Freie Universität Berlin
Bachelorarbeit am Institut für Informatik
Arbeitsgruppe Software Engineering

Implementierung einer Unterstützung von Versionsverwaltung in einem Plug-In für verteilte Paarprogrammierung

Christian Zygmunt Jeschke
Matrikelnummer: 4935108
christian.jeschke@fu-berlin.de

Betreuer: Kelvin Glaß

Eingereicht bei: Prof. Dr. Lutz Prechelt

Zweitgutachten bei: Prof. Dr. Claudia Müller-Birn

14. März 2019

Zusammenfassung

Die Paarprogrammierung basiert auf der Idee das zwei Programmierer zusammen Programmcode und dessen Dokumentation erstellen. Ziel ist es Qualitätssicherung zu betreiben. Erreicht wird dies durch folgende zwei Aspekte der Paarprogrammierung. Erstens erfolgt durch die zweite Person eine ständige Durchsicht und zweitens gibt es immer zwei Personen, die ein tiefes Verständnis über das Geschriebene besitzen. Erfolgt diese Arbeit an zwei Rechnern nennen wir dies verteilte Paarprogrammierung. Eine Software die dies ermöglicht ist **Saros** die an der Freien Universität entwickelt wird.

Versionsverwaltung erlaubt es, das Speichern von aktuellem Inhalt des Arbeitsverzeichnisses, erleichtern somit den Umgang mit mehreren Versionen einer Software und dient der Koordination von gleichzeitiger Entwicklung verschiedener Änderungen. Sie erleichtert vor allem das anschließende zusammenführen. Git ist eine Versionsverwaltung, die sich dadurch auszeichnet, dass sie verteiltes Arbeiten unterstützt und das sie den Speicher effizient nutzt.

Ich möchte in dieser Arbeit diskutieren warum die Software Saros um eine Unterstützung für Git erweitern werden sollte. Dazu habe ich eine Anforderungsanalyse durchgeführt, daraus Anforderungen bestimmt und diese priorisiert. Eine Implementierung der am höchst priorisierten Anforderungen war ebenfalls Teil dieser Arbeit, wobei ich bei der Implementierung auf jedem Fall Datenverlust vermeiden wollte. Mehrere Nutzer können nun mit Saros Daten der Versionsverwaltung ausgleichen.

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.

Christian Zygmunt Jeschke

14. März 2019

Inhaltsverzeichnis

1 Einleitung

Saros funktioniert für die Entwicklungsumgebung (auf Englisch *Integrated Development Environment* oder kurz IDE) Eclipse und ist für die IDE IntelliJ in Arbeit. Eine Hauptfunktion der Software ist das der Programmcode mehrerer Rechner synchron gehalten wird. Beide Nutzer können so stets den gleichen Programmcode bearbeiten, auch ohne, dass die Dateien manuell gespeichert und verschickt werden müssen. Auch ermöglicht Saros, über die IDE miteinander zu kommunizieren (z.B. über Chat) und die Zeilen zu sehen die der Partner gerade betrachtet. Wichtiges Konzept ist dabei die **Session**. Eine Session (auf Deutsch Sitzung) [1] ist die Abstraktion davon, wenn zwei Entwickler zusammen programmieren. Es ist also ein virtueller Arbeitsplatz. Nutzer von Saros können nur dann zusammenarbeiten, wenn sie sich innerhalb einer Session befinden. Der **Session Start** ist dabei der Beginn der Zusammenarbeit. Dabei werden Dateien vom Initiator der Session (dem *Host*), zu ein oder mehreren *Clients* verschickt, falls diese nicht exakt die selbe Datei bereits besitzen.

Versionsverwaltung (auf Englisch *Version Control System* oder kurz VCS) erlaubt das Speichern von aktuellem Inhalt des Arbeitsverzeichnis. Dies erleichtert somit den Umgang mit mehreren Versionen einer Software. Sie dienen der Koordination von gleichzeitiger Entwicklung verschiedener Änderungen. Sie erleichtern vor allem das anschließende zusammenführen.

Die Unterstützung von VCS in Saros wird hiermit nicht zum ersten Mal angesprochen. Ich möchte dazu die Arbeit von Riad Djemili vorstellen. Das Thema der Arbeit ist die Entwicklung der ersten Version von Saros. Die schriftliche Ausarbeitung enthält bereits Aussagen über die Nutzung von VCS bei der Entwicklung und unter anderem Ideen, wie eine mögliche Unterstützung in Saros aussehen könnte. [1] Dazu folgen einige Zitate. Der erste bezieht sich auf die Umfrage, die innerhalb der Arbeit durchgeführt wurde.

> Meine erste Frage bezog sich auf verwendete Versionsverwaltungssysteme (”Which revision control system(s) do you use?”). Verschiedene Antworten mit Möglichkeit eigener Ergänzungen und Mehrfachwahl waren vorgegeben. Motivation dieser Frage war die tendenzielle Bestätigung oder Widerlegung meiner Annahme, dass eine Vielzahl der Projekte mit mehreren beteiligten Entwicklern über eine Versionskontrolle verfügt. Aus den Antworten resultiert, dass alle Teilnehmer Versionsverwaltungssysteme einsetzen. Die Antwort ”I use no revision control system” wurde nur in der Nennung mit anderen Antworten gewählt. Am Verbreitetsten waren bei den Teilnehmern SVN (Subversion) 38% und CVS (Concurrent Versions System) (33%)

Weiter beschreibt der Autor das er bei Tests von damals bereits erhältlichen Alternativen zu Saros große Verzögerungen erlebte.

> Ungeachtet der Tatsache, dass ich bereits eine identische Version des Projekts auf meinem System installiert hatte, wurden alle Daten immer von dem entfernten Host der Sitzung übertragen. Diese Entwurfsentscheidung scheint die Annahme zugrunde zu liegen, dass die meisten DPP-Anwender vor Beginn einer Sitzung, nicht bereits die gleiche Version eines Projekts auf ihren Systemen vorliegen hätten. Dies widersprach jedoch meiner persönlichen Erfahrung: Betrieb ich beispielsweise auf einer Arbeitsstelle Paarprogrammierung, dann mit Kollegen, die an dem gleichen Projekt arbeiteten, und mit denen mein Projektstand abgesehen von minimalen Änderungen seit dem letzten Übertragen in das Versionskontrollsystem übereinstimmte. Um meine eigene Annahme zu überprüfen, fragte ich in der von mir realisierten Umfrage ... , wie wahrscheinlich es sei, dass die Anwender bereits vor Beginn einer DPP-Sitzung das zu bearbeitende Projekt auf ihrem System installiert hätten. Die Teilnehmer der Umfrage gaben im Durchschnitt 72,5% bei einer Standardabweichung von 37,36 an. Ein Drittel der Teilnehmer gab dabei sogar 100% an

[1] Es gibt im Kontext dieser Arbeit einige englische Vokabeln die auch im Deutschen verwendet werden. Damit ich eindeutig bei der Bezeichnung bleibe, dient eine Übersetzung nur zur Erklärung des Konzepts aber anschließend verwende ich meist die englischen Begriffe und benutze sie wie Eigennamen.

Nachfolgend will ich Herr Djemili's eigene Vorstellung zu einer VCS Unterstützung für seine Software zitieren und abschließend eine Diskussion zur Wahl der Dateien, die über Saros geteilt werden soll(t)en.

> Öffentliche Repositorien (auf Englisch repositories) sind ein hervorragender Mechanismus, um den aktuellen Projektstand zur Veröffentlichung zu stellen. Ein möglicher Anwendungsfall bei der Benutzung von Saros wäre auch sich den Projektcode vom Repositorium zu laden und dann auf Basis dieser Version die Sitzung einzuleiten.
>
> Eine weitere Ausnahme sind Dateien, bei denen das Attribut team-private gesetzt ist. Dateien dieser Art werden von Versionsverwaltungssystemen benutzt, um Informationen über den lokalen Entwicklungsstand zu speichern. Für die verteilte Paarprogrammierung sind sie nicht relevant.

Eine weitere Abschlussarbeit an der Gruppe Software Engineering hat bereits eine VCS Unterstützung in Saros integriert. [2] Das gewählte VCS ist Apache Subversion (SVN). [2] Der Nutzer kann beim Start einer Session auswählen, ob er die Unterstützung von SVN anschalten will. Ist sie angeschaltet, werden die Dateien für das VCS genauso synchron gehalten wie Dateien. Dafür werden Operationen, die bei einem Nutzer stattfinden verschickt und dann beim Partner wiederholt. Genutzt wird dazu ein Saros Protokoll, welches sogenannte Activities versendet. Eine Anforderungsanalyse wird innerhalb der Arbeit nicht beschrieben, sodass ich davon ausgehe das die Unterstützung von SVN auf die gewählte Art, nach Einschätzung des Autors erfolgte.

In meiner Arbeit soll nun untersuchen werden, ob eine Unterstützung des VCS Git in Saros erfolgen sollte. Dazu habe ich sie folgendermaßen gegliedert. In folgendem Abschnitt werde ich das VCS Git und die dahinterliegenden wesentlichen Technologien vorstellen. Abschnitt vier habe ich in zwei Teile geteilt. Im ersten Teil beschreibe ich wie Saros mit Git umgegangen ist bevor ich diese Arbeit begonnen habe. Im zweiten Teil werde ich den Ausgangspunkt meiner Untersuchung beschreiben, anschließend die Anforderungserhebung beschreiben und schließlich die Funktionalität festhalten, die daraus resultiert. Im fünften Abschnitt beschreibe ich die Implementierung.

[2] https://subversion.apache.org/ (besucht am 12.03.2019)

2 Git

Ich möchte hier einige Aspekte davon genauer erklären wie das VCS Git funktioniert, weil dies zum Verständnis und Bau meiner Implementierung notwendig ist.[3] Zuallererst möchte ich erläutern welche Datenstruktur Git zugrunde liegt und anschließend wichtige Befehle. Auch will ich noch mal auf Saros eingehen, im speziellen will ich die Datenübertragung von Git und Saros vergleichen. Abschließen möchte ich diesen Abschnitt mit einer Vorstellung der Java-Bibliotek JGit, die ich bei der Implementierung genutzt habe.

2.1 Das *.git* Verzeichnis

Ich möchte hier erklären wie Git intern die Versionen von Dateien abspeichert, weil dies erklärt, wieso der Speicher effektiv genutzt wird. Auch soll dies einen Einblick darauf geben, warum ich im Verlauf der Entwicklung abgewogen habe ob ich diese Dateien direkt *anfasse* (siehe Abschnitt 5.0.1).

Praktisch befindet sich die Versionsverwaltung eines **Arbeitsverzeichnisses** innerhalb eines Unterverzeichnisses mit dem Namen *.git*.[4] Beides zusammen wird als **Repository** bezeichnet. Innerhalb des *.git* Verzeichnisses befinden sich folgende Unterverzeichnisse und Dateien: *config*, *description*, *info*, *hooks*, *objects*, *refs*, *HEAD* und *index*. Ich werde im Folgenden die hier relevanten, letzten vier, genauer beschreiben.

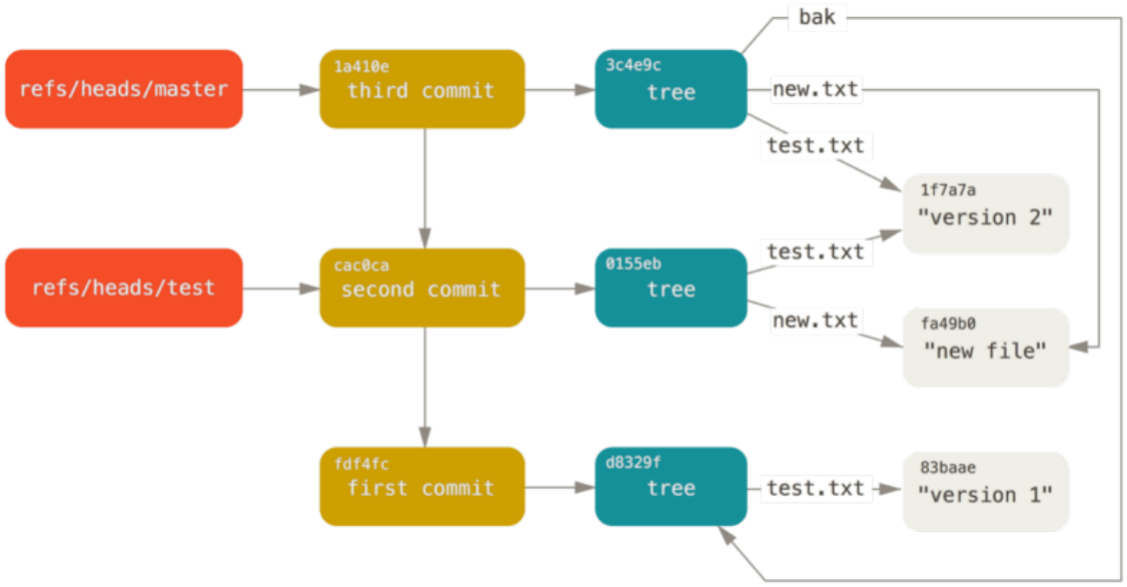

Abbildung 1: Objekte im *.git* Verzeichnis. **Branches**, **Commits**, **Trees** und **Blobs**, von links nach rechts respektiv. Quelle: [3]

Der Inhalt des Verzeichnisses **objects** (auf Deutsch Objekte/Identitäten) hält die Informationen über alle Versionen. Der Ordner refs enthält Informationen darüber wo sich Branches momentan befinden.

[3] Die Quelle für dieses Kapitel ist [3], falls nicht anders angegeben.

[4] Es gibt Ausnahmen: Insbesondere erlaubt die Option *–seperate-git-dir* beim initialisieren des Versionsverwaltungssystems einen Speicherort für das *.git* Verzeichnis zu wählen der nicht im selben Verzeichnis ist wie das Arbeitsverzeichnis. [4]. Auch befindet sich die Datei *.gitignore* nicht im *.git* Verzeichnis, sondern auf selber Ebene.

Dabei enthalten die Blobs die Inhalte der abgespeicherten Dateien, aber in komprimierter Form. Die Tree Objekte referenzieren mehrere Blobs oder wiederrum andere Trees. Sie dienen einerseits als Verzeichnis und vor allem dazu eine Version des Systems festzuhalten. Für effektive Speichernutzung erstellt Git nur ein neues Blob wenn sich die Datei ändert. Ändert sich zwischen zwei Versionen ein Blob nicht, so wird der alte Blob referenziert.

Branches sich Zeiger die auf einen Commits zeigen. Die Vokabel Commit wird bei Git für die Versionen benutzt, wobei ein Commit mehr enthält als nur die eigentlichen Daten, sondern auch Meta-Informationen (wie den Autor, Datum beim Autor, *Commiter*, Datum beim Commiter und eine Commit-Nachricht). Wenn ich deshalb den momentane Version meiner Arbeit speichern möchte so erstelle ich, bei Git, einen Commit, indem ich alle Änderungen *commite*.[5] Ein Commit enthält neben den eigentlichen Daten auch Meta-Informationen (wie den Autor, Datum beim Autor, *Commiter*, Datum beim Commiter und eine Commit-Nachricht). Branches benötigen wir falls wir einen Fehler beseitigen möchten oder eine neue Funktionalität implementieren. Damit es dabei aber weiterhin eine stabile Version gibt, werden die Änderungen nicht in diese stabile Version geschrieben. Diese bleibt weiterhin bestehen und die Version mit den Änderungen wird mit einem neuen Branch gekennzeichnet. Der Branch der weiterhin auf die stabile Version zeigt ist bei Git standardmäßig als *master* benannt. Das Erzeugen von Branches ist mit Git deshalb so sparsam (sowohl in Zeit als auch im Speicher), weil wie bereits erwähnt ein Branch lediglich ein Zeiger zu einem Commit ist. Dieser besteht nur aus 40 Zeichen, weil Git keinen Dateinamen verwendet, sondern stattdessen einen sogenannten SHA1-Hash. Berechnet wird dieser mit dem SHA-1 Algorithmus dessen Ausgabe ein String der Länge 40 ist (Wie in Abbildung 2 werden aber meist nur die ersten Zeichen angegeben, weil dies zur Identifikation ausreicht). Bei einem Commit wird der Hash aus alle in ihm enthaltenen Informationen berechnet, also sowohl die Blobs, die ebenfalls erstmals zu Hashes gewandelt werden, die Trees, (ebenfalls als Hash) und die Meta-Informationen. Auch wenn Git Operation ausführt, so benutzt er nicht den Dateinamen, sondern stets den Hashwert. Ein Hash ist einzigartig, d.h. das sobald ich ein Bit verändere, auch der Hash der Datei verändert wird. Jede Änderung an Dateien oder sogar Datenverlust wird durch die Änderung des Hash registriert. [5] Benötigen wir einen neuen Branch, so müssen wir nur einen neuen Zeiger erstellen. Mit Git können deshalb viele Branches benutzt werden, was bedeutet, dass an vielen Änderungen gleichzeitig gearbeitet werden kann ohne sich dabei zu behindern. Erst beim Vereinigen von Branches (dem sogenannten **Merge**) müssen Überschneidungen beachtet werden. Erzeugen wir einen neuen Commit so zeigt der Branch automatisch auf die neue Version wenn sich der **HEAD** auf dem Branch befindet. Der HEAD ist der wichtigste Zeiger, denn ändert sich der HEAD, so ändert sich auch das Arbeitsverzeichnis. Wenn der Nutzer eine bestimmte Version hinter einem Commit betrachten will muss er dazu den HEAD ändern, sodass er auf den Commit zeigt oder auf einen Branch der wiederum auf den Commit zeigt.

[5] Analog zu den üblichen englischen Eigennamen gibt es auch Verben die auch im Deutschen ohne Übersetzung verwendet werden und so auch in dieser Arbeit

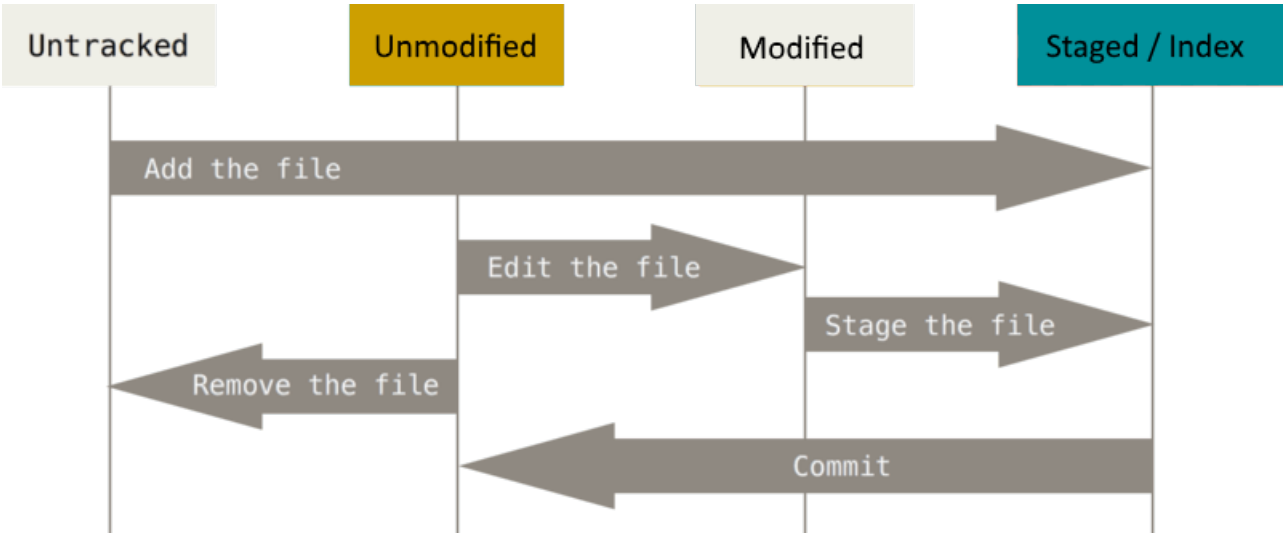

Abbildung 2: Eine Datei, die im aktuellen Commit enthalten ist oder sich seit einem früheren Commit nicht verändert hat ist Unmodified. Verändere ich eine Datei so ist sie Modified. Erstelle ich eine neue Datei so ist ihr Zustand zuerst Untracked. Bevor der Zustand sich von Modified oder Untracked in Unmodified ändern kann muss er erst zu Staged wechseln d.h. bevor die Änderung in den aktuellen Commit gelangen kann, muss sie in den Index.

2.2 git add und git commit

Änderungen die durch den Befehl *git add* in den **Index** gebracht werden (auch als *gestaged* oder *geadded* bezeichnet), können im Anschluss in einen Commit gebracht werden, mit dem Befehl *git commit*.
Der Index ist ein Zwischenspeicher. Besonders ist dieser Bereich vor allem deshalb weil er Änderungen von Dateien festhalten kann, die weder in einem Commit noch im **Arbeitsverzeichnis** enthalten sind. Intern erstellt Git genau dann einen Tree wenn Dateien in den Index gebracht werden oder fügt sie dem aktuellen Tree hinzu, wenn der Index bereits Änderungen enthält.

2.3 git bundle

Der Befehl *git bundle* erlaubt es eine komprimierte Datei zu erstellen, die alle Versionen beinhalten kann oder nur Teile davon. Dies dient vor allem dazu, Übertragungen von Repositories zu ermöglichen wenn von der Übertragung über die Protokolle von Git abgesehen werden muss/soll. Für das Erstellen des Bundles wird eine **aktuelle Version** benötigt. Dies ist in aller Regel der HEAD des Senders. Außerdem muss eine **Basis** angegeben werden, wenn nur einige Versionen übertragen werden sollen. Wichtig ist die gewählte Basis beim Sender und Empfänger besteht, weil nur Dateien ab der Basis übertragen werden.

2.4 git fetch

Durch die Ausführung dieses Befehls wird eine Verbindung zwischen dem eigenen und einem weiteren Repository hergestellt. Das Ziel ist es das eigene Repository zu aktualisieren. Das heißt es werden Versionen hinzugefügt und Branches aktualisiert. Das *fetchen* verändert die eigenen Branches **nicht**, sondern das eigene Repositry besitzt für jeden Branch des anderen Systems eine Kopie oder erstellt diese gegebenenfalls. Zum Verändern der eigenen Branches folgt nach dem *git fetch* der Befehl *git merge*.(Beides in einem Befehl wird mit *git pull* erreicht.)

2.5 git merge

Das Ziel beim Befehl *git merge* ist es zwei Commits oder Branches zusammenzuführen. Ein in dieser Arbeit besonders interessanter Spezialfall ist der **Fast Forward Merge** (siehe Abschnitt 5.1). Bei einem Fast Forward Merge ist der eine Commit/Branch Vorgänger des anderen. In den einfachen Fällen wird bei einem Fast Forward Merge nur der Branch des Vorgängers verschoben und das Arbeitsverzeichnis auf das des Nachfolgers aktualisiert. Es gibt jedoch auch Fälle, in denen danach das Arbeitsverzeichnis Änderungen beinhaltet, die **nicht** beim Nachfolger existierten. Verantwortlich dafür sind Änderungen im Index. [6]

2.6 Datenübertragung mit Git

Git hat mehrere Protokolle. Es gibt ein *dumb* und ein *smartes* Protokoll wobei das erstere nur die Übertragung über *HTTPS* erlaubt und das zweite sowohl *HTTPS* als auch *SSH*. Bei der dump Variante gibt es keine Besonderheiten nur weil es sich um Dateien der Versionskontrolle handelt. Die benötigten Dateien werden ausfindig gemacht und verschickt / empfangen. Bei der smarten Varianten werden vorab Verhandlungen geführt um die Transfermenge gering zu halten. Dazu werden 2 Prozesse benötigt, die es Sender und Empfänger erlauben Informationen auszutauschen. So wird zum Beispiel der SHA-1-Hash beim zu aktualisierenden Branch mit dem des Senders abgeglichen und nur falls sich dieser verändert hat oder zuvor noch nicht existiert hat, die benötigte Differenz versendet. [6]

2.7 Datenübertragung mit Saros

Saros benutzt für den Datentransfer folgende Möglichkeiten:
Zum einen werden **Message**'s versendet. Sie können versendet werden auch wenn die Nutzer noch nicht in einer Session verbunden sind und werden vor allem dazu genutzt eine Session aufzubauen, also beim Session Start.
Zum anderen werden zum Datentransfer ebenfalls **Activities** genutzt. Sie können **nur** während einer Session genutzt werden und dienen vor allem dazu die Dateien beider Nutzer synchron zu halten. Activities sind *Value Objects* was bedeutet, dass sie nicht verändert werden sollten. Bei Activitys wird eine Verbindung über einen Server hergestellt, wobei der Server, bei Saros, der Host der Sitzung ist. [7] [8]

2.8 JGit

JGit ist eine Java-Bibliothek zum Benutzen von Befehlen von Git. Verantwortlich für dieses Projekt ist die *Eclipse Foundation* die ebenfalls für **Eclipse** zuständig sind. JGit kann jedoch auch unabhängig von der IDE genutzt werden. [9] Projekte die JGit nutzen sind u.a. Jenkins oder das Eclipse Plug-In EGit zur Nutzung von Git mit Eclipse. [10] Die Sprache Java habe ich für meine Implementierung verwendet. Saros selbst ist ebenfalls größtenteils mit Java geschrieben.

[6] Siehe dazu *Two-way merge* in https://raw.githubusercontent.com/github/git-msysgit/master/Documentation/technical/trivial-merge.txt (besucht am 12.02.2019)

3 Der Ist-Zustand

In Anbetracht meine Arbeit sind die benannten Gedanken und Entscheidungen der vorherigen Versuchen ein VCS in Saros zu unterstützen (siehe Abschnitt 1) insoweit interessant das sich nach meiner Meinung nach bis auf die Technologie (Git ist im selben Jahr wie Saros erschienen) nicht viel verändert haben sollte. Deshalb werden die Argumentationen und Erkenntnisse auch von mir als wichtige Entscheidungsgrundlage genommen.
Ich möchte jetzt hier erwähnen wie ich Saros vorgefunden habe als ich mit der Einarbeitung anfing. Der Fokus dabei war natürlich wie die Arbeit mit einem VCS und Saros gleichzeitig funktioniert.
Gestartet habe ich mit einem manuellen Test für den bereits bestehenden SVN-Support damit ich feststellen konnte wie sich die Nutzung mit aktiviertem Support und ohne unterscheidet. Vorweg: Einen Unterschied merkte ich nicht.
Daraus folgte meine erste Erkenntnis: **Ein Unterstützung von VCS innerhalb von Saros ist nicht vorhanden**.

Ich wollte anschließend überprüfen wie Saros sich verhält, wenn es trotzdem mit Git genutzt wurde. Es gab Indizien das dies zu Problemen führt.[7] Ich wollte auch das *Risky* und *Safe* Layout (siehe Abbildung 3) überprüfen.

I'm working with a version control system, can I use Saros?

SVN: Yes, you can. In theroy, there should be no problems since so called *derived files* are not shared by Saros. Usually, the corresponding Eclipse plugins (such as Subclipse) take care of setting the "derived" attribute for ".svn" directories (as well as Java ".class" files, which are also not synced).
If in doubt or if you don't use such plugins, you can always set this attribute on your own: Just right-click on the respective directory, select "Properties", and check the attribute "Derived".

Git: To make completely sure that Saros won't mess with your precious versioning data, you might consider a folder layout where your Eclipse project(s) reside on a level below the .git folder (see below). That's the way we organize our own source code (even though we do so for other reasons).
We're working on Saros's Git-friendliness so it also supports repositories adhering to the one-project-per-repository policy.

Risky Layout ("one project per repository")	Safe layout ("multiple projects per repository")

`my-project/  <-- you shouldn't share this one`
`  .git/`
`  src/`
`  test/`
`  readme.html`
	`my-project/`
	`  .git/`
	`  module-a/  <-- totally safe to share`
	`  src/`
	`  test/`
	`  module-b/  <-- so is this one`
	`  src/`
	`  test/`

Abbildung 3: Auf der Website des Saros - Projekts [11] befindet sich eine Regel auf die der Saros Nutzer bei gleichzeitiger Nutzung von Git und Saros achten soll.

Ich wollte wissen was bei einem Session Start passiert, wenn dabei ein Repository mit Saros übertragen wird und der Partner noch **keine** Dateien hat. Dafür habe ich ein Test erstellt (siehe #427).
Der Test hat ergeben das der *.git* Ordner bei Nutzung mit Eclipse nicht geteilt wird, unabhängig vom Layout. Daraus folgt das, wenn ich will das beide Saros Nutzer auch Git benutzen, erreichen ich dies **nicht** mit den Saros Protokollen. Die Protokolle von Saros übertragen nicht den wichtigen *.git* Ordner und damit existiert auch kein Repository bei den Clients sondern nur das Arbeitsverzeichnis.

[7] https://sourceforge.net/p/dpp/bugs/790/(besucht am 21.08.2019)

4 Der Soll-Zustand

Dieser Abschnitt dient dazu das zu beschreiben was am Ende der Arbeit implementiert werden soll. Zu Anfang beschreibe ich jedoch erst einmal den Ausgangspunkt. Anschließend wo und wie ich die Anforderungserhebung durchgeführt habe und die Resultate.

Ausgangspunkt waren die Ideen, formuliert von Franz Ziers (Wissenschaftler der Gruppe Software Engineering [8]), die ich auf der Seite der offenen Abschlussarbeiten vorgefunden habe:

Background: In principle, Saros works orthogonal to traditional version control systems: It keeps the respective files in sync, regardless of whether they are part SVN or Git working copies – or just plain folders and files. This also means that, by default, any VCS actions (such as svn/git commit, svn checkout, git pull) need to be performed by both Saros session participants to achieve consistency of the meta-data. There are multiple possible implementations for Git support in Saros:

1. *The easy case: Both developers have access to a shared Git repo. For this centralized scenario, the Git implementation is (almost) straight-forward and consists of coordinated sequences of git push, git pull, and git checkout.*
2. *The Git-way (1): Given that one of the developer's repository can already be accessed by the other developer through one of the standard "Git ways"(e.g. SSH). Then, a session start can be achieved by a git pull; git commit should be followed by a git reset + git pull (or git reset on one side/=git push= on the other, respectively).*
3. *The Git-way (2): If none of the standard protocols between the two developers is possible, Saros could provide a bridge, e.g. if Alice and Bob are in a Saros session, Bob pulls from a "remote"repository at localhost: which is a proxy for Alice's Git repo – the actual communication is done via Saros.*

4.1 Anforderungsanalyse

Aus den gesammelten Informationen, aus den Erkenntnissen des Tests und durch Introspektion, habe ich ein *Szenario* formuliert welches mögliche Anforderungen an einen Git-Support darstellt. Das Issue #213[9] enthielt dieses und war der Startpunkt für zahlreiche Diskussionen auf der Plattform Github. Dort ist der Quellcode von Saros auch öffentlich zugänglich, also *Open-Source*. Auf der Plattform kann man u.a. den Code runterladen, Fehler am Programm oder an der Dokumentation angeben (in den sogenannten *Issues*) und Änderungen hochladen (in sogenannten *Pull Requests (PR)*. Ich habe zur Einführung meiner eigenen Idee zur Anforderung und zum Ausdiskutieren meist diese Plattform verwendet. Während der Entwicklung wurde der Chat Gitter für das Projekt Saros wiedereröffnet, sodass ich Diskussionen oder Fragen, die meist kurzlebige Probleme ansprechen hier auslagern konnte. Ein Beispiel für ein solches Problem findet sich im Issue #388 Neben den Online Plattformen habe ich großen Gebrauch von Gesprächen mit meinem Betreuer gemacht. In wöchentliches Meeting haben wir Schrittweise geplant und abgewogen. Auch habe ich beim Saros Meeting teilgenommen welches zwei Mal wöchentlich stattfindet. Die Teilnehmer sind meist nur die festen (studentischen) Entwicklern und Hr. Glaß. Kommilitonen, die wegen einer Arbeit im Projekt sind (wie ich) nehmen nur unregelmäßig am Meeting teil. Externe Entwickler sind nicht Teil des Meetings. Vorteil an diesem Meeting sehe ich weitere Teammitglieder zu sehen. Im Meeting wird reihum erzählt welche Arbeit im vergangenen Zeitraum geleistet wurde, wobei dieser Zeitraum sehr unterschiedlich sein kann (von drei Tagen bis mehrere Wochen). Genutzt habe ich diese Meetings um Fragen zu stellen, meine eigenen Fortschritte zu berichten, z.B. anhand eines Prototyps, und die Teammitglieder über relevante Entscheidungen aufzuklären. Außerhalb der Uni nutze ich das Forum der Entwickler der Bibliothek JGit und dessen Alternativen zu Github (Gerit und Bugtracker) zur Diskussion und zum Senden von entdeckten Bugs in der Bibliothek. [10] Außerdem habe ich den Entwickler-Chat

[8] http://www.mi.fu-berlin.de/w/Main/FranzZieris(besucht am 10.03.2019)

[9] Links sind einsehbar unter https://github.com/saros-project/saros/issues/ oder https://github.com/saros-project/saros/pull/ und anschließend der Nummer

[10] https://bugs.eclipse.org/bugs/show$_bug.cgi?id = 543390$ (eingereicht am 12.01.2019)

von Git benutzt um eine Entscheidung zu treffen, ob ich die API von Git benutze oder lieber selber die einzelnen Dateien aus dem *.git* Verzeichnis versende. Auch erhielt ich dort von einem Entwickler den Hinweis zur Nutzung von Bundles.

4.2 Anforderungen

Das Ergebnis der Anforderungsanalyse werde ich nun hier vorstellen wobei beim Abschluss der Arbeit die Grundfunktionalität implementiert werden sollte und falls noch Zeit bleibt die ergänzende Funktionalität. Als Grundfunktionalität wird das bezeichnet, was nach Anforderungserhebung am höchsten priorisiert wurde. Dabei hat sich herausgestellt das es ein bestimmtes Szenario gibt, welches eine Git Unterstützung in Saros wünscht. Dieses wird deshalb als **das erwartete Szenario** benannt.

4.2.1 Grundfunktionalität – Das erwartete Scenario

Alice und Bob arbeiten gemeinsam an der Datei Hallowelt.java mithilfe von Saros. Beide haben ebenfalls ein Repository in welchem sich diese Datei befindet. Sie sind beide einverstanden mit dem derzeitigen Ergebnis und wollen einen Commit erstellen (d.h. die Datei in ihrem jetzigen Zustand nun auch im Repository ablegen). Damit dies lokal bei Alice stattfindet, muss sie folgende Befehle ausführen: *git add Hallowelt.java ; git commit*. Nun möchte Bob innerhalb seines Repository dasselbe tun, jedoch will er keinen neuen Commit erstellen, weil dies innerhalb von Git zu Dopplungen führen würde. Das bedeutet, dass, wenn Beide unabhängig voneinander einen Commit erstellen es zwei Commits gibt, die dieselben Zustand beschreiben, aber einen anderen hash-Wert zugeteilt bekommen. Dies passiert aufgrund der unterschiedlichen Meta-Daten .

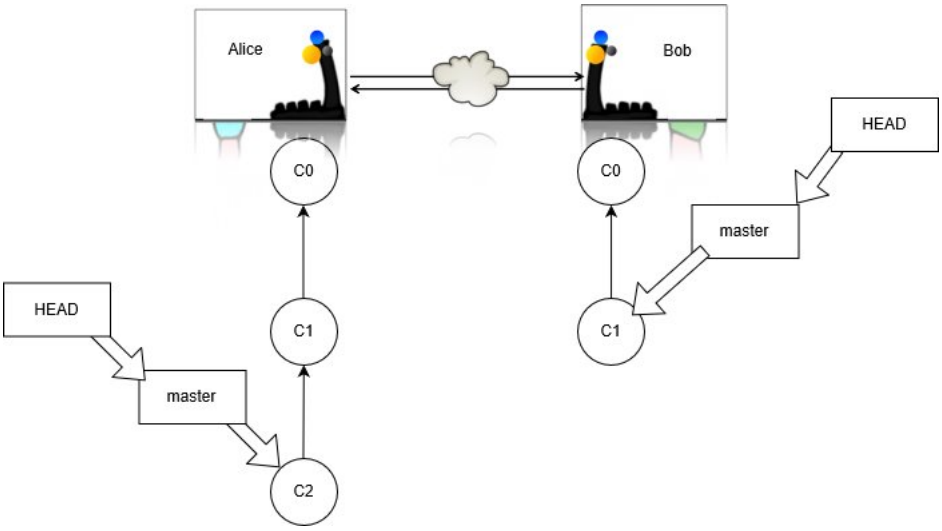

Abbildung 4: Diagramm für das erwartete Szenario (bearbeitet aus [12] mit [13])

Dies ist kein erwünschtes Verhalten deswegen muss Bob den von Alice erstellten Commit bei sich wiederfinden. Hierfür soll diese Arbeit eine Automatisierung anbieten. Allgemein gesagt soll sie erlauben die Differenz von Alice HEAD und dem von Bob auszugleichen. Dies soll dadurch erfolgen das Bob seinen aktuellen Commit an Alice sendet und die Differenz der Zustände berechnet wird. Die Differenz wird in eine Datei gepackt, die anschließend mithilfe von Saros zu Bob versendet wird. Daraufhin kann Bob manuell den Zustand von Alice Repository herstellen.

Um den Umfang der Grundfunktionalität einzuschränken treffe ich folgende Annahmen. Ich nehme an, dass Alice und Bob in einer Saros Session sind, beide lokal dasselbe Projekt besitzen und dieses zur

Versionskontrolle Git nutzt. Auch erwarte ich das beide Nutzer das **gesamte** Projekt per Saros teilen und dass es alle Änderungen bei Alice (also beim Sender) in einem Commit enthalten sind. Zusätzlich gehört noch zur Grundfunktionalität die Erstellung und Anpassung von Dokumentation. Vor allem soll die veraltete Information entfernt werden über die Einschränkung des Layouts von Projekten die Saros und Git gleichzeitig nutzen wollen. (siehe Abschnitt 3)

4.2.2 Erweiterte Funktionalität

- Die Abschwächung der Annahmen die in der Grundfunktionalität beschrieben sind. Insbesondere soll es möglich sein die neue Funktion zu benutzen auch wenn Bob keine Versionskontrolle nutzt und teilen auch wenn es Änderungen bei Alice gibt, die in keinem Commit enthalten sind.
- Die Annahmen sollen automatisch überprüft werden und es soll geregelt sein, was passiert, wenn die Überprüfung ein negatives Ergebnis gibt.
- Es soll eine Benutzeroberfläche (auf Englisch *Graphical User Interface* oder kurz GUI) geben. Die Elemente der GUI sollen es erlauben die Funktionen der Grundfunktionalität auszuführen. Auch soll es Benachrichtigung an den Benutzer geben, wenn die Annahmen verletzt sind, bei Erfolg und bei Versagen beim Senden oder Empfangen der Dateien. Dabei soll das Design der neuen Elemente sich an das bereits bestehende GUI anpassen.
- Ein Automatischer Merge der verschickten Differenz, sollte keine manuelle Anpassung notwendig sein (Fast-Forward-Merge).
- Erstellung von Unit- und Integrationstests zur Absicherung meiner Implementierung.

4.3 Nicht gewünschte Anforderungen

Bei der Analyse habe ich ebenfalls eine Anforderung gefunden, die nicht besteht, weil sie verändern würde wie Saros aktuell eine Session aufbaut. Meine Idee war es Anstelle des Versendens von Dateien beim Session Start, das Versenden von Repositories, die sowohl die Dateien als auch Inhalte der Versionskontrolle enthielten. Dadurch wäre das Versenden von Dateien womöglich nicht mehr notwendig. Siehe dazu das Issue #304.

5 Implementierung

In diesem Abschnitt werde ich nun meine Implementierung vorstellen und den Prototypen. Auch formuliere welche Annahmen beim Merge wichtig sind. Anschließen möchte ich diesen Abschnitt damit, dass ich die Formulierungen der Grundfunktionalität und der erweiterten nochmal aufgreife und angebe was tatsächlich implementiert wurde. Beginnen will ich mit wichtigen Entscheidungen, die ich vor der Implementierung treffen musste, weil sie vor allem die Technik bestimmen, die ich einsetzen muss.

5.0.1 Wichtige Entscheidungen

- Bundles zu benutzen ist eine der wichtigsten Entscheidungen, weil sie mir dies ermöglicht Protokollen zu benutzen die nicht von Git angeboten werden. Das ist vor allem deshalb nützlich, weil keine zusätzlichen Schwierigkeiten beim Aufbauen einer Verbindung durch meine Implementierung hinzukommen können, sondern ich *einfach* die bereits in Saros implementierten Protokolle nutzen kann. Es wird auch keine Authentifizierung benötigt, die ansonsten von Git gebraucht wird um verteilt zu arbeiten und es kommen keinerlei Probleme aufgrund von Firewalls. Außerdem können Bundles auch nur Teile des Repositories übertragen und damit verringert sich die Transfergröße um ein Vielfaches. Vor allem in Anbetracht des erwarteten Szenarios ist dies von großer Bedeutung, weil das bedeutet das die Übertragungszeit dort sehr kurz bleibt. Auch kann dadurch die Übertragung des Bundles mit Activities erfolgen (anstelle der schnelleren aber nur am Session Start implementierten direkte Verbindung).
- Activities nutze ich für die Verhandlung vor der Übertragung und für die Übertragung selbst. Die Wahl Activities zu benutzen, insbesondere um Commits zu verschicken, habe ich getroffen, weil das erwartete Szenario das Verschicken von **einzelnen** Commits beschreibt. Ist die Dateimenge klein genug (<1kB), was bei nur einzelnen Commits realistisch ist, bleibt das Versenden mit Activities effizient.[14] Das ermöglicht eine einfache Implementierung, weil das Erstellen von neuen Activities in der Architektur von Saros vorgesehen ist.
- Eine weitere wichtige Entscheidung ist es **nicht** Inhalte aus dem Git Verzeichnis direkt zu kopieren und zu versenden. Der Vorteil davon wäre es, dass nur Dateien die wirklich benötigt werden *angefasst* und übertragen werden. Der Nachteil ist das beim Empfänger nur zusätzliche Dateien eingefügt werden. Die *config* Datei welche Informationen über die lokalen Verzweigungen enthält oder die *HEAD* Datei müsste manuell verändert werden. Dieser Nachteil bedeutet das Dateien nicht nur übertragen werden müssen, sondern das beim Empfänger Dateien angepasst werden müssen. Außerdem ist es keine gute Idee, die internen Strukturen eines Repository zu verändern in Anbetracht der Anforderung, dass kein Datenverlust stattfinden darf.
- Nutzen werde ich die Bibliothek JGit um mit dem Versionsverwaltungssystem zu arbeiten. Alternativ wäre es möglich Befehle von Git direkt einzusetzen, welches jedoch die Integration in das das auf Java-basierende Saros stark erschwerte oder die Java-Bibliotheken gitblit [11] oder Git (library) von Project Nayuki[12] welche jedoch nicht mehr aktuell gehalten werden. JGit wird im Gegensatz aktuell gehalten. [13]
- Ich habe mich gegen die Nutzung von **statischen** Methoden entschieden, weil Klassen, die diese anbieten nicht in Tests ersetzt werden können, was das Testen erheblich erschwert. [15] [16]

[11] http://gitblit.com/index.html (besucht am 09.03.2019)
[12] https://www.nayuki.io/page/git-library-java (besucht am 09.03.2019)
[13] Der von mir erstellte *Bugreport* wurde kurze Zeit später, durch einen Entwickler bearbeitet. Das Problem ist gelöst und soll mit der Version 5.3 behoben werden, die am 20 März 2019 erscheint.

In diesem Teil der Arbeit werde ich nun genauer auf den bei der Arbeit entstandenen Programmcode eingehen. Die Klasse, die alle benötigten Funktionen aus der Bibliothek zur Verfügung stellen soll ist eine Fassade. Die Klasse die den Ablauf und den Transfer von Daten steuert ist der *GitManager*.

5.1 JGitFacade

Aufgrund dessen das ich für Saros die Nutzung der Bibliothek mithilfe des Fassaden-Musters erlauben möchte, habe ich mich entschieden, die folgenden Funktionen anzubieten.[14] Ermöglichen will ich das **Erstellen eins Bundles** und der **Fetch aus einem Bundle**. Auch soll es möglich sein **einen Merge durchzuführen** wenn ein **Fast-Forward-Merge** möglich ist und es keine Änderungen gibt die nicht innerhalb der Session sind. Eine weitere Funktion habe ich implementiert die es ermöglicht den **Hash eines Commits oder Branches** zu bekommen. Die Fassade dient in der gewählten Implementierung zum Bereitstellen der Funktionalität von Git und soll zur Laufzeit vom Benutzer als Objekt initialisiert werden. Übergeben werden muss das Verzeichnis in welchem sich das *.git* Verzeichnis befindet, also das Arbeitsverzeichnis. Im englischen wird dieses als **Working Directory Tree** bezeichnet und somit hier als **workDirTree**. [15]

5.1.1 Erstellen eines Bundles

```
public byte[] createBundle(String actual, String basis)
```

Das ist die Funktion[16] welche das Erstellen von Bundles erlaubt. Innerhalb von JGit wird ein Bundle mithilfe der *BundleWriter*-Klasse geschrieben. Diese benötigt (analog bei Nutzung von Git) die Angabe einer aktuellen Version und optional einer Basis. Als aktueller Zustand kann ein Branch oder der HEAD angegeben werden. Als Basis einen **Revision String** [17]. Somit kann ich mit der Eingabe die Anzahl der Commits festlegen, die im Bundle enthalten sein sollen. Will ich zum Beispiel nur den letzten Commit versenden, so ist actual gleich HEAD und basis HEAD^1 (dies ist der direkte Vorgänger vom HEAD). Die Implementierung erlaubt jedoch beliebige Spannen. So ist es zum Beispiel ebenfalls möglich die Basis leer zu lassen (also einen leeren Zeichenkette zu übergeben) und somit alle Commits im Repository einzupacken. Zwar erwarte ich das Versenden aller Commits nicht, jedoch beschränke ich die Funktionalität auf das erwartete Szenario erst mit dem Objekt der Klasse *GitManager*, welche ich nach der Fassade vorstellen will. Der Rückgabewert der Funktion ist ein *Bytearray*, weil diese Serialisierbar sind, heißt, sie können übertragen werden.

5.1.2 Fetch aus einem Bundle

```
public void fetchFromBundle(byte[] bundle)
```

Das *fetchen* vom Bundle habe in dieser Prozedur implementiert. Nochmal erinnern möchte ich, dass hiermit nur die Inhalte aus dem Bundle ins Repository geladen werden, nicht jedoch gemerged.

[14] Das Fassaden-Entwurfsmuster dient dazu die Funktionen eines komplexen Systems durch eine neue Stelle zu vereinfachen. Das Vereinfachen ist hierbei eine wichtige Unterscheidungsmerkmal, den insbesondere soll eine Fassade keine neuen Funktionalitäten anbieten.[17] Auch kann dadurch Abhängigkeit reduziert werden indem ein komplexes System (hier: eine Bibliothek) an nur eine Stelle vorkommt.[18]

[15] Eine Diskussion hierrüber ist unter dem Link https://stackoverflow.com/questions/39128500/working-tree-vs-working-directory (besucht am 13.02). Ich entscheide mich für die Bezeichnung weil ich die Argumentation von Nutzer *Timothy L.J. Stewart* nachvollziehbar finde. Die Bezeichnung als Tree folgt übrigens, weil Linux und Git sehr ähnliche Dateisysteme besitzen und dort ebenfalls Tree geläufig ist. Entwickler von Linux und Initiator von Git ist Linus Torvalds. Siehe dazu die Antwort des Nutzers *Greg Burghardt*

[16] Prozeduren haben Seiteneffekte und keinen Rückgabewert. Funktionen hingegen keine Seiteneffekte aber einen Rückgabewert. Trotz Diskussionen ob dies in der OO Programmierung nicht alles einfach als Methode benannt werden sollte, mag und mach ich diese Unterscheidung. https://stackoverflow.com/questions/2786796/do-you-call-them-functions-procedures-or-methods(besucht am 05.03)

[17] oder auch benannt als *gitrevision*. Diese dienen in Git zum Identifizieren der Zeiger.

5.1.3 Merge

```
public void fastForwardMerge(String revString)
```

Ich habe den Merge in dieser Prozedur implementiert. Der eingegebene String ist auch hier ein revision String. Dabei habe ich die Einschränkung gemacht das nur ein Fast-Forward-Merge möglich sein soll. Beispielsweise könnte ich für *revString* gleich *remote/bundle* angegeben, damit der *HEAD* auf den Branch *remote/bundle* gemerged wird. Weil ich nur einen Fast-Forward-Merge erlaube wird dieser also nur dann ausgeführt wenn *HEAD* ein Vorgänger von *remote/bundle* ist. Die Einschränkung folgt aus dem erwarteten Szenario. Weitere Szenarien unterstütze ich hier nicht, aber die entstehende Problematik will ich hier kurz erwähnen. Mergen wir in einen Branch der nicht Vorgänger des anderen ist, könnte der Empfänger nach dem Merge einen weiteren Commit erstellen der die Branches verbindet (einen sogenannten Merge-Commit). Das Problem ist, das dieser dann wieder zum Sender übertragen werden muss.

Es kann, auch wenn ich nur einen Fast-Forward-Merge erlaube, zu Überschreibungen kommen. Um dies vorzubeugen formuliere ich weitere Einschränkungen, wobei ich Empfänger als auch Sender einzeln betrachten kann. Dabei nehme ich an, dass alle Dateien, die sich im workDirTree befinden, und somit von Git beobachtet werden, auch von Saros geteilt werden. Das sind Dateien die *Unmodified* sind, aber auch *Untracked* und *Modified* (Ich erinnere an Abbildung 2). Dies ist im Allgemeinen erwartet. Abgesichert kann dies jedoch nicht, weil ich nicht ohne unverhältnismäßigen Aufwand innerhalb von Saros Dateien, die in einer Session geteilt werden mit denen Vergleichen kann die von Git im workDirTree genutzt werden.

Im Allgemeinen ist die Nutzung von *git merge* abzuraten, wenn es *non-trivial uncommitted changes* gibt , weil es hierbei zu Änderungen kommen kann die nicht zurückgesetzt werden können. [19][18] Bei dem erwarteten Szenario muss es aber Änderungen geben, weil Bob innerhalb der Session an seinen lokalen Dateien Änderungen gemacht hat, aber keinen eigenen Commit erstellen will in welchem diese vorkommen. **Das heißt es muss Sichergestellt werden, das der workDirTree durch den Merge nicht verändert wird.** Eine Überschreibung einer Datei, nach einem Merge bei Bob, würde nämlich dazu führen, dass sich die Datei ebenfalls bei Alice ändert.

Einen Fast-Forward-Merge durchführen ohne das Änderungen entstehen kann ich dann wenn die Vorbedingung gilt das es keine *Fehlenden Änderungen* geben darf (Das sind solche die im Index enthalten ist, aber nicht im Arbeitsverzeichnis). Dateien die diese Änderungen enthalten erhalten wir bei JGit mit der Funktion getMissing(). Können wir gewährleisten das es keine solchen Dateien gibt, folgt die Nachbedingung, dass es keine Änderungen zwischen dem HEAD, dem Index und dem Arbeitsverzeichnis gibt.[19] Diese Bedingungen, die ich im Programmcode abprüfe, habe ich hier noch einmal in OCL zusammengefasst.

Listing 1: Vor- und Nachbedingungen für einen Merge beim Empfänger (in OCL)

```
context JGitFacade::fastForwardMerge(String revString) pre:
        status.getMissing() -> isEmpty()

context JGitFacade::fastForwardMerge(String revString) post:
        status.isClean() = true
```

Zum Zustand von Alice (also des Senders): Sie sollte keine Dateien im workDirTree enthalten die im Zustand Modified oder Untracked sind (wieder verweise ich auf Abbildung 2). Mithilfe von *git diff HEAD* kann der Benutzer solche sehen. Da der HEAD von Alice somit **genau** die Dateien enthält die auch über Saros geteilt werden und wir im Anschluss an den Merge bei BOB diesen HEAD vorfinden und auch dort alle von Saros sichtbaren Dateien genau wie im HEAD sind, **können wir absichern das der Merge keine Änderungen vornimmt**.

[18] was dies bedeutet sind kann in Abschnitt 2 unter git merge eingesehen werden.

[19] Zustände eines Repository sind in Glossaren einheitlich als *Clean* und *Dirty* benannt, aber in verschiedenen Quellen unterschiedlich definiert. Die hier gewählte Definition folgt aus der Dokumentation der verwendeten booleschen Funktion von JGit. Eine Diskussion darüber unter folgendem Link https://stackoverflow.com/questions/20642980/does-git-dirty-mean-files-not-staged-or-not-committed-glossary-conflict (besucht am 13.02.2018).

Der Index darf im Falle des Senders beliebige Änderungen beinhalten, also vor allem auch solche die sich zum HEAD bzw. zum workDirTree unterscheiden. Die von Saros geteilten Dateien *merken* den Index nicht, wenn wir ein Bundle erstellen. Somit sind sie ungefährlich.

5.1.4 Hash eines Commits oder Branches

```
public String getSHA1HashByRevisionString(String revString)
```

Benötigen wir den SHA1-Hash dann können wir diese Funktion verwenden. Ich werde diese benötigen um Bob's *HEAD* an Alice zu senden, um anschließend die *richtige* Größe des Bundles festzulegen.

5.2 GitManager

Die Klasse, die den Ablauf steuert, werde ich nun hier vorstellen. Der Ablauf besteht dabei aus der Verhandlung welche Commits übertragen werden sollen und aus der Übertragung selbst. Hierzu wurden 3 neue Aktivitäten erstellt, die jeweils zuständig sind für das Versenden der Anfrage, das Versenden des SHA1-Hash und für das Versenden der Commits. Auf beiden Seiten sorgt ein Objekt der Klasse *GitManager* dafür dies zu steuern.

```
public class GitManager extends AbstractActivityProducer
```

Als Erbe eines *Producers* ist diese Klasse zum Erstellen von *Activities* befähigt. Ebenso wird beim Start des Objekts ein *Consumer* innerhalb des Objektes initialisiert welcher es erlaubt *Activities* zu empfangen. Durch Versenden eines Send Request wird eine Anfrage an alle anderen Nutzer der Session gesendet. Weil für jeden Benutzer ein Objekt der Klasse erstellt wird, kann darin auch das zu benutzende Arbeitsverzeichnis abgespeichert werden. Dieses soll zu Beginn einer Session von jedem Benutzer eingestellt werden und kann im Laufe der Session auch wieder gewechselt werden. Eine Automatische Einstellung findet nicht statt, weil es keine Einschränkungen für seinen Speicherort gibt.

Das folgende Diagramm soll den soeben beschrieben Ablauf veranschaulichen den der GitManager steuert.

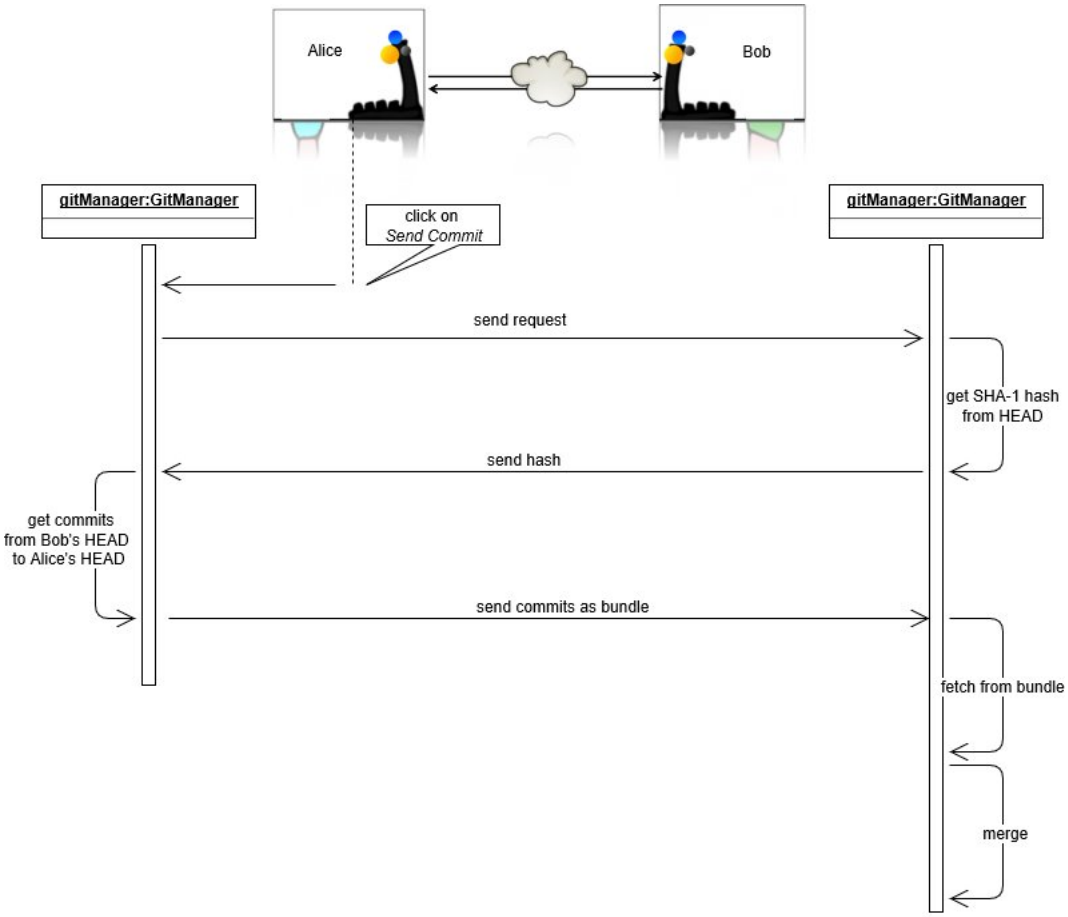

Abbildung 5: Diagramm für die implementierte Nutzung (bearbeitet aus [12] mit [13])

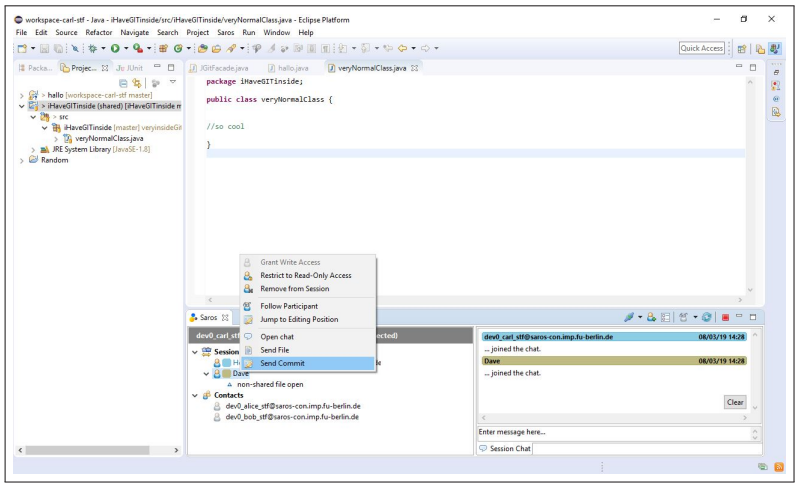

Abbildung 6: All der zuvor beschriebene Code befindet sich im Core. Das bedeutet das die Implementierung unabhängig von der benutzen IDE die Funktionen anbietet. Die GUI muss speziell für die IDE implementiert werden oder in die IDE unabhängige HTML GUI, welche sich jedoch noch in Arbeit befindet. Ich habe mich für die Implementierung in die Eclipse abhängige GUI entschieden. Hier sichtbar ist ein Screenshot des implementierten Knopfs in Eclipse.

5.3 Implementierung der Grundfunktionalität

Die Grundfunktionalität konnte implementiert werden. In meiner Arbeit habe ich die Klassen und Methoden mit besten Gewissen kommentiert so dass zukünftige Entwickler mit diesen Arbeiten können. Meine Ergebnisse des Tests und die Erkenntnisse zum *Safe* und *Risky* Layout (siehe Abschnitt 3) habe ich habe ich den anderen Entwicklern mitgeteilt im Issue 472.

5.4 Implementierung der erweiterten Funktionalität

5.4.1 Abschwächung der Annahmen

Durch die Abschwächung der Annahmen will ich alle Szenarios, bei Nutzung von Saros mit Git, sinnvoll behandeln. Sinnvoll bedeutet dabei nicht immer eine Automatisierung, sondern es kann auch zum Beispiel eine Mitteilung an den Benutzer oder ein Hinweis/Beschreibung in der Dokumentation sein. In diesem Abschnitt habe ich dazu die in der Grundfunktionalität erwähnten Annahmen aufgelistet und wie ich sie abschwäche.

Sender und Empfänger besitzen lokal dasselbe Projekt
Bei der Implementierung der Grundfunktionalität treffe ich keine Annahmen über den Zustand eines Projekts. Wichtig ist nur das auf beiden Seiten lokal ein workDirTree existiert und in ihm das *.git* Verzeichnis. Damit ist diese Annahme nicht mehr notwendig.

Beide nutzen für das Projekt Git
Diese Annahme muss wahr bleiben. Es wird in keinem Fall ein neues Git Verzeichnis eingerichtet. Es hat sich gezeigt das erwartet wird, das beide Nutzer ein Verzeichnis mit Versionsverwaltung bereits besitzen.

Alle Änderungen beim Sender sind in einem Commit enthalten und **Das gesamte Projekt wird per Saros geteilt**
Will der Nutzer einen Merge ausführen, so gelten beide Annahmen. Ohne den Merge sind auch diese Annahmen nicht mehr notwendig.

Sender und Empfänger sind in einer Saros Session
Diese Annahme muss auch weiterhin wahr sein. Das folgt einerseits aus dem erwarteten Szenario und andererseits kann nur so die Implementierung funktionieren, weil Activities nur innerhalb einer Session verschickt werden können.

5.4.2 Automatische Überprüfung der Annahmen und Merge

Im vorherigen Punkt habe ich bereits die Handhabung bei Verletzung der Annahmen beschrieben. Eine automatische Überprüfung für die Annahmen, die nicht mehr gelten entfällt.

Für die Annahme **Beide nutzen für das Projekt Git** existiert eine Überprüfung, den sowohl beim Sender als auch Empfänger werden Methoden benutzt die ein gewähltes workDirTree verlangen welches wiederrum ein *.git* enthalten muss.

Die Annahme **Sender und Empfänger sind in einer Saros Session** wird folgendermaßen überprüft: Existiert keine Session, so existiert auch das *GitManager* Objekt nicht weil es erst zum Session Start initialisiert wird. Das bedeutet das Funktionen erst dann verfügbar sind, wenn Sender und Empfänger in einer Saros Session sind und somit ist die Überprüfung automatisiert.

Die Überprüfung der Annahmen **Das gesamte Projekt wird per Saros geteilt** und **Alle Änderungen sind bei Alice(also beim Sender) in einem Commit enthalten** konnten im Rahmen dieser Arbeit nicht automatisiert werden. Diese müssen beim Alice manuell überprüft werden und zwar dann, wenn sie einen Merge bei Bob erwartet. Beim Bob habe ich andere Annahmen gefunden, die ich wiederrum

automatisch abprüfe. Die Implementierung des Merge selbst habe ich auch geschafft. So kann zusammengefasst werden das ich beim Merge das erwähnte Ziel erreicht habe, und zwar einen Merge durchzuführen, wenn ein automatischer Merge möglich ist. Beim Festlegen der Zielsetzung habe ich fälschlicherweise noch gedacht das es ausreicht einen Fast-Forward-Merge durchzuführen.

5.4.3 GUI

Mein Ziel war es eine GUI für die Nutzung der eigenen Funktionen und für Benachrichtigungen zu erstellen. Die GUI sollte dabei an das bisherige GUI angepasst werden. Eine Implementierung für die Eclipse GUI für *Send Commit* und *Change Working Directory Tree* existiert und kann dort bereits genutzt werden. Eine Implementierung für die IntelliJ GUI oder die HTML GUI habe ich nicht implementiert, weil beide Projekte noch in Arbeit sind. Die Benachrichtigungen konnte ich im Rahmen der Arbeit nicht in die GUI implementiert. Bemüht habe ich mich nochmal darum genauer auszuformulieren wie ich mir die GUI im Gesamten vorstelle, also auch wann Benachrichtigung stattfinden sollen. Diese spare ich mir an dieser Stelle jedoch und verweise auf den Ausblick.

5.4.4 Erstellung automatischer Tests

Für die implementierten Methoden der JGitFacade und des Managers existieren Testfälle für die erwarteten Szenarios (sogenannten *happy paths*), aber auch für die Fälle in welchen Fehler erwartet werden. Dies sind vor allem solche in welchen *falsche* Parameter übergeben werden.

6 Schluss

Im folgenden Abschnitt werde ich die Arbeit in einigen Sätzen zusammenfassen und abschließend mit einem Ausblick formulieren wie zukünftige Arbeiten diese ergänzen können.

6.1 Zusammenfassung

Der Git-Support in Saros konnte mit dieser Arbeit in folgenden Punkten erreicht werden:

- Es existiert die Möglichkeit bei Nutzung von Saros seinem Partner (oder seinen Partnern) neue Versionen/Commits zuzusenden. Dafür ist keine zusätzliche Verbindung notwendig. Sie werden automatisch in seine Revision gefetched. Danach können sie bei ihm gemerged werden. Ziel bzw. das Szenario bei welchem dies eingesetzt werden soll, ist wenn nach gemeinsamer Bearbeitung des Programmcodes eine neuer Commit bei einem der Partner erstellt werden soll und dem/den anderen dieser nur noch gesendet.
- Der vorherige Punkt ist die gewünschte Anforderung an eine Unterstützung von Git. Es wurden ebenfalls Anforderungen gefunden die **nicht** existieren. Eine davon: Beim Session Start ein Repository senden, anstatt nur die Dateien.
- Eine GUI -Implementierung des ersten Punkts für die IDE Eclipse
- Das Einbinden von ausgewählten Funktionen der Bibliothek JGit über die *JGitFacade*
- Das Einführen vom *GitManager*, einem Objekt welches beim Benutzer am Beginn einer Sitzung erstellt wird und welches Zuständig ist den Speicherorts festzuhalten in welchem sich das *.git* Verzeichnis befindet und für das Steuern beim Versenden von Commits
- Den implementieren von Merge. Diesen habe ich so implementiert das Dateien, die über Saros geteilt werden respektiert werden. Dazu habe ich die Annahme formuliert das alle Dateien im Arbeitsverzeichnis auch von Saros geteilt werden, weil dies im Allgemeinen erwartet ist. Ich unterscheide, beim Empfänger, den Zustand in denen Änderungen im Index, aber nicht im Arbeitsverzeichnis enthalten sind (also auch nicht von Saros sichtbar) und der wo dies nicht der Fall ist. Für den ersten Fall erlaube ich keinen Merge. Der Sender ist nicht so stark eingeschränkt aber bei ihm gilt die Einschränkung: Alle Änderungen sollten in einem Commit enthalten sein, wenn sie von Saros sichtbar sind. Im Programmcode habe ich die Einschränkung des Empfängers abgesichert und auch das es nur ein Fast-Forward-Merge sein darf.

6.2 Ausblick

Außerhalb des erwarteten Szenarios ist ein Git-Support nur mit weiteren Überlegungen möglich. Dazu will ich die Ideen verschriftlichen, die ich während der Arbeit entwickelt habe.

6.2.1 Merge

Ein Vorteil wäre es, wenn ich Dateien, die in einer Session sind mit denen vergleichen kann die im Repository sind. So könnte ich für z.B. die Schnittmenge die bereits implementierten Vorgehensweise wählen und für den Rest (also Dateien die nicht von Saros geteilt werden aber im Repository vorhanden oder umgekehrt) eine neue. Hier wäre es sinnvoll eine Unterscheidung in Zustände zu treffen bei denen keine Überschreibung passiert und solche bei welchen dies automatisch passiert. Dazu wäre die Kenntnis

über den genauen Vorgang beim Merge vorteilhaft [20] und wie wir dies in der Bibliothek verwenden können [21]. **Damit müsste ich zukünftig beachten, dass beim Empfänger Inhalte des workDirTree die nicht über Saros geteilt werden durch den Merge verändert werden könnten**. Für die neuen Vorgehensweisen bietet sich an das Teilen (also das Versenden von Activities) für die Zeit des Merge zu stoppen und den Einsatz von *git stash* [22]

6.2.2 Benachrichtigung und GUI

- Wenn Alice auf *Send Commits* geklickt hat soll Bob sich in einem Fenster entscheiden können ob er die Commits von Alice empfangen möchte oder nicht.
- In beiden Fällen soll Alice die Antwort erfahren.
- Zusätzlich soll Alice bei positiver Antwort erfahren wenn sie den aktuellen Hashwert des *HEAD*s von Bob erhalten hat.
  - Sollte der Bob ein Git Verzeichnis ausgewählt haben welches eine Version besitzt die Alice noch nicht hat (technischer: Kann der Hash Wert von Bob's *HEAD* nicht in der Versionskontrolle von Alice gefunden werden) so werden beide über einen Fehlschlag benachrichtigt.
  - Anschließend wird Bob dazu aufgefordert ein neues Git Verzeichnis zu wählen. Bei der Benachrichtigung soll trotz Geheimnisprinzip stehen, das der Hash des *HEAD* beim Sender nicht gefunden werden konnte, damit es diesem durch diese Benachrichtigung möglich ist den geforderten Zustand manuell einzurichten.
- Sollte der Hash erfolgreich als Vorgänger identifiziert werden sollte eine Nachricht kommen wenn das Bundle erstellt wurde und sobald sie das Bundle gesendet hat.
- Bob sollte anschließend erfahren, wenn er die Commits erhalten hat, sie gefetched wurden und danach eine Auswahl treffen dürfen ob gemerged werden soll oder nicht. Zusätzlich sollte im Auswahlbildschirm für den Merge stehen welche Annahmen getroffen wurden, sowohl für die abgesicherten, als auch die die nicht durch das Programm überprüft werden können.

- Beim Klick auf *Change Work Directory Tree* und *OK* sollte der Benutzer sehen können welchen er aktuell ausgewählt hat. Wichtig wäre auch das beim Start einer Session angezeigt wird das er noch eins auswählen muss.
- Hat der Sender kein Verzeichnis ausgewählt und will die neue Funktion benutzen wird er aufgefordert eines auszuwählen.
- Hat der Empfänger kein Verzeichnis ausgewählt, wird auch er dazu aufgefordert.

Generell halte ich es für sinnvoll beim Starten den Nutzer auswählen zu lassen ob er mit dem Git-Support arbeiten möchte. Beim ersten Mal sollten auch die implementierten Funktionen und wichtigsten Annahmen genannt werden.

[20] (besucht am 12.02.2019)
https://www.quora.com/How-does-Git-merge-work
https://stackoverflow.com/questions/14961255/how-does-git-merge-work-in-details
https://stackoverflow.com/questions/1824264/what-is-a-trivial-merge-in-git
https://github.com/git/git/blob/master/Documentation/technical/api-merge.txt

[21] https://github.com/eclipse/jgit/blob/master/org.eclipse.jgit.test/tst/org/eclipse/jgit/merge/SimpleMergeTest.java

[22] Wollen wir unsere Dateien im Arbeitsverzeichnis zwischenspeichern ohne einen neuen Commit zu erstellen können wir diesen Befehl dazu nutzen. Der *Stash* ist dabei der Ort wo die Änderungen gespeichert werden.

6.2.3 Alternativer Session Start

In der im Abschnitt 1 erwähnten Arbeit habe ich ebenfalls eine weitere Idee für die Benutzung bekommen. Dazu möchte ich erneut eine Stelle aus dieser zitieren.

> Ein möglicher Anwendungsfall bei der Benutzung von Saros wäre auch sich den Projektcode vom Repositorium zu laden und dann auf Basis dieser Version die Sitzung einzuleiten.

Der Versand des Hashes der aktuellen Version an den Partner könnte den im Zitat erwähnten Anwendungsfall ermöglichen. **Vor** der Session könnten die selben Version, mithilfe eines zugeschickten Hash manuell eingestellt werden. Meine Implementierung erlaubt bereits das Auslesen des Hashes. Sie müsste noch erweitert werden, so dass dem Partner der Hash auch außerhalb der Session zugesendet werden kann. Davor sollte die Priorität dieser Anforderung sichergestellt werden.

6.2.4 ignorieren des *.git* Verzeichnis

Die Vermutung das sich das*.git* Verzeichnis nur im selben Verzeichnis befinden darf wie das Projekt (siehe Abschnitt 3), weil es ansonsten ebenfalls von Saros geteilt wird, führt zu einer Diskussion im Github im Issue 370 (besucht am 08.03.2019). Nach anschließender Diskussion im Stand-Up hat sich eine gewünschte Anforderung herausgestellt, die das ignorieren von Dateien für eine Session wünscht. Die gewünschte elegante Lösung wäre es eine erweiterbare *schwarze* Liste zu erstellen die Dateien zum Teilen über Saros ausschließt.[23] Diese Lösung ist im Rahmen der Bachelorarbeit zu aufwendig.

Ich will auch hier nochmal auf Abschnitt 1 in [1] und 3 verweisen, da es bereits eine Lösung gibt die Dateien von Saros ausschließt. Im Test konnte dies auch bewiesen werden. Dies sind Dateien die als *derived* oder als *team-private* gekennzeichnet sind. Ich vermute das dies automatisch auch mit dem *.git* Verzeichnis geschieht. Da dies eine Mechanik ist, die sich nur auf Eclipse einschränkt müssten Saros-Nutzer von anderen IDE's andere Lösungen finden. Glücklicherweise hat IntelliJ eine analoge Lösung jedoch mit der Bezeichnung *excluded*. Eine Benachrichtigung an den Nutzer könnte eine sinnvolle Zwischenlösung sein:
Setzen sie ihr Git Verzeichnis auf *derived*(Eclipse) bzw. *excluded*(IntelliJ) damit dieses nicht geteilt werden kann.

6.2.5 Dokumentation

Die Website vom Projekt Saros wird bald unter einer neuen Adresse erreichbar sein [12] und wird dabei überarbeitet. Die Konzepte und Annahmen aus dieser Arbeit sollten sobald die neue Website steht dort platziert werden.

6.2.6 Working Directory Tree oder .git Verzeichnis

While FileRepositoryBuilder has a setGitDir() as well as a setWorkTree() method, I recommend to always use setGitDir(), because (B)y default, the git directory is a direct subdirectory of the work directory, but this can be overridden through an environment variable or a configuration setting. And then there are bare repositories that don't have a work directory at all.[20]

Im jetzigen Zustand habe ich das Working Directory Tree gewählt, weil die Arbeit mit dem *.git* Verzeichnis nicht funktionierte. Beim Versuch habe ich Fehlermeldungen erhalten das auf das Repository nicht zugegriffen werden konnte.

[23] Ironischerweise hat der Testfall (siehe #472) aufgedeckt das die Datei *.gitignore*, die eine solche schwarze Liste für die Software Git ist, durch Saros geteilt wird.

Abbildungsverzeichnis

Referenzen

[1] Riad Djemili. *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung*. Freie Universitat Berlin. 2006.

[2] Andreas Haferburg. *Integration of Version Control Systems in a Tool for Distributed Pair Programming*. Freie Universitat Berlin. 2010.

[3] Scott Chacon und Ben Straub. *ProGit*. Webseite. 2018. URL: https://git-scm.com/book/de/v2 (besucht am 06. 08. 2018).

[4] Rakhesh Sasidharan (aka. rakhesh). *How to move/ separate the .git folder out of your working tree*. 19 Februar 2014. URL: https://rakhesh.com/coding/how-to-move-separate-the-git-folder-out-of-your-working-tree (besucht am 24. 01. 2019).

[5] Christoph Burgdorf. *The anatomy of a Git commit*. November 18, 2014. URL: https://blog.thoughtram.io/git/2014/11/18/the-anatomy-of-a-git-commit.html (besucht am 03. 08. 2018).

[6] Scott Chacon und Ben Straub. *ProGit - Git Protocols*. Webseite. 2018. URL: https://git-scm.com/book/de/v2/Git-on-the-Server-The-Protocols (besucht am 06. 08. 2018).

[7] Saros Team. *Website des Projekts Saros - Activities*. URL: http://www.saros-project.org/activitySending (besucht am 08. 01. 2019).

[8] Saros Team. *Website des Projekts Saros - Overview*. URL: http://www.saros-project.org/specoverview (besucht am 08. 01. 2019).

[9] Eclipse Foundation. *JGit Website*. URL: https://projects.eclipse.org/projects/technology.jgit (besucht am 09. 03. 2019).

[10] Twitter Nutzername frodriguez. *MVN Repository - Usage of JGit*. URL: https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit/usages (besucht am 09. 03. 2019).

[11] Saros Team. *Website des Projekts Saros - FAQ*. URL: http://www.saros-project.org/faq (besucht am 08. 01. 2019).

[12] Saros Team. *Neue Website des Projekts Saros (noch in Arbeit)*. 2019. URL: http://saros-project.github.io/saros/ (besucht am 08. 03. 2019).

[13] JGraph Ltd. *draw.io*. URL: https://www.draw.io/ (besucht am 02. 02. 2019).

[14] *Diskussion ber Nutzung von Activities*. URL: https://github.com/saros-project/saros/issues/324#issuecomment-445760423 (besucht am 09. 03. 2019).

[15] Lasse Koskela. *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Oktober 19, 2007.

[16] Miko Hevery. *Static Methods are Death to Testability*. Dezember 17 2008. URL: https://testing.googleblog.com/2008/12/static-methods-are-death-to-testability.html (besucht am 07. 03. 2019).

[17] E. Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[18] Alexander Shvets. *refactoring.guru*. URL: https://refactoring.guru/design-patterns/facade/java/example (besucht am 08. 01. 2019).

[19] Scott Chacon et. al. *man page git merge*. URL: https://git-scm.com/docs/git-merge#_description (besucht am 12. 02. 2019).

[20] Rdiger Herrmann. *How to Access a Git Repository with JGit*. September 22, 2014. URL: https://www.codeaffine.com/2014/09/22/access-git-repository-with-jgit/ (besucht am 08. 03. 2019).

7 Beiträge im GitHub

7.1 Issue's

- #213 Scenario for Git support in Saros
- #214 Start a new session with Git
- #301 Adding the Git Support into the Invitation Process
- #304 Implementation of the Git Support
- #324 Send a commit via Saros
- #370 Ignore the .git directory during a Saros session
- #388 Amount of activities
- #472 Delete wrong faq post regarding working with Git

7.2 PR's

- #428 [CORE][FEATURE]add JGitFacade and Test
- #444 [CORE][FEATURE] Add Manager And Activities To Share Commits
- #470 [CORE][FEATURE] Add merge to Git support
- #471 [E] Add buttons for change (Git) Work Dir Tree and Send Commit

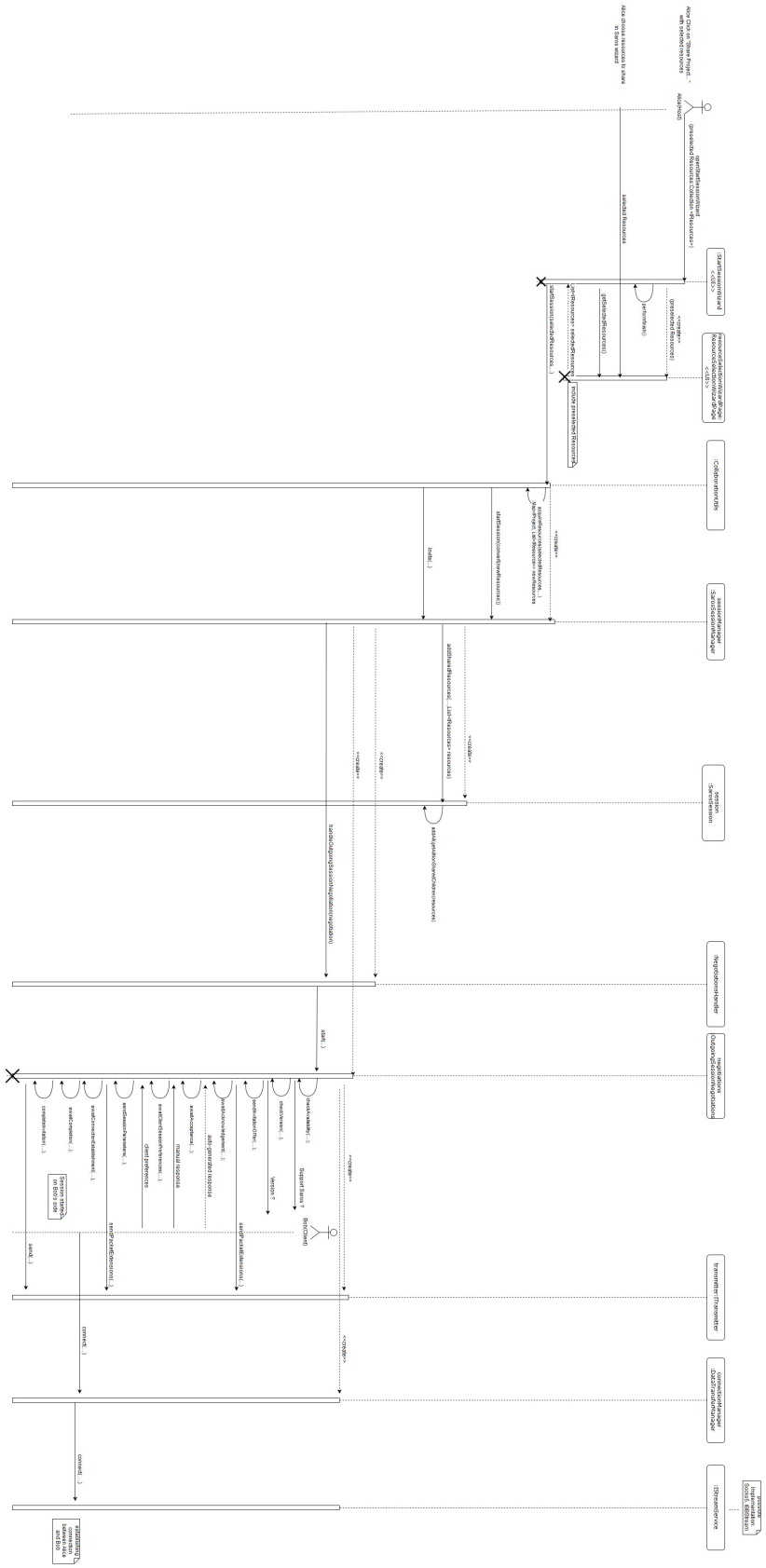

Abbildung 7: UML Diagramm für Saros Session Start (gezeichnet mit [13])

Danksagung

Abschließend möchte ich mich bei all denen bedanken die mich beim Schreiben der Arbeit unterstützt und mir damit zum Abschluss meines Bachelors verholfen haben.

Zu aller erst bei meinen Eltern. Ihr habt dies ermöglicht. Danke auch an meine Brüder und Großmutter für die Motivation. Namentlich möchte ich noch diejenigen nennen die diese Arbeit gelesen haben und mir Rückmeldung gaben: Abdullah Barhoum, mein Betreuer Kelvin Glaß, Christian Claus und Rainier Raymond Robles. Nicht vergessen möchte ich die Entwickler von Saros. Ich habe gerne an diesem Projekt mitgearbeitet.