Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Verbesserung und Erweiterung der Core-Bestandteile von Saros

David Sungaila
Matrikelnummer: 4665940
david.sungaila@arcor.de

Betreuer: Franz Zieris
Eingereicht bei: Prof. Dr. Prechelt

Berlin, 15.04.2016

Zusammenfassung

Das Saros-Projekt ist eine Software für in Echtzeit stattfindende, verteilte Softwareentwicklung, womit mehrere Nutzer an den selben Quelltextdateien arbeiten können. Dieses wird in Form von Plug-Ins für integrierte Entwicklungsumgebungen (IDE) erreicht. Zu diesem Zeitpunkt wird nur Eclipse unterstützt, wobei ein Plug-In für IntelliJ in Entwicklung ist. Ursprünglich begann die Arbeit an Saros ohne die Absicht weitere IDEs, abgesehen von Eclipse, zu unterstützen. Dies führte zu Problemen mit der Codequalität und einer Erschwerung des Entwicklungsprozesses, da während der Entwicklung vom IntelliJ-Plug-In sehr viele Codeduplikate angelegt worden sind. Es folgten Arbeiten, die sich diesem Problem widmeten, wofür die vorhandene Kern-Komponente (genannt Core) ausgebaut wurde. Diese Komponente enthält plattformunabhängigen Quelltext, der die Codeduplikate nach und nach abschafft. In dieser Bachelorarbeit führte ich dieses Vorhaben fort, wodurch komplexere Duplikate erfolgreich abgeschafft wurden. Zusätzlich unterstützte ich das HTML-UI-Projekt, welches eine platformübergreifende Oberfläche für Saros bereitstellen soll, mit der Implementierung fehlender Funktionalität.

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.

15.04.2016

David Sungaila

Inhaltsverzeichnis

1 Einführung

1.1 Saros-Projekt

Im Jahre 2006 begannen Arbeiten an einer neuen Software an der Freien Universität Berlin innerhalb der AG Software Engineering. Es dient als Forschungsgegenstand für verteilte Paarprogrammierung. Die Weiterentwicklung wird hauptsächlich durch Abschlussarbeiten (wie etwa dieser Bachelorarbeit) vorangetrieben, wobei die AG Software Engineering die Koordination übernimmt und notwendige Ressourcen zur Verfügung stellt. Das Softwareprojekt wurde Saros genannt; dessen Projekthomepage lautet http://www.saros-project.org/.

Paarprogrammierung ist eine Arbeitstechnik, bei derer zwei Entwickler an einem Rechner sitzen, um gemeinsam Quelltext zu bearbeiten. [CW00] Saros ist der Versuch, diese Arbeitstechnik auf verteilte Rechner zu verschieben. Das heißt, dass zwei Entwickler an zwei verteilten Rechnern sitzen und gleichzeitig den gleichen Quelltext bearbeiten. Beide Rechner sind übers Netzwerk miteinander verbunden, wobei Saros die Aufgabe übernimmt, den Quelltext zu verteilen und konsistent zu halten.

Um die fehlende direkte Kommunikation auszugleichen, welche ansonsten bei Programmierung am gleichen Rechner gegeben ist, bietet Saros weitere Werkzeuge an, wie etwa Sofortnachrichten oder eine virtuelle Weißwandtafel. Saros wird als Plug-In für Entwicklungsumgebungen (IDEs) implementiert.

1.2 Ausgangszustand

1.2.1 Entwicklung des Eclipse-Plug-Ins

Zu Beginn der Entwicklung von Saros wurde als erste und vorerst einzige Entwicklungsumgebung (IDE) Eclipse gewählt. Eclipse[1] ist eine quelloffene IDE, welche hauptsächlich für Java-Programmierung, aber auch für andere Sprachen genutzt werden kann. Sie zeichnet sich durch eine gute Erweiterbarkeit aus, die von Saros genutzt wird, um sich selbst als Plug-In in die Oberfläche und Quelltextverwaltung einzuhängen. Durch Verwendung der Plug-In-Schnittstelle von Eclipse, ist Saros in der Lage, das Verteilen und konsistent halten von Quelltext zu realisieren.

Da zu Beginn keine Unterstützung weiterer IDEs vorgesehen war, bestand Saros aus einem einzigen Plug-In-Modul für Eclipse. Die Geschäftslogik[2] und

[1] Projektwebseite: https://www.eclipse.org/

[2] Beschreibt jene Abstraktion der Logik, die sich um Regeln, Umgang und Verarbeitung von Daten kümmert. Diese Logik ist dabei unwissend über die genaue Implementierung der technischeren Ebenen wie z.B. der Nutzeroberfläche. Dieser Begriff wird unscharf benutzt,

Oberfläche waren eng miteinander verbunden.

1.2.2 Entwicklung des IntelliJ-Plug-Ins

In der späteren Entwicklung, nachdem Saros die primären Funktionen unterstützte und stabil wurde, stellte sich heraus, dass zwar seitens der Endnutzer ein großes Interesse an Saros bestand, jedoch viele Entwickler die Entwicklungsumgebung IntelliJ IDEA[3] bevorzugen. Daraus ergab sich die neue Anforderung, Saros für IntelliJ zur Verfügung zu stellen.

Für die Entwicklung des IntelliJ-Plug-Ins schloss sich ein externer Mitarbeiter dem Saros-Team an. Es entstand ein Prototyp eines IntelliJ-Plug-Ins, welcher eine problematische Codequalität aufwies. Es wurden Codeduplikate angelegt, um den bisherigen Stand vom ausgereiften Eclipse-Plug-In zu übernehmen. An diesen Codeduplikaten wurden Anpassungen vorgenommen, um das IntelliJ-Plug-In lauffähig zu bekommen.

Zu diesem Zeitpunkt existierte bereits die Kern-Komponente (genannt Core), welche jedoch noch nicht ausreichend plattformunabhängigen Code enthielt. Anstatt die benötigte Geschäftslogik aus dem Eclipse-Plug-In in den Kern zu verschieben und damit den weiteren Entwicklungsprozess zu vereinfachen, wurde der (vermutlich leichtere) Weg mit den Codeduplikaten gewählt.

1.2.3 Aufräumarbeiten und Verbesserung der Codequalität

Nach der Entwicklung des IntelliJ-Plug-In-Prototyps sah der Stand von Saros wie folgt aus: Es gab drei Module, einmal das stabile Eclipse-Plug-In, die Kern-Komponente und das neue IntelliJ-Plug-In. Letzteres stellte noch nicht die grundlegenden Funktionen bereit, die Saros für Eclipse bot.

Diese zwei Module waren komplett unabhängig von einander. Die Geschäftslogik von Saros existierte nun in zwei Varianten: Eclipse und IntelliJ. Dies führte zu Problemen in der Weiterentwicklung, da aufgrund dieser Aufteilung, zwei unterschiedliche Projekte gepflegt werden mussten. Änderungen an der Geschäftslogik wurden meist nur im Eclipse-Plug-In vorgenommen, sodass das die Codeduplikate im IntelliJ-Plug-In veralteten. Eine genauere Problemanalyse folgt im Abschnitt 2.1 Probleme mit Ausgangszustand.

Um die einhergehenden Probleme zu bewältigen und zukünftige Entwicklungsprozesse von neu unterstützten IDEs zu verbessern, wurden Arbeiten

weshalb ich mich für diese Definition entschied: [Log]

[3] Projektwebseite: https://www.jetbrains.com/idea/

an der Kern-Komponente vorangetrieben. Diese Komponente soll sämtliche Geschäftslogik aus dem Eclipse- und IntelliJ-Plug-In extrahieren und in sich auslagern. Zukünftig würde jedes Plug-In-Modul nur noch aus IDE-spezifischen Code und Verwendung der Kern-Komponente bestehen. Änderungen an der Kern-Komponente werden automatisch auf alle Plug-Ins angewandt.

Die letzte direkte Arbeit an der Kern-Komponente[4] wurde Anfang 2015 in Form einer Bachelorarbeit durchgeführt (genaueres dazu im Abschnitt 1.4 Verwandte Abschlussarbeiten). Durch den Umfang von Saros, der großen Inkonsistenz zwischen Eclipse- und IntelliJ-Geschäftslogik und der beschränkten Bachelorarbeitszeit, konnte die Arbeit nicht sämtliche Codeduplikationen entfernen.

1.3 Motivation

Diese Bachelorarbeit soll die vorhergehende Arbeiten an der Kern-Komponente fortführen. Die meisten und einfachsten Codeduplikate wurden bereits verschoben. Meine Aufgabe es ist, die verbliebenen nicht-trivialen Codeduplikate zu verschieben, wobei größere und wichtigere Geschäftslogik priorisiert wird.

Die Motivation hierbei ist es, technische Schulden[5] mit Fokus auf die Kern-Komponente zu beseitigen. Direkte Profiteure sind dabei jetzige und zukünftige Entwickler an Saros, die von einem verbesserten Entwicklungsprozess profitieren. Endnutzer sollen von meiner Arbeit keine Auswirkungen wahrnehmen, mit Ausnahme von indirekten Einflüssen, wenn etwa die IntelliJ-Plug-In-Entwicklung beschleunigt wird.

1.4 Verwandte Abschlussarbeiten

Als direkter Vorgänger meiner Arbeit an der Kern-Komponente ist die Bachelorarbeit *Refaktorisierung des Eclipse Plugins Saros für die Portierung auf andere IDEs* [Las15] vom April 2015 zu sehen. Diese widmete sehr viel Zeit der Problemanalyse mit den Codeduplikaten und stellte einen strukturierten Plan auf, in deren Reihenfolge duplizierte Klassen entfernt wurden. Meine Arbeit kann als Fortsetzung dieser Bemühungen angesehen werden.

In der Masterarbeit *Entwicklung einer IDE-unabhängigen Benutzeroberfläche für Saros* [Boh15] wird die Benutzeroberfläche von den Plug-Ins abstra-

[4] Gemeint sind Abschlussarbeiten, die an der Kern-Komponente stattfanden, um das Problem mit den Codeduplikaten anzugehen.

[5] Technische Schuld beschreibt die Reduktion von Codequalität zugunsten gesenkten Aufwandes. In der Konsequenz kann es zu erhöhtem Aufwand bei Weitereinwicklung jenes Codes kommen. [Fow03]

hiert. Die neue Saros-Oberfläche soll HTML-basierend sein und in jeder IDE gleich aussehen. In den Plug-In-Komponenten soll damit nur noch Code für die Einbettung der HTML-Oberfläche und spezifische API verwendet werden, wobei der Rest aus Kern- und UI-Komponente kommt. Diese Arbeit kam ebenfalls mit der Kern-Komponente in Berührung.

Die Masterarbeit *Entwicklung und Evaluation eines unabhängigen Sitzungsservers für das Saros-Projekt* [Was16] extrahiert die Host[6]-Logik aus den Saros-Plug-Ins in eine eigene Komponente. In dieser Komponente befindet sich ein Server, der die meiste Geschäftslogik übernimmt und die Clients (unabhängig von deren IDE) verwaltet. Es wurden viele noch ausgelassene Codeduplikate in die Kern-Komponente verschoben, um die Erstellung des Servers möglich zu machen. Diese Masterarbeit hat einen großen Beitrag zur Weiterentwicklung der Kern-Komponenten geleistet, wenn auch aus anderen Motiven.

Eine weitere (zu diesem Zeitpunkt laufende) Bachelorarbeit ist *Erweiterung der HTML-Oberfläche in Saros* [Web16]. Diese setzt die Arbeit an der HTML-Oberfläche fort und kümmert sich um das neue Frontend. Meine Arbeit steht mit dieser insofern in Beziehung, dass ich Code zum Backend geliefert habe, um Schnittstellen zwischen Kern-Komponente und HTML-Frontend zu schaffen. Hauptsächlich beschäftigte sich meine Arbeit mit Aufräumarbeiten im Kern, doch mit den Beiträgen zum Backend habe ich auch Funktionalitäten für Saros erschaffen.

1.5 Aufbau dieser Arbeit

In den vorherigen Abschnitten wurde das Saros-Projekt vorgestellt, ein Überblick zu dem Ausgangszustand gewährt, die Motivation dieser Arbeit erklärt und in Relation stehende Abschlussarbeiten genannt. Es folgt nun eine grobe Vorschau der folgenden Abschnitte:

Problemstellung, Anforderungen und Zielsetzung zeigt die Probleme mit dem Ausgangszustand auf und wodurch diese entstanden sind. Außerdem nennt es die Anforderungen an diese Arbeit und dessen Umfang.

Vorgehensweise beschreibt den Saros-Entwicklungsprozess und wie die zuvor genannten Probleme technisch gelöst werden sollen. Benutzte Entwurfsmuster werden hier genannt.

Umsetzung dokumentiert die geleistete technische Arbeit. Dabei wird mehr auf den Entscheidungsprozess eingegangen, als die tatsächliche

[6] Übernimmt die Verantwortung u.a. fürs Verteilen und konsistent halten von Quelltext. Er ist der Mittelpunkt, welcher die Saros-Sitzung hält und alle Teilnehmer verwaltet.

Umsetzung in Quelltext.

Ergebnisse stellt die Ergebnisse in den Vergleich zu den gesetzten Anforderungen und Zielen. Zudem gibt es einen Ausblick auf das, was meiner Meinung nach dieser Arbeit noch zu tun ist. Außerdem folgt ein persönliches Fazit zum Gesamtergebnis.

Anhang enthält eine Liste aller Gerrit-Patches, die aus der Entwicklungsarbeit hervorgegangen sind. Außerdem eine Liste von verwandten Hyperlinks.

2 Problemstellung, Anforderungen und Zielsetzung

2.1 Probleme mit Ausgangszustand

Codeduplikate sind eine technische Schuld, die die Plug-In-Entwicklung ausbremst. Änderungen an dem Eclipse-Plug-In, die als Codeduplikat vorliegen, müssen auch im IntelliJ-Plug-In übernommen werden. Häufig wurde das IntelliJ-Plug-In ausgelassen, wodurch sich Original und Duplikat immer mehr voneinander unterschieden und das Problem verschlimmerten.

Weiterentwicklungen und Verbesserungen flossen so nicht mehr in das IntelliJ-Plug-In zurück, wodurch es stellenweise, trotz laufender Implementierung fehlender Funktionalitäten, veraltete. Je älter das Duplikat wurde, desto schwerer wurde die spätere Refaktorisierung[7].

Auch ist eine schwach ausgebaute Kern-Komponente hinderlich für die Entwicklung neuer Plug-Ins. Soll von Saros eine neue Entwicklungsumgebung unterstützt werden, so stehen zu wenig Klassen zur Wiederverwendung zur Verfügung. Dies kann zu erneutem Anlegen von Codeduplikaten verleiten.

2.2 Anforderungen an diese Arbeit

In dieser Arbeit wurden zwei Ziele verfolgt, die jeweils unterschiedliche Anforderungen hatten.

Der erste Teil umfasst die Refaktorisierung und Verbesserung der Kern-Komponente. Hier sind die Anforderungen, dass keine sichtbaren Änderungen für den Endnutzer entstehen sollen und möglichst bestehender Code nicht in ihrem Verhalten verändert wird. Dies soll sicherstellen, dass mit meiner Arbeit allein die Codequalität verbessert wird, ohne bisherige Funktionalität zu verändern. Komponenten- und Integrationstests können helfen, das Einhalten dieser Anforderungen zu überprüfen.

Der zweite Teil ist die Implementierung neuer Funktionalität. Ich half einer verwandten Abschlussarbeit, indem ich benötigte und noch fehlende Schnittstellen lieferte. Genauere Beschreibung dieser Aufgabe im späteren Abschnitt 4.5 Erweiterung von HTML UI backend. Die Anforderungen umfassten hier, bei der Implementierung das bisherige Architekturdesign beizubehalten und die Priorisierung mit der anderen Abschlussarbeit abzustimmen.

Für die unterschiedlichen Aufgaben und Anforderungen kann die Metapher

[7] Eine Umstrukturierung von Quelltext, die technische Schulden abbaut, ihn besser lesbar macht und dabei das wahrnehmbare Verhalten nicht verändert. [Fow04a]

der zwei Hüte genutzt werden. Es gibt einen Hut fürs Refaktorisieren, dass bestimmte Anforderungen erzwingt und einen entsprechenden Hut fürs Implementieren. Je nach Aufgabe wird der aufgesetzte Hut gewechselt, jedoch können niemals beide Hüte zeitgleich aufgesetzt werden. [Fow14]

In meinem Fall konnte ich für den ersten Teil nur den Refaktorisierungshut tragen, im zweiten Teil nur den Implementierungshut. Der kommende Abschnitt 4 Umsetzung beschreibt Vorfälle, in denen Entscheidungen getroffen wurden, die diese Idee strikt einhielten.

2.3 Arbeitsumfang und Ziele

Innerhalb der dreimonatigen Bearbeitungszeit wurde zum Ziel gesetzt, möglichst viele Codeduplikate zu entfernen. Die Priorität wurde auf Klassen gelegt, dessen Verschieben in den Kern zwar aufwändig ist, jedoch einen hohen Nutzen erzielt.

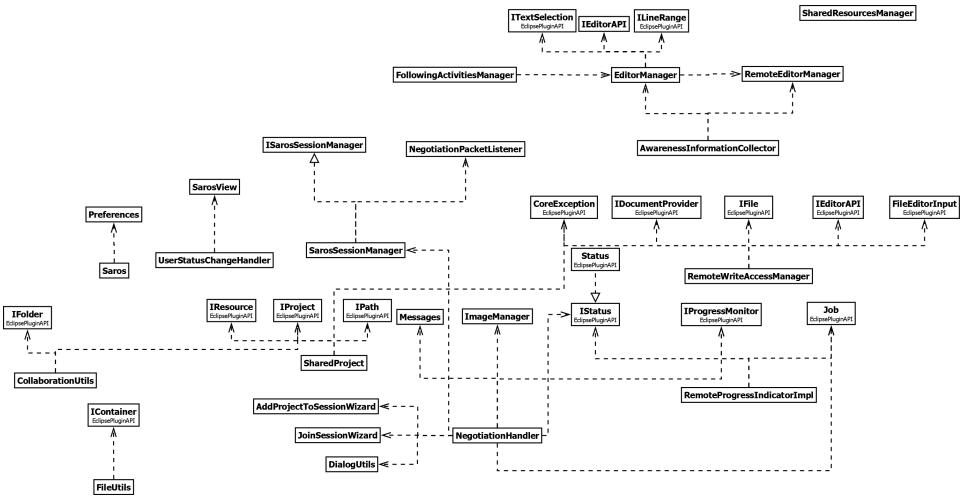

Abbildung 1: Auszug einer angefertigten Übersicht über verbliebene Codeduplikate. Zu sehen sind die vielen Abhängigkeiten zwischen den duplizierten Klassen, die ein Verschieben erschweren.

So stand zu Beginn nicht genau fest, welche Codeduplikate angegangen werden sollen. Zunächst war eine Analyse der bisherigen Lage notwendig. Die Wahl fiel auf das Verschieben vom `RemoteEditorManager`, von den `ConsistencyWatchdogs` und eine Refaktorisierung der Starten-/Beendenlogik von Saros.
Danach wechselte der Fokus auf die Unterstützung der HTML-Oberflächen-Entwicklung. Hier sollte ich die zeitnah nötigsten Implementierungen nachliefern, wobei die Wahl auf die Annahme von Sitzungsanfragen in Form von Dialogen fiel.

3 Vorgehensweise

3.1 Saros-Entwicklungsprozess

Der Projekt-Quelltext steht unter der GNU General Public License [Fou07] öffentlich zur Verfügung.

3.1.1 Team

Zwar kann jeder[8] an diesem Projekt mitentwicklen, doch die meisten Beiträge stammen aus Abschlussarbeiten von Studenten der Freien Universität Berlin.

Da die Abschlussarbeiten zu verschiedensten Zeitpunkten starten und dessen Bearbeitungszeiten variieren, wechselt das vorhandene Studententeam häufig. Aus der AG Software Engineering nehmen weitere Personen an der Entwicklung teil. Sei es durch Koordination und Betreuung der laufenden Abschlussarbeiten und/oder Weiterentwicklung des Codes. Die Arbeitsgruppe stellt die notwendigen Ressourcen zur Verfügung, wie etwa einen separaten Arbeitsraum fürs Saros-Team.

Pro Woche findet zweimal ein Standup mit den Studenten statt. Dieses Standup ähnelt dem Daily Scrum Meeting [Yip16]: Teilnehmer stehen im Kreis, berichten von ihrer Arbeit und halten sich möglichst kurz. Es wird eine Zeitspanne von maximal 15 Minuten anvisiert.

3.1.2 Java-Plattform und IDEs

Saros ist in der Programmiersprache Java geschrieben und wird jeweils mit Eclipse und IntelliJ entwickelt. Diese IDEs werden verwendet, um die entsprechenden Saros-Plug-Ins schreiben und testen zu können. Für die Entwicklung der Kern-Komponente ist keine spezielle IDE vorausgesetzt.

3.1.3 Reviewprozess mit Gerrit

Es werden keine Änderungen am Quelltext akzeptiert, die nicht den Reviewprozess von Saros durchlaufen. Gerrit[9] dient als Software für die Durchführung der Reviews durch Integration mit der Versionsverwaltung Git[10] und einer Weboberfläche.

Mitarbeiter an Saros laden ihre Änderungen auf Gerrit hoch, welche dann

[8] Für die Arbeit am Code ist eine Teilnahme am Reviewprozess zwingend notwendig, wofür das Saros-Team kontaktiert und um Erlaubnis gebeten werden muss.

[9] Projekthomepage: https://www.gerritcodereview.com/

[10] Projekthomepage: https://git-scm.com/

von den anderen Mitarbeitern begutachtet, bewertet und diskutiert werden. Erreicht die Änderung eine Akzeptanz, fließt diese erst dann durch Git in das Projekt ein. Vorgeschlagene Änderungen können mehrere Iterationen durchlaufen, bis ein akzeptables Ergebnis vorliegt.

Es sind alle Mitarbeiter aufgefordert, auf eine ihrer Änderungen zwei Reviews fremder Änderungen durchzuführen, um den Reviewprozess am laufen zu halten. Auch meine Arbeitsergebnisse liefen durch diesen Prozess. In dieser Bachelorarbeit werden meine Änderungen als *Gerrit-Patch* bezeichnet.

3.2 Allgemeines Vorgehensmuster

Zunächst muss sich klar gemacht werden, um welche Tätigkeit es sich handelt. Ist es eine Refaktorisierung, muss ein bestimmtes Vorgehen angewandt werden: Analysiere den vorhandenen Code. Was soll er tun und wie erreicht er dies? Stelle sicher, dass deine Umbauarbeit nicht den gewünschten Effekt des Codes verändert. Lege fest, was der Umfang der betroffenen Codestellen ist, um dann sicher zu gehen, dass die Annahmen[11] außerhalb dieses Umfangs korrekt bleiben.

Es werde beispielsweise eine Refaktorisierung innerhalb einer Klasse vorgenommen. Wird nun von außen erwartet, dass eine Klassenmethode einen Wert größer `0` zurückgibt, dann muss nach der Refaktorisierung diese Annahme weiterhin stimmen. Innerhalb der Klasse können Annahmen durch die Refaktorisierung verändert werden.

Für meine Arbeit bedeutete dies vor allem das Beibehalten von UI-Funktionalität. Wenn Codeduplikate an die Oberfläche von Plug-Ins gebunden sind, muss ein Weg gefunden werden, diese exakten Interaktionen beizubehalten.

3.2.1 Technische Entwurfsmuster

In meiner Arbeit wurde eine Reihe von Entwurfsmustern im Rahmen der Refaktorisierung verwendet. Entwurfsmuster sind wiederverwendbare Lösungen, die angepasst auf bestimmte Probleme angewandt werden können. [GHJV94] Mit ihnen können technische Schulden mit bewährten Mustern behoben und die Codequalität insgesamt erhöht werden. Es folgt eine Aufzählung von den wichtigsten in dieser Arbeit angewandter Muster:

Singleton stellt sicher, dass genau ein oder kein Exemplar einer Klasse existieren kann. Dies war nützlich für eine von mir eingebaute Restriktion, die das mehrfache Erstellen des Saros-Kontext verhindern soll.

[11] Gemeint sind bspw. Abfragen, die nur einen bestimmten Wertebereich überprüfen, weil der Entwickler Annahmen zu möglichen Eingabewerten hatte.

Fassade stellt eine vereinfachte Schnittstelle für darunterliegenden Code bereit. Mein in Saros eingeführter *Lebenszyklus* ist eine solche Facade, die den Umgang mit dem Saros-Kontext auf ein einfaches Starten und Stoppen reduziert.

Schablonenmethode definiert eine Methode, dessen Teilschritte durch andere Methoden bereitgestellt werden können. Die von mir eingeführte `AbstractSarosLifecylce`-Klasse nutzt dieses Muster.

Entwurfsmuster erwiesen sich für mich als besonders hilfreich, als es darum ging, Erweiterungen oder Restriktionen einzubauen.

4 Umsetzung

4.1 Aufteilung in Arbeitspakete

Meine Arbeit wurde in Pakete aufgeteilt, wobei jedes Paket einem größeren Vorhaben entspricht. Ein solches Arbeitspaket umfasst beispielsweise das Verschieben einer größeren Klasse in die Kern-Komponente. Durch die vielen Referenzen können weitere Klassen betroffen sein. Von kleinen Änderungen, wie etwa das Anpassen von Referenzen, bis hin zu kompletten Überarbeitungen.

Die von mir gewählten Arbeitspakete werden als groß angesehen. Groß bedeutet hier, dass sehr viel Kontextwissen, Anwendung von Entwurfsmustern und Zeit benötigt wird, um dessen Ziel zu erreichen. Automatisierte Refaktorisierungen durch Werkzeuge wurden genutzt, reichen aber alleine nicht für die Bearbeitung aus.

Arbeitspakete mit der Tätigkeit *Verschieben* haben den Hintergrund, dass Klassen in den Plug-In-Komponenten existieren, die eigentlich in die Kern-Komponente gehören. Dabei liegen oft Codeduplikate dieser Klassen im IntelliJ-Plug-In vor. Der Abschnitt *Verbesserung [...] der Core-Bestandteile von Saros* des Arbeitstitels bezieht sich auf diese Tätigkeit.

Die Teilaufgaben eines Arbeitspaketes werden in Form von Gerrit-Patches für den Reviewprozess bereitgestellt. In den folgenden Abschnitten werden Infotabellen, wie dieses Beispiel rechts, angezeigt.

[NOP] Making the tutorial	
Status	Abandoned
Erstellt am	15.10.2015
Geändert am	15.10.2015
Umfang	+3, -0

In der ersten Zeile wird der Name des Patches, wie er auf Gerrit zu lesen ist, angezeigt. Die jeweilige URL kann im Anhang nachgelesen werden.

Es wird drei mögliche Status geben: *Open* (Reviewprozess noch nicht abgeschlossen), *Merged* (Review bestanden und in Saros eingeflossen) und *Abandoned* (vom Autor zurückgezogen).

Erstellt am ist das Datum des erstmaligen Hochladen des Gerrit-Patches. *Geändert am* nennt das letzte Datum, an dem der Patch bearbeitet wurde. Das kann eine neue Iteration, Kommentare von Gutachtern oder der Wechsel zum *Merged*-Status sein.

Der Umfang nennt die Anzahl der hinzugefügten (+) und entfernten (-)

Quelltextzeilen. Eine relativ neutrale oder negative Summe deutet auf entfernte Codeduplikate durch Refaktorisierung hin.

4.2 Verschieben von RemoteEditorManager

Der `RemoteEditorManager` speichert die Zustände der Quelltext-Editoren anderer Teilnehmer zwischen. Dies beinhaltet Eigenschaften wie etwa alle geöffneten Editoren, den sichtbaren Textausschnitt oder der selektierte Text.

Bei der Paarprogrammierung ist es prinzipbedingt, dass beide Entwickler am gleichen Rechner die gleiche Oberfläche der IDE sehen. Wenn lokal an Quelltext gearbeitet wird, navigieren und bearbeiten beide Teilnehmer zwangsweise mit dem selben Editor. Dies ist eine wichtige Eigenschaft, die von Saros auf die verteilte Paarprogrammierung abgebildet wird. Die Teilnehmer können die Bearbeitung des jeweils anderen in Echtzeit sehen, beispielsweise in Form von Textselektionen.

Muss der Code plattformspezifisch sein?

Nein, weder aus design-, noch programmiertechnischen Gründen. Für die meisten Entwicklungsumgebungen hat sich das Konzept von mehreren Quelltext-Editoren durchgesetzt. Für die von Saros unterstützten IDEs (Eclipse und IntelliJ) trifft dies definitiv zu. Es muss designtechnisch also nicht plattformspezifisch sein.

Da der RemoteEditorManager nur sehr generelle Eigenschaften verwaltet, kann der Code dahingehend umgeschrieben werden, dass dieser sowohl für Eclipse, als auch IntelliJ passt.

Gehört der Code in die Kern-Komponente?

Ja, denn die Verwaltung der Quelltext-Editoren ist auch eine Netzwerkaktivität. Der Host der Saros-Sitzung hat die Aufgabe, die Quelltext-Editor-Zustände aller Teilnehmer konsistent zu halten. Eine solche Verwaltungstätigkeit gehört in den Kern und nicht in die Oberfläche[12].

Was muss getan werden?

Die Klasse `RemoteEditorManager` hält den sichtbaren Textbereich in der `LineRange`-Klasse und den selektierten Textbereich in der `TextSelection`-Klasse fest. Im Eclipse-Plug-In wurde mit Klassen aus der Eclipse-API gearbeitet, wobei das IntelliJ-Plug-In eine eigene, gleichnamige Implementierung nutzte. Damit der `RemoteEditorManager` verschoben werden kann, müssen

[12] Gemeint sind die Saros-Plug-Ins, welche nur eine Schnittstelle zur IDE sein sollen.

zunächst diese beiden plattformspezifischen Klassen bearbeitet werden. Alle anderen referenzierten Klassen sind bereits vom Kern aus erreichbar, sodass kein weiterer Aufwand entsteht.

Es stellte sich bei genauerer Betrachtung heraus, dass die Implementierung von `LineRange` und `TextSelection` trivial ist. Sie beide sind simple Datenbehälter, die auch generisch für den Kern implementiert werden können. Sobald die Neuimplementierung im Kern liegt, müssen nur noch alle Referenzen von den alten auf die neuen Klassen umgestellt werden. Dies ist leicht möglich, da sich die Signaturen[13] der Neuimplementierungen nicht von den Vorgängern unterscheiden. Zusätzlich fallen für das IntelliJ-Plug-In die nun obsoleten Klassen weg.

Nachdem `LineRange` und `TextSelection` im Kern liegen, kann der `RemoteEditorManager` problemlos mit in den Kern verschoben werden. Das Codeduplikat vom IntelliJ-Plug-In ist, mit Ausnahme der Referenzen, identisch mit dem Original, sodass keine weiteren Anpassungen notwendig sind und das Duplikat entfernt werden kann.

Wie kann diese Aufgabe aufgeteilt werden?

Da die Verschiebung von `RemoteEditorManager` nur noch von den Klassen `LineRange` und `TextSelection` aufgehalten werden, bietet sich folgende Einteilung für den Gerrit-Reviewprozess an:

LineRange: Implementiere diese Klasse neu, lege sie in den Kern und passe alle Referenzen an.

TextSelection: Analog zu LineRange.

RemoteEditorManager: Verschiebe die Klasse in den Kern und entferne Codeduplikat aus dem IntelliJ-Plug-In.

4.2.1 LineRange

Eine Neuimplementierung von `LineRange` ist simpel, denn es werden nur zwei Werte festgehalten. Diese werden mit dem Konstruktor gesetzt und können von da an nur noch durch getter-Methoden abgerufen werden. Der Quelltext umfasst somit nur 22 Zeilen.

[API] Move LineRange to core	
Status	Merged
Erstellt am	12.11.2015
Geändert am	03.12.2015
Umfang	+65, -89

[13] Hier sind Signaturen von Konstruktoren und Methoden gemeint. [Kel03] Bleiben diese gleich, ist im Falle von Aufrufen keine Änderung von Quelltext nötig.

Die Signaturen von beiden zuvor verwendeten `LineRange`-Implementierungen sind identisch. Es bot sich somit an, auch für die Neuimplementierung diese Signaturen beizubehalten. Der Vorteil dessen ist, dass alle Verwendungsstellen nur auf die Neuimplementierung umreferenziert werden müssen, ohne weiteren Quelltext anzupassen. In Java gelingt dieser Referenzaustausch mit der Anpassung der `imports` im Quelltextkopf.

4.2.2 TextSelection

Diese Teilaufgabe ist analog zu `LineRange`: Die Klasse dient nur zum Halten von zwei Werten, die simple Neuimplementierung behält die alten Signaturen bei und nur noch Referenzen müssen angepasst werden.

[API] Move TextSelection to core	
Status	Merged
Erstellt am	12.11.2015
Geändert am	08.12.2015
Umfang	+109, -89

Ein interessanter Vorfall während der Bearbeitung dieser Aufgabe war, dass ein Defekt im bisherigen Quelltext entdeckt wurde. Zunächst das gewünschte Verhalten:
Saros versucht die Entwicklungsumgebungen der Teilnehmer möglichst synchron zu halten. Wenn ein Teilnehmer mit der Textselektion den sichtbaren Textbereich eines Teilnehmers überschreitet, so wird der Textbereich entsprechend gescrollt.[14] Somit haben alle Teilnehmer die vorgenommenen Selektionen im Blick.

Die Implementierung des obigen Verhaltens hatte jedoch den Mangel, dass die Überprüfung auf Notwenigkeit des Scrollens immer `true` als Ergebnis zurückgab. Hier wurde bei der Eclipse-Implementierung von `TextSelection` der Gebrauch von der Hilfsmethode `getEndLine` gemacht. Diese soll zurückgeben, in welcher Zeile sich das letzte Zeichen des selektierten Textes befindet.

Doch während der Arbeiten fand ich heraus, dass dieser Aufruf in jedem Fall `-1` zurück gab und nicht die erwartete Zeilennummer. Das lag daran, dass im Saros-Quelltext nicht der richtige Konstruktor der Klasse verwendet wurde, um die `getEndLine`-Hilfsmethode verwenden zu können (Rückgabewert `-1` signalisiert Fehler). In Folge dieser falschen Annahme, wird in Saros ein if-Block jedes Mal `true` zurückgeben, da `-1` als Zeilennummer die Bedingungen immer erfüllt.

Der Schaden dieses Defektes ist minimal. Denn sollte die Scrollen-Methode unnötigerweise aufgerufen werden, tritt kein Scrollen und somit kein sichtbarer Effekt ein. Es ist höchstens eine kleine Verschwendung von Rechenzeit,

[14] Diese Funktionalität ist nur im `FollowMode` aktiv.

weshalb dieser Fehler auch nicht früher wahrgenommen und/oder entdeckt wurde.

Da dieser entdeckte Defekt keinen nennenswerten Schaden brachte und eine Behebung deutlich höheren Zeitaufwand erforderte, entschied ich mich, dieses Problem zu kennzeichnen und das defekte Verhalten beizubehalten. Die betroffene Codestelle wurde mit erklärenden Kommentaren versehen und der Aufruf von `getEndLine` entfernt. Um keine Verhaltensänderung des folgenden Quelltexts zu provozieren, wurde dort der Wert `-1` hart eincodiert.

Hier wurde deutlich, welchen Einfluss die gestellten Anforderungen auf meine Arbeit hatte. Es hieß, dass Refaktorisierung weder wahrnehmbare Verhaltensänderungen hervorrufen solle, noch fremde Progammabläufe zu stören habe.
In dem Arbeitspaket *Verschieben von RemoteEditorManager* sollen Klassen in die Kern-Komponente verschoben werden, ohne bisheriges Verhalten zu verändern. Beim Auffinden des oben genannten Defekts bin ich sicher gegangen, dass trotz meiner Umbauarbeiten der fehlerhafte Programmablauf gleich bleibt. Die Fehlerbehebung muss in einer separaten Aufgabe stattfinden.

4.2.3 RemoteEditorManager

Zuletzt musste nur noch der `RemoteEditorManager` in die Kern-Komponente verschoben werden. Hierzu konnte ich das Refaktorisierungswerkzeug von Eclipse nutzen, um die Klasse automatisiert aus dem Eclipse-Plug-In in die Kern-Komponente zu verschieben. Dies funktionierte problemlos, da keinerlei Referenzen auf die Plug-Ins (wie z.B. `LineRange` oder `TextSelection`) mehr bestanden. Das Codeduplikat im IntelliJ-Plug-In konnte entfernt werden.

[API] Move RemoteEditorManager to core	
Status	Merged
Erstellt am	12.11.2015
Geändert am	08.12.2015
Umfang	+2, -387

Hier wurde beispielhaft Quelltext durch Löschen verbessert und der von Gerrit berechnete Umfang reflektiert dies sehr schön: 2 hinzugefügte und 387 entfernte Quelltextzeilen. Verschobener Code wird nicht mitgezählt.

4.3 Verschieben von ConsistencyWatchdogs

Um die Arbeit mit dem gleichen Quelltext verteilt möglich zu machen, muss Saros die Quelltextinhalte aller Teilnehmer konsistent halten. Dies wird mit einer Implementierung des Jupiter-Algorithmus [NCDL95] erreicht. Er garantiert, dass das Endergebnis gleich bleibt, unabhängig von der Reihenfolge der eingehenden Änderungen.

Dies funktioniert zuverlässig, solange keine Änderungen von außen vorgenommen werden, welche der Jupiter-Algorithmus nicht abdeckt. Ein solches Ereignis könnte das Ändern des Quelltextes außerhalb des IDE-Editors sein.

Um zu verhindern, dass die Quelltexte unter den Teilnehmern inkonsistent werden, hat Saros ein Sicherheitsnetz eingebaut: Es überprüft in regelmäßigen Abständen die Prüfsumme offener Quelltexte und verschickt die Dateien des Gastgebers an jene Teilnehmer, bei denen eine Abweichung entdeckt wurde. Folgende Klassen sind an dieser Aufgabe hauptsächlich beteiligt: `ConsistencyWatchdogSever`, `ConsistencyWatchdogClient`, `ConsistencyWatchdogHandler` und `DocumentChecksum`.

Muss der Code plattformspezifisch sein?

Nein. Alle hierfür notwendigen plattformspezifischen Elemente, wie etwa die Verwaltung von Dateien, können abstrahiert und in den Kern gelegt werden. Zum Teil ist dies bereits geschehen.
Die Erstellung von Prüfsummen und der Dateiaustausch zwischen den Teilnehmern ist etwas, das nicht fest von den Plug-In-APIs abhängig ist.

Gehört der Code in die Kern-Komponente?

Ja, denn auch hier geht es um Netzwerkaktivitäten zwischen Gastgeber und Teilnehmern, welche über alle unterstützten IDEs gleich funktionieren sollen.

Was muss getan werden?

Die betroffenen Klassen sind `ConsistencyWatchdogSever`, `ConsistencyWatchdogClient`, `ConsistencyWatchdogHandler` und `DocumentChecksum`. Diese müssen in die Kern-Komponente verschoben werden, wobei es größere Hindernisse gibt. Auf den ersten Blick finden die Lese- und Schreiboperationen in diesen Klassen statt. Und diese sind bisher plattformspezifisch implementiert. Zu Beginn war die Erwartung, eine Abstraktion dieser Operationen erstellen zu müssen.

4.3.1 ConsistencyWatchdogServer und DocumentChecksum

Diese Teilaufgabe durchlief mehrere Iterationen. Zuerst umfasste es allein das Verschieben von `DocumentChecksum`, einer Klasse, die von einer Quelltextdatei den relativen Pfad und die Prüfsumme festhält.

[API] Move WatchdogServer and DocumentChecksum to core	
Status	Merged
Erstellt am	19.11.2015
Geändert am	13.12.2015
Umfang	+6, -427

In der nächsten Iteration wurden auch `ConsistencyWatchdogServer` und `IsInconsistentObservable` mit verschoben. Durch Washingtons Abschlussarbeit [Was16] wurden bereits alle Klassen, die von `ConsistencyWatchdogServer` referenziert werden, in den Kern verschoben. `ConsistencyWatchdogServer` lag aber noch im Eclipse-Plug-In und als Codeduplikat im IntelliJ-Plug-In vor. Das Verschieben war ohne weiteres möglich.

Während dieser Iteration wurde festgestellt, dass ein Defekt in `ConsistencyWatchdogServer` existiert. Dieser wurde entdeckt, da die *Sonarqube QA*-Software eine Analyse des Quelltexts vornahm und eine fragwürdige Abfrage vorfand: Es wurde auf einer generischen Menge `Set<SPath>` die Methode `contains` aufgerufen. Leider benutzt Java für diese Methode nicht wie erwartet den Datentyp `SPath`, sondern verlangt den Basistyp `object`. So ist es plausibel, dass der Entwickler nicht bemerkte, wie der `contains`-Methode kein `SPath` übergeben wurde. Die Abfrage gab immer `false` zurück. Zunächst wurde dieses Problem angesprochen und dessen Behebung aufgeschoben.

Eine dritte Iteration wurde nach einer Feststellung gestartet, dass `IsInconsistentObservable` nicht in dieser Teilaufgabe bearbeitet werden sollte, da es allein von `ConsistencyWatchdogClient` verwendet wird. Beide Klassen sollten in einem gemeinsamen Patch abgearbeitet werden. Von hier an sind nur noch `ConsistencyWatchdogServer` und `DocumentChecksum` in dieser Teilaufgabe enthalten.

Der weitere Verlauf der Iterationen der Gerrit-Patches betraf nur noch kleinere Verbesserungen wie etwa Dokumentation und Quelltextformatierungen. Ein weiterer Fehler fiel in `DocumentChecksum` auf: Hier werden Prüfsummen als Werttyp `int` errechnet. Wenn keine Prüfsumme vorliegt, wird `-1` zurück gegeben. Problematisch ist die Tatsache, dass die Prüfsummenfunktion ebenfalls eine `-1` berechnen kann. Bei einem solchen Wert würde Saros davon ausgehen, dass die Prüfsumme gar nicht vorliege, was zu Fehlern im Programmablauf führt.

Da dieser Fehler bereits vorher bestand und dessen Eintrittswahrscheinlichkeit sehr gering ist (1 zu 2^{32}), fiel die Entscheidung, diese Behebung in einem anderen Patch durchzuführen. Dieser Patch soll sich nur um das Verschieben von Klassen in die Kern-Komponente kümmern. Diese Entscheidung ähnelt der Begründung und dem Vorgehen aus dem Kapitel 4.2.2 TextSelection.

4.3.2 ConsistencyWatchdogClient

Es wurden `ConsistencyWatchdogClient` und `IsInconsistentObservable` in die Kern-Komponente verschoben. Und auch hier hat Washingtons Vorarbeit [Was16] dafür gesorgt, dass alle referenzierten Klassen bereits im Kern vorliegen, weshalb ein Verschieben nicht weiter aufwändig war.

[API] Move WatchdogClient to core	
Status	Merged
Erstellt am	09.12.2015
Geändert am	17.12.2015
Umfang	+10, -491

Die Durchführung entsprach dem bisherigen Vorgehen: Mache alle referenzierten Klassen im Kern erreichbar (bereits gegeben), verschiebe die Klasse und passe alle eingehenden Referenzen an. Lösche zuletzt das Codeduplikat aus dem IntelliJ-Plug-In.

4.3.3 ConsistencyWatchdogHandler

Den `ConsistencyWatchdogHandler` zu Verschieben erwies sich zu Beginn als schwer. Denn diese Klasse kümmert sich um das Austauschen inkonsistenter Quelltextdateien und die Fehlerausgabe an den Benutzer. Und das wiederum sind plattformspezifische Tätigkeiten in den Plug-Ins.

[API] Move WatchdogHandler to core	
Status	Abandoned
Erstellt am	09.12.2015
Geändert am	16.12.2015
Umfang	+230, -329

Als erste Iteration wurde der `ConsistencyWatchdogHandler` aufgeteilt: Eine abstrakte Klasse `AbstractConsistencyWatchdogHandler` im Kern und die implementierenden Klassen `EclipseConsistencyWatchdogHandler` und `IntelliJConsistencyWatchdogHandler` in den jeweiligen Plug-Ins.

Die Idee dahinter war es, sämtlichen unabhängigen Code in die abstrakte Klasse zu legen. Alle plattformspezifischen Zugriffe, wie Dateioperationen oder Fehlermeldungen, würden dann von den konkreten Klassen implementiert werden müssen. Doch dieses Vorgehensmuster wurde in einer nächsten Iteration verworfen.

Anstatt abstrakte Klassen einzuführen, wurden Schnittstellen eingeführt und andere Klassen angepasst, um vom Kern aus Dateioperationen und Fehlermeldungen erreichbar zu machen. Doch auch diese Lösung war nicht zufriedenstellend.

So war der Code immer noch sehr weit voneinander verteilt. Eine neue Schnittstelle `IRecoveryProgressUI` lag im Kern, die Implementierungen `EclipseRecoveryProgressUI` und `IntelliJRecoveryProgressUI` in den Plug-Ins. Exemplare dieser Klassen wurden wiederum in anderen Klassen außerhalb des Kerns erstellt und viele weitere solcher Codeverflechtungen. Es war klar, dass meine Lösung in dieser Form nicht akzeptabel ist, da sie sehr komplex und schwer zu warten ist.

In Absprache mit den Teilnehmern des Gerrit-Reviewprozesses, wurde entschieden, die UI-Interaktionen, also Fehlermeldungen an den Benutzer, auszubauen. Dieses Verhalten sei schon von Beginn an unverständlich implementiert und bringe kaum Mehrwert. Ein Mitarbeiter ergriff sofort die Initiative und erstellte einen Patch[15], noch ehe ich meine Lösung anpassen konnte.

Außerdem stellte selbiger Mitarbeiter fest, dass die Dateioperationen im `ConsistencyWatchdogHandler` redundant sind und an ganz anderer Stelle ausgeführt werden. Dieser Code kann also ebenfalls entfernt werden. Dies war ein Fakt, welcher durch meine Arbeiten nicht hätte in Erfahrung gebracht werden können, da er auf Hintergrundwissen anderer Saros-Mitarbeiter beruht.

[15] Siehe [INTERNAL] ConsistencyWatchdogHandler - remove UI access auf Gerrit

Anstatt den vorherigen Patch weiter zu iterieren entschied ich mich, einen komplett neuen Patch anzulegen. Denn der Ausgangszustand hat sich während der Bearbeitung geändert und meine vorherigen Lösungswege sind nicht mehr von Relevanz.

[API] Move WatchdogHandler to core and update core context	
Status	Merged
Erstellt am	13.01.2016
Geändert am	28.01.2016
Umfang	+44, -296

Mit dem neuen Ausgangszustand, dass die UI-Interaktion entfernt wurde und die Dateioperationen obsolet sind, vereinfachte sich das Verschieben von `ConsistencyWatchdogHandler` stark.

Nun gab es keine vom Kern aus unerreichbaren Klassen mehr. Das Standardvorgehen wurde angewandt: Referenzierte Klassen angepasst, `ConsistencyWatchdogHandler` in die Kern-Komponente verschoben, alle eingehenden Referenzen aktualisiert und zuletzt das Codeduplikat aus dem IntelliJ-Plug-In entfernt.

4.4 Abstraktion vom Lebenszyklus

Saros macht Gebrauch vom Dependency Injection-Entwurfsmuster. Dieses Entwurfsmuster wird in der objektorientierten Programmierung genutzt, um Abhängigkeiten zwischen Objekten zur Laufzeit zu regeln. [Fow04b] Der Effekt ist, dass Objekte ihre Abhängigkeiten nicht selbst bei der Initialisierung erzeugen müssen, sondern von außen injiziert bekommen. Welche Abhängigkeiten wie und wo verwendet werden, kann an einem zentralen Punkt festgelegt werden.

Als Implementierung dieses Entwurfsmusters werden in Saros sog. PicoContainer[16] verwendet. Mit deren Hilfe wird das Saros-Projekt in Module aufgeteilt, die zur Laufzeit geladen werden. So kümmert sich `SarosCoreContextFactory` um das Erstellen aller benötigten Exemplare aus der Kern-Komponente, während `SarosEclipseContextFactory` sich um Exemplare aus dem Eclipse-Plug-In kümmert. Zur Laufzeit werden nun diese Factories benutzt, je nach Bedarf jener Plattform, auf welcher Saros läuft.

Welches Problem liegt vor?

Jede einzelne Plattform (Eclipse, IntelliJ, Server) hat eine eigene Implementierung für die Verwendung dieser PicoContainer. Eigentlich dürften diese identisch sein, der einzige Unterschied sind die geladenen Factories. Doch tatsächlich handelt es sich um leicht angepasste Codeduplikate.

Auch der Ort der Initialisierung ist nicht konsistent. Bei Eclipse ist diese in der Hauptklasse, welche der Einstiegspunkt des Eclipse-Plug-Ins ist. Im IntelliJ-Plug-In wird wiederum diese Logik in einer separaten Klasse ausgelagert. Und der Sever hat diese Logik direkt in der *main*-Methode.

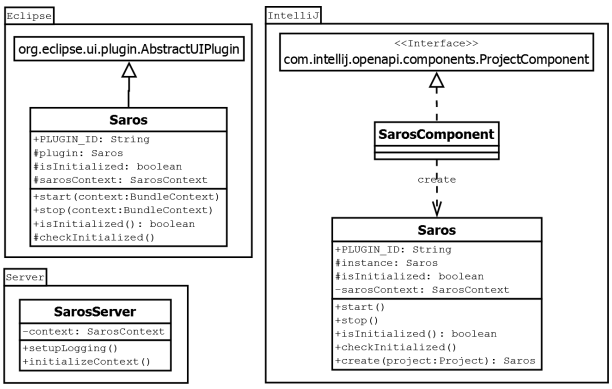

Abbildung 2: Der bisherige Zustand. Nur relevante Klassen sichtbar.

[16] Projektwebseite: http://picocontainer.com/

Die fehlende Konsistenz in der Platzierung dieser Logik macht es schwer, dessen Quelltext zu warten. In Abbildung 2 ist anhand der Klassennamen nicht zu erkennen, wo danach zu suchen ist.

Des weiteren ist problematisch, dass sämtliche Logik nicht von einem zentralen Punkt aus geändert werden kann. Änderungen müssen separat auf jede Implementierung angewandt werden.

Was ist der gemeinsame Nenner?

Alle Implementierung nutzen die `SarosContext`-Klasse, welche den PicoContainer beinhaltet und das Hinzufügen und Entfernen von Komponenten steuert. Komponenten werden durch die erwähnten Factories (z.B. `SarosEclipseContextFactory`) zusammengefasst, welche nun von den jeweiligen Plattformen (Eclipse, IntelliJ und Server) an den `SarosContext` übergeben werden müssen.

Außerdem gibt es zwei grundlegende Funktionen: Den `SarosContext` initialisieren und wieder aufzulösen.

Wie sieht eine Lösung aus?

Die Verwaltung dieser PicoContainer teilt sich in zwei Teile auf: Im ersten Teil wird die Logik fürs Initialisieren und Finalisieren des `SarosContext` bereitgestellt. Welche Factories dazu verwendet werden muss ein zweiter Teil übernehmen, der diese an den ersten Teil übergibt.

Dieser erste Teil gehört in die Kern-Komponente und der zweite Teil, welcher plattformspezifisch ist, existiert jeweils in den Saros-Plattformen. Es exisitert also ein abstrakter Teil, der durch konkrete Teile vervollständigt wird.

Es folgte die Designentscheidung, eine abstrakte Klasse im Kern anzulegen und dessen Implementierungen in die Plug-Ins bzw. den Server zu legen. Das verwendete Entwurfsmuster nennt sich Schablonenmethode.

Wie soll dieses Konstrukt benannt werden?

Die Dependency Injection ist sehr tief in Saros verankert. Saros kann weder gestartet, noch gestoppt werden, ohne Dependency Injection und die Klasse `SarosContext` zu benutzen. Es könnte gesagt werden, dass die Initialisierung und Finalisierung der Lebenszyklus von Saros ist. Als Entscheidung fiel, das englische Wort Lifecycle für die Namensgebung zu verwenden.

Die abstrakte Klasse im Kern wird `AbstractSarosLifecycle` und dessen Implementierungen nach dem Muster `{Plattformname}SarosLifecycle` benannt (z.B. `EclipseSarosLifecycle`).

Wie sieht die Lösung konkret aus?

Eine abstrakte Klasse namens `AbstractSarosLifecycle` wird in der Kern-Komponente angelegt. Diese beinhaltet ein `SarosContext`-Exemplar und regelt das Initialisieren und Finalisieren des PicoContainers. Dann werden die implementierenden Klassen `EclipseSarosLifecycle` und `IntellijSarosLifecycle` angelegt.

Bei diesem Vorhaben wurden nur diese zwei Plattformen beachtet, da der Server zu diesem Zeitpunkt noch in Arbeit war und andere Plug-Ins noch nicht zur Verfügung standen.

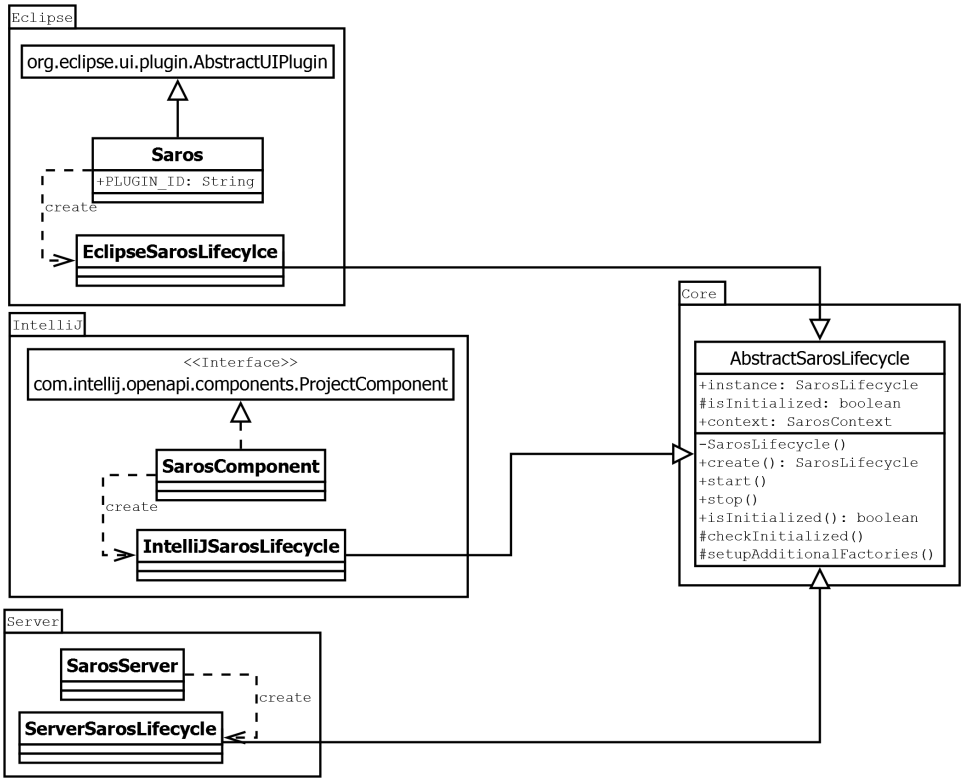

Abbildung 3: Entwurf der Plattformen mit den neuen `Lifecycle`-Klassen.

Der UML-Entwurf aus Abbildung 3 zeigt, wie das Einfügen der neuen `Li-`

`fecycle`-Klassen aussehen könnte.

Ist diese Arbeit getan, muss sämtliche alte Logik aus den Plattformen entfernt werden, die sich bisher um den `SarosContext` gekümmert haben. Stattdessen wird nur noch an den entsprechenden Stellen der Lebenszyklus gestartet oder gestoppt. Für das Eclipse-Plug-In wären dies jene Stellen, an denen Eclipse das Saros-Plug-In startet oder stoppt. Dies sind geeignete Zeitpunkte zum Steuern des Lebenszyklus.

4.4.1 AbstractSarosLifecycle

Die erste wichtige Designentscheidung ist, dass die Kern-Komponente selbst keine Implementierung von `AbstractSarosLifecycle` erhält. Denn diese Kern-Komponente ist *tot*, sie ist nur eine Sammlung von Klassen und Funktionalitäten, in welchen kein *Leben* steckt. Sie kann von alleine nichts tun, sondern nur unter Verwendung von lebendigen Komponenten, wie etwa der Saros-Plug-Ins, etwas bewirken.

[API] Create AbstractSaros-Lifecycle	
Status	Merged
Erstellt am	25.02.2016
Geändert am	03.03.2016
Umfang	+128, -0

Mögliche Bezeichnungen für die Kern-Komponente sind API oder Klassenbibliothek, in jedem Fall macht eine Implementierung von `CoreSarosLifecycle` keinen Sinn. Denn ohne Leben gibt es auch keinen Lebenszyklus.

Die `AbstractSarosLifecycle`-Klasse hat die folgenden Bestandteile:

SarosContext Die für den PicoContainer zuständige Klasse wird innerhalb der `Lifecycle` gehalten.

getSarosContext Eine getter-Methode, die das `SarosContext`-Exemplar nach außen zur Verfügung stellt.

start Startet den Lebenszyklus, indem alle Factories (mindestens die `SarosCoreContextFactory`) erstellt und dann dem `SarosContext` übergeben werden.

stop Fordert den `SarosContext` auf, sich und den PicoContainer aufzulösen, womit der Lebenszyklus stoppt.

Alle aufgezählten Methoden sind vollständig in der abstrakten Klasse implementiert und dürfen von den Implementierungen nicht überschrieben werden. Die unterbundene Veränderlichkeit ist eine Designentscheidung, die mit meiner defensiven Programmierung einhergeht.

Meine Annahme ist, dass die Methoden `getSarosContext`, `start` und `stop` auf allen Plattformen funktionieren und keine plattformspezifischen Änderungen von Nöten sind. Wenn es einen Grund zur Veränderung gäbe, etwa eine Fehlerbehebung oder Erweiterung, dann solle dies in der `AbstractSarosLifecycle`-Klasse vorgenommen werden, so dass diese automatisch auf alle Plattformen angewandt wird.

Folgende Methoden stehen den konkreten Klassen zum Implementieren zur Verfügung:

additionalContextFactories Diese Methode muss von der konkreten Klasse implementiert werden. Sie wird in der `start`-Methode aufgerufen, um alle plattformspezifischen Factories zu erhalten. Beispielsweise könnte die `SarosEclipseContextFactory` zurückgegeben werden.

initializeContext Optionale Methode. Diese wird innerhalb von `start` aufgerufen, sodass die konkrete Klasse diesen Einstiegspunkt für eigenen Code nutzen kann.

finalizeContext Analog zu `initializeContext` ist dies ein optionaler Einstiegspunkt innerhalb der `stop`-Methode.

Im besten Fall muss die konkrete Klasse, also die Implementierung von `AbstractSarosLifecycle`, nur `additionalContextFactories` implementieren und übernimmt den Rest aus der Basisklasse. Dies verschiebt die Verantwortung für den richtigen Umgang mit dem PicoContainer in die Kern-Komponente. Die Plattformen haben nur noch die notwendigen Factories zu übergeben, und den Lebenszyklus zu starten und zu stoppen.

Innerhalb dieses Gerrit-Patches wurde nur die abstrakte Klasse angelegt. Durch die Iterationen und den Rückmeldungen der Teilnehmer, haben sich in erster Linie die Dokumentation und die Methoden-Signaturen verändert. Das grundlegende Design von `AbstractSarosLifecycle` blieb erhalten.

Die Entwicklung der abstrakten Klasse lief parallel zu den konkreten Klassen `EclipseSarosLifecycle` und `IntellijSarosLifecycle`. So kam währenddessen die neue Anforderung hinzu, dass das Eclipse-Plug-In eine Methode aus `SarosContext` aufrufen muss. Zu diesem Zeitpunkt wurde die abstrakte Klasse so entworfen, dass der `SarosContext` nicht von außen erreichbar ist.

Hier musste eine Lösung gefunden werden. Meine Argumentationskette war, dass die `SarosContext`-Klasse ein fester Bestandteil vom `Lifecycle` ist und nicht von außen manipuliert werden dürfte. Ich berief mich auf das Geheimnisprinzip, welche meine defensive Programmierung rechtfertigen sollte.

Demnach implementierte ich eine Proxy-Methode[17], die nach außen genau die eine benötigte Methode `SarosContext#getComponent` zugreifbar machte, ohne das `SarosContext`-Exemplar erreichbar zu machen.

Für dieses Vorgehen gab es Kritik von den Teilnehmern des Gerrit-Reviewprozesses. Es hieß, dass dies eine unnötige Restriktion sei und es kein Problem wäre, das gesamte `SarosContext`-Exemplar von außen zugreifbar zu machen. Ich argumentierte weiter, dass dadurch der `Lifecycle` umgangen werden kann und durch destruktive Methodenaufrufe einen invaliden Zustand hinterlassen kann.

Doch meine defensive Programmierung sei hier deplatziert. Sie mache zwar Sinn über Softwaregrenzen hinweg (etwa APIs nach außen), doch innerhalb der Saros-Software sei dies nicht notwendig. Wie auch *null-checks* von Parametern nicht notwendig seien.
Ich akzeptierte diese Begründung und führte eine getter-Methode `getSarosContext` ein, wie sie zuvor beschrieben wurde.

4.4.2 EclipseSarosLifecycle

Diese Teilaufgabe verlief unproblematisch. Zuerst wurde die konkrete Klasse `EclipseSarosLifecycle` angelegt, welche von `AbstractSarosLifecycle` erbt. Sie wurde als Singleton implementiert.

[API] Extend AbstractSaros-Lifecycle (Eclipse)	
Status	Merged
Erstellt am	25.02.2016
Geändert am	29.03.2016
Umfang	+100, -71

Meine Entscheidung zur Verwendung eines Singletons lag darin begründet, dass es innerhalb des Eclipse-Plug-Ins keinen Sinn ergäbe, mehrere Kontexte und Lebenszyklen laufen zu lassen. Als Singleton ist garantiert, dass genau ein oder kein Exemplar von `EclipseSarosLifecycle` existiert.

Auch diese Entscheidung wurde von den Teilnehmern des Gerrit-Reviewprozesses kritisiert, mit der Begründung, dass dies eine unnötige Restriktion sei, da mehrere Lebenszyklen keinen Schaden verursachen könnten.

Bei einer genaueren Betrachtung der Verwendungsstellen der Eclipse-API fiel mir eine mögliche Bruchstelle auf, die das Singleton-Entwurfsmuster rechtfertigte: Aufrufe wie `getPreferenceStore` oder `getBundle` arbeiten mit der Eclipse-API zusammen und werden von Komponenten aus dem Lebenszyklus verwendet. Es war nicht abzusehen, welche Folgen es haben könnte,

[17] Hier wurde eine Methode geschrieben, die einfach den Rückgabewert von `SarosContext#getComponent` weitergibt.

wenn mehrere Komponenten parallel diese Aufrufe nutzen. Beispielsweise kann aus der Dokumentation jener Aufrufe nicht in Erfahrung gebracht werden, ob diese threadsicher[18] sind.

Die Implementierung des Singleton-Entwurfsmusters wird dadurch erreicht, dass der einzige Konstruktor auf die Sichtbarkeit `private` gesetzt wird und ein Exemplar nur mit der öffentlichen Methode `getInstance` zu erstellen ist. Diese Methode sorgt auch dafür, dass nie mehr als ein Exemplar erstellt werden kann. Zumindest von außerhalb der Klasse nicht.

Die folgenden Methoden aus der Basisklasse werden überschrieben:

additionalContextFactories Es wird die `SarosEclipseContextFactory` zurückgegeben. Sollte der SwtBrowser[19] eingeschaltet sein, werden zusätzlich `HTMLUIContextFactory` und `EclipseHTMLUIContextFactory` zurückgegeben.

initializeContext Hier wird `FeedbackPreferences#setPreferences` aufgerufen, was zwangsweise während der Initialisierung des Lebenszyklus passieren muss.

finalizeContext Analog zu `initializeContext` werden Aufrufe noch vor dem Stoppen des Lebenszyklus getätigt. In diesem Falle `SarosSessionManager#stopSarosSession` und `ConnectionHandler#disconnect`.

Zuletzt musste die `Saros`-Klasse angepasst werden. Alle Codestellen zum Umgang mit dem `SarosContext` wurden entfernt und durch Aufrufe des neuen `EclipseSarosLifecycle` ersetzt.

4.4.3 IntellijSarosLifecycle

Der Umfang dieser Teilaufgabe war hoch. Denn anstatt nur die neue Klasse `IntellijSarosLifecycle` anzulegen und dessen Aufrufe einzuprogrammieren, musste ein Fehler im bisherigen Design vom IntelliJ-Plug-In ausgebaut werden.

[API] Extend AbstractSarosLifecycle (IntelliJ)	
Status	Merged
Erstellt am	28.02.2016
Geändert am	29.03.2016
Umfang	+210, -277

Bisher befand sich die Logik für das Aufsetzen des PicoContainers innerhalb der `Saros`-Klasse im IntelliJ-Plug-In. Doch das ist nicht dessen einzige

[18] „A procedure is thread safe when the procedure is logically correct when executed simultaneously by several threads.“ [Mul15]

[19] Dies ist die Implementierung der HTML-Oberfläche, an welcher Bohnstedt [Boh15] zuletzt arbeitete. Da diese noch in Entwicklung ist, muss sie manuell in der Entwicklungsumgebung eingeschaltet werden.

Aufgabe, es hält auch noch Exemplare von `Project` und `Workspace` bereit, welche von anderen Komponenten erfragt werden.

Und in der Art und Weise, wie auf diese zwei Exemplare Zugriff gegeben wird, offenbart eine zweifelhafte Designentscheidung. Denn die `Saros`-Klasse wird selbst an den PicoContainer übergeben, um per Dependency Injection für andere Klassen erreichbar zu sein.

Das heißt also, dass jene Klasse, die das Initialisieren und Zerstören der PicoContainer aufruft, wiederum selbst Komponente vom PicoContainer ist. Die Verwaltungsklasse `Saros` verwaltet den PicoContainer, welcher wiederum dessen Verwaltungsklasse `Saros` verwaltet. Hier ist ein Kreis von Verantwortlichkeiten entstanden.

Wäre das Finalisieren seitens `Saros` implementiert worden, hätte es zu Problemen geführt. Hier der beispielhafte Ablauf: `Saros` wird aufgefordert, den PicoContainer aufzulösen. Der PicoContainer fordert nun alle Komponenten auf, sich aufzulösen. Da `Saros` wiederum eine Komponente ist, wird diese aufgelöst, während es selbst noch am finalisieren ist. Dieser Kreis ist höchst problematisch.

In den ersten Iterationen des Gerrit-Patches habe ich diese Problematik übersehen. Durch mein bisheriges Vorgehensmuster habe ich die `Saros`-Klasse durch das neue `IntellijSarosLifecycle` ersetzt, in der Annahme, dass diese die gleiche Aufgaben abdecken und die bisherige Implementierung solide sei. Das stellte sich als falsch heraus.

Der Fehler war es, den neuen `IntellijSarosLifecycle` das Aufbewahren von `Project` und `Workspace` anzuhängen, was schon gar nicht zu seinem Aufgabenbereich passt. Außerdem ging damit die problematische zirkuläre Abhängigkeit einher, die oben beschrieben wurde und somit von der alten `Saros`-Klasse übernommen wurde.

Als Lösung dieses Problems wurde versucht, diese gefragten Klassen `Project` und `Workspace` auf eine andere Weise verfügbar zu machen. So entschied ich mich, diese zwei Exemplare über Dependeny Injection zu verbreiten. Dazu wurden diese zwei Exemplare an `SarosIntellijContextFactory` übergeben, welches diese zu Komponenten des PicoContainers werden ließ. Jetzt musste jede Klasse, die auf diese Exemplare zugreifen will, eine Dependency Injection einprogrammiert bekommen.

Dieses Vorhaben gelang, führte jedoch zu einem hohen Umfang des Gerrit-Patches. Eigentlich wären nur fünf Klassen von den Umbauarbeiten betroffen, doch die neu eingeführte Dependency Injection sorgte dafür, dass 16

weitere Klassen angepasst werden mussten.

Bei der Implementierung von `IntellijSarosLifecycle` wurde nur die `additionalContextFactories`-Methode überschrieben. Sie gibt `SarosIntellijContextFactory` zurück und zusätzlich die `HTMLUIContextFactory`, falls die HTML-Oberfläche eingeschaltet worden ist.

4.5 Erweiterung von HTML UI backend

Nachdem die drei großen Arbeitspakete *RemoteEditorManager*, *ConsistencyWatchdogs* und *Lebenszyklus* abgeschlossen wurden, verblieb nur noch ein Viertel meiner dreimonatigen Arbeitszeit.

In Absprache mit meinem Betreuer wurde der Aufgabenfokus verschoben. Zwar standen noch weitere Codeduplikate bereit bearbeitet zu werden, doch ich hatte die Gelegenheit, eine andere Abschlussarbeit mit meinem erworbenen Wissen zur Kern-Komponente zu unterstützen.

Zu diesem Zeitpunkt lief parallel Webers Abschlussarbeit [Web16], welche sich um die Weiterentwicklung des Frontends der neuen HTML-Oberfläche kümmert. Zu dem Frontend gehört auch ein Backend, das eine Schnittstelle zwischen Kern-Komponente aus der Java-Welt und dem Frontend aus der JavaScript-Welt bietet.

Meine neue Aufgabe war es nun, das Backend zu erweitern, da noch viel Code fehlte, um auf Funktionalitäten aus der Kern-Komponente zugreifen zu können. Der Ausschnitt *Erweiterung der Core-Bestandteile von Saros* aus meinem Arbeitstitel bezieht sich auf diese Tätigkeit. Denn hier geht es erstmals nicht allein um Refaktorisierungen, sondern um die Implementierung neuer Funktionalitäten.

4.5.1 IncomingSessionNegotiation

In Saros werden Sitzungen zur Paarprogrammierung mittels Dialogen erstellt. Der Benutzer wird durch eine grafische Oberfläche geführt, die ihm dabei unterstützt, die benötigten Einstellungen zu setzen.

[HTML] IncomingSessionNegotiation UI preparation	
Status	Open
Erstellt am	12.02.2016
Geändert am	24.03.2016
Umfang	+211, -0

Ein solcher Dialog ist das Anzeigen, dass der Benutzer eine Einladung zu einer Sitzung erhalten hat. Der Nutzer kann nun die Sitzung ablehnen oder akzeptieren, worauf dann weitere Einstellung abgefragt werden.

Dieser Dialog existiert bereits für das Eclipse-Plug-In, die HTML-Oberfläche hat diesen jedoch noch nicht. Es fehlt die UI aus dem Frontend, aber auch die Schnittstellen aus dem Backend.
Meine Aufgabe ist es, das Backend zu erweitern, sodass eine Implementierung der HTML-UI möglich wird.

Die Bearbeitung dieser Aufgabe lief noch zum Zeitpunkt der Abgabe dieser Bachelorarbeit. Ein erster Gerrit-Patch wurde hochgeladen, welcher erste Klassenhülsen[20] anlegt, worauf das Frontend schon mal binden kann.

Aufwändig ist bei dieser Angelegenheit, dass bestehende Logik aufgebohrt werden muss, sodass sie zwei verschiedene Oberflächen ermöglicht werden. Ist die HTML-Oberfläche abgeschaltet, dann wird die bisherige Logik ausgeführt. Bei eingeschaltetem Zustand muss mit dem Backend gesprochen werden, um die HTML-Dialoge anzuzeigen und Daten ans Frontend fließen zu lassen.

In den Wochen vor der Abgabe dieser Bachelorarbeit wurde bereits ein erstes Konzept ausgearbeitet. Sie beinhaltet die Designentscheidung, eine hybride Form zwischen HTML- und IDE-Oberfläche zu erstellen. Eclipse hat die Möglichkeit, langlaufende Vorgänge in der *Progress*-Sicht anzuzeigen und abbrechbar zu machen. Mit der Einführung der HTML-Oberfläche soll diese native Funktion nicht verloren gehen. Dieses Vorhaben ist nicht weiter trivial.

Der Zusammenspiel aller notwendigen Klassen ist sehr komplex. Denn es werden viele Grenzen überschritten. Komponentengrenzen wie auch programmiersprachliche Grenzen. Und diese Grenzüberschreitungen sind teils gerichtet, sodass beachtet werden muss, welche Komponente auf welchen Komponenten zugreifen kann. Eine Einführung dieses hybriden Modells durchdringt jeder dieser Sichten mehrmals in beide Richtungen, um Daten und Ereignisse zu transportieren.

Da die Arbeit noch am laufen ist und die Implementierungen sich mit jeder Iteration stark unterscheiden, verzichte ich an dieser Stelle auf eine genauere Vorgehensbeschreibung. Mein Ziel ist es, bis zur Verteidigung dieser Bachelorarbeit, dieses Vorhaben abzuschließen.

[20] Gemeint sind Klassen mit noch fehlender Funktionalität.

5 Ergebnisse

Innerhalb der dreimonatigen Bearbeitungszeit konnten große Codeduplikate entfernt werden. Nach der Vorarbeit von Lasarzik [Las15] und Washington [Was16] verblieben keine kleinen Klassen mehr, dessen Verschieben in die Kern-Komponente simpel wäre. Stattdessen widmete ich mich den größeren Klassen, wie etwa dem `RemoteEditorManager` oder den ConsistencyWatchdogs.

Dabei stellte sich heraus, dass meine Arbeit keine einfache Refaktorisierungen mehr waren, die allein mit Entwicklungswerkzeugen hätten bearbeitet werden können. Geschätzt waren nur ein Viertel der Zeit für die Implementierung. Die restliche Zeit entfiel auf die Codeanalyse, den Austausch mit Saros-Entwicklern und der Gerrit-Reviewprozess.

Ehe irgendeine Klasse verschoben werden konnte, musst deren Abhängigkeiten geklärt werden. Viele Zusammenhänge und Interaktionen im Code konnten erst in Erfahrung gebracht werden, nachdem andere Entwickler befragt wurden. Und auch im Reviewprozess wurden Randfälle entdeckt, die ich bei der Implementierung übersehen hatte.

Mein stetiger Respekt vor dem Code war teils hinderlich. So habe ich bei Refaktorisierungen einen sehr hohen Wert darauf gelegt, bestehende Funktionalitäten beizubehalten. Das hat dann zu komplexen Konstrukten geführt, nur um alte Funktionalität beizubehalten, welche nach Absprache mit anderen Entwicklern im Nachhinein als obsolet abgestuft worden sind. Siehe die Teilaufgabe ConsistencyWatchdogHandler.

Auch hat mich die stetige Annahme, dass bisheriger Code korrekt und designtechnisch akzeptiert sei, auf Irrwege geleitet. So habe ich mit dieser Einstellung Codekonstrukte übernommen, die sich später als schlecht herausstellten. Übernommener Code ist zu analysieren, ob dieser auch wirklich das beabsichtigte Verhalten hat. Blindes Übernehmen kann zu fehlgeleiteten Design führen, wie es mir mit der Teilaufgabe IntellijSarosLifecycle erging.

Ich musste mich schon vor befreundeten Softwareentwicklern rechtfertigen, dass meine Bachelorarbeit doch nur simple Anwendung des Refactor-Tools aus Eclipse sei. Doch dies war bei weitem nicht so.

5.1 Ausblick

Trotz meiner geleisteten Arbeit ist das Thema Codeduplikate im IntelliJ-Plug-In noch nicht vollständig abgeschlossen. Die Situation hat sich schon deutlich verbessert, allzu viele Duplikate sind nicht mehr vorhanden.

Im IntelliJ-Plug-In befindet sich ein Paket mit vermeintlichen Codeduplikaten. Dieses Paket wird Core genannt, um zu signalisieren, dass diese Klassen eigentlich in den Kern sollen. Es ist noch zu überprüfen, was davon in die Kern-Komponente verschoben werden muss.

Auch sollten Stellen gesucht werden, dessen Logik in den Kern verschoben werden kann, weil sie über alle Plattformen gleich ist. Meine Abstraktion des Lebenszyklus ist eine solche Arbeit.

Die bisherige Arbeit am Kern sollte definitiv fortgesetzt werden, da sie jede weitere Arbeit an den Plug-Ins vereinfacht. Insbesondere wenn weitere IDE-Unterstützungen ihre Entwicklung beginnen, ist eine ausgebaute Kern-Komponente eine erhebliche Zeitersparnis.

Zu diesem Zeitpunkt läuft mit der HTML-Oberfläche eine weitere sehr große Umbauarbeit. Ich befürchte, dass die Übergangsphase aus Entwicklersicht sehr unangenehm wird. Denn bevor die neue HTML-Oberfläche vollständig ist, wird der Quelltext mit Fallunterscheidungen gespickt sein, um beide Oberflächen parallel betreiben zu können.

5.2 Fazit

Meine Arbeit betraf fast ausschließlich die Codequalität von Saros. Das führte dazu, dass ich, im Gegensatz zu anderen Abschlussarbeiten, mein Ergebnis für Außenstehende wenig beeindruckend sichtbar machen konnte.

Es war jene Form von Arbeit, wovon nur die Saros-Entwickler profitieren. Der Aufwand in der Wartung und Erstellung von Plug-Ins nimmt mit jeder Klasse ab, die zurecht in die Kern-Komponente verschoben wurde.
Ich habe meinen Beitrag dazu geleistet: Acht Klassen verschoben, drei neue Klassen angelegt. In Anbetracht der Komplexität dieser verschobenen Klassen, ist *acht* eine respektable Anzahl an entfernten Codeduplikaten.

6 Anhang

6.1 Offene und übernommene Gerrit-Patches

- [API] Move LineRange to core
  http://saros-build.imp.fu-berlin.de/gerrit/2946
- [API] Move TextSelection to core
  http://saros-build.imp.fu-berlin.de/gerrit/2947
- [API] Move RemoteEditorManager to core
  http://saros-build.imp.fu-berlin.de/gerrit/2948
- [API] Move AwarenessInformationCollector to core
  http://saros-build.imp.fu-berlin.de/gerrit/2958
- [API] Move WatchdogServer and DocumentChecksum to core
  http://saros-build.imp.fu-berlin.de/gerrit/2959
- [API] Move WatchdogClient to core
  http://saros-build.imp.fu-berlin.de/gerrit/2979
- [API] Move WatchdogHandler to core and update core context
  http://saros-build.imp.fu-berlin.de/gerrit/2996
- [API] Move SarosCoreContextFactory to core
  http://saros-build.imp.fu-berlin.de/gerrit/2971
- [API] Create AbstractSarosLifecycle
  http://saros-build.imp.fu-berlin.de/gerrit/2982
- [API] Extend AbstractSarosLifecycle (Eclipse)
  http://saros-build.imp.fu-berlin.de/gerrit/2984
- [API] Extend AbstractSarosLifecycle (IntelliJ)
  http://saros-build.imp.fu-berlin.de/gerrit/2983
- [HTML] IncomingSessionNegotiation UI preparation
  http://saros-build.imp.fu-berlin.de/gerrit/3019

6.2 Zurückgezogene Gerrit-Patches

- [API] Move IFollowModeChangesListener into core
  http://saros-build.imp.fu-berlin.de/gerrit/2924
- [API] Move IsInconsistentObservable to core
  http://saros-build.imp.fu-berlin.de/gerrit/2960
- [API] Move ConsistencyWatchdogClient to core
  http://saros-build.imp.fu-berlin.de/gerrit/2967
- [API] Move WatchdogHandler to core
  http://saros-build.imp.fu-berlin.de/gerrit/2980

6.3 Verwandte Hyperlinks

- Bearbeitungsseite auf ThesesHome
  http://www.mi.fu-berlin.de/w/SE/ThesisSarosPluginRefactoring

7 Literatur

[Boh15] Matthias Bohnstedt. Entwicklung einer IDE-unabhängigen Benutzeroberfläche für Saros. Masterarbeit, Freie Universität Berlin, Institut für Informatik, August 2015. Online erhältlich unter http://www.inf.fu-berlin.de/inst/ag-se/theses/Bohnstedt15-IDE-unabhaeng-benutzeroberflaeche.pdf; aufgerufen am 9. April 2016.

[CW00] Alistar Cockburn and Laurie Williams. The costs and benefits of pair programming. Technical report, 2000. Online erhältlich unter http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF; aufgerufen am 13. April 2016.

[Fou07] Free Software Foundation. Gnu general public license, Juli 2007. Online erhältlich unter http://www.gnu.org/licenses/gpl-3.0.html; aufgerufen am 13. April 2016.

[Fow03] Martin Fowler. Technicaldebt, Oktober 2003. Online erhältlich unter http://martinfowler.com/bliki/TechnicalDebt.html; aufgerufen am 12. April 2016.

[Fow04a] Martin Fowler. Definitionofrefactoring, September 2004. Online erhältlich unter http://martinfowler.com/bliki/DefinitionOfRefactoring.html; aufgerufen am 13. April 2016.

[Fow04b] Martin Fowler. Inversion of control containers and the dependency injection pattern, Januar 2004. Online erhältlich unter http://martinfowler.com/articles/injection.html; aufgerufen am 13. April 2016.

[Fow14] Martin Fowler. Workflows of refactoring, Januar 2014. Online erhältlich unter http://martinfowler.com/articles/workflowsOfRefactoring/#2hats; aufgerufen am 13. April 2016.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Kel03] Björn Kellermann. 1.3 signatur, definition, Juni 2003. Online erhältlich unter http://www.fh-wedel.de/~si/seminare/ss03/Ausarbeitung/9.hxml/haskell/signatur_definition.html; aufgerufen am 12. April 2016.

[Las15] Arndt Lasarzik. Refaktorisierung des Eclipse Plugins Saros für die Portierung auf andere IDEs. Bachelorarbeit, Freie Universität Berlin, Institut für Informatik, April 2015. Online

erhältlich unter http://www.inf.fu-berlin.de/inst/ag-se/theses/Lasarzik15-refaktorisierung.pdf; aufgerufen am 9. April 2016.

[Log] Business logic - wikipedia, the free encyclopedia. Online erhältlich unter https://en.wikipedia.org/wiki/Business_logic; aufgerufen am 12. April 2016.

[Mul15] Multithreaded programming guide. Technical report, Oracle, 2015. Online erhältlich unter https://docs.oracle.com/cd/E53394_01/html/E54803/index.html; aufgerufen am 10. April 2016.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupitar collaboration system. Technical report, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, 1995. Online erhältlich unter http://lively-kernel.org/repository/webwerkstatt/projects/Collaboration/paper/Jupiter.pdf; aufgerufen am 8. April 2016.

[Was16] Denis Washington. Entwicklung und Evaluation eines unabhängigen Sitzungsservers für das Saros-Projekt. Masterarbeit, Freie Universität Berlin, Institut für Informatik, Januar 2016. Online erhältlich unter http://www.inf.fu-berlin.de/inst/ag-se/theses/Washington16-saros-server.pdf; aufgerufen am 9. April 2016.

[Web16] Nina Weber. Erweiterung der HTML-Oberfläche in Saros. Bachelorarbeit, Freie Universität Berlin, Institut für Informatik, 2016. Unveröffentlichte Arbeit.

[Yip16] Jason Yip. It's not just standing up: Patterns for daily standup meetings, Februar 2016. Online erhältlich unter http://www.martinfowler.com/articles/itsNotJustStandingUp.html; aufgerufen am 13. April 2016.