Bachelor-Arbeit

am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Implementierung eines Mechanismus‘ zur Übertragung von Informationen der Eclipse-Konsole in Saros

Bernd Bieber
Matrikelnummer: 3714025
bieber@zedat.fu-berlin.de

Betreuer: Franz Zieris, M.Sc.
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Elfriede Fehr

Berlin, 12.06.2014

Zusammenfassung

Diese Arbeit befasst sich mit dem Teilen der Konsolenansicht der Eclipse Entwicklungsumgebung als Erweiterung für Saros. Saros ist ein Plug-In für Eclipse und ermöglicht es Entwicklern, verteilte Paarprogrammierung zu betreiben. Es werden anfangs die Grundlagen erläutert, die zum weiteren Verständnis notwendig sind, gefolgt von einem Einblick in die Planungsphase, in der unter anderem der Einsatz der benutzten Methoden und Werkzeuge beschrieben wird. Im Anschluss wird die Machbarkeit anhand eines Prototypen aufgezeigt. Es folgt der Entwurf einer Anforderung mithilfe von Anwendungsfällen. Diese wurden neben der Entwicklung kontinuierlich erweitert. Zusätzlich sind sie auch als Anleitung zur Benutzung der fertigen Erweiterung verwendbar. Es wird nicht genau auf die Implementierung eingegangen, vielmehr auf die Entscheidungen, aus denen sie resultierte. Abschließend wird noch ein Ausblick auf optionale Erweiterungen der Implementierung gegeben.

Eidesstattliche Erklärung

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

Berlin, 12.06.2014

_____

Bernd Bieber

Inhalt

1. Kapitel: Einleitung

Die Arbeitsgruppe Softwareentwicklung der Freien Universität Berlin konzentriert sich auf die Weiterentwicklung des Open Source Projektes „Saros“. Saros ermöglicht die verteilte Paarprogrammierung in der Eclipse-Entwicklungsumgebung. Dabei geht es oftmals um Sitzungen mit zwei oder mehreren Teilnehmern, die gleichzeitig Änderungen an einem oder mehreren Projekten vornehmen. Diese Änderungen werden synchronisiert, so dass die bearbeiteten Projekte jederzeit identisch sind. Die Ausgaben in der Konsole werden jedoch nur lokal angezeigt und die Informationen der Konsole nicht mit den anderen Entwicklern geteilt. Über die Feedback-Möglichkeit von Saros wurde seitens mehrerer Nutzer der Wunsch geäußert, die Möglichkeit der Konsolen-Teilung einzurichten.

Bevor ich mit der Umsetzung anfangen konnte, erfolgte eine Einarbeitungs- und Planungsphase. Diese beinhaltet das Prüfen der Anwendbarkeit von Methoden und Techniken, wie z. B. Testgetriebene Entwicklung und Timeboxing, sowie ein Kennenlernen der Saros-Architektur und des Codes. Außerdem behandle ich kurz, welche Kenntnisse notwendig sind, um zum Saros-Projekt etwas beitragen zu können und welche Werkzeuge beherrscht werden müssen.

Das Ziel dieser Arbeit ist es, die Möglichkeit in Saros zu schaffen, dass die Konsole in Eclipse zwischen den an der Sitzung beteiligten Entwicklern geteilt werden kann. Dafür habe ich zuerst anhand eines Prototypen die Machbarkeit gezeigt. Dieser soll den aktuellen Inhalt einer Konsole auslesen, diese Informationen transportieren und wieder ausgeben können. Mithilfe von Anwendungsfällen soll diese Anforderung iterativ erweitert und verbessert werden. Die konkrete Implementierung soll kontinuierlich in den bestehenden Saros-Code integriert werden. Abschließend soll das Saros-Plug-In um dieses Feature erweitert werden.

2. Kapitel: Grundlagen

Zu Beginn dieser Arbeit erläutere ich einige Grundbegriffe, die für das Verständnis der nachfolgenden Bachelorthesis hilfreich bzw. nötig sind. Dabei gehe ich davon aus, dass Vorkenntnisse der allgemein üblichen Begriffe und Vorgehensweisen eines Software Entwicklers bekannt sind.

2.1. Paarprogrammierung

In der Softwareentwicklung gibt es verschiedene Techniken zur Umsetzung von Softwareprojekten, eine davon ist die Paarprogrammierung. Bei dieser Technik arbeiten zwei Entwickler zeitgleich am gleichen Problem und am selben Computer. Über das Vorgehen sprechen sich beide Entwickler fortwährend ab, so dass Probleme frühzeitig auffallen und behoben werden können. Das bezieht sich sowohl auf Probleme bzw. Fehler im Design, im Code und auch im Konzept. Durch diese beständige Durchsicht und Prüfung des Codes – auch *Permanent Review* genannt – kann die Qualität der Software deutlich gesteigert werden, da Fehler schneller entdeckt und korrigiert werden können. Gleichzeitig wird auch die Geschwindigkeit erhöht, da die Codeerstellung und ihre Überprüfung zeitgleich stattfindet, anstatt nacheinander. [Vgl. 8]

Verteilte Paarprogrammierung

Bei der Paarprogrammierung sitzen beide Programmierer an ein und demselben Computer. Befinden sich die Programmierer räumlich getrennt, so können sie auf die Technik der verteilten Paarprogrammierung bzw. *„Distributed Pair Programming“* zurückgreifen. Dafür benötigen beide Programmierer jedoch die Möglichkeit, an demselben Quelltext zu arbeiten und sich permanent miteinander abzusprechen. [Vgl. 8]

2.2. Eclipse

Eclipse ist eine integrierte Entwicklungsumgebung (abgekürzt als IDE vom englischen *Integrated Development Environment*), in dem Programmiersprachen wie beispielsweise Java und C++ genutzt werden können um Software zu entwickeln. Darüber hinaus bietet es eine vollständige Rahmenstruktur, mithilfe derer Software entwickelt wird. Eclipse wurde ursprünglich von IBM erschaffen, ist aber im Jahr 2004 in ein Open Source-Unternehmen namens Eclipse Foundation übergegangen. In Eclipse sind bereits verschiedene Plug-Ins vorhanden, jeder Entwickler hat aber auch die Möglichkeit, diese anzupassen und eigene Plug-Ins zu entwickeln, um Eclipse für andere Anforderungen zu erweitern. [Vgl. 2, S. 1137 und 3, S. 9]

2.3. Saros

Saros ist eine Erweiterung (Plug-In) für Eclipse. Es bietet seinen Nutzern die Möglichkeit, ein oder mehrere Projekte in einer Sitzung zu teilen und unterstützt auf diese Weise die verteilte Paarprogrammierung, in dem sich an beiden Computern jeweils durch ständige Synchronisation jederzeit identische Versionen der zu bearbeitenden Projekte befinden. Jeder Programmierer arbeitet an der Datei auf seinem Computer, wo die Datei auch gespeichert wird. Saros synchronisiert die Dateien beider Programmierer permanent, indem jede einzelne Änderung zum jeweils anderen Programmierer übertragen wird. [Vgl. 9]

2.4. Eclipse Konsole

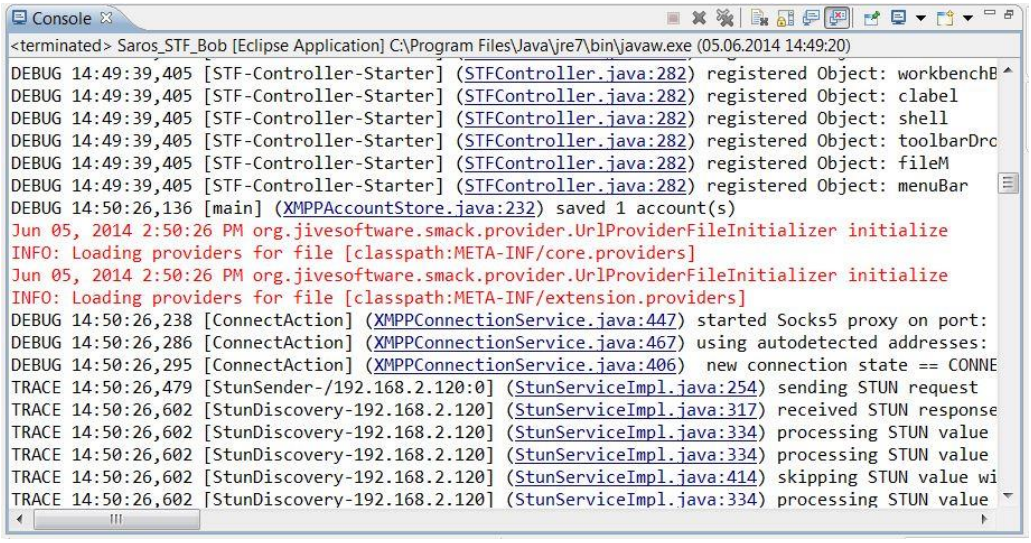

Abbildung 1: Eclipse Konsole

Eclipse bietet seinen Nutzern die Möglichkeit eine Konsolenansicht anzuzeigen. In dieser Konsolenansicht, oder *console view*, können unter anderem Ausgaben, wie z. B. die Standardausgabe von Java oder spezielle Logger[1], von ausgeführten Programmen dargestellt werden. Es ist auch möglich, sich mehrere Konsolenansichten anzeigen zu lassen. In jeder Ansicht lässt sich nun genau eine Ausgabe (aus der systemweiten Sammlung aller Ausgaben) betrachten. Auch das betrachten derselben Ausgabe in verschiedenen Ansichten ist durchaus möglich.

Diese Ausgaben können nicht nur als einfacher Text dargestellt werden, sondern verfügen auch über Hyperlinks, mit denen direkt zu der entsprechenden Datei bzw. Stelle im Editor gesprungen werden kann. Es besteht auch die Möglichkeit, dass der Text eingefärbt wird, um ihn schneller lokalisieren zu können. Darüber hinaus kann die Konsolenansicht auch mittels Tastatureingabe zum Einlesen einer Standardeingabe genutzt werden. [Vgl. 3, S. 46 und S.133]

[1] Ein Logger ist ein Teilstück einer Software und wird zur Protokollierung eingesetzt, Saros z. B. nutzt die Log4J Bibliothek.

Da es möglich ist, mehrere Programme gleichzeitig zu starten, gibt es für jedes Programm eine eigene Konsolenausgabe. Diese sind über ein DropDown-Menü umschaltbar, so dass der Nutzer jederzeit die Möglichkeit hat, sich die entsprechende Ausgabe anzeigen zu lassen. Für die Zuordnung der Ausgabe zum entsprechenden Programm dient der eindeutige Name der Konsolenausgabe, z. B.:

„<terminated> Saros_STF_Bob [Eclipse Application] C:\Program Files\Java\jre7\bin\javaw.exe (05.06.2014 14:49:20)“

Dieser Name enthält zusätzlich auch noch Informationen darüber, ob das Programm noch läuft oder bereits beendet wurde, sowie die Art der Ausführung, der ausführende Kompilierer und den Zeitstempel des Ausführungsbeginns.

Neben der Ausführung von Programmen besteht auch die Möglichkeit, eine Konsolenansicht für eigene Eclipse-Plug-Ins zu erstellen und für beliebige Inhalte zu nutzen, wie z. B. SVN[2], OSGi[3] und ANT-Builds[4].

[2] Für mehr Informationen: http://subversion.apache.org/
[3] Für mehr Informationen: http://www.osgi.org/Main/HomePage
[4] Für mehr Informationen: http://ant.apache.org/

3. Kapitel: Vorgehen

3.1. Methoden und Techniken

Bei der Überlegung, wie das Teilen der Konsole in Saros implementiert werden soll, muss ich zuerst die Entscheidung fällen, mithilfe welcher Methoden und Techniken ich entwickeln möchte. Dabei kamen diverse in Betracht, die nachstehend kurz erläutert werden:

3.1.1. Testgetriebene Entwicklung

Das augenscheinlichste Merkmal der testgetriebenen Entwicklung ist, dass ein Test zur Codeprüfung angefertigt wird und erst im Anschluss daran der Programmcode geschrieben wird, der die Anforderungen dieses Tests erfüllen kann. Daraus resultierend ist der Name „Test-First“ oder „Test-Driven Development“ (TDD), entstanden. [Vgl. 2, S.662]

Kurz gefasst lässt sich eine Softwareentwicklung unter Nutzung von TDD in drei, immer wiederkehrende Schritte einteilen:

1. Erstellung eines Tests, der scheitern wird.
2. Der Programmcode muss geschrieben werden, so dass der Test aus 1) gelingt.
3. Überprüfen des Programmcodes aus 2) ohne die Funktionalität zu ändern, aber den Code sauberer zu schreiben um eine einfachere und bessere Wartung und Lesbarkeit zu gewährleisten. [Vgl. 6, S.36 ff.]

Der zu programmierende Code wird also in viele kleine Teile unterteilt, die einzeln getestet und anschließend programmiert und gesäubert werden. Dadurch ist eine hohe Anzahl von Tests notwendig – diese Tests können auf der einen Seite Erfolgserlebnisse darstellen, den Entwickler jedoch auch frustrieren, da die Entwicklung scheinbar langsam vorangeht. Da diese Tests von den Entwicklern selbst geschrieben werden, kann es vorkommen, dass gewisse Teile nie getestet werden und es trotz aller Tests Fehler im Programm gibt. Durch das Fehlen einer externen und damit objektiveren Überprüfung kann es also zu einer trügerischen Sicherheit kommen. [Vgl. 7, S. 155 ff.]

Studien und Erfahrungsberichte haben gezeigt, dass Entwickler, die erstmals testgetrieben entwickeln, schnell in Frustration und Überforderung gedrängt werden. Vor dem allerersten Test müssen dem Entwickler die entsprechenden *Test Frameworks* bekannt und geläufig sein. Und wenn innerhalb eines Entwicklungsprozesses die Softwareentwicklung auf TDD umgestellt wird, so produziert das zu Beginn eine große Anzahl von Tests, die für das bereits Programmierte bestimmt sind und eine hohe Wahrscheinlichkeit mit sich bringt, dass ein große Anzahl an bereits geschriebenem und funktionierendem Code geändert werden müsste. [Vgl. 7, S. 155 ff. und 6, S.36]

In der Vorbereitungsphase habe ich versucht, testgetrieben zu entwickeln. Ich habe mich aus den oben genannten Gründen und den nachfolgend aufgeführten *gegen* die Technik der testgetriebenen Entwicklung entschieden:

1. Die bestehende Codebasis von Saros ist so groß, dass die Einarbeitungszeit zu sehr verlängert würde.
2. Das eigene Saros Test Framework (STF) hätte zuerst von mir erweitert werden müssen, da nicht alle nötigen Testfunktionalitäten zu Verfügung standen, was wiederrum zu viel Zeit gekostet hätte.

3.1.2. Nutzerorientierte Gestaltung

Der Begriff der nutzerorientierten Gestaltung (vom englischen User-Centered-Design, UCD) ist am Begriff der Gebrauchtstauglichkeit angelehnt, wird aber ausschließlich für die Beziehung von Menschen zu EDV-Produkten angewandt und dort insbesondere für das Softwaredesign. Nutzerorientierte Gestaltung im Softwareentwicklungsprozess soll dazu führen, dass das fertige Programm tatsächlich die Bedürfnisse des Nutzers erfüllt und dieser es gut nutzen kann. Um nutzerorientiert zu gestalten, sollte auf Erfahrungen der Nutzer eingegangen werden und das in einem Maße, dass die Erfahrungen statistisch angemessen gewertet werden können – und auf diese Weise zu einer objektiven Anforderung werden können.[Vgl. 5]

Wird sich bei der Softwareentwicklung für UCD entschieden, so gibt es diverse Methoden, wie z. B. Nutzer-Befragungen in verschiedenen Formen und Erstellung von Prototypen für diverse Nutzertypen, um die Erfahrungen der Nutzer festzuhalten, vgl. [13].

Leider habe ich während der Bearbeitungszeit keinen echten Nutzer für das Teilen der Konsole gefunden. Daher habe ich mich *gegen* UCD entschieden (mit Ausnahme der Anwendungsfälle, die ich aber ohne Kundeneinfluss entwickelte – siehe Kapitel 3.3).

3.1.3. Risikoorientierte Entwicklung

Mittels der risikoorientierten Entwicklung möchte man die größten Risiken frühzeitig ausräumen. Wer diese Vorgehensweise nutzt, erstellt möglichst früh eine Liste von Fällen, die das gesamte Projekt scheitern lassen könnten, sollten diese schiefgehen. Diese Liste wird dann in der Reihenfolge des wahrscheinlichsten Scheiterns abgearbeitet. Beim Teilen der Eclipse-Konsole bestand diese Liste aus einem Element: die Machbarkeit. (Vgl. [11])

3.1.4. Timeboxing

Timeboxing ist eine Methode oder Technik, die innerhalb einer Projektdurchführung eingesetzt wird. Timeboxing arbeitet mit der festen Vorgabe, dass an der zeitlichen Planung keine Änderungen durchgeführt werden. Sollte der Fortschritt nicht so zügig vorwärts gehen wie geplant, so sind Abstriche im Entwicklungsstand, nicht je-

doch in der zeitlichen Komponente vorzunehmen. Das führt dazu, dass zu dem Zeitpunkt, der bei Projektbeginn festgelegt wurde, definitiv ein Programmiercode vorliegt. Es ist möglich, dass dieser nicht so weit ausgereift ist wie ursprünglich geplant, es existiert aber ein System, das funktioniert, vgl. [1].

Da ich zu einem festen Zeitpunkt die Implementierung eines Mechanismus‘ zum Teilen und Übertragen der Eclipse Konsole fertig gestellt haben möchte, habe ich mich *für* die Technik des Timeboxing entschieden und diese bei verschiedenen Teilbereichen angewendet. Beispielsweise habe ich einige Videos ausgewertet, in dem zwei Programmierer mittels verteilter Paarprogrammierung mit Saros arbeiten. Anhand dieser Videos wollte ich erkennen, wann das Teilen und Übertragen der Konsole wichtig ist. Bevor ich mit der Auswertung der Videos begann, habe ich mir einen festen Zeitrahmen gesetzt – eine Woche – innerhalb dessen ich mich mit den Videos und der Auswertung beschäftige, aber nicht länger, da der Nutzen für die Implementierung unklar war.

3.2. Vorbereitungen

Nachdem ich mich für das Projekt des Teilens und Übertragens der Eclipse Konsole in Saros entschieden hatte, begann ich mich mit den Vorbereitungen. Als erstes erstellte ich einen Entwurf für eine Anforderung. Bevor ich anfing, mich mit Saros und den zu verwendenden Technologien und Prozessen vertraut zu machen, war es wichtig (siehe Kapitel 3 Risikoorientierte Entwicklung), die Machbarkeit nachzuweisen. Dazu entwickelte ich zuallererst einen Prototyp. Mit seiner Hilfe konnte ich mich mit der Architektur des Eclipse-Konsolen-Plug-Ins vertraut machen.

3.2.1. Teilen der Konsole und Paarprogrammierung

Ohne einen Nutzer, der eine Anforderung für ein konkretes Szenario erheben könnte und somit auch Gründe für das Teilen der Konsole liefern könnte, kann ich nur Vermutungen über ein solches Szenario anstellen. Wie kann also das Teilen der Konsole die verteilte Paarprogrammierung in Saros verbessern?

Ein Szenario ist beispielsweise, dass nur ein Teilnehmer in einer Saros-Sitzung die Möglichkeit hat, eine Konsolenausgabe zu erzeugen. Das könnte den anderen dabei helfen zu verstehen, was, wie und in welcher Reihenfolge passiert.

Ein anderes Szenario könnte aus sehr unterschiedlichen Hardwareressourcen der Beteiligten bestehen. Das Teilen der Ausgaben der laufenden Programme könnte nun Aufschluss darüber geben, an welchen Stellen die Laufzeit, also Zeitdauer zur Bearbeitung von Prozessen, verbessert werden muss. Auch das Analysieren nebenläufiger Prozesse kann auf diese Weise vereinfacht werden.

3.2.2. Anforderung

Mein erster Entwurf einer Anforderung sah folgendermaßen aus:

> **Die eigene Konsolenansicht in Eclipse soll in einer laufenden Saros-Sitzung allen anderen Teilnehmern zur Verfügung gestellt werden.**

Dieser Entwurf ist entstanden ohne Einfluss eines echten Nutzers. Vielmehr bietet Saros seinen Nutzern nach einer Sitzung die Möglichkeit einen kurzen Fragebogen auszufüllen und diesen an das Saros Projekt zu schicken und so etwaige Fehler zu melden oder Verbesserungsvorschläge anzubringen. Über diese Feedback-Möglichkeit wurde mehrfach von diversen Nutzern der Wunsch geäußert, auch die Konsole während einer Saros-Sitzung teilen zu können.

3.2.3. Machbarkeit / Prototyp

Der die Machbarkeit darstellende Prototyp genügte den folgenden Bedingungen:

1. Er ist von Saros komplett unabhängig.
2. Der Text konnte aus der Konsolenausgabe extrahiert werden.
3. Der Text konnte in die Konsolenausgabe geschrieben werden.

Nachdem ich diesen Prototyp erfolgreich fertiggestellt hatte, wurde dieser verworfen und nicht mehr zur Weiterentwicklung genutzt.

3.2.4. Verwendete Technologien

Git

Git ist eine Versionsverwaltungssoftware (wie z. B. SVN) und kann zur automatischen Versionsverwaltung des Programmcodes verwendet werden. Der Quellcode kann von allen Teilnehmern einer spezifischen Sitzung aufgerufen werden und wird dann auf dem jeweiligen Computer mit all denen durch diesen Entwickler vorgenommenen Änderungen gespeichert. Um die Änderungen dem ursprünglichen Quellcode hinzuzufügen, muss der Befehl „Push to Upstream“ durchgeführt werden. [Vgl. 3, S.76 ff. und 2, S. 671 ff.]

Egit (Git for Eclipse) ist ein Eclipse-Plug-In, das es einem ermöglicht, viele Git-Befehle direkt aus Eclipse heraus zu nutzen.

Gerrit

Gerrit ist ein webbasiertes System für Code-Durchsichten (auch *Review System* genannt, vgl. [1] S. 138 ff.), mit dessen Hilfe alle Entwickler in einem Projekt eine Änderung am Code (Commit) manuell durchsehen können. Gerrit verwaltet ein Git-Repository, und die Entwickler haben keinen direkten Schreibzugriff auf den Hauptzweig, sondern können nur Commits zur Begutachtung bereitstellen, die dann gemeinschaftlich beurteilt und danach in den Hauptzweig integriert werden. Es gibt ein Bewertungssystem, nachdem erst eine bestimmte Wertung für diesen Commit

erreicht werden muss, bevor er in den Hauptzweig integriert werden kann. Bei Saros hat jeder Entwickler die Möglichkeit, eine Wertung von -2 bis +2 abzugeben[5]. Ein Commit braucht eine Wertung von mindestens +2 für die Integration in den Hauptzweig.

Gerrit dient aber auch als Kommunikationsforum, da hier jede Codezeile kommentiert werden kann. Dies kann zu angeregten Diskussionen führen und gibt dem Ersteller des Commits die Möglichkeit, diesen anzupassen und erneut durchsehen und bewerten zu lassen. Gerrit erstellt dann automatisch neue Versionen des Commits, die aufeinander aufbauen und sich paarweise nach Unterschieden untersuchen lassen können.

Sofern es genügend Entwickler gibt, die sich gut mit dem Code auskennen, kann mit der Anwendung dieses Durchsicht-Systems das Integrieren fehlerhaften Codes in den Hauptzweig reduziert und die Codequalität erheblich gesteigert werden.

Jenkins

Jenkins ist ein Open Source Werkzeug, das automatisiert Test- und Buildprozesse ausführen kann. Es wird unter anderem für die kontinuierliche Integration genutzt.

Bei Saros ist Jenkins direkt mit Gerrit gekoppelt und stößt bei jedem Commit automatisch einen Build- und Testprozess an, der genau protokolliert wird. Gerrit wird anschließend benachrichtigt, ob alle Tests erfolgreich verlaufen sind oder es zu Fehlern kam. Liegt ein Fehler vor gibt Jenkins als Entwickler in Gerrit dem entsprechenden Commit eine -1 Bewertung. Zusätzlich gibt es von Gerrit unabhängige automatische Tests, die z. B. jede Nacht angestoßen werden. Das erhöht die Wahrscheinlichkeit, mit der seltene Fehler gefunden werden können.

3.2.5. Kontinuierliche Integration

Kontinuierliche Integration (Vgl. [12]) ist eine Methodik aus dem Bereich *Extreme Programming*. Sie wird genutzt, um Software nach bestimmten Kriterien automatisch zu produzieren. Dazu wird die Software in Teilkomponenten zerlegt, die einzeln gebaut und automatisch getestet werden können. Saros setzt die kontinuierliche Integration mit den oben beschriebenen Technologien Git, Gerrit und Jenkins um[6].

3.2.6. Verhalten der Oberfläche

Bei all meinen Überlegungen und Entscheidungen gehe ich davon aus, dass alle Teilnehmer während einer Saros-Sitzung miteinander sprechen können (z. B. per Telefon oder Voice over IP). Aus diesem Grund, und weil Saros ein leichtgewichtiges[7]

[5] Für mehr Informationen: http://www.saros-project.org/node/141
[6] Für eine umfangreiche Lektüre empfehle ich die Bachelorarbeit von Holger Freyther: Beseitigung von Stolpersteinen im Saros-Entwicklungsprozess, S.13 ff.
[7] Leichtgewichtig im Sinne von einfach zu benutzen und einfach zu konfigurieren.

Plug-In ist, habe ich darauf geachtet, den Nutzer nicht mit unnötigen Nachrichten an der Oberfläche zu stören, wie zum Beispiel Frage- oder Bestätigungsdialoge. Ein konkretes Beispiel hierfür ist das Weglassen einer Benachrichtigung beim Empfänger, dass das Teilen der Konsole gestartet wurde, oder sogar die Frage, ob er geteilte Konsolen empfangen möchte.

Als Teilnehmer einer Saros-Sitzung kann man aktiv an der Weiterentwicklung eines Projektes bzw. mehrerer Projekte teilnehmen oder aber passiv in einem Folgemodus genau das beobachten, was der Partner gerade ändert oder markiert. Das Teilen der Konsole soll diesem Verhalten so weit wie möglich gleichen. Man kann aktiv programmieren und sich bei Bedarf die vom Partner geteilte Konsole anschauen oder im Folgemodus jede Änderung sofort angezeigt bekommen. Wobei das Teilen der Konsole immer als aktiv betrachtet wird: Man kann nur seine eigene Konsole teilen und nicht die Konsole eines Teilnehmers geteilt bekommen.

Die Eclipse Konsole bietet – wie in Kapitel 2.4 beschrieben – ihrem Nutzer die Möglichkeit, ihr Verhalten auf verschiedene Arten anzupassen. Ich habe darauf geachtet, dieses Verhalten weder zu verändern noch einzuschränken. Jeder Nutzer hat die Verantwortung über seine Konsolenansicht. Welche Konsolenansicht und wie er diese angezeigt bekommen möchte, entspricht genau dem Verhalten der Standard-Konsolenansicht. Einzige Ausnahme ist der oben beschriebene Folgemodus, bei dem sich der Nutzer bewusst entscheidet diese Kontrolle abzugeben.

3.2.7. Entkoppelung

Zum aktuellen Zeitpunkt finden Arbeiten an einer Portierung von Saros als Eclipse-Plug-In zu einem IntelliJ- und NetBeans-Plug-In[8] statt. Um diese Portierung zu ermöglichen, werden alle Eclipse-unabhängigen Bausteine aus dem bisherigen Projekt herausgelöst und in einem eigenen Projekt gebündelt, dem so genannten „Kern (Core) von Saros“. Durch die kontinuierliche Integration (vgl. Kapitel 3.2.5) war es mir möglich, den Fortschritt dieser Arbeiten aktuell zu verfolgen und zu helfen, das Teilen der Konsole auch bei diesen IDEs voranzutreiben. Ich habe bei der Implementierung von Anfang an darauf geachtet, Eclipse-Abhängigkeiten zu erkennen und entsprechende abstrakte Klassen und Interfaces im Kern anzulegen. Bei dieser Entkoppelung musste ich festlegen, wie den implementierenden oder erbenden Klassen Zugang gewährt wird und welche Forderungen ich an sie stelle. Hier hatte ich z. B. die Möglichkeit, auf der einen Seite eine abstrakte Methode einen Rückgabewert fordern zu lassen, so dass einem Entwickler bei der Implementierung sofort auffällt, wenn er den falschen Wert zurückgibt. Oder andererseits nichts zu fordern und es dem Entwickler zu überlassen, bei der Implementierung die Korrektheit aller Werte sicherzustellen. Ein Fehler würde so erst zur Laufzeit auffallen. Der Vorteil ist aber die bessere Verständlichkeit bei der Nutzung und die größere Flexibilität. Sofern ich sicherstellen konnte, dass ein Fehler zur Laufzeit keinen Schaden anrichten kann, habe ich mich für die zweite Möglichkeit entschieden.

[8] IntelliJ- und NetBeans sind ähnlich wie Eclipse Entwicklungsumgebungen.

3.3. Anwendungsfälle

Mit Anwendungsfällen, oder *use cases*, sind Verhaltensanforderungen oder Anwendergeschichten gemeint. Laut [10] ist ein Anwendungsfall eine „Vereinbarung zwischen den Beteiligten eines Projektes über das Verhalten des Systems“. Es handelt sich dabei um einen kurzen Text in klar verständlicher Sprache. Dieser Text enthält ausreichend Details, damit alle Beteiligten wissen, was programmiert werden soll und welches Ziel damit erreicht werden soll. Ein Anwendungsfall sollte nur so viel enthalten, dass der Entwickler den Programmiercode innerhalb von drei Wochen inklusive Tests und eventuellen Korrekturen umsetzen kann. [Vgl. 4, S.229 ff.]

Anwendungsfälle können für vielfältige Zwecke eingesetzt werden und unterscheiden sich je nach Zweck gegebenenfalls auch in ihrer Detailtiefe. Ich habe Anwendungsfälle genutzt, um die funktionalen Anforderungen an die durch mich zu implementierende Komponente zu beschreiben.

Bei der Erstellung eines Anwendungsfalles sollte man sich immer an den folgenden drei Punkten orientieren:

1. Ein Anwendungsfall ist auf ein konkretes Ziel fokussiert – jeder einzelne Satz sollte daher ein Teilziel enthalten.
2. Seine Formulierung ist nicht aus Sicht des Entwicklers vorzunehmen, sondern entspricht einer Draufsicht eines Außenstehenden.
3. Er muss klar und verständlich formuliert sein, allen muss klar sein, was erreicht werden soll. [Vgl. 4, S. 263 f.]

Ich habe drei Anwendungsfälle erstellt und werde diese nachfolgend aufführen. Zuerst gehe ich auf die Punkte ein, die allen Anwendungsfällen gemein ist.

3.3.1. Gemeinsamkeiten der genutzten Anwendungsfälle

Hauptakteur: Alice

Alice ist der Host einer laufenden Saros-Sitzung. Alice hat Bob eingeladen, an der Sitzung teilzunehmen.

System: Saros

Niveau: Benutzerziel

Beteiligte: Bob

Voraussetzung: Es läuft eine aktuelle Saros-Sitzung mit Alice und Bob als Teilnehmern.

Invarianten[9]:

1. Alice und Bob befinden sich in einer Saros-Sitzung.
2. Alice und Bob haben eine Konsolenansicht zur Verfügung.

[9]Eine Invariante ist die Beschreibung eines Zustandes des Systems, der vor, während und nach einem Prozess erhalten bleibt. Eine Invariante ist sozusagen garantiert bzw. unveränderlich, vgl. [4] S. 111 f.

3.3.2. Bemerkungen

- Die Konsole beinhaltet alle eigenen Konsolenausgaben.
- Eine eigene Konsolenausgabe ist eine Konsolenausgabe, die nicht durch Saros geteilt wurde.
- Eine Änderung einer Konsolenausgabe beinhaltet das Hinzufügen und Löschen einer Konsolenausgabe sowie Änderungen an deren Inhalt.
- Der Teilen-Button ist ein Toggle-Button in der Konsolenansicht, der sich entweder aktivieren oder deaktivieren lässt.
- Alice muss nicht Host einer Sitzung sein, um ihre Konsole zu teilen.
- Alice und Bob sind durchaus austauschbar und können ihre Konsole auch gleichzeitig teilen.

3.3.3. Genutzte Anwendungsfälle

Anwendungsfall 1: Teilen der Konsole

Vorbedingung: Alice hat das Teilen der Konsole nicht gestartet.

1. Alice aktiviert den Teilen-Button, um das Teilen der Konsole zu starten.
2. Saros erstellt für jede eigene Konsolenausgabe von Alice eine neue Konsolenausgabe bei Bob.
3. Saros füllt die neu erstelle Konsolenausgabe mit dem entsprechenden kompletten Inhalt.
4. Saros überträgt ab jetzt jede Änderung einer eigenen Konsolenausgabe von Alice an Bob.
5. Bob hat den aktuellen Inhalt von Alices Konsole nun jederzeit in der eigenen Konsole verfügbar.

Anwendungsfall 2: Teilen der Konsole beenden

Vorbedingung: Alice hat das Teilen der Konsole gestartet.

1. Alice deaktiviert den Teilen-Button, um das Teilen der Konsole zu stoppen oder die Sitzung wird beendet(Ausnahme zu Invariante 1)[10].
2. Saros reagiert nicht mehr auf Änderungen der Konsolenausgaben von Alice.
3. Saros beendet die laufende Übermittlung an Bob.
4. Bob hat den letzten Inhalt von Alices Konsole immer noch in der eigenen Konsole verfügbar.

[10] Es ist dabei unwichtig, durch wen die Sitzung beendet wird, ob durch Alice, Bob oder äußere Einflüsse.

Anwendungsfall 3: Entfernung einer geteilten Konsolenausgabe

Anmerkung: In diesem Fall ist Bob der Akteur und Alice nicht beteiligt.

Vorbedingung: In Bobs Konsole befindet sich eine von Alice geteilte Konsolenausgabe.

1. Bob drückt auf den Button „Close Console“.[11]
2. Die Konsolenausgabe wird aus Bobs Konsolenansicht entfernt.
3. Saros teilt Bob die Änderungen dieser geteilten Konsolenausgabe nicht mehr mit.

3.3.4. Entwicklung der Anwendungsfälle

Die obigen Anwendungsfälle sind nicht der erste Entwurf, sondern das Resultat ständiger Anpassungen und Verbesserungen. Das betrifft sowohl die Wortwahl als auch den Inhalt.

Beispielhaft zeige ich den Verlauf vom ersten Anwendungsfall “Teilen der Konsole“:

> Die erste Version war noch sehr unausgereift und hatte die Invariante 1 als Vorbedingung:
>
> > Vorbedingung: Alice und Bob befinden sich in einer Saros-Sitzung.
> >
> > 1. Alice wählt per Knopfdruck aus, dass sie den Inhalt ihrer aktiven Konsolenausgaben andauernd Bob zur Verfügung stellt.
> > 2. Die Konsolenausgabe von Alice erscheint bei Bob in einem separaten Fenster.
>
> Diese fortgeschrittene Version entstand, als Änderungen nicht sofort erkannt wurden, sondern nach einiger Zeit ausgelesen wurden. Das Löschen und Hinzufügen von Konsolenausgaben galt auch noch nicht als Änderung.
>
> 1. Alice aktiviert den Teilen-Button, um das Teilen der Konsole zu starten.
> 2. Saros liest unmittelbar den Inhalt aller aktiven Konsolen aus und leitet diesen weiter an Bob.
> 3. Saros wartet auf Änderungen der Konsole und übermittelt diese an Bob.
> 4. Für jede Konsolenausgabe von Alice erstellt Saros eine neue Konsole (sofern noch nicht vorhanden) und füllt diese mit dem von Alice übertragenem Inhalt.
> 5. Bob kann sich nun den aktuellen Inhalt von Alices Konsole anschauen.

[11] Der „Close Console“-Button befindet sich in der Konsolenansicht der entsprechenden Konsolenausgabe.

4. Kapitel: Implementierung

Saros ist ein agiles Open Source Softwareprojekt. Es kann schnell auf Erweiterungen reagieren und wird bei Problemen schnell angepasst. Aus diesem Grund vermeide ich es an dieser Stelle auf konkrete Umsetzungen einzugehen. Gerade Namen von Klassen und Methoden können sich sehr schnell ändern. Vielmehr gehe ich auf einige Designentscheidungen ein, die ich bei der Entwicklung getroffen habe. Diese sollten etwas mehr Bestand haben und zukünftigen Entwicklern ihre Arbeit erleichtern.

4.1. Button in der Konsolenansicht

Für die erste Entscheidung habe ich die Rolle eines Nutzers eingenommen. Wie würde ich meine Konsole während einer Saros-Sitzung teilen wollen? Zur Auswahl standen folgende Möglichkeiten.

1. Automatischer Start des Teilens, z. B. bei Sitzungsbeginn

Das hat den Vorteil, sehr komfortabel für den Nutzer zu sein, da er nichts tun muss – allerdings dafür den Nachteil, sehr unflexibel zu sein. Denkbar wäre stattdessen ein intelligentes Starten und Stoppen, also genau dann, wenn es gebraucht wird. Der Aufwand ist allerdings hoch und ohne Nutzertests nicht umsetzbar.

2. Ein Button in der Benutzeroberfläche

Flexibilität ist hier ein klarer Vorteil. Der Nutzer kann jederzeit das Teilen der Konsole starten und stoppen. Dass nun noch ein Button zu der Flut an Interaktionsmöglichkeiten, die Eclipse bietet, hinzukommt, könnte als Nachteil gesehen werden.

Ich habe mich für die zweite Möglichkeit entschieden, da mir als Nutzer Flexibilität wichtiger ist als Komfort.

Die Entscheidung, an welche Stelle ich diesen Button setze, war schnell getroffen: So dicht wie möglich an dem zu bewirkenden Effekt: Die Toolbar der Eclipse-Konsole (siehe nachfolgende Abbildung 2).

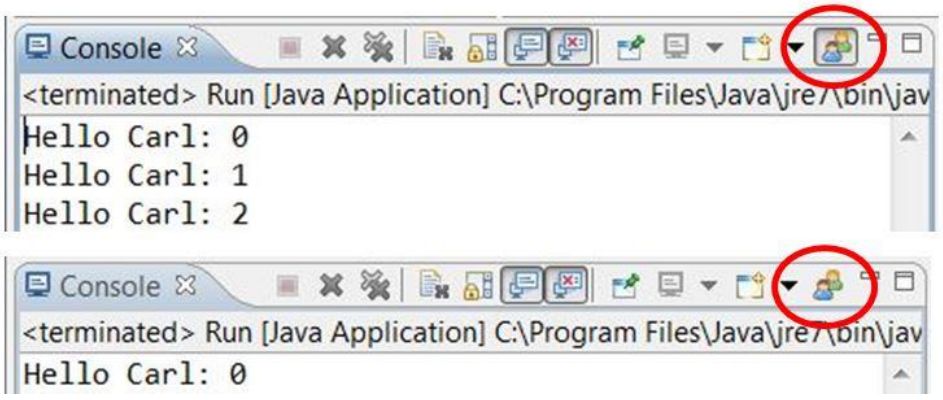

Abbildung 2: Button gedrückt / nicht gedrückt

4.2. Art des Teilens

„Divide et impera“

– Grundprinzip der Außenpolitik im antiken Rom

Auch für das Teilen der Konsole galt es Entscheidungen zu treffen:

1. Wann und wie wird der Inhalt einer Konsolenausgabe ausgelesen?

Eine Möglichkeit war es, nach einem bestimmten Zeitintervall den gesamten Inhalt auszulesen:

- **Vorteil**: Unabhängig von der Anzahl der Änderungen, wird der Vorgang nur einmal pro Intervall angestoßen. Das Anpassen der Laufzeit ist auf diese Weise sehr einfach.
- **Nachteil**: Die Summe der Änderungen seit dem letzten Durchlauf muss bestimmt werden.

Die Alternative bestand im Reagieren auf Änderungen am Inhalt:

- **Vorteil**: Alle Änderungen lassen sich schnellstmöglich in der richtigen Reihenfolge bei allen anderen Teilnehmern nachbauen.
- **Nachteil**: Kann sehr unterschiedliche Effekte auf die Laufzeit des Systems haben, je nachdem wie schnell die Änderungen erzeugt werden und wie zeitintensiv das Reagieren auf diese und das Übermitteln ist.

Ich habe mich für das Reagieren auf Änderungen entschieden, da mir auch als Entwickler Flexibilität wichtig ist und der Nachteil umgangen werden kann. Generell sollte man als Entwickler die ereignisorientierte Programmierung dem Polling vorziehen.

3. Inhalte sofort übermitteln oder gepuffert?

Wie oben erwähnt, kann die sofortige Übermittlung die Laufzeit verringern. Ich habe also alle Änderungen in Reihenfolge ihres Auftretens in einer Liste zwischengespeichert und diese Liste zyklisch übermittelt. Dieser Zyklus kann während einer Sitzung verändert werden, um die Laufzeit zu verbessern.

5. Kapitel: Auswertung

5.1. Erweiterungen

Ich habe eine den Anforderungen entsprechende Version des Teilens der Eclipse Konsole fertiggestellt. Diese ist jedoch keine vollständige bzw. abgeschlossene Umsetzung. Einige Zusatzfunktionalitäten konnte ich aus Zeitgründen nicht mehr einbauen.

Die folgenden Punkte sind als Vorschlag für Erweiterungen zu verstehen. Zu jedem Punkt gebe ich eine Aufwandsabschätzung ab. Diese basiert auf der Zeit (in Manntagen), die ich für vergleichbare Aufgaben gebraucht habe und beinhaltet nicht den Review- und Test-Prozess. Dabei gibt es folgende Abstufungen:

Gering: weniger als 2 Manntage

Hoch: mehr als 10 Manntage

5.1.1. Der Folgemodus

Diese Erweiterung soll dafür sorgen, dass sich die Konsolenansicht des Folgenden genauso verhält wie die des Gefolgten. Jeder Wechsel einer Konsolenausgabe, jedes Navigieren im Text und jeder Verhaltenswechsel der Konsolenansicht wird synchron umgesetzt.

Aufwand: *Hoch*

5.1.2. Der Tooltip des Buttons

Ich habe über die plugin.xml den Tooltip: „Start/Stop sharing the Console" eingerichtet. Dieser sollte aber abhängig vom aktuellen Zustand sein und beschreiben, was passiert, wenn man ihn drückt. Dies erfordert eine programmatische Anpassung.

Aufwand: *Gering*

5.1.3. Speichern des Buttonzustandes

Es ist möglich, das Starten und Stoppen des Teilens der Konsole auch von anderen GUI-Elementen aus oder programmatisch aufzurufen. Dies kann dann zu einem inkonsistenten Zustand des Buttons führen. Es muss also Logik hinzugefügt werden, die diesen Zustand anpasst.

Aufwand: *Gering*

5.1.4. Farben und Hyperlinks übertragen

Das Eclipse Konsolen Plug-In stellt Funktionalitäten zur Verfügung, um Farben und Hyperlinks einer Konsolenansicht auszulesen und sie einzufügen. Es müssen aber vorher die entsprechenden Farbeinstellungen der Teilnehmer der Saros-Sitzung abgeglichen werden um Konflikte (z. B. Vertauschen von Grün und Rot) zu vermeiden.

Bei den Hyperlinks ist es möglich, dass die Teilung nicht sinnvoll ist, da dem Empfänger die entsprechende Ressource nicht unbedingt zur Verfügung steht.

Aufwand: *Hoch*

5.2. Ausblick

Zusätzlich zu den konkreten Erweiterungen gebe ich noch einen allgemeinen Ausblick, was für die Verbesserung des Teilens der Konsole getan werden kann.

5.2.1. Testfälle

Wie in Kapitel 3 erwähnt, hatte ich nicht genug Zeit, um das STF ausreichend zu erweitern und einen Testfall zu implementieren. Alle Tests habe ich manuell auf meinem PC vorgenommen. Ich halte es für sehr wichtig, einige Testfälle zu entwickeln und werde versuchen, diese in den nächsten Wochen zu entwickeln.

Diese Tests sollten mindestens folgende Punkte sicherstellen:

- Die Oberfläche darf nicht einfrieren.
- Alle Änderungen müssen in der richtigen Reihenfolge und vollständig gesendet und empfangen werden.
- Performantes Verhalten auf verschiedensten Umgebungen.

5.2.2. Benutzerorientierte Gestaltung

Um die Anforderung zu verbessern und somit die Benutzertauglichkeit zu erhöhen, sollten echte Benutzer gefunden werden, die das Teilen der Konsole in Saros benötigen. Nur dann ist es auch möglich, den gesamten iterativen Prozess der benutzerorientierten Gestaltung zu durchlaufen.

5.2.3. Portierungen

In Kapitel 3.2.7 hatte ich die Portierung zu anderen IDEs angesprochen. Ich gehe davon aus, dass es in den nächsten Monaten eine lauffähige Version von Saros für IntelliJ und netBeans gibt. Es müssten dann die entsprechenden ActivityConsumer und ActivityProducer für das Teilen der Konsole implementiert werden und diese dann am besten mit einem neuen Button an der Oberfläche verknüpft.

5.2.4. Unterstützung beliebiger Konsoleninhalte

In Kapitel 2.4 erwähnte ich, dass andere Eclipse Plug-Ins die Möglichkeit haben, beliebige Inhalte in einer Konsolenausgabe darzustellen. Eine Analyse dieser Inhalte und der Möglichkeit sie auszulesen, habe ich aus Zeitgründen nicht vorgenommen. Für jeden Inhaltstyp muss das Auslesen und Füllen der Konsolenausgabe einzeln erweitert werden.

5.3. Schlusswort

Abschließend lassen sich meine Erkenntnisse in zwei Gruppen einteilen:

1. die Arbeit im Saros-Team und
2. die Probleme, denen ich mich bei der Ausarbeitung stellen musste.

5.3.1. Arbeiten im Saros Team

Wenn man als Entwickler im Saros Team mitarbeitet, so bringt das einige Pflichten mit sich: Zweimal in der Woche fand das sogenannte Standup statt. Alle Studenten, die gerade eine Abschlussarbeit zum Saros-Projekt anfertigen, stehen sich im Kreis gegenüber. Franz Zieris als Betreuer ist ebenfalls anwesend. Alle Teilnehmenden berichten kurz in ca. zwei bis fünf Minuten, was sie seit dem letzten Standup getan haben und welche Pläne sie für die Zeit bis zum nächsten Standup haben. Das Standup ist an das sogenannte *daily SCRUM meeting* angelehnt.[12] Wenn es dabei zu konkreten Fragen kommt, können diese nach dem Standup persönlich mit Einzelnen besprochen werden. Auch zu Problemen, die aufgetreten sind, findet meistens jemand eine Lösung oder zumindest einen Ansatz, um das Problem zu lösen.

Das Arbeiten im Saros Entwickler Team brachte aber auch diverse Vorteile mit sich. Das regelmäßige Standup führt neben den oben aufgeführten Punkten auch dazu, dass die Körperhaltung und -sprache, die Art und Weise zu Formulieren und die Sicherheit beim Reden regelmäßig trainiert werden. Das Arbeitsklima innerhalb der Arbeitsgruppe war durchgehend sehr gut und die Motivation sehr hoch.

Ich habe schon in anderen Projekten mitgearbeitet und nach Abschluss gab es oft eine sogenannten „Lessons Learned“ Workshop. Der Punkt: Erreichbarkeit im Team war immer einer derjenigen, die am höchsten priorisiert wurden! Die Erreichbarkeit im Saros Team war immer sehr gut. Ich musste nie länger als einen Tag auf eine Antwort auf eine Mail warten. Und bei Skype waren tagsüber mindestens zwei Leute ansprechbar. Das reduziert mögliche Zwangspausen, weil man gerade irgendwo nicht weiterkommt oder von jemand abhängig ist, erheblich.

Ein weiterer Vorteil dieser Arbeitsgruppe ist das Durchsicht-System. Bei der Entwicklung von Software entstehen immer Fehler und es gibt immer Verbesserungsmöglichkeiten. Schaut nun ein anderes Projektmitglied auf den abgegebenen Code,

[12] Mit „Standup“ ist hier ein kurzes Meeting gemeint, bei dem alle stehen, damit das Meeting auch tatsächlich kurz bleibt. Die Vorlage, das *daily SCRUM meeting*, gehört zur einer agilen Vorgehensweise der Softwareentwicklung namens „SCRUM“, vgl. [15] S. 71.

wird er mindestens eines von beiden finden. Dieses Vorgehen führte, zumindest bei mir, zu einem regelmäßigen Training zur Vermeidung von Fehlern und Erkennung von Verbesserungen.

5.3.2. Probleme

Durch die große Codebasis und die mir zuvor unbekannten Technologien wie Git, Gerrit und Jenkins, habe ich sehr lange gebraucht, um mich in Saros einzuarbeiten und erste Anpassungen vorzunehmen. Dazu kommt, dass jede Klasse und jede Methode von anderen verändert werden kann und es bei der Benutzung nach einem Update zu anderen Ergebnissen kommen kann. Über dieses Problem war ich mir aber vorher bewusst, sodass ich davon nicht komplett überrascht wurde, sondern höchsten vom Ausmaß dessen.

Ein anderes Problem, dass ich nicht erwartet hatte, war, dass ich kein Vergleichsprojekt finden konnte. Ich war sicher, jemand anderes hätte bestimmt schon die Konsolenausgabe von Eclipse ausgelesen und in eine andere kopiert. Dieser Umstand führte zu einem deutlich längeren Bearbeitungszeitraum, als ich geplant hatte.

Zusätzlich ist mir bewusst geworden, welche Schwierigkeiten die Informationssuche im Internet aufwerfen kann. Leider ist es einige Male vorgekommen, dass jemand anders innerhalb der Arbeitsgruppe die von mir gesuchte Information finden konnte bzw. diese deutlich schneller fand als ich.

5.3.3. Fazit

Das Ziel dieser Arbeit war es, Saros um die Möglichkeit zu erweitern, die Konsole in Eclipse zwischen den an einer Sitzung beteiligten Entwicklern zu teilen. Zu diesem Zweck habe ich anhand eines Prototypen die Machbarkeit nachgewiesen, mithilfe von Anwendungsfällen eine Anforderung entwickelt und meine Implementierungen kontinuierlich in den bestehenden Saros-Code integriert. Dabei hat sich gezeigt, dass es möglich ist dieses Ziel zu erreichen ohne einen echten Nutzer zu haben, der für eine Anforderungsentwicklung und abschließende Tests genutzt werden kann. Allerdings ist davon abzuraten, da die Qualität des Ergebnisses mit Einbeziehung eines Nutzers immer besser sein wird.

In fachlicher Hinsicht kann man abschließend sagen, dass davon ausgegangen werden kann, dass das Teilen einer Konsole in Saros die Qualität der verteilten Paarprogrammierung erhöht, da der den Beteiligten verfügbare Informationsgehalt verbessert wird und somit diese der Qualität der lokalen Paarprogrammierung ein Stück angenähert wird.

Insgesamt konnte ich mir, insbesondere dank der Arbeit im Saros-Team, einige Fähigkeiten aneignen, die mir die Vorgehensweise im weiteren Berufsleben als Softwareentwickler erleichtern und die Qualität meiner Arbeit erhöhen werden.

Literaturverzeichnis

[1] M. Broy und M. Kuhrmann, „7.6.6.3 Time Boxing“ in *Projektorganisation und Management im Software Engineering*, Springer-Verlag, 2013.

[2] S. Kersken, *Handbuch für Fachinformatiker: der Ausbildungsbegleiter*, 2. Ausgabe, Galileo computing, 2005.

[3] E. Burnette und J. Staudemeyer, *Eclipse IDE kurz & gut*, O'Reillys, 2013.

[4] A. Cockburn, *Use Cases effektiv erstellen*, Hüthig Jehle Rehm, 2008.

[5] T. Lowdermilk, *User-Centered Design: A developer's guide to building user-friendly applications*, O'Reillys, 2013.

[6] R. Osherove, *The Art of Unit Testing: Deutsche Ausgabe*, Hüthig Jehle Rehm, 2008.

[7] M. Müller, *Analyse leichtgewichtiger Softwareentwicklungsmethoden*, Logos Verlag Berlin GmbH, 2008.

[8] T. Walther, „Pair Programming – Ausarbeitung im Rahmen des Seminars Agile Softwareprozesse“, Freie Universität Berlin, Wintersemester 2004/05.

[9] Saros, http://www.saros-project.org, Stand: 17.06.2014

[10] L. Prechelt, *Anwendungsfälle (Use Cases)*, Vorlesung „Softwaretechnik“ Buchkapitel 4, Freie Universität Berlin, Institut für Informatik, 2012.

[11] R. Breu, *Software-Engineering: objektorientierte Techniken, Methoden und Prozesse in der Praxis*, Oldenbourg Verlag, 2005.

[12] M. Fowler, *Continuous Integration*: http://martinfowler.com/articles/continuousIntegration.html

[13] F. Sardowick und H. Brau, *Methoden der Usability-Evaluation* in “Methoden der Usability Evaluation”, 2. Auflage, Verlag Hans Huber, 2011.

[14] J. F. Smart, *Jenkins: The definitive Guide*, O'Reilly, 2011.

[15] E. Hanser, *Agile Prozesse*, Springer-Verlag, 2010.