchten Eclipse-Erweiterungen.

7.2 Konflikte und Lösungsansätze

Im Wesentlichen traten während des Testens drei verschiedene Arten von Konflikten auf. In zwei Fällen bereitete die gleichzeitige Installation von Saros und dem Testkandidaten Probleme (Flex Builder und EasyEclipse). In einem weiteren Fall war die Installation zwar problemlos möglich und auch das Starten einer Saros-Sitzung bereitete keinerlei Probleme. Leider fand jedoch keinerlei Synchronisation durch Saros statt. Im letzten Fall bezieht sich der Konflikt auf die Darstellung von Awareness-Informationen durch Saros, die nicht nach Definition erfolgte.

So war das detaillierte Testen des Adobe Flex Builder nicht möglich, da bereits die Installation fehlschlug. Abweichend von der definierten Vorgehensweise beim Test von Plugins musste hier die Eclipse Version 3.3 herangezogen werden, da diese zur Verwendung des Plugins von Adobe empfohlen wird. Die zunächst getroffene Annahme, es handle sich hier um eine Mindestanforderung und neuere Versionen von Eclipse würden ebenfalls unterstützt, stellte sich als unzutreffend heraus. Eine Installation von Adobe Flex Builder war in der Eclipse Version 3.4 bzw. 3.5 zumindest unter MS Windows XP sowie MS Windows 7 nicht möglich. Die anschließende Installation von Saros schlug dann ebenfalls fehl, da Eclipse einen Konflikt bezüglich einer bestimmten Komponente feststellte. Betroffen war

hier konkret die Eclipse-Komponente `osgi.bundle/org.eclipse.debug.ui`, die eine generische Benutzerschnittstelle für den Debugger bereitstellt und die Flex Builder offenbar in einer älteren Version benötigt als Saros.

Ein ähnlicher Konflikt trat beim Testen von EasyEclipse auf. Bei keiner der verfügbaren Versionen von EasyEclipse ist es gelungen, das Saros-Plugin erfolgreich zu installieren. In den meisten Fällen verlief die Durchführung der Installationsschritte völlig problemlos, doch nach einem Neustart von Eclipse war keine der verschiedenen Saros-Komponenten in der Benutzeroberfläche auffindbar, obwohl sich das Saros-Paket im Plugin-Verzeichnis von EasyEclipse befand und auch über die Einstellungen aktiviert und deaktiviert werden konnte. Lediglich bei der EasyEclipse-Version für Ruby-Entwicklung schlug bereits die Installation fehl, da Eclipse einen Versionskonflikt mit einer RadRails-Komponente erkannte. Dieser Konflikt trat ebenfalls bei allen weiteren Versuchen auf, ein beliebiges Plugin in dieser EasyEclipse-Version zu installieren, was schließlich einen Fehler in dieser Version vermuten ließ.

Die Tests von Aptana Studio verliefen im Wesentlichen erfolgreich; Lediglich bei der Benutzung der Editoren zur Bearbeitung von HTML-, CSS- und ERB-Inhalten traten Fehler bei der Darstellung von Awareness-Informationen auf. Konkret schien es zunächst so zu sein, dass Text-Markierungen der verschiedenen Sitzungsteilnehmer nicht auf Seiten der anderen Teilnehmer dargestellt würden. Bei genauerer Untersuchung stellte sich jedoch heraus, dass die Markierungen sichtbar wurden, wenn ein Teilnehmer selbst seinen Cursor in eine Zeile bewegte, die ein anderen Teilnehmer markiert hatte. Das Problem besteht also darin, dass die Ansicht der Sitzungsteilnehmer nicht zum Zeitpunkt der Markierung durch einen Teilnehmer aktualisiert wird, sondern erst durch eine weitere Aktion der übrigen Teilnehmer.

Ein weiteres Problem bei der Benutzung der HTML-, CSS- und ERB- sowie auch des Javascript-Editors des Aptana Studio Plugins besteht darin, dass verschiedene Schaltflächen zum einfachen Einfügen diverser Sprachelemente zur Verfügung gestellt werden, die bei der Umsetzung der rollenspezifischen Benutzerrechte nicht berücksichtigt werden. So ist es einem Observer durch einen einfachen Klick auf eine dieser Schaltflächen möglich, die Sperre der Schreibrechte zu umgehen. Saros erkannte zwar durchweg die durch eine solche Vorgehensweise entstandenen Inkonsistenzen, doch sollte an dieser Stelle nach Möglichkeit versucht werden, die Schaltflächen bei Sitzungsteilnehmern in der Observer-Rolle zu sperren, da deren Verwendung mit hoher Wahrscheinlichkeit eher die Regel als die Ausnahme darstellt.

8 Fazit

Um die weitere Verbreitung von Saros zu unterstützen und genauere Erkenntnisse über die Möglichkeiten zu dessen Einsatz bei der Entwicklung mit anderen Programmiersprachen als Java zu gewinnen, sollte die Benutzbarkeit von Saros im Zusammenhang mit anderen Eclipse-Erweiterungen untersucht werden.

Aus den vielen verfügbaren Eclipse-Erweiterungen wurden dazu zunächst anhand verschiedener Auswahlkriterien einige Plugins ausgewählt, die im späteren Verlauf auf ihre Kompatibilität mit Saros hin untersucht wurden.

Hierzu wurden im Rahmen einer umfangreichen Anforderungsanalyse einerseits typische Funktionalitäten von Saros herausgestellt, die unabhängig vom eingesetzten Plugin und somit der verwendeten Programmiersprache unterstützt werden müssen.

Andererseits wurden die ausgesuchten Plugins analysiert und verschiedene Funktionalitäten identifiziert, die in ähnlicher Form bei allen untersuchten Plugins vorhanden sind. Zudem wurden einige Ansätze aufgezeigt, wie bei der Untersuchung eines beliebigen Plugins vorgegangen werden kann, um relevante Features für die Zusammenarbeit mit Saros erkennen zu können.

Die zuvor ermittelten Funktionalitäten wurden jeweils zu Anforderungen zusammengefasst und der entstandene Anforderungskatalog (siehe Anhang B) kann zukünftig als Grundlage zur Ermittlung der Kompatibilität von weiteren Plugins mit Saros herangezogen werden.

Aus den Anforderungen wurden nun konkrete Testfälle abgeleitet und diese entsprechend einer genau definierten Vorgehensweise zum Test der vorher ausgesuchten Plugins verwendet. Zum Testen der Plugin-spezifischen Funktionalitäten wurden dabei Testfälle für Feature-Klassen entworfen, die generisch auf die verschiedenen Features innerhalb einer Klasse angewendet werden können.

Die Tests verliefen im Allgemeinen sehr erfreulich. So kann nun mit Sicherheit festgestellt werden, dass Saros nicht nur die Entwicklung mit der Programmiersprache Java, sondern auch mit C bzw. C++, PHP, Python, Ruby, Perl und Javascript unterstützt.

Eine Analyse der Testergebnisse zeigte, dass Probleme, die während der Tests vereinzelt auftraten, fast immer auf Eigenschaften anderer Plugins zurückzuführen waren.

9 Ausblick

Im Verlauf dieser Arbeit wurde eine Kompatibilitätsliste erarbeitet, die alle mit Saros kompatiblen Eclipse-Erweiterungen enthält. Diese Liste stellt gewissermaßen eine Momentaufnahme dar, da sich sowohl Saros als auch die untersuchten Plugins sowie die Eclipse IDE selbst ständig weiterentwickeln.

Es gilt also zukünftig, diese Entwicklungen im Auge zu behalten und einzelne Tests ggf. zu wiederholen, sollten neue Versionen der einzelnen Komponenten erscheinen. Konkret ist zum aktuellen Zeitpunkt bereits die nächste Version 2.2 des PDT geplant, welche im Juni 2010 veröffentlicht werden soll. Ebenfalls für den Juni 2010 geplant ist die Veröffentlichung der neuen „Helios“ genannten Eclipse Version 3.6 – in diesem Zusammenhang wird außerdem auch die Version 7.0 des CDT verfügbar sein.

Anhand der entwickelten Anforderungen und Testfälle können die neuen Versionen bei Erscheinen schnell und zielsicher auf ihre Kompatibilität mit Saros hin geprüft werden, sodass die größtmögliche Aktualität gewährleistet werden kann.

9.1 Automatisierung der Tests

Die Ausführung der in Abschnitt 6.1 erstellten Testfälle erfolgt derzeit manuell.

Diese manuelle Ausführung stellt einen erheblichen Aufwand dar, der besonders im Hinblick auf die große Anzahl der verfügbaren Eclipse-Erweiterungen und die schnelle Weiterentwicklung des Saros-Plugins dringend reduziert werden sollte.

Die Diplomarbeit [Sz10] von Sandor Szücs befasst sich aktuell jedoch unter anderem mit der Implementierung eines speziell auf die Bedürfnisse von Saros zugeschnittenen Test-Frameworks, das zukünftig die Automatisierung vieler Tests ermöglichen könnte.

Das Test-Framework soll speziell die verschiedenen Probleme in Bezug auf die Netzwerk-Komponente lösen. Doch auch in Bezug auf das Testen von Plugins bietet es einige Interessante Funktionen, wie z.B. die Möglichkeit zur vereinfachten Konfiguration der im Test verwendeten Eclipse-Instanzen. Aktuell sind realistische Tests mit mehreren realen Anwendern auf separaten Systemen extrem aufwändig, da erst das zu testende Plugin installiert und konfiguriert werden muss.

Da sich das Framework noch in der Entwicklung befindet und noch keine genaue Dokumentation vorliegt, kann eine genaue Vorgehensweise zur Automatisierung der Testfälle noch nicht geliefert werden. Das das automatische Testen aber sehr viel Zeit und Aufwand sparen kann und zudem sehr viel genauer möglich ist, als das manuelle Testen, ist eine Umstellung zukünftig unbedingt anzustreben.

9.2 Neue Saros-Features

Da sich Saros in einem kontinuierlichen Entwicklungsprozess befindet, werden zukünftig immer wieder Erweiterungen bestehender Features sowie auch vollkommen neue Features in neue Versionen des Saros-Plugins integriert werden. Sobald dies geschieht, sind alle Aussagen, die über die Kompatibilität von Saros zu einem bestimmten Plugin getroffen wurden, grundsätzlich in Frage zu stellen. Dies bedeutet natürlich nicht, dass alle gewonnenen Testergebnisse nun wertlos wären. Es wird aber eine genaue Analyse notwendig,

um die Stellen zu identifizieren, an denen ein neues Feature innerhalb des Saros-Plugins eingreift und vorherige Verhaltensweisen verändert.

Im Speziellen bedeutet dies, dass für jede Anforderung an die Plugin-Kompatibilität eindeutig festgestellt werden muss, ob ein neues Feature oder eine Veränderung des Saros-Plugins Einfluss auf die geforderte Funktionalität hat. Dieser Fall liegt insbesondere immer dann vor, wenn Teile der Editor-Synchronisations-Logik berührt werden, denn hier unterscheiden sich die verschiedenen Sprach-Plugins, hier stellt sich im Wesentlichen heraus, ob und in welchem Umfang Saros mit einem Plugin zusammenarbeitet.

Im Gegensatz dazu besteht ebenso auch die Möglichkeit, dass eine Veränderung des Saros-Plugins ausschließlich Komponenten berührt, die keinen Einfluss auf das Zusammenarbeiten mit anderen Plugins haben. Als Beispiel sei hier eine Änderung an der Roster Sicht genannt, die vielleicht neue Möglichkeiten zur Organisation der Jabber-Kontakte einführt.

Ist der erste Schritt getan und es wurden diejenigen Anforderungen identifiziert, die von der Änderung berührt werden, müssen die zur jeweiligen Anforderung gehörigen Tests durchgeführt werden, um sicherzustellen, dass alle geforderten Funktionalitäten nach wie vor ohne Einschränkungen funktionieren. Ist das der Fall, können die Änderungen in die Liste der kompatiblen Features für das getestete Plugin aufgenommen werden. Andernfalls muss der Fehler ausfindig gemacht und behoben werden.

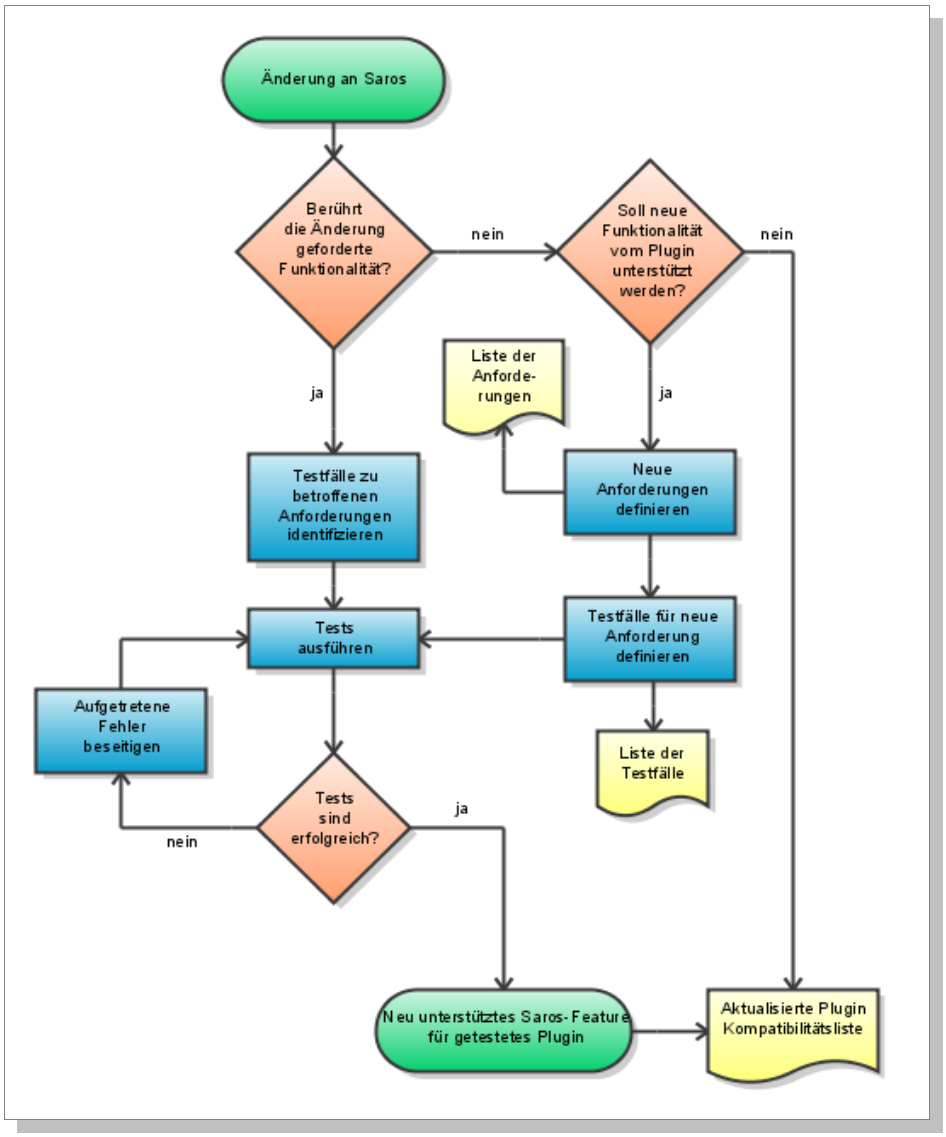

Abbildung 10: Ablaufdiagramm zur Integration neuer Saros-Features.

Nach erfolgreichem Test kann das neue Saros-Feature schließlich in die Kompatibilitätsliste aufgenommen werden.

10 Verzeichnisse

Glossar

Driver Von Saros vergebene Benutzerrolle. Der Driver besitzt Schreibrechte und dient als Ausgangspunkt der Projekt-Synchronisation.

Follow Mode Von Saros bereitgestellter Modus, der das automatische Verfolgen der Aktivitäten anderer Sitzungsteilnehmer ermöglicht.

IDE Abkürzung für englisch „Integrated Development Environment“, also „Integrierte Entwicklungsumgebung“.

Observer Von Saros vergebene Benutzerrolle. Der Observer besitzt keine Schreibrechte und verfolgt die Aktivitäten der übrigen Teilnehmer, insbesondere des Drivers.

Plugin Erweiterung der Eclipse IDE

RIA Abkürzung für englisch „Rich Internet Application“.

Wizard Etwa: „Assistent“; Leitet den Anwender durch eine Folge von Dialogen, um z.B. komplexe Konfigurationsschritte übersichtlicher zu gestalten.

Workbench: Die für den Benutzer sichtbare grafische Oberfläche der Eclipse IDE.

Workspace Eclipse verwaltet hier die vom Benutzer angelegten Projekte.

Literaturverzeichnis

[Doh09]: Lisa Dohrmann, *Erhebung von Benutzerfeedback aus der Nutzung eines Werkzeugs zur verteilten Paarprogrammierung*, 2009, http://www.inf.fu-berlin.de/inst/ag-se/theses/Dohrmann09-saros-benutzerfeedback.pdf

[Lau10]: Stephan Lau, *Verbesserte Präsenz durch Screensharing für ein Werkzeug zur verteilten Paarprogrammierung*, 2010, https://www.inf.fu-berlin.de/w/SE/ThesisDPPXIII

[PP]: Tilman Walther, *Pair Programming*, 2004/2005, http://www.tilman.de/uni/PairProgramming.pdf

[Sta10]: Eike Starkmann, *Verteilte Paarprogrammierung in Open Source Projekten*, 2010, https://www.inf.fu-berlin.de/w/SE/ThesisDPPIX

[STU]: RFC, *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*, 2003, http://tools.ietf.org/html/rfc3489

[Sz10]: Sandor Szücs, *Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung*, 2010, https://www.inf.fu-berlin.de/w/SE/ThesisDPPVI

[TC]: Tiobe Company, *TIOBE Software: General Info*, 2010, http://www.tiobe.com/index.php/content/company/GeneralInfo.html

[TID]: Tiobe Company, *TIOBE Software: Tiobe Index Definition*, 2010, http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm

[TW]: Tilman Walther, *Architektur und Konzepte von Eclipse 3*, 2004/2005, http://www.tilman.de/uni/Eclipse3.pdf

[UML]: Object Management Group, *Introducing to OMG UML*, 2010, http://www.omg.org/gettingstarted/what_is_uml.htm

[XJ]: XMPP Standard Foundation, *About XMPP*, 2010, http://xmpp.org/about/

[XML]: W3C, *Extensible Markup Language*, 2010, http://www.w3.org/XML/

Abbildungsverzeichnis

Anhang A – Verwendete Eclipse Plugins

Java Development Tools (JDT)

Beschreibung:	Diese Komponente ist standardmäßig in Eclipse enthalten, da sie zur Unterstützung der Entwicklung in Java dient (vgl. Abschnitt 2.1 Eclipse, Seite 6).
Projektseite:	http://www.eclipse.org/jdt
Eclipse-Update-Seite:	*- bereits beim Download von Eclipse Classic enthalten -*
Unterstützte Sprache(n), bzw. Editoren:	Java (*.java)
Vorliegende Version:	3.5.1
Features:	Quelltext-Analyse & automatische Modifikation, Debugging, Refactoring, Testing u.v.m.

C/C++ Development Tooling (CDT)

Beschreibung:	Plugin zur Entwicklung mit C/C++ unter Eclipse.
Projektseite:	http://www.eclipse.org/cdt/
Eclipse-Update-Seite:	http://download.eclipse.org/tools/cdt/releases/galileo
Unterstützte Sprache(n), bzw. Editoren:	C/C++ (*.c)
Vorliegende Version:	6.0.1
Features:	Quelltext-Analyse & automatische Modifikation, Debugging, Refactoring, Testing u.v.m.

PHP Development Tools (PDT)

Beschreibung:	Umfangreiche Entwicklungsumgebung für jegliche Art von PHP-Entwicklung.
Projektseite:	http://www.eclipse.org/pdt/
Eclipse-Update-Seite:	http://download.eclipse.org/releases/galileo → Programming Languages → PHP Development Tools
Unterstützte Sprache(n), bzw. Editoren:	PHP (*.php)
Vorliegende Version:	2.1
Features:	Detaillierte Auflistung siehe http://www.zend.com/en/products/studio/comparison

Aptana PyDev

Beschreibung:	Dieses Plugin kann direkt oder in Kombination mit Aptana Studio unter Eclipse verwendet werden. Letztere Variante richtet sich v.a. an Entwickler, die Python im Rahmen von Internet-Anwendungen einsetzen. Zusätzlich ist die Installation von Python (oder Jython bzw. IronPython) auf dem System notwendig.
Projektseite:	http://pydev.org/

Eclipse-Update-Seite:	http://pydev.org/updates
Unterstützte Sprache(n), bzw. Editoren:	Python (*.py), Jython, IronPython (Jython und IronPython sind Implementierungen der Sprache Python in Java bzw. C)
Vorliegende Version:	1.5.4
Features:	Detaillierte Auflistung unter http://pydev.org/manual_adv_features.html

Aptana Studio

Beschreibung:	Aptana Studio ist neben der Version als Eclipse Plugin auch als eigenständige IDE erhältlich und wendet sich vornehmlich an Entwickler von Internet-Anwendungen. Durch das Hinzufügen weiterer Plugins wird eine breite Unterstützung von im Web häufig verwendeten Sprachen bzw. Technologien erreicht.
Projektseite:	http://www.aptana.org/studio
Eclipse-Update-Seite:	http://download.aptana.org/tools/studio/plugin/install/studio
Unterstützte Sprache(n), bzw. Editoren:	HTML (*.html), CSS (*.css), JavaScript (*.js)
Vorliegende Version:	2.0.3
Features:	Quelltext-Vervollständigung, Datenbank-Tools, Fehlersuche, Integration mehrerer Browser, u.v.m.

Aptana RadRails

Beschreibung:	Dieses Plugin stellt eine Entwicklungsumgebung für das Web-Framework Ruby on Rails dar. Zur Verwendung muss zusätzlich Ruby auf dem System installiert werden. Die Unterstützung von Ruby wird durch die Integration der Ruby Development Tools (RDT) gewährleistet.
Projektseite:	http://www.radrails.org/
Eclipse-Update-Seite:	http://download.aptana.com/tools/radrails/plugin/install/radrails-bundle
Unterstützte Sprache(n), bzw. Editoren:	Ruby (*.rb), Embedded Ruby (*.erb)
Vorliegende Version:	2.0.2
Features:	Integriertes Rails-Kommandozeilenwerkzeug, verschiedene Wizards, Debugging, Refactoring, Cloud-Integration, Test-Unterstützung, integrierte WEBrick-Server

Aptana Adobe® AIR™

Beschreibung:	Die Grundvoraussetzungen für die Unterstützung von Adobe® AIR™ werden bereits durch das Aptana Studio Plugin geschaffen. Über die mitgelieferte Plugin-Verwaltung von Aptana kann die Adobe® AIR™ Unterstützung aktiviert werden.

Projektseite:	http://www.aptana.org/air
Eclipse-Update-Seite:	*- wird über die Aptana Studio Update-Seite installiert -*
Unterstützte Sprache(n), bzw. Editoren:	HTML (*.html), JavaScript (*.js), CSS (*.css)
Vorliegende Version:	2.0.2
Features:	Beispielprojekte (Vorschau oder Import als Projekt), integrierte Adobe® AIR™ Laufzeitumgebung, Debugging, Projekt-Wizard, Export-Wizard, integrierter Inhalts-Assistent, einfacher Import von AJAX-Bibliotheken, Erstellung und Verwaltung von Zertifikaten, Hilfe und Online-Dokumentation

Adobe® Flex Builder

Beschreibung:	Adobe® Flex Builder unterliegt einer kommerziellen Lizenz, kann aber für akademische Zwecke kostenlos genutzt werden. Unabhängig davon steht eine sechswöchige Testversion zur Verfügung (Hinweis: Registrierung erforderlich!), die in diesem Fall verwendet wurde. Dieses Plugin richtet sich ähnlich wie Aptana Studio an die Entwickler von RIAs, insbesondere unter Verwendung von Flash im Rahmen der Adobe Flex Technologie.
Projektseite:	http://www.dev3.org
Eclipse-Update-Seite:	*- wird über ein separates Installationsprogramm installiert -*
Unterstützte Sprache(n):	HTML, JavaScript, CSS, AJAX, Flash
Vorliegende Version:	3
Features:	*nicht weiter untersucht, da schon Installation fehlgeschlagen*

Dev3

Beschreibung:	Dieses Plugin dient vornehmlich der Unterstützung bei der Entwicklung in der für das Content Management Typo3 eingesetzten Sprache TypoScript.
Projektseite:	http://www.dev3.org
Eclipse-Update-Seite:	http:www.dev3.org/update
Unterstützte Sprache(n), bzw. Editoren:	TypoScript, PHP
Vorliegende Version:	1.0.3
Features:	Quelltext-Vervollständigung, Syntax-Highlighting, verschiedene Wizards u.a.

EPIC – Eclipse Perl Integration

Beschreibung:	Dieses Plugin unterstützt als eines der wenigen die Entwicklung in Eclipse mit Perl.
Projektseite:	http://www.epic-ide.org/
Eclipse-Update-Seite:	http://e-p-i-c.sf.net/updates/testing
Unterstützte Sprache(n):	Perl

Vorliegende Version:	0.6.x
Features:	Siehe http://www.epic-ide.org/index.php

EasyEclipse

Beschreibung:	EasyEclipse in mehreren verschiedenen Versionen erhältlich, die jeweils auf die Entwicklung mit einer bestimmten Sprache bzw. Technologie abgestimmt sind. Es handelt sich hier nicht um ein Plugin, sondern um ein Paket aus Eclipse-Plattform und zusätzlich einer bestimmten Kombination aus Plugins.
Projektseite:	http://www.easyeclipse.org
Eclipse-Update-Seite:	*- eigenständige Installation notwendig -*
Unterstützte Sprache(n):	Java, C/C++, PHP, Perl, Ruby, Python
Vorliegende Version:	1.2.2.2
Features:	*Je nach Version variierend, allerdings nicht weiter untersucht, da bereits Installation fehl schlug.*

Anhang B – Liste der Kompatibilitäts-Anforderungen

1. Installation und Aufbau einer Saros-Sitzung
   1.1. Die gleichzeitige Installation von Saros und dem zu testenden Plugin muss möglich sein.
   1.2. Eine Saros-Sitzung muss zwischen mindestens zwei verschiedenen Teilnehmern hergestellt werden können.
2. Benutzerrollen
   2.1. Der Driver einer Saros-Sitzung besitzt Schreibrechte, d.h. insbesondere, dass er den Inhalt einer sich im Projekt befindlichen Datei editieren kann.
   2.2. Jeder Observer einer Saros-Sitzung besitzt ausschließlich Leserechte, d.h. insbesondere, dass er den Inhalt einer sich im Projekt befindlichen Datei nicht editieren kann.
   2.3. Findet ein Rollentausch statt, d.h. der bisherige Driver übernimmt die Observer-Rolle und ein bisheriger Observer wird Driver, bleibt die Rechteverteilung erhalten.
3. Awareness-Informationen
   3.1. Hat der Driver Datei A geöffnet und ist diese Datei gerade aktiv, so ist Datei A bei allen anderen Teilnehmern mit einem grünen Punkt markiert. Ist sie geöffnet, aber nicht aktiv, ist sie mit einem gelben Punkt markiert.
   3.2. Markiert Sitzungsteilnehmer A eine Textstelle innerhalb einer Datei, so wird diese Textstelle bei allen übrigen Teilnehmern mit der zu Teilnehmer A gehörigen Farbe hinterlegt.
   3.3. Scrollt der Driver in einer Datei, so ist die Position seines Scrollbalkens bei allen übrigen Sitzungsteilnehmern in dieser Datei durch die zum Driver gehörige Farbe markiert.
   3.4. Fügt der Driver Text zu einer Datei hinzu, so wird dieser neu hinzugefügte Text bei allen übrigen Sitzungsteilnehmern in der zum Driver gehörigen Farbe hinterlegt.
   3.5. Die aktuelle Cursor-Position des Drivers ist bei allen übrigen Sitzungsteilnehmern in der zum Driver gehörigen Farbe markiert.
4. Der Follow Mode
   4.1. Wechselt Teilnehmer A in den Follow Mode zu Teilnehmer B, so springt seine aktuelle Ansicht in die gleiche Position, in der sich die von Teilnehmer B befindet.
   4.2. Befindet sich Teilnehmer A im Follow Mode zu Teilnehmer B...
      4.2.1. ...entspricht die aktive Datei von Teilnehmer A immer der von Teilnehmer B.
      4.2.2. ...scrollt die Ansicht von Teilnehmer A in der aktuellen Datei immer dann, wenn Teilnehmer B in der Datei scrollt.

4.3. Der Follow Mode wird verlassen, wenn...

4.3.1. ...Teilnehmer A die aktive Datei verlässt.

4.3.2. ...Teilnehmer A den Follow Mode über den dafür vorgesehenen Befehl be endet.

5. Dateioperationen und Re-Factoring

5.1. Nach dem Erstellen sowie Löschen von Dateien (bzw. Ordnern) müssen die Projekt-Kopien aller Sitzungsteilnehmer identisch sein.

5.2. Nach dem Umbenennen sowie Verschieben von Dateien (bzw. Ordnern) müssen die Projekt-Kopien aller Sitzungsteilnehmer identisch sein.

6. Vermeidung und Auflösung von Inkonsistenzen

6.1. Gelingt es dem Observer, die Sperrung seiner Schreibrechte zu umgehen und das Projekt zu modifizieren, so muss Saros die entstandene Inkonsistenz als solche erkennen.

6.2. Hat Saros eine Inkonsistenz erkannt und den Benutzer darauf hingewiesen, so muss die Inkonsistenz durch den Aufruf des entsprechenden Befehls aufgelöst werden können.

7. Ausführen von Programmcode

7.1. Das einfache Ausführen des Programmcodes darf keine Auswirkung auf die aktuelle Saros-Sitzung haben. Insbesondere muss das unveränderte Weiterarbeiten nach Beendigung der Ausführung möglich sein.

7.2. Wie 7.1 nur in Bezug auf Debugging.

7.3. Wie 7.1 nur in Bezug auf automatisiertes Testen.

8. Quelltext-Analyse

8.1. Funktionalitäten, die das Markieren von Quelltext bewirken, dürfen keine Konflikte mit den Awareness-Informationen von Saros hervorrufen.

8.2. Die Anzeige von Zusatzinformationen muss im Kontext der Saros-Sitzung passieren. Insbesondere darf der Follow Mode nicht verlassen werden, sollte ein Sprung innerhalb von Dateien oder dateiübergreifend stattfinden.

8.3. Das Ausblenden von Quelltext darf die Arbeit im Follow Mode nicht negativ beeinflussen, d.h. die Textstelle, die der verfolgte Teilnehmer sieht, muss auch für den Verfolger sichtbar sein.

9. Quelltext-Modifikation

9.1. Das automatische Vervollständigen von Quelltext über Tastenkombination, das Kontextmenü o.ä. muss ebenfalls von Saros synchronisiert werden.

9.2. Das Refactoring von Sprachelementen muss durch Saros synchronisiert werden.

9.3. Jegliche automatisierte Formatierung des Quelltextes muss von Saros snychronisiert werden.

Anhang C – Testfälle

Im Folgenden findet sich eine Auflistung aller Testfälle, die zur Überprüfung eines Plugins auf Kompatibilität mit Saros ausgeführt werden müssen. Zu jedem Testfall ist zusätzlich vermerkt, welche Anforderung(en) er jeweils abdeckt.

Um die Lesbarkeit und Verständlichkeit der Testfälle zu erhöhen, werden zur Bezeichnung mehrerer Benutzer statt einer fortlaufenden Nummerierung (Teilnehmer A, Teilnehmer B usw.) die Vornamen Alice, Bob sowie ggf. Carol und Dave verwendet.

Die Testfälle beschreiben im Allgemeinen ein Szenario mit zwei Sitzungsteilnehmern. Zusätze in Klammern weisen ggf. darauf hin, wie sich der Testverlauf in einem Szenario mit mehr als zwei Teilnehmern darstellt.

Allgemeine Testfälle

Installation von Saros

Voraussetzung(en):	Eclipse sowie das zu testende Plugin sind auf dem System installiert*.
Schritte:	1. Saros wird entsprechend der Installationsanleitung über die Eclipse-Update-Seite (als Dropin) installiert.
Erwartete Ergebnisse:	Saros ist zusammen mit dem zu testenden Plugin in der Eclipse IDE auf dem System installiert.
Anforderung(en):	1.1

Initialisierung einer Saros-Sitzung

Voraussetzung(en):	Es werden zwei (drei oder vier) gültige IM-Accounts, d.h. Benutzername sowie das zugehörige Passwort, benötigt*. Alice verfügt über ein Projekt, das mit Bob geteilt werden kann*.
Schritte:	1. Alice und Bob tragen jeweils ihre Account-Informationen in den Saros-Präferenzen ein. 2. Alice und Bob verbinden sich jeweils mit ihren zuvor angelegten Accounts und fügen sich gegenseitig zu ihren Kontaktlisten hinzu. 3. Alice wählt per Rechtsklick auf das Beispielprojekt im Kontextmenü den Eintrag „Share Project...“ aus und lädt Bob zu einer Saros-Sitzung ein.

* Diese Voraussetzung gilt ebenfalls für alle folgenden Testfälle, auch wenn sie dort nicht explizit angegeben ist.

4. Bob akzeptiert diese Einladung und wählt im folgenden Dialog die Option, das geteilte Projekt als neues Projekt anzulegen, aus.

Erwartete Ergebnisse: Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung und verfügen über identische Kopien des Beispiel projekts.

Anforderung(en): 1.2

Benutzerrollen

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer..

Schritte:

1. Alice und Bob öffnen die Datei namens „Testfile“.
2. Alice versucht etwas in die Datei „Testfile“ zu schreiben.
3. Alice versucht etwas aus der Datei „Testfile“ zu löschen.
4. Bob versucht etwas in die Datei „Testfile“ zu schreiben.
5. Bob versucht etwas aus der Datei „Testfile“ zu löschen.
6. Alice und Bob wechseln die Benutzerrollen.
7. Die Schritte 2. bis 5. werden erneut durchgeführt.

Erwartete Ergebnisse:

1. Die Datei „Testfile“ ist bei beiden Benutzern geöffnet.
2. Der Versuch ist erfolgreich und Bob kann sehen, was Alice geschrieben hat.
3. Der Versuch ist erfolgreich und Bob sieht, dass Alice etwas gelöscht hat.
4. Der Versuch ist nicht erfolgreich. Der Inhalt der Datei ändern sich nicht.
5. Der Versuch ist nicht erfolgreich. Der Inhalt der Datei ändern sich nicht.
6. Alice ist nun Observer, Bob ist Driver.
7. Analog der Ergebnisse 2. bis 5. mit jeweils vertauschten Benutzernamen.

Anforderung(en): 2.1, 2.2 und 2.3

Awareness-Informationen über den Driver der Sitzung

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice öffnet die Datei namens „Testfile“.
2. Alice öffnet die Datei namens „TestfileTwo“.
3. Alice wechselt zurück in die Datei „Testfile“.
4. Alice scrollt in der Datei „Testfile“ bis zum Ende der Datei.
5. Alice schließt die Datei „Testfile“.
6. Alice schließt die Datei „TestfileTwo“.

Erwartete Ergebnisse:

1. Bei Bob (allen Observern) ist die Datei „Testfile“ mit einem grünen Punkt markiert.
2. Bei Bob (allen Observern) ist die Datei „TestfileTwo“ mit einem grünen Punkt und die Datei „Testfile“ mit einem gelben Punkt markiert.
3. Bei Bob (allen Observern) ist die Datei „Testfile“ mit einem grünen Punkt und die Datei „TestfileTwo“ mit einem gelben Punkt markiert.
4. Bei Bob (allen Observern) scrollt die Datei mit. Außerdem ist rechts ein Balken erkennbar, der die Position des Scrollbalkens von Bob anzeigt.
5. Bei Bob (allen Observern) ist die Datei „TestfileTwo“ mit einem grünen Punkt markiert.
6. Bei Bob (allen Observern) ist jede Markierung aufgehoben.

Anforderung(en): 3.1, 3.3

Awareness-Informationen beim Bearbeiten von Datei-Inhalten

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice und Bob öffnen jeweils die Datei namens „Testfile“.
2. Alice markiert eine Textstelle in dieser Datei.
3. Bob markiert eine Textstelle in dieser Datei..
4. Alice fügt 3 Zeilen Text zu der Datei hinzu.

Erwartete Ergebnisse:

1. Die Datei „Testfile“ ist bei beiden Teilnehmern geöffnet.
2. Bei Bob (allen übrigen Teilnehmern) ist die von Alice markierte Textstelle mit der zu Alice gehörigen Farbe

hinterlegt.

3. Bei Alice (allen übrigen Teilnehmern) ist die von Bob markierte Textstelle mit der zu Bob gehörigen Farbe hinterlegt.
4. Bei Bob (allen übrigen Teilnehmern) erscheint der von Alice geschriebene Text mit der zu Alice gehörigen Farbe hinterlegt. Außerdem ist am Ende der Zeile die aktuelle Cursor-Position von Alice erkennbar.

Anforderung(en): 3.2, 3.4, 3.5

Follow Mode

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice öffnet die Datei namens „Testfile“.
2. Bob wechselt in den Follow Mode zu Alice.
3. Alice öffnet die Datei namens „TestfileTwo“.
4. Alice wechselt zurück in die Datei „Testfile“.
5. Alice scrollt ans Ende der Datei „Testfile“.
6. Bob verlässt den Follow Mode über den dafür vorgesehenen Befehl.
7. Alice wechselt in die Datei „TestfileTwo“.
8. Bob wechselt wieder in den Follow Mode zu Alice.
9. Bob öffnet die Datei „AnotherFile“.

Erwartete Ergebnisse:

1. Die Datei „Testfile“ ist bei Alice geöffnet.
2. Die Datei „Testfile“ öffnet sich auch bei Bob.
3. Die Datei „TestfileTwo“ öffnet sich auch bei Bob und seine Ansicht wechselt zu dieser Datei.
4. Auch Bob wechselt zurück zur Datei „Testfile“.
5. Auch Bob scrollt ans Ende dieser Datei.
6. Der Follow Mode ist beendet.
7. Bob wechselt nicht in die Datei „TestfileTwo“.
8. Bob wechselt in die Datei „TestfileTwo“.
9. Der Follow Mode ist beendet.

Anforderung(en): 4.1, 4.2, 4.3

Erstellen, Editieren und Löschen von Dateien und Ordnern

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice erstellt einen neuen Dateiordner namens „Testfolder“.
2. Alice erstellt im Ordner „Testfolder“ eine neue Datei namens „Testfile“.
3. Alice editiert die Datei „Testfile“ und speichert sie.
4. Alice erstellt eine neue Datei namens „AnotherFile“.
5. Alice löscht die Datei „Testfile“.
6. Alice löscht den Ordner „Testfolder“.

Erwartete Ergebnisse: Nach jedem einzelnen Schritt gleichen sich die Projekte von Alice und Bob in Bezug auf die Projektstruktur und die Da teiinhalte exakt (beide Projekte sind identisch).

Anforderung(en): 5.1

Umbenennen und Verschieben von Dateien und Ordnern

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice benennt die Datei „Testfile“ in „RenamedFile“ um.
2. Alice benennt den Ordner „Testfolder“ in „RenamedFolder“ um.
3. Alice verschiebt die Datei „RenamedFile“ in den Ordner „AnotherFolder“.
4. Alice verschiebt den Ordner „AnotherFolder“ in den Ordner „RenamedFolder“.

Erwartete Ergebnisse: Nach jedem einzelnen Schritt gleichen sich die Projekte von Alice und Bob in Bezug auf die Projektstruktur und die Da teiinhalte exakt (beide Projekte sind identisch).

Anforderung(en): 5.2

Inkonsistenzen

Voraussetzung(en): Alice und Bob befinden sich in einer gemeinsamen Saros-Sitzung. Alice ist Driver, Bob ist Observer.

Schritte:

1. Alice und Bob öffnen die Datei „Testfile“.

2. Bob verursacht durch das Einfügen von Text per Copy & Paste o.ä. eine Inkonsistenz.
3. Bob beseitigt die Inkonsistenz durch Verwendung des von Saros bereitgestellten Befehls.

Erwartete Ergebnisse:

1. Die Datei ist bei beiden Teilnehmern geöffnet.
2. Saros erkennt die Inkonsistenz und benachrichtigt Bob. Das Warndreieck in der Shared Session Sicht wird aktiviert.
3. Die Inkonsistenz ist beseitigt. Alice und Bob verfügen über identische Kopien der Datei „Testfile“.

Anforderung(en): 6.1, 6.2

Plugin-spezifische Testfälle

Die hier aufgelisteten Testfälle abstrahieren durchgehend vom konkret zu testenden Feature. Mit hoher Wahrscheinlichkeit gibt es je untersuchtem Plugin mehrere Features, die mit ein und demselben Testfall getestet werden können. Bei der Beschreibung der Testfälle steht daher weniger die Art und Weise der konkreten Anwendung eines Features, sondern viel mehr das erwartete Ergebnis im Vordergrund.

Ausführen von Programmcode

Vorgehensweise: Während einer laufenden Saros-Sitzung wird der Programmcode auf die vorgesehene Weise ausgeführt.

Erwartete Ergebnisse: Der Zustand der Saros-Sitzung vor und nach der Ausführung ist identisch. D.h. sowohl die Verteilung der Benutzerrollen, bestehende Follow Modi und Kopien des geteilten Projektes sind bei allen Sitzungsteilnehmern unverändert.

Anforderung(en): 7.1

Debugging von Programmcode

Vorgehensweise: Während einer laufenden Saros-Sitzung per jeweiligem Debugging-Feature eine dynamische Fehlersuche durchgeführt.

Erwartete Ergebnisse: Der Zustand der Saros-Sitzung vor und nach der Ausführung ist identisch. D.h. sowohl die Verteilung der Benutzerrollen, bestehende Follow Modi und Kopien des geteilten Projektes sind bei allen Sitzungsteilnehmern unverändert.

Anforderung(en): 7.2

Markierung von Quelltext

Vorgehensweise: Das jeweilige Feature wird während einer laufenden Saros-Sitzung auf den erstellten Quelltext angewendet bzw. deren automatische

Ausführung durch das getestete Plugin beobachtet.

Erwartete Ergebnisse: Vom Plugin verursachte Markierungen dürfen keine Konflikte mit den von Saros verursachten Markierungen auslösen.

Anforderung(en): 8.1

Anzeige von Zusatzinformationen

Vorgehensweise: Das jeweilige Feature wird während einer laufenden Saros-Sitzung auf den erstellten Quelltext angewendet. Ein Benutzer in der Observer-Rolle muss dabei dem Driver im Follow Mode folgen.

Erwartete Ergebnisse: Die Funktionalität des Follow Modes darf durch die Anwendung des Features nicht beeinflusst werden, d.h. die Ansichten des Drivers und des ihm folgenden Benutzers müssen sich weiterhin zu jeder Zeit gleichen.

Anforderung(en): 8.2

Ausblenden von Quelltext

Vorgehensweise: Während einer laufenden Saros-Sitzung wird durch Anwendung des jeweiligen Features ein Teil des Quelltextes ausgeblendet. Ein Benutzer in der Observer-Rolle muss dabei dem Driver im Follow Mode folgen.

Erwartete Ergebnisse: Die Funktionalität des Follow Modes darf durch die Anwendung des Features nicht beeinflusst werden, d.h. hier, dass der Verfolger die Aktivitäten des Verfolgten – trotz möglicher Unterschiede im Detail der Ansicht – mitverfolgen kann.

Anforderung(en): 8.3

Automatische Quelltext-Vervollständigung

Vorgehensweise: Das jeweilige Feature zur Vervollständigung des Quelltextes wird während einer laufenden Saros-Sitzung auf den erstellten Quelltext angewendet.

Erwartete Ergebnisse: Jede durch das Feature verursachte Änderung des Quelltextes wird ebenfalls von Saros synchronisiert und ist bei allen Sitzungsteilnehmern sichtbar.

Anforderung(en): 9.1

Refactoring

Vorgehensweise: Das jeweilige Feature zum Refactoring des Quelltextes wird während einer laufenden Saros-Sitzung auf den erstellten Quelltext angewendet.

Erwartete Ergebnisse: Jede durch das Feature verursachte Änderung des Quelltextes wird

ebenfalls von Saros synchronisiert und ist bei allen Sitzungsteilnehmern sichtbar.

Anforderung(en): 9.2

Automatische Quelltext-Formatierung

Vorgehensweise: Das jeweilige Feature zur automatischen Formatierung des Quell textes wird während einer laufenden Saros-Sitzung auf den erstellten Quelltext angewendet.

Erwartete Ergebnisse: Der Zustand der Saros-Sitzung vor und nach der Ausführung ist identisch. D.h. sowohl die Verteilung der Benutzerrollen, bestehende Follow Modi und Kopien des geteilten Projektes sind bei allen Sitzungsteilnehmern unverändert.

Anforderung(en): 7.1