Hochschule München

Fakultät 07

Studiengang Bachelor Wirtschaftsinformatik

Generalthema:

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Thema der Studienarbeit: Versionsverwaltung mit Git

Betreuer: Michael Theis

Verfasser: Christian Ondreka

Fachsemester: 4. Semester

Abgabedatum: 24.05.2013

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Bevor ich mit Ausführungen zum Versionierungssystem Git beginne, soll erst der Nutzen einer Versionsverwaltung geklärt werden. Dieser wird schnell klar, wenn man sich Ablauf und Bedingungen eines typischen Software-Projektes vergegenwärtigt. Meistens arbeiten mehrere Leute an einem Projekt. Daraus lassen sich viele Gründe für den Einsatz einer Versionsverwaltung ableiten:

- „Letzte Änderungen rückgängig machen“[1]
- Projekt auf einen vergangenen Stand zurücksetzen[2]
- Aus Versehen gelöschte Dateien wiederherstellen[3]
- Riskante Änderungen ausprobieren[4]

Beim genaueren Betrachten der genannten Gründe fällt auf, dass diese auch von einem vernünftigen Backup-Programm bewältigt werden können. So wäre es denkbar, täglich mit einer Backup-Software zeitgesteuerte Sicherungen durchzuführen[5]. Dies lässt Programme zur Versionsverwaltung auf den ersten Blick als überflüssig erscheinen. Allerdings gibt es – besonders wenn es zur verstärkten Teamarbeit im Projekt kommt – eine Reihe von Gründen für den Einsatz einer Versionierungssoftware:

- Änderungen anderer Entwickler mitbekommen[6]
- Festellen „wer, wann, was und wieso geändert hat“[7]
- Verhindern, dass „mein Partner gestört wird, während ich am Quelltext experimentiere“[8]

1 Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4.
2 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4.
3 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4.
4 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4.
5 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4
6 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 5.
7 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 5
8 Vgl. Hafner, U., Versionsverwaltung mit Git und Subversion, 2013, S. 4.

1.2 Tools für die Versionsverwaltung

Wie es für jeden Zweck eine Vielfalt an verfügbarer Software gibt, so gilt das auch für den Bereich der Versionsmanagement-Tools. Einige namhafte Beispiele sind Subversion, Git, Bazaar, CVS und GNU arch.[9] Diese Programme unterscheiden sich hauptsächlich in Bedienung (Kommandozeilenbefehle, Oberfläche, etc.) und ob es sich um ein zentrales oder ein dezentrales System handelt. Dazu mehr im nächsten Abschnitt.

1.3 Begriffsdefinitionen

Im folgenden werden wichtige Begriffe erklärt, die von zentraler Bedeutung für den Hauptteil meiner Arbeit sind. Für den Leser ist es entscheidend, sich der Definitionen bewusst zu sein, um meinen Ausführungen folgen zu können.

Repository: „Ein Repository ist eine serverbasierte Datenstruktur zur Speicherung der Änderungen an Dateien und Verzeichnissen“[10]. Vereinfacht ausgedrückt enthält ein Repository archivierte Verzeichnisbäume – also ein oder mehrere Verzeichnis(se) und die darin enthaltenen Unterverzeichnisse mit Quelltext – des lokalen Verzeichnissystems.

Commit: Neben dem Repository der wichtigste Begriff. Ein Commit bezeichnet den Entwicklungsstand eines Projekts. Oft wird auch von einem Versionsstand gesprochen. Zudem gibt es den gleichnamigen Befehl commit. Der Begriff wird zum einen für den Entwicklungsstand (Zustand) an sich und zum anderen zum Speichern (Aktion) eines definierten Versionsstandes verwendet.

Branch: Ein Branch ist ein einzelner Entwicklungszweig. Der Synonym Ast ist auch gebräuchlich. Vermehrt kommt es vor, dass mehrere Entwickler in einem Projekt arbeiten. In diesem Fall ist die Bildung von Verzweigungen quasi unvermeidbar.

Workspace: „Ein Projektverzeichnis mit einem Repository nennt man einen Workspace“[11].

Update: Auch dieser Befehl ist bei allen Programmen verfügbar. Es wird ein Abgleich zwischen dem Repository und dem Working Directory durchgeführt. Falls das Repository aktuellere Versionen enthält, wird das Working Directory entsprechend aktualisiert.

---

9 Vgl. ubuntu Deutschland e. V. Versionsverwaltung. http://wiki.ubuntuusers.de/Versionsverwaltung
10 Ullrich Hafner.: Versionsverwaltung mit Git und Subversion, a. a. O., S. 6.
11 Vgl. Preißel, R./Stachmann B., Git, 2012, S. 45.

2. Grundlegende Konzepte

2.1 Prinzip der Versionsverwaltung

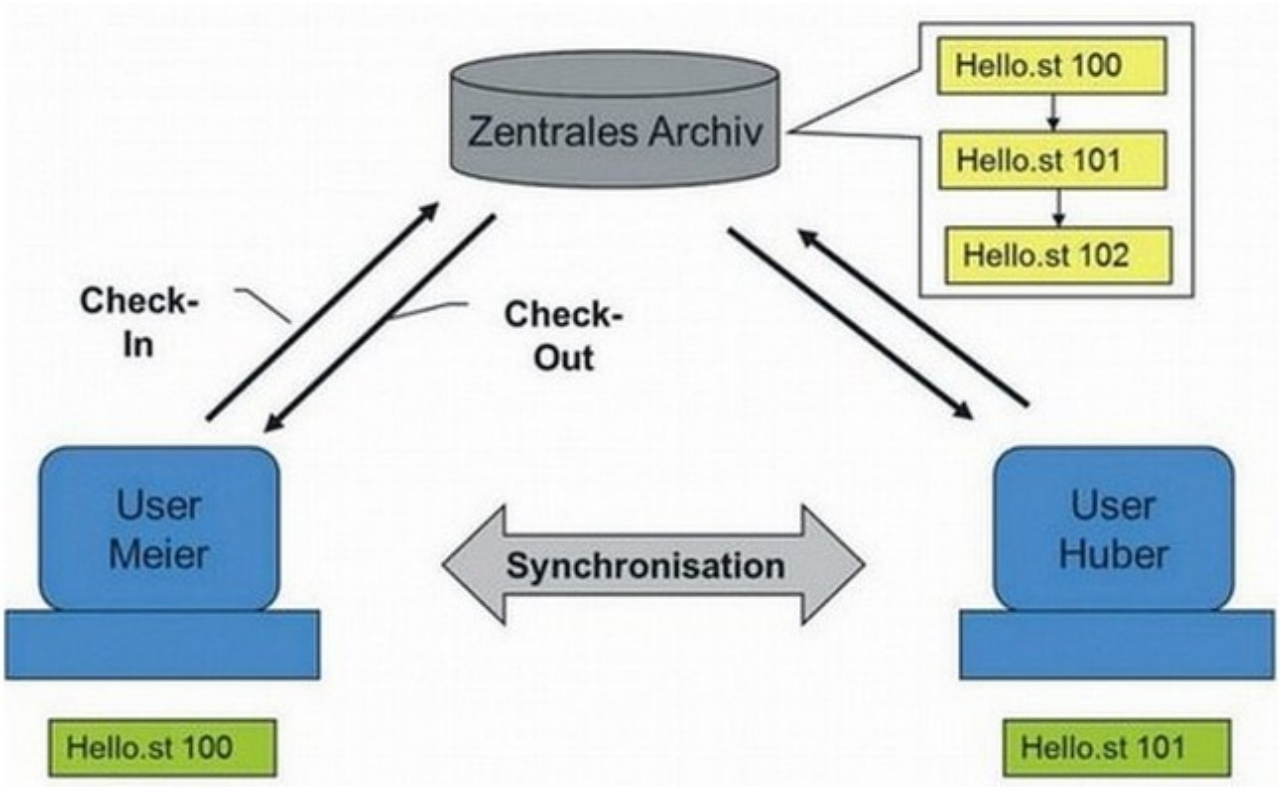

Abbildung 1: Das Grundprinzip der Versionsverwaltung

Quelle: Herkommer, Günter: Die Trends beim Versionsmanagement, 2013.

Bevor die verschiedenen Konzepte der Versionsverwaltung beleuchtet werden, werfen wir erst mal einen Blick auf das Prinzip der Versionsverwaltung. Abbildung 1 zeigt dies auf anschauliche Art und Weise. Grundsätzlich funktioniert ein Versionsverwaltungssystem so, dass es eine zentrales Repository gibt. Dorthin werden die Projektdaten abgelegt. Möchte ein Entwickler z. B. eine neue Funktion hinzufügen, werden die Daten aus dem Repository geladen („auschecken“) und lokal auf dem Computer bearbeitet. Nach der Bearbeitung werden die modifizierten Dateien zurück ins Repository geladen („einchecken“).[12]

12 Günter Herkommer: Die Trends beim Versionsmanagement.http://www.computer-automation.de/steuerungsebene/steuernregeln/fachwissen/article/79313/1/Die_Trends_beim_Versionsmanagement/, Zugriff am 24.04.2013.

2.2 Arten der Versionsverwaltung

2.2.1 Lokales Versionsmanagement

„Bei der lokalen Versionsverwaltung wird die Historie einfach lokal auf dem Entwicklungsrechner gespeichert“[13].

2.2.2 Zentrales Versionsmanagement

Bei diesem Ansatz muss lokal auf dem Rechner ein Client installiert sein, mit dessen Hilfe man zum Ein- und Aus-Checken „auf einen zentralen Server zugreift“[14]. Auf diesem Server befindet sich das zentrale Repository, das eine Master-Kopie jeder Datei enthält. Anwender haben die Möglichkeit, im Repository zu sperren, „um zu signalisieren, dass die Datei geändert wird“[15]. Wird eine modifizierte Datei mit einem Commit ins Repository eingecheckt, muss sichergestellt werden, dass „alle Unterschiede zwischen dem, was auf dem Server ist, und dem, was eingecheckt wird, aufgelöst werden“[16]. Dies gilt insbesondere dann, wenn die veränderte Datei von mehreren Entwicklern separat bearbeitet wurde.

2.2.3 Dezentrales Versionsmanagement

Beim dezentralen Versionsmanagement verhält es sich umgekehrt. Verteiltes Versionsmanagement lässt die Entwicklung von Dateien in unterschiedlichen Versionen und das anschließende Einfügen in das zentrale System (Repository) zu. Im Gegensatz zur zentralen Verwaltung „gibt es keine Trennung zwischen Entwickler-und Serverumgebung“. Zusätzlich zum Workspace hat jeder Entwickler „ein eigenes lokales Repository (genannt Klon) mit allen Versionen, Branches und Tags“[17]. Dies macht es möglich, trotz fehlendem Internetzugang weiterzuarbeiten. Wie schon beim zentralen Management werden Änderungen durch ein Commit festgeschrieben, jedoch nur im eigenen lokalen Repository. Dies hat zur Folge, dass andere Entwickler die getätigten Änderungen nicht sofort sehen. Mit Hilfe von Push- und Pull-Befehlen können „Änderungen […] von einem Repository“[18] zu einem anderen vorgenommen werden. Das Programm Git, welches Gegenstand dieser Arbeit ist,

13 Günter Herkommer: Die Trends beim Versionsmanagement, http://www.computer-automation.de/steuerungsebene/steuernregeln/fachwissen/article/79313/1/Die_Trends_beim_Versionsmanagement/, Zugriff am 24.04.2013.
14 Günter Herkommer: Die Trends beim Versionsmanagement, a. a. O.
15 Günther Herkommer: Die Trends beim Versionsmanagement, a. a. O.
16 Günter Herkommer: Die Trends beim Versionsmanagement, a. a. O.
17 Preißel, R./Stachmann, B., Git, 2012, S. 2.
18 Preißel, R./Stachmann, B., Git, 2012, S. 2.

eignet sich für verteilte Versionsverwaltung.

2.2.4 Branching und Merging

Vereinfacht ausgedrückt ist ein Branch eine Historie von Commits. Arbeiten mehrere Entwickler parallel an einem Projekt entstehen automatisch Branches. Dies ist mit einem Satz nicht so einfach zu verstehen. In Abbildung 2 ist der Sachverhalt anschaulich dargestellt.

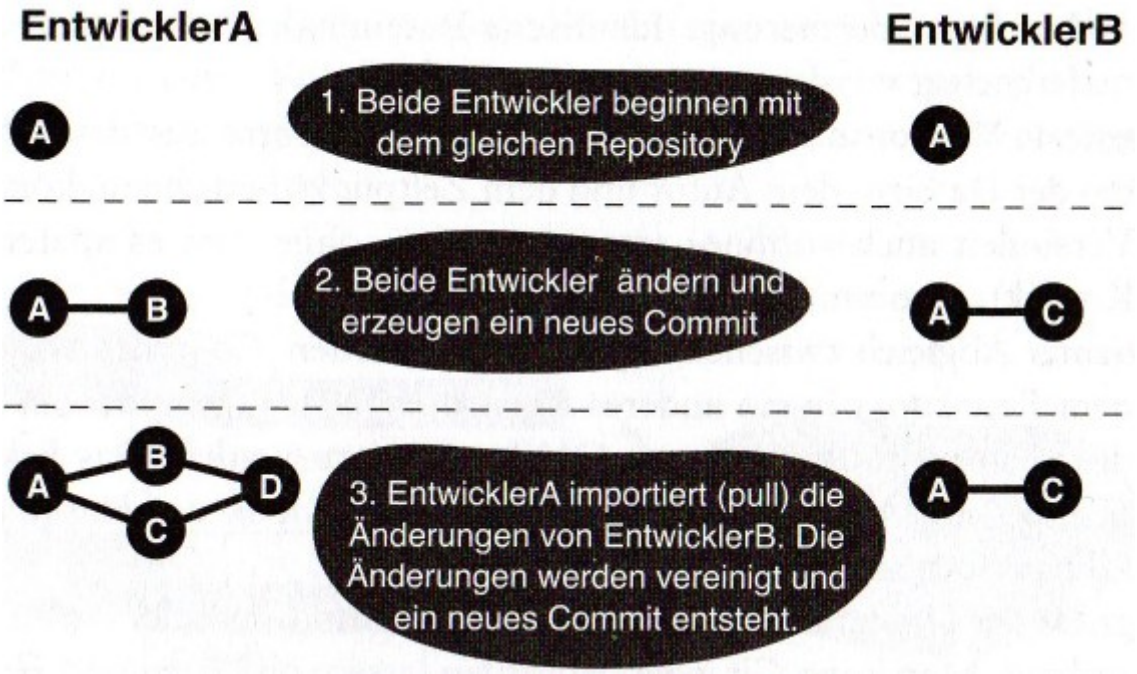

Abbildung 2: Mehrere Branches entstehen durch paralleles Arbeiten

Quelle: Preißel, René/Stachmann, Bjørn: Git, Heidelberg, 2012

Daher stellt man sich am besten zwei Entwickler vor, die ausgehend von einem gemeinsamen Basisstand beginnen weitere Funktionen und Klassen zu implementieren. Nachdem beide Änderungen vorgenommen haben, legt jeder ein neues Commit in seinem lokalen Repository an. Es existieren zwei unterschiedliche Versionen des Projekts. Durch die jeweiligen Änderungen sind zwei Branches entstanden. Möchte einer der beiden die Änderungen des Partners importieren, so bietet Git die Möglichkeit die beiden Versionen zusammenführen zu lassen. Verläuft das Zusammenführen erfolgreich, so entsteht daraus ein sogenanntes Merge-Commit. Holt der andere Partner dieses Commit ab, so sind beide auf dem selben Stand und haben die Änderungen des jeweiligen Kollegen im Blick. Es

sind zwei Branches durch die Zusammenarbeit mehrerer Entwickler entstanden. Branches können aber auch voll beabsichtigt sein. In der Praxis kommt es durchaus oft vor, dass man gezielt einen Branch vom Hauptzweig abzweigt. Dies ist dann sinnvoll, wenn man mehrere Features parallel entwickelt oder nebenbei Fehler korrigiert.[19] Dieser Sachverhalt ist in Abbildung 3 auf anschauliche Art und Weise dargestellt.

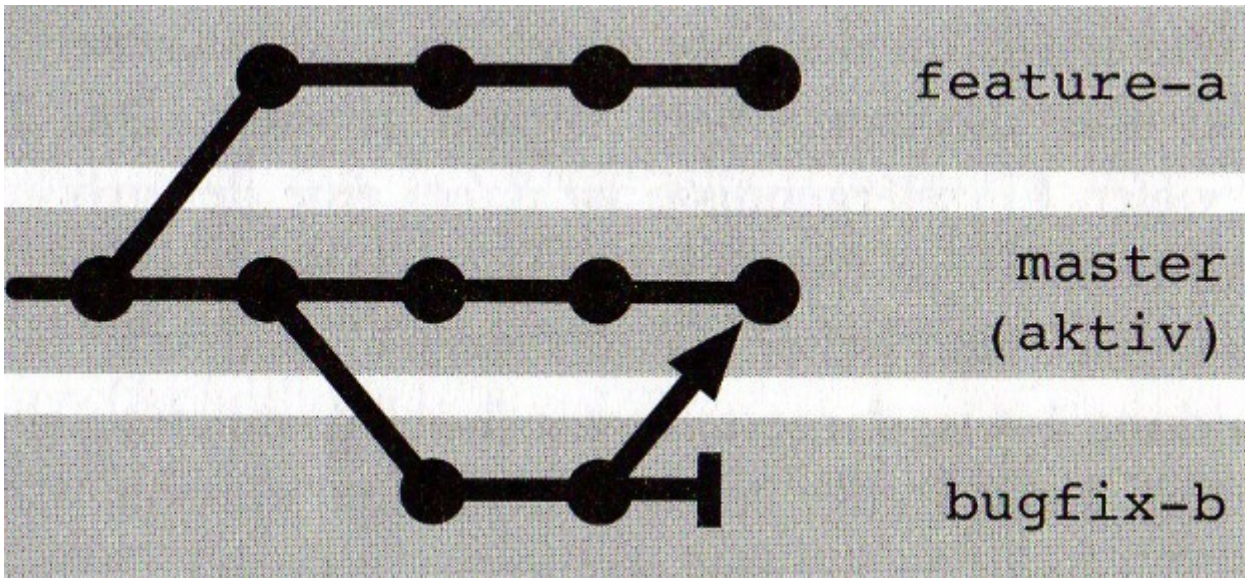

Abbildung 3: Mehrere Branches zur Erledigung verschiedener Aufgaben

Quelle: Preißel, René/Stachmann, Bjørn: Git, Heidelberg, 2012

3. Das Versionierungssystem Git

3.1 Allgemeines

Git ist eine freie Software, die zur verteilten Versionsverwaltung in Software-Projekten dient. Es wurde von Linus Torvalds, dem Initiator des Linux-Projektes im Jahr 2005 geschrieben. In den Jahren zuvor nutzten die Kernel-Entwickler von Linux noch das proprietäre Programm Bitkeeper, das aufgrund von Lizenzverschärfungen den Zugang zu den Kernelquellen für viele Kernel-Entwickler unmöglich machte. Deshalb entschied sich Linus

19 Vgl. Preißel R./Stachmann B., Git, 2012, S. 6.

Torvalds, ein neues Versionsierungssystem zu entwickeln. Torvalds gab ihm den Namen Git (engl.: Blödmann).[20] Git ist für Windows, Linux und Mac OS X verfügbar.

3.2 Warum Git?

Git ist nur ein namhafter Vertreter unter den verteilten Versionierungssystemen. Neben Git gibt es noch andere prominente Vertreter wie Mercurial, GNU Arch oder darcs.[21] Es gibt eine Vielzahl an Gründen, auf Git zu setzen, wie im Folgenden ersichtlich ist.

Performance: Wenn es um Schnelligkeit und Performance geht, sticht Git unter den Vertretern verteilter Versionsverwaltungssysteme hervor. „In weniger als einer halben Minute wechselt es zum Beispiel von der aktuellen Version auf eine sechs Jahre ältere Version der Linux-Kernel-Sourcen – auf einem kleinen MacBook Air. Das kann sich sehen lassen, wenn man bedenkt, dass über 200000 Commits und 40000 veränderte Dateien dazwischen liegen“[22].

Robust gegen Fehler und Angriffe: Ein Datenverlust ist äußerst unwahrscheinlich, da die Historie auf mehrere verteilte Repositorys verteilt wird. Eine geniale und „simple Datenstruktur im Repository“[23] ermöglicht es, dass die Daten weiterhin interpretierbar bleiben.

Offline-Entwicklung: Aufgrund der dezentralen Architektur ist es möglich, offline zu entwickeln. Dies ist dann besonders hilfreich, wenn man beispielsweise mit dem Laptop im Zug sitzt und kein Internet verfügbar ist[24].

Zusammenführen von Branches: Ein weiterer Vorteil besteht darin, dass eine Vielzahl von Entwicklern parallel auf verteilten Repositorys arbeiten kann. Dadurch entstehen „zwangsläufig mehrere Entwicklungsstränge“[25].

Open-Source-Community: Typisch für nahezu alle Software-Projekte aus der Linux-Welt ist eine aktive Community. Dies trifft auch auf Git zu. „Neben der detaillierten offiziellen Dokumentation unterstützen zahlreiche Anleitungen, Foren, Wikis den Anwender“[26].

20 Vgl. ubuntu Deutschland e. V.: Git. In: http://wiki.ubuntuusers.de/Git, Zugriff am: 20.04.2013.
21 Vgl. ubuntu Deutschland e. V.: Versionsverwaltung. In: http://wiki.ubuntuusers.de/Versionsverwaltung, Zugriff am 30.04.2013.
22 Preißel, R./Stachmann B., Git, 2012, Vorwort vii.
23 Preißel, R./Stachmann B., Git, 2012, Vorwort vii.
24 Vgl. Preißel, R./Stachmann B., Git, 2012, Vorwort vii.
25 Preißel, R./Stachmann B., Git, 2012, Vorwort vii.
26 Preißel, R./Stachmann B., Git. 2012, Vorwort vii.

3.3 Eigenschaften

Das dezentrale System Git zeichnet sich durch folgende Eigenschaften aus:

- „Einfache und effiziente Arbeitsweise nach KISS-Prinzip“[27]
- „Kein zentraler Server nötig“[28]
- „Unterstützung vieler Übertragungsprotokolle“[29]
- „Absicherung durch GnuPG-Signierung“[30]
- Arbeit am Projekt auch ohne Internetzugang möglich

Lesern, denen das KISS-Prinzip nicht geläufig sein sollte, hier die Bedeutung des Begriffs: Das Prinzip besagt laut Wikipedia, dass „eine möglichst einfache Lösung eines Problems gewählt werden sollte“[31].

3.4 Installation

Damit man in den Genuss der zahlreichen Vorteile von Git kommt, muss es erst erfolgreich installiert und eingerichtet werden. Je nach verwendetem Betriebssystem gibt es verschiedene Wege, Git zu installieren. Dazu eine kurze Installationsanleitung für die drei gängigen Betriebssysteme Windows, Linux und Mac OS X.

3.4.1 Installation mit Windows

Die Installation unter Windows ist sehr einfach gehalten. Alles was benötigt wird, ist das Installationsprogramm für Windows. Dazu besuchen Sie mit einem Webbrowser die GitHub Webseite und laden Sie sich das Installationsprogramm herunter. Nach erfolgter Installation kann man die Kommandozeile oder eine Standard-GUI nutzen.

3.4.2 Installation mit Linux

Standardmäßig ist Git bei allen Distributionen in den Paketquellen enthalten. Damit der hier zur Verfügung stehende Rahmen nicht gesprengt wird, beschränke ich mich haupt-

27 ubuntu Deutschland e. V.: Git. in http://wiki.ubuntuusers.de/Git, Zugriff am 20.04.2013.
28 ubuntu Deutschland e. V.: Git. in http://wiki.ubuntuusers.de/Git, Zugriff am 20.04.2013.
29 ubuntu Deutschland e. V.: Git. in http://wiki.ubuntuusers.de/Git, Zugriff am 20.04.2013.
30 ubuntu Deutschland e. V.: Git. In http://wiki.ubuntuusers.de/Git, Zugriff am 20.04.2013.
31 Wikipedia Foundation: KISS-Prinzip. in http://de.wikipedia.org/wiki/KISS-Prinzip, Zugriff am 21.04.2013.

sächlich auf debian-basierte Linux-Distributionen (Debian, Ubuntu Linux, Linux Mint u. a.). Grundsätzlich gibt es drei Möglichkeiten, unter Linux Programme (Pakete) zu installieren:

- über grafische Paketverwaltung
- im textbasierten Modus (auch Shell genannt)
- Quelltext herunterladen und kompilieren (nur für Fortgeschrittene)

Vor allem für weniger erfahrene Linux Nutzer bietet es sich an, Git über die grafische Paketverwaltung zu installieren. Dazu öffnen Sie das Software-Center. Wie das durchzuführen ist, hängt von der verwendeten Oberfläche (GNOME, KDE, Unity, Xfce) ab. „Die Bedienung des Software-Centers ist sehr einfach“[32]. Oben rechts befindet sich ein Suchfeld. Geben Sie dort den Suchbegriff „Git“ ein. Wählen Sie per Klick den gewünschten Eintrag aus. Neben näheren Details zum Programm im Hauptfenster erscheint auch eine Schaltfläche zum Installieren. Klicken Sie diese einmal mit der rechten Maustaste an, um den Installationsvorgang zu starten. Danach fragt Software-Center nach dem Passwort. Während Nutzer von Ubuntu Linux und Linux Mint an dieser Stelle ihr normales Passwort zur Systemanmeldung eingeben, müssen Debian-Nutzer das Root-Passwort eingeben.[33] „Ein Fortschrittsbalken […] informiert Sie anschließend, dass etwas passiert“[34].

Am schnellsten lässt sich Git über die Kommandozeile bzw. über ein Terminal installieren. Halten Sie dazu die Tasten Strg + Alt + T gleichzeitig gedrückt. Es öffnet sich ein Terminal. Nutzer von Ubuntu Linux oder Linux Mint geben die in Abbildung 4 dargestellte Befehlszeile ein: Nach der Eingabe der Befehlszeile wird man nach seinem Passwort gefragt. Dieses muss blind eingeben werden, d. h. der Nutzer sieht bei der Eingabe keine Sternchen.[35] Weiterhin ist zu beachten, dass die Installation noch durch eine Eingabe von „J“ (Abkürzung für Ja) sowie Enter-Taste bestätigt werden muss.[36] Erfahrene Benutzer, denen die in den Paketquellen verfügbare Version nicht aktuell genug ist, können sich den Quelltext herunterladen und diesen eigenhändig kompilieren. Dazu benötigt man die Bibliotheken curl, zlib, openssl, expat, und libiconv[37].

32 Fischer, M., Ubuntu GNU/Linux, 2010, S. 321.
33 Jurzik, H., Debian GNU/Linux, 2011, S. 129.
34 Jurzik, H., Debian GNU/Linux, 2011, S. 146.
35 Vgl. Fischer, M., Ubuntu GNU/Linux, 2010, S. 573.
36 Vgl. Jurzik, H., Debian GNU/Linux, 2011, S. 131.
37 Vgl. o. V.: Los geht’s – Git installieren. in: http://git-scm.com/book/de/Los-geht's-Git-installieren, Zugriff am 26.04.2013.

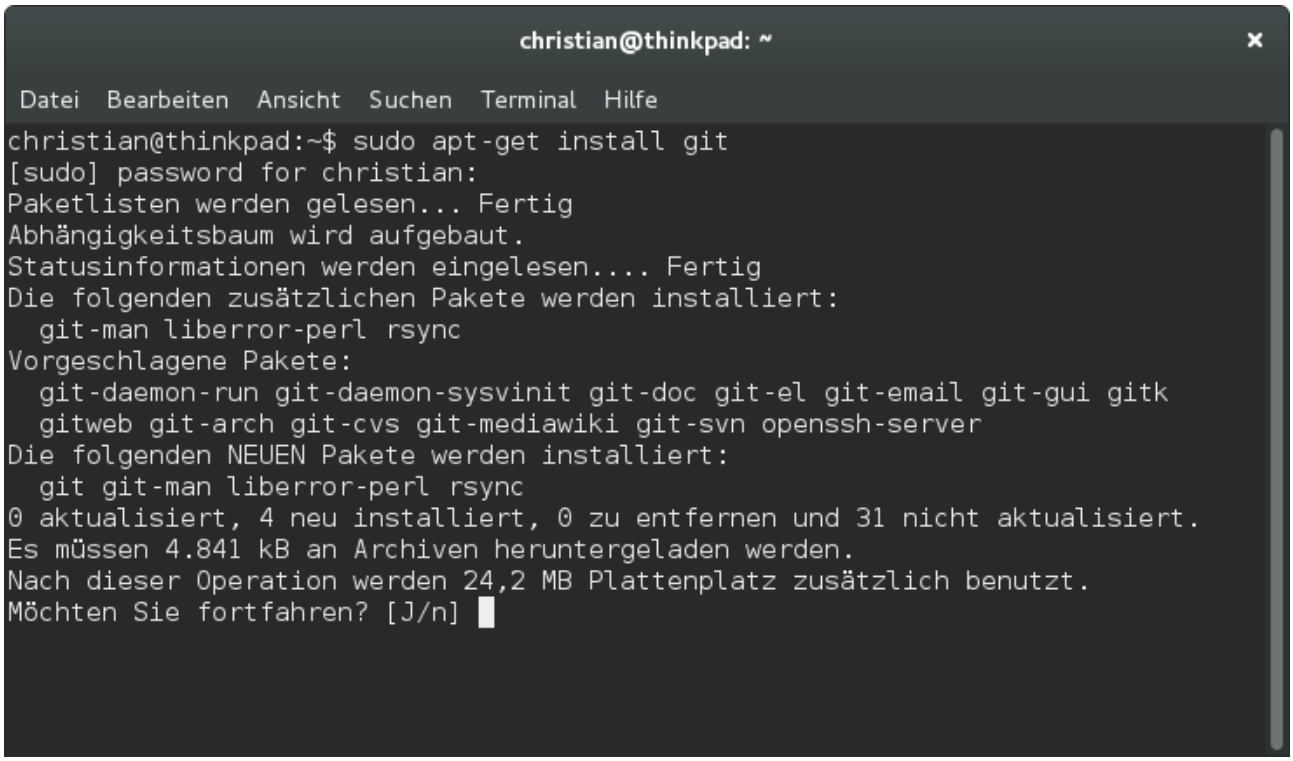

Abbildung 4: Installation in debianbasierten Linux Distributionen in einem Terminal

3.4.3 Installation Mac OS X

Besitzer eines Macs haben wie Linux-Nutzer mehrere Möglichkeiten. Wer gerne grafische Installer nutzt, ist mit dem grafischen Git Installationsprogramm bestens bedient. Das Installationsprogramm kann man unter http://code.google.com/p/git-osx-installer herunterladen. Die zweite Möglichkeit besteht darin, Git über MacPorts zu installieren.[38] Sollte MacPorts bei Ihnen installiert sein, installieren Sie mit folgender Befehlszeile Git auf ihrem System:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

3.5 Hinweise zur Benutzung

Wie viele andere populäre Programme stammt Git aus der Linux-Welt. Klassischen Windows-Nutzern wird in den nächsten Kapiteln auffallen, dass die Bedienung von Git über

38 Vgl. o. V.: Los geht's – Git installieren. in: http://git-scm.com/book/de/Los-geht's-Git-installieren, Zugriff am 30.04.2013.

Kommandos erfolgt. Insgesamt stehen mehr als 100 Befehle zur Bedienung von Git zur Verfügung, von denen man meist nur ein Dutzend benötigt. In der Linux-Welt ist es trotz der Existenz von graphischen Benutzeroberflächen nicht unüblich, mit der Kommandozeile zu arbeiten.

Wem es schwer fällt, sich mit der Bedienung über Befehle anzufreunden, dem stehen als Alternative zur Kommandozeile grafische Benutzeroberflächen zur Verfügung. Es gibt zwei grafische Oberflächen, die offizieller Bestandteil des Git-Projekts sind. Die beiden grafischen Werkzeuge werden namentlich Git-Gui und GitK genannt. Während sich Git-Gui für die schnelle Erzeugung eines neuen Versionsstandes (Commit) eignet, ist GitK für die Betrachtung der Historie eines Projekts gedacht.[39] Darüber hinaus existieren zahlreiche grafische Werkzeuge, die nicht offizieller Bestandteil des Git-Projekts sind. Diese Alternativen werden extern entwickelt und der Vollständigkeit halber erwähnt, werden aber nicht näher betrachtet.[40] Die folgende Aufzählungsliste benennt die drei grafischen Alternativen.

- Qgit
- gitg (nur für Linux verfügbar)[41]
- Giggle (nur für Linux verfügbar)[42]

4. Arbeiten mit Git

4.1 Git einrichten

Bevor man an einem Projekt zu arbeiten beginnt, sollte Git entsprechend eingerichtet sein. Zuerst sollte man Git mitteilen, wer man ist. Eine eindeutige Identifikation ist, wie in Abbildung 5 zu sehen, mit einer Kombination eines Benutzernamen und einer Mail gegeben.[43]

39 Vgl. Preißel R./Stachmann B., Git, 2012, S. 234.
40 Vgl. dazu Ubuntu Deutschland e. V.: Git. In http://wiki.ubuntuusers.de/Grafische_Oberflächen_für_Git, Zugriff am 20.04.2013.
41 Vgl. dazu Ubuntu Deutschland e. V.: Git. In http://wiki.ubuntuusers.de/Grafische_Oberflächen_für_Git, Zugriff am 20.04.2013.
42 Vgl. dazu Ubuntu Deutschland e. V.: Git. In http://wiki.ubuntuusers.de/Grafische_Oberflächen_für_Git, Zugriff am 20.04.2013.
43 Vgl. Preißel R./Stachmann B., Git, 2012, S. 9.

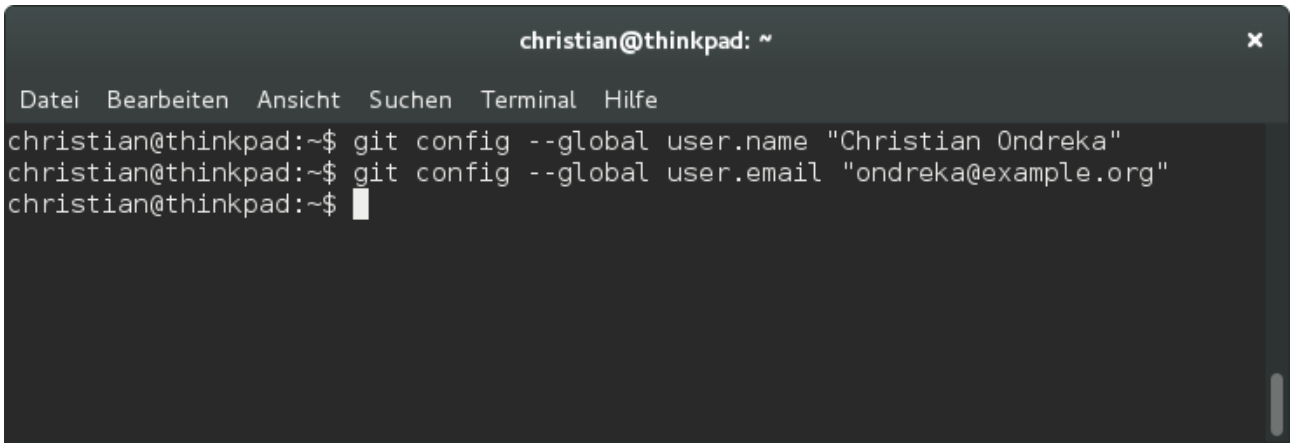

Abbildung 5: Einrichtung von Git

4.2 Erste Schritte

Sollten Sie vorhaben, die hier gezeigten Schritte am eigenen Rechner nachzuvollziehen, ist es ratsam, dies an einem Projektverzeichnis von nicht allzu hoher Priorität zu machen. Im Normalfall fragt Git vor gefährlichen Handlungen explizit nach. In meinem Beispielprojekt simuliere ich die Arbeitsschritte eines Entwicklers, der an einem kleinen Projekt zu arbeiten beginnt. Git ist schon erfolgreich installiert und eingerichtet. Im nächsten Schritt wird das Arbeitsverzeichnis im Ordner „Entwicklung“ angelegt. Ich habe das Verzeichnis teampartner01 genannt und gleich zwei Java-Dateien angelegt. Es soll die einzelnen und Arbeitsschritte darstellen. Im nächsten Schritt wird das Repository erzeugt, in dem die Historie gespeichert wird. Dies erledigt man mit dem init-Befehl, der im Arbeitsverzeichnis ausgeführt werden. Die Anwendung des init-Befehls ist in Abbildung 6 veranschaulicht.

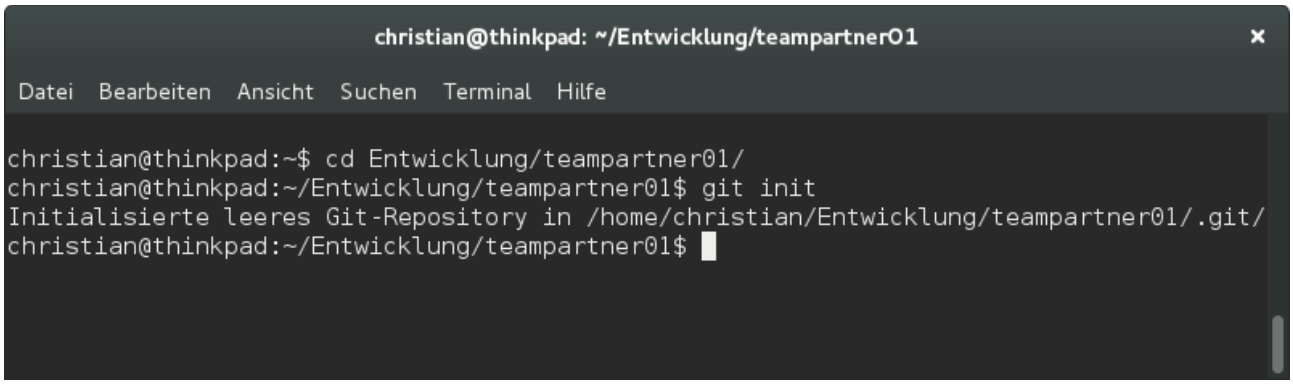

Abbildung 6: Erzeugung des Git-Repositorys in einem Terminal

In Abbildung 7 ist zu sehen, wie das erstellte Verzeichnis mit einem Dateimanager betrachtet aussieht. Die beiden Dateien Main.java und MyClass.java sind erwartungsgemäß vorhanden. Im versteckten Verzeichnis .git werden die Commits in Zukunft abgelegt.

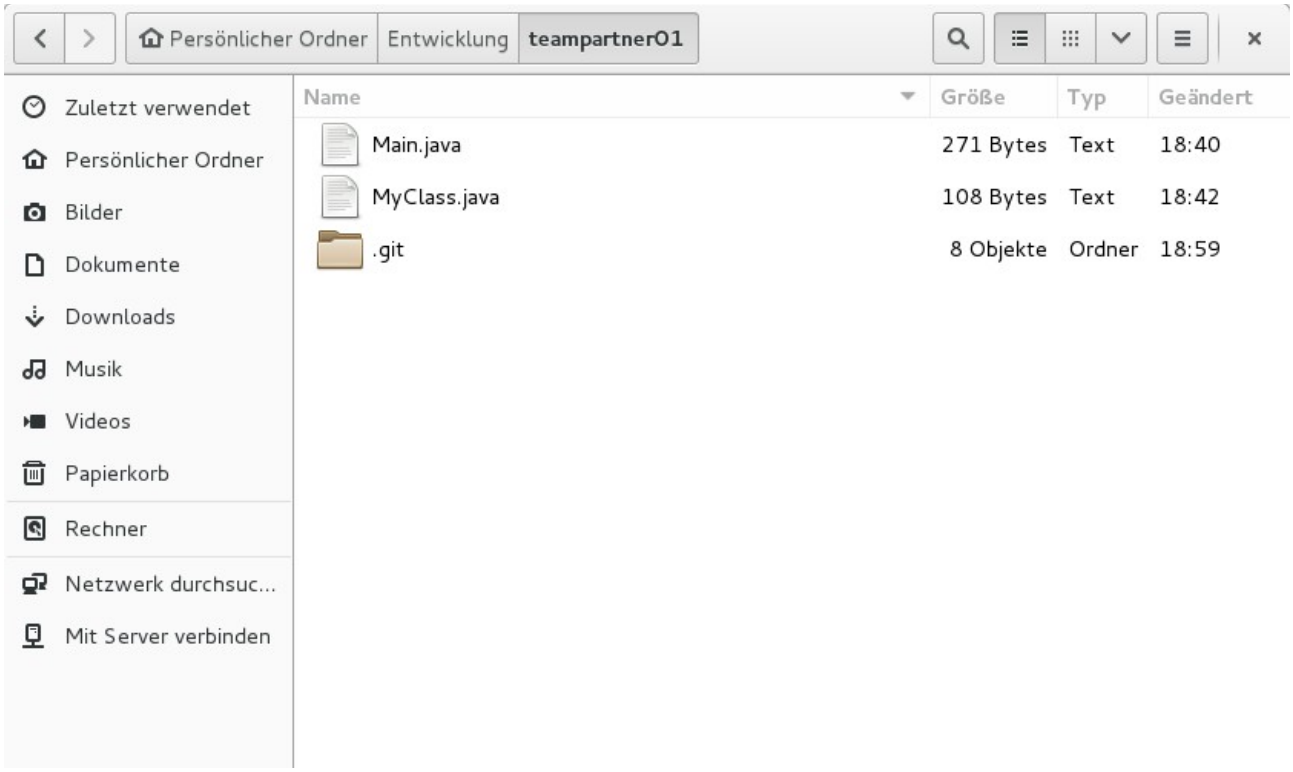

Abbildung 7: Blick mit einem Dateimanager auf Arbeitsverzeichnis

Als nächstes sollen die Dateien Main.java und MyClass.java ins Repository gebracht werden. Dies erfolgt in zwei Schritten: „Als erstes bestimmt man mit dem add-Befehl, welche Dateien in das nächste Commit aufgenommen werden sollen. Danach überträgt der Commit-Befehl“[44] die Dateien ins Repository. Abbildung 8 zeigt die Durchführung.

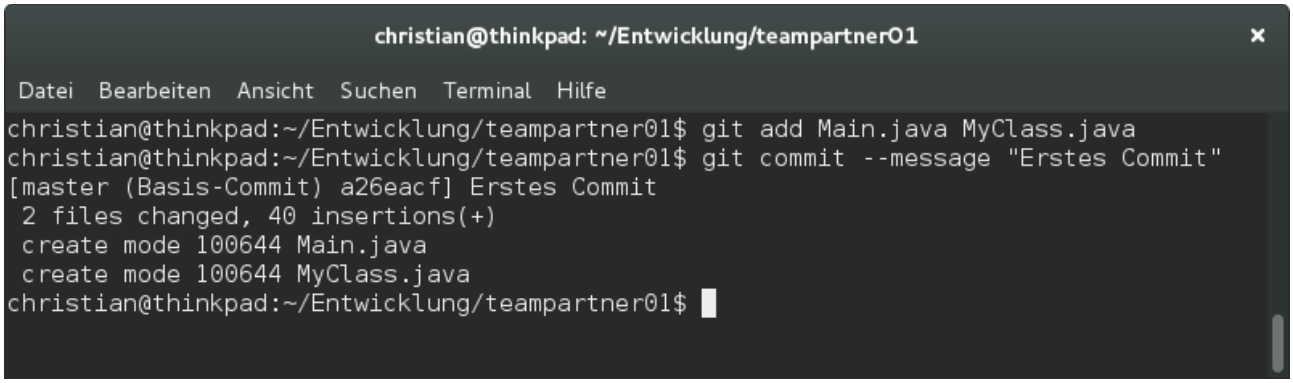

Abbildung 8: Erstes Commit erzeugt ersten Versionsstand

44 Preißel R./Stachmann B., Git, 2012, S. 10.

In Abbildung 9 ist das Arbeitsverzeichnis in einem Dateimanager betrachtet dargestellt. Auf den ersten Blick hat sich nichts gravierendes geändert. Bei genauerem Hinsehen fällt auf, dass der versteckte Ordner .git 11 Objekte enthält. Diese 11 Objekte sind im Rahmen des ersten Commits entstanden.

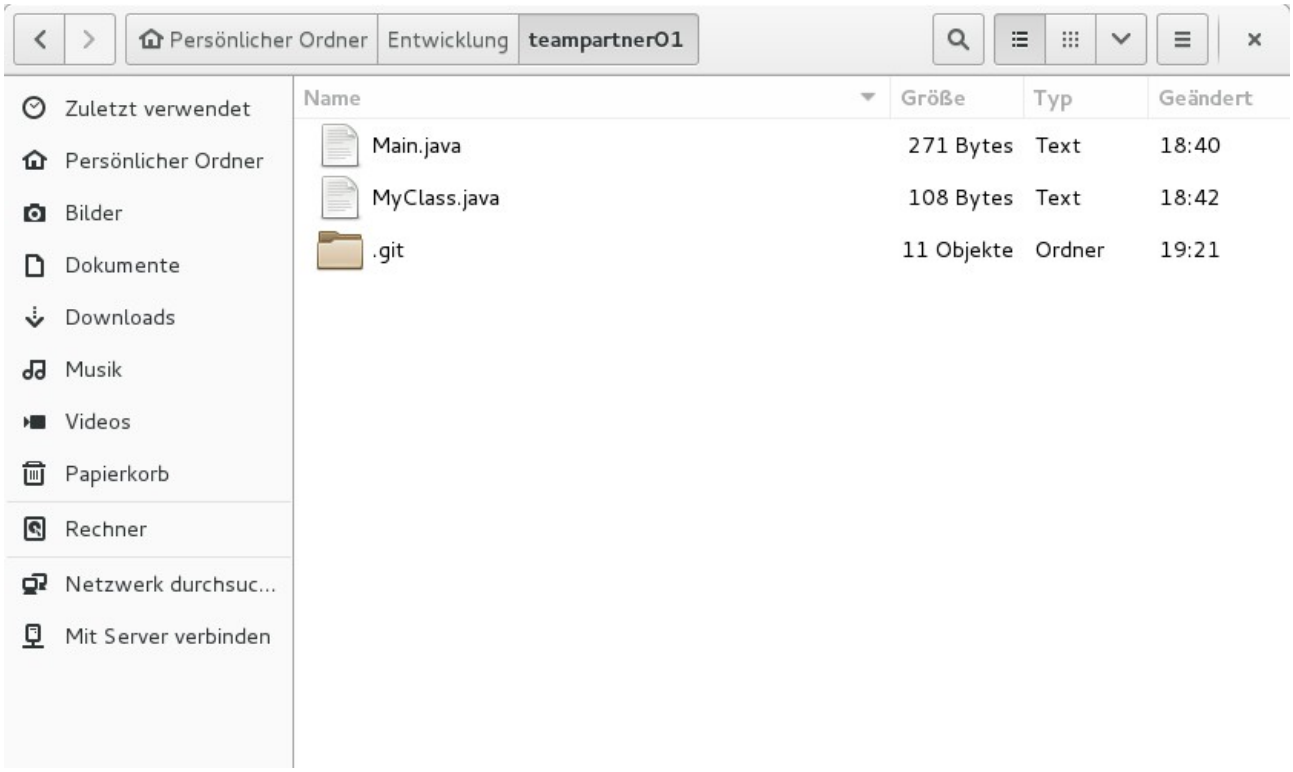

Abbildung 9: Das Arbeitsverzeichnis nach dem ersten Commit

In vielen Softwareprojekten finden Änderungen in Form von Modifikationen, Erweiterungen und Löschungen statt. Aus diesem Grund wird die Datei MyClass.java modifiziert und eine neue Datei Test.java hinzugefügt. In Abbildung 10 ist der aktuelle Zustand des Arbeitsverzeichnisses zu entnehmen. Wie bereits erwähnt, befindet sich die neue Datei Test.java im Arbeitsverzeichnis. Die Modifikation der Datei MyClass.java ist auf den ersten Blick nicht zu entdecken. Bei genauerer Betrachtung fällt jedoch auf, dass sich im Vergleich zu Abbildung 9 die Dateigröße von 108 Bytes zu 300 Bytes vergrößert hat. Darüber hinaus bestätigt dies die in Abbildung 11 dargestellte Ausgabe des status-Befehls.

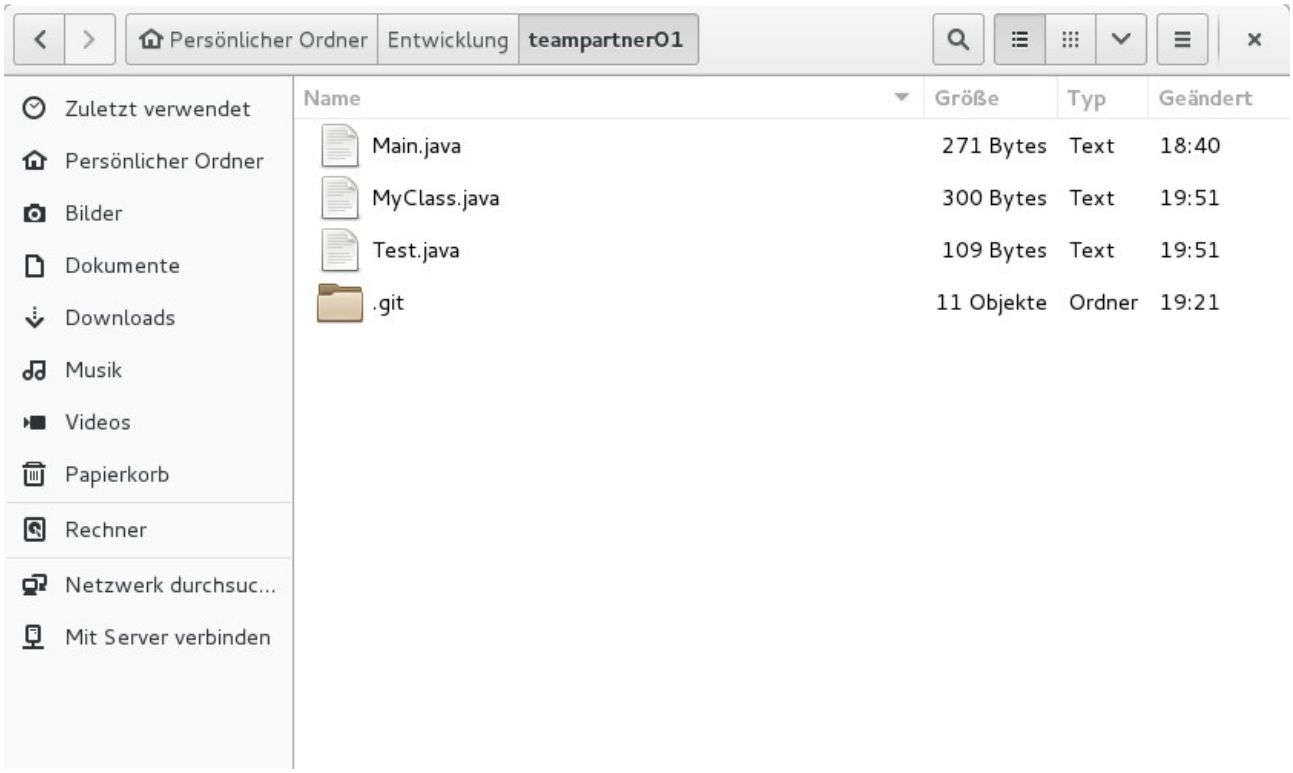

Abbildung 10: Blick mit einem Dateimanager auf das Arbeitsverzeichnis

Mit dem Status-Befehl lassen sich alle Änderungen seit dem letzten Commit anzeigen.

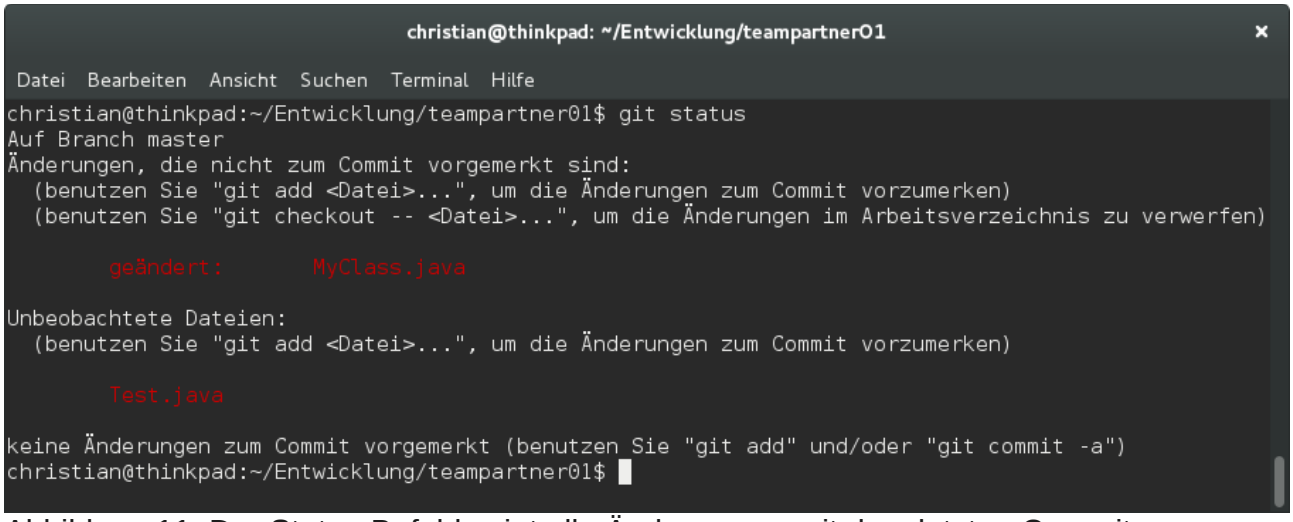

Abbildung 11: Der Status-Befehl zeigt alle Änderungen seit dem letzten Commit an

Wie Abbildung 11 zu entnehmen ist, wird die neue Datei in der Kategorie „Untracked files“ gelistet, weil diese im vorherigen Commit nicht enthalten war und noch nicht mit dem add-

Befehl für das nächste Commit angemeldet wurde.[45]

Im nächsten Schritt führt der Entwickler nach den ersten Änderungen ein Commit aus und erzeugt damit einen neuen Versionsstand im Repository. Wie schon beim ersten Commit müssen die Änderungen mit dem add-Befehl angemeldet werden. Die Option –all bewirkt, dass alle Änderungen beim nächsten Commit übernommen werden.[46]

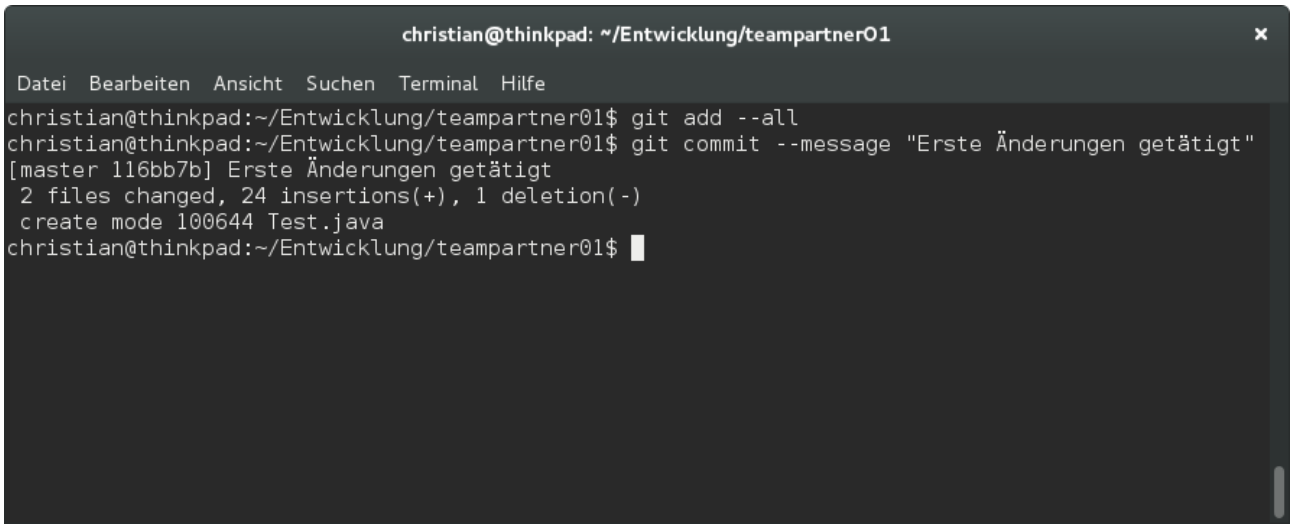

Abbildung 12: Durchführung des zweiten Commits

Gelegentlich kommt es vor, dass Dateien überflüssig werden oder später durch andere ersetzt werden. Im Rahmen dieses Beispiels soll die Datei Test.java gelöscht werden. Damit diese Änderung für das nächste Commit angemeldet werden kann, müssen nicht mehr benötigte Dateien im Arbeitsverzeichnis gelöscht und neue Dateien im Arbeitsverzeichnis vorhanden sein. Nachdem die Datei Test.java gelöscht wurde, wird die Datei Test.java mit dem rm-Befehl für das nächste Commit als gelöscht markiert.[47] Dieser Schritt ist in Abbildung 14 nachvollziehbar dargestellt. Zusätzlich wird dem Arbeitsverzeichnis eine neue Datei namens NewClass.java hinzugefügt, die mit dem bereits bekannten add-Befehl angemeldet wird. Bei Anwendung des add-Befehls werden alle Änderungen in einem Zwischenspeicher (Stage-Bereich) vermerkt. Der commit-Befehl transferiert anschließend die gesammelten Änderungen aus dem Zwischenspeicher in das Repository.[48] Abbildung 12 zeigt den aktuellen Zustand des Arbeitsverzeichnisses.

45 Vgl. Preißel R./Stachmann B., Git, 2012, S. 11.
46 Preißel R./Stachmann B., Git, 2012, S. 20.
47 Vgl. Preißel R./Stachmann B., Git, 2012, S. 12.
48 Vgl. Preißel R./Stachmann B., Git, 2012, S. 27.

Im Vergleich zu vorher (siehe Abbildung 10) ist die Datei Test.java nicht mehr existent.

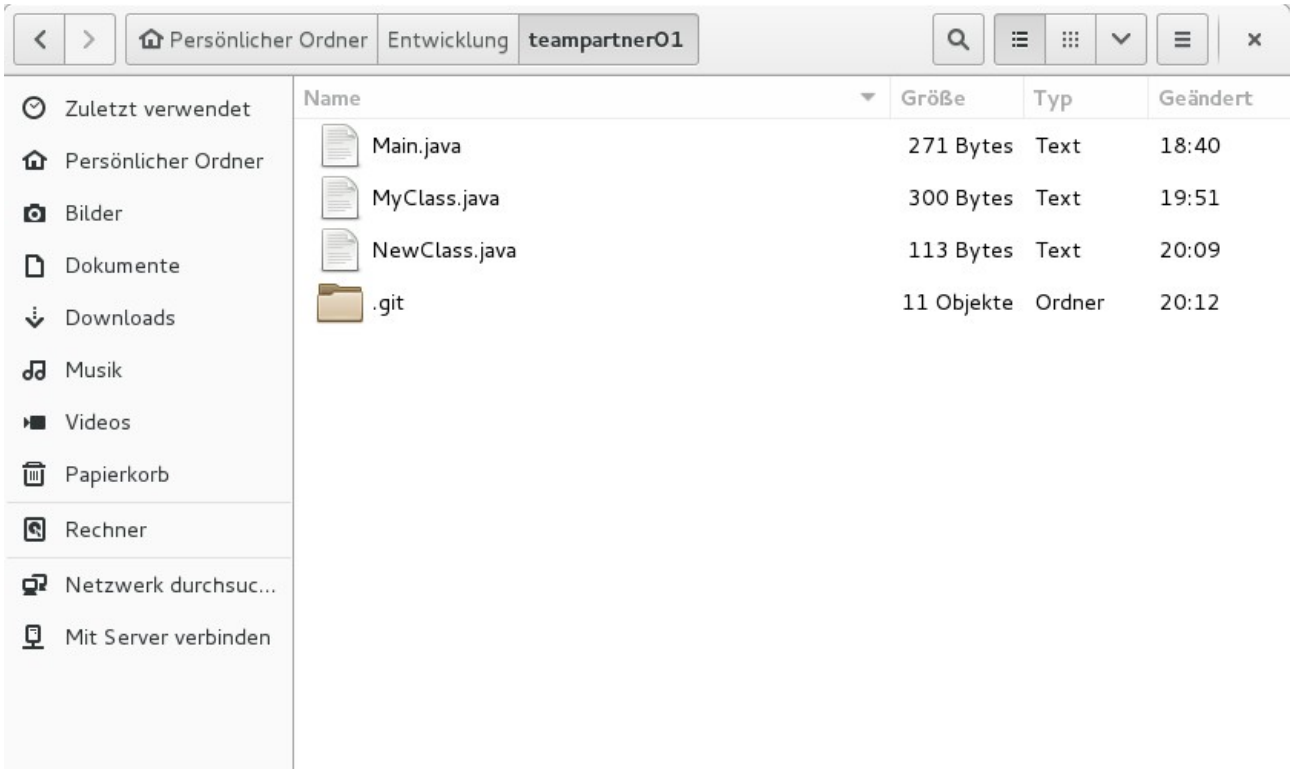

Abbildung 13: Das Arbeitsverzeichnis nach dem Löschen und Hinzufügen von Dateien

Stachmann und Preißel weisen explizit darauf hin, dass „für jedes neue Commit Änderungen angemeldet werden [müssen]. Für geänderte und neue Dateien erledigt dies der add-Befehl. Gelöschte Dateien müssen mit dem rm-Befehl als gelöscht markiert werden“.[49]

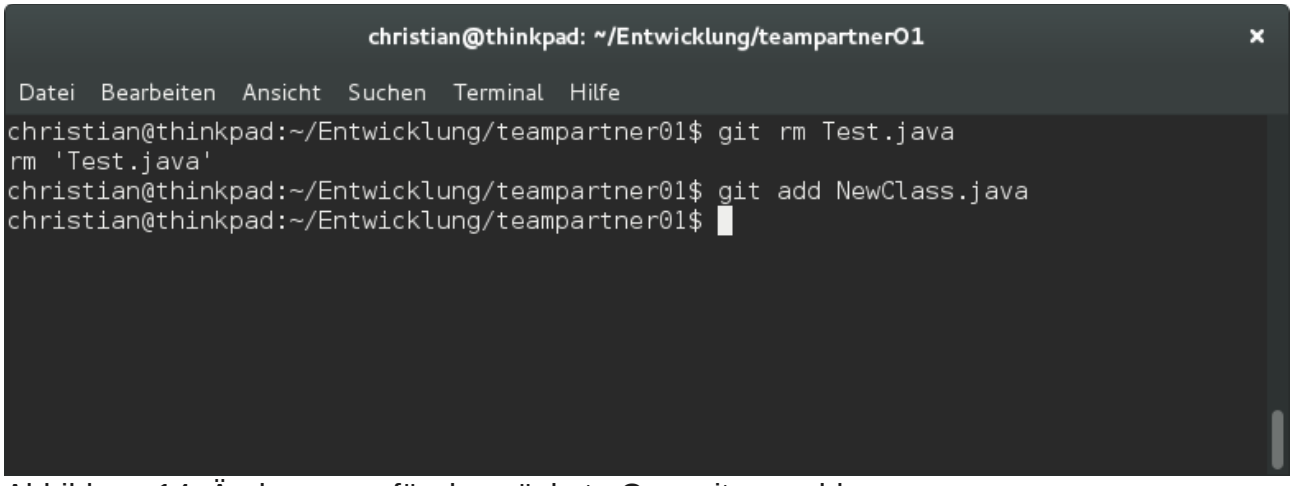

Abbildung 14: Änderungen für das nächste Commit anmelden

49 Preißel R./Stachmann B., Git. 2012, S. 12.

Ein weiterer Aufruf des status-Befehls gibt Aufschluss darüber, welche Änderungen in den nächsten Commit einfließen. In Abbildung 15 ist die Ausgabe des Status-Befehl zu sehen.

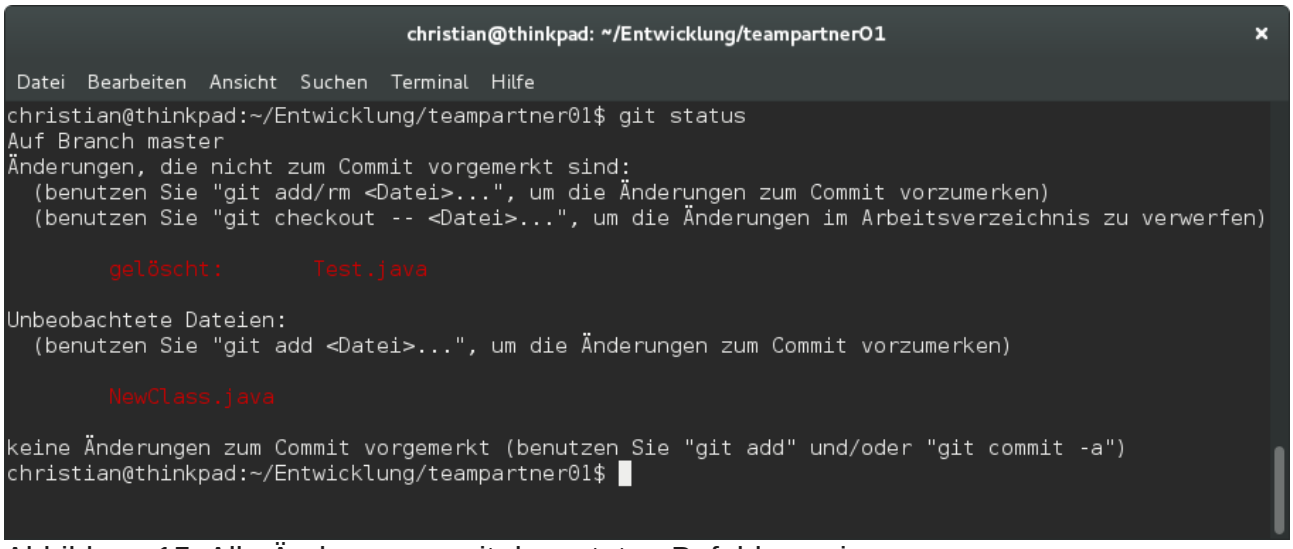

Abbildung 15: Alle Änderungen mit dem status-Befehl anzeigen

Anschließend wird das Commit durchgeführt. Die Anwendung des commit-Befehls und der damit einhergehenden Erzeugung einer neuen Version ist in Abbildung 16 zu sehen.

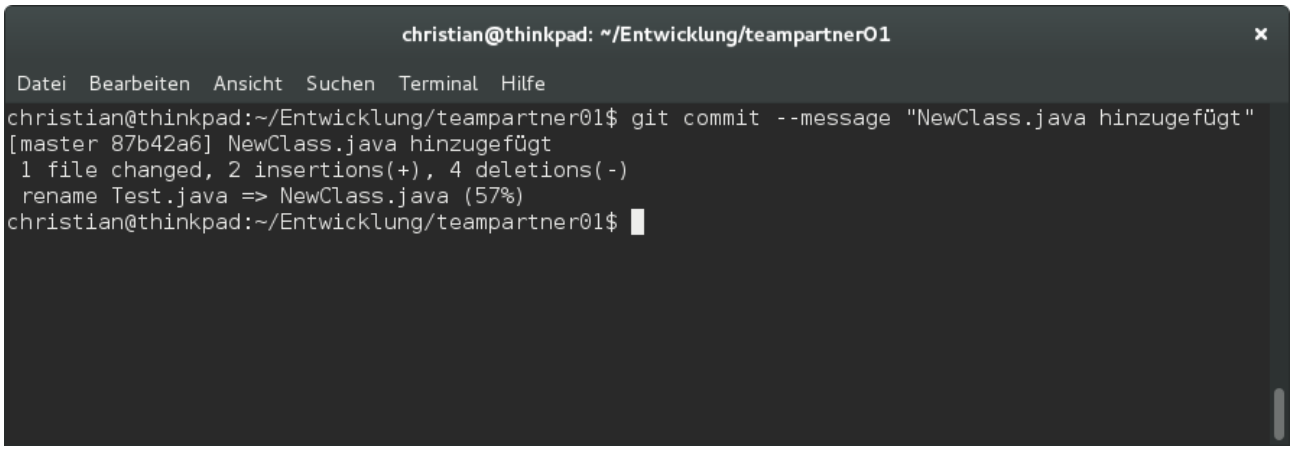

Abbildung 16: Durchführung des dritten Commits

Zusätzlich hat man die Möglichkeit, sich die Historie des Projekts ausgeben zu lassen. Möglich macht das der log-Befehl. Wie in Abbildung 17 zu entnehmen ist, werden die Commits chronologisch absteigend sortiert.[50]

50 Vgl. Preißel R./Stachmann B., Git, 2012, S. 12.

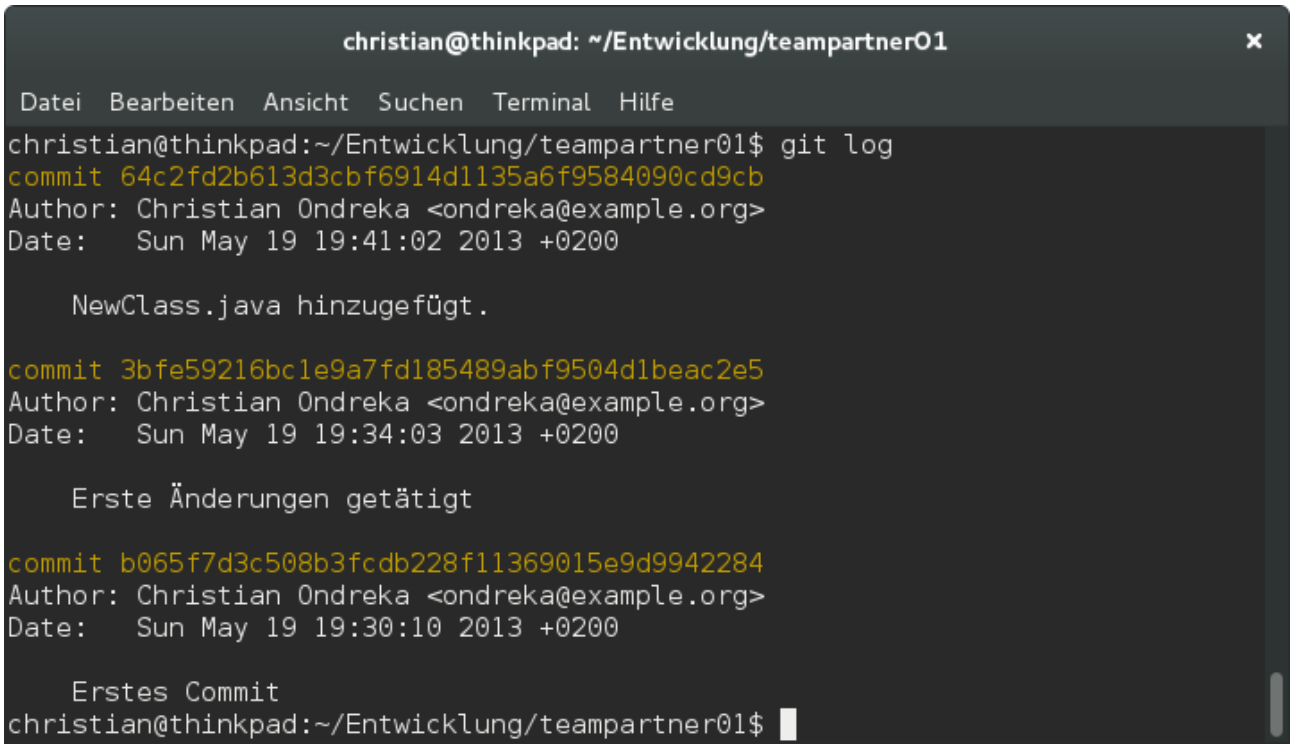

Abbildung 17: Chronologische Auflistung aller Commits

4.3 Zusammenarbeit mit Git

In der Praxis kommt es äußerst selten vor, dass ein Entwickler alleine an einem Projekt arbeitet. Aufgrund der dezentralen Architektur von Git können mehrere Entwickler ihre Commits zwischen ihren Repositorys austauschen. Um dies im kleinen Maße zu veranschaulichen, lege ich ein zweites Arbeitsverzeichnis an, das die Aktivitäten eines zweiten Entwicklers simulieren soll. Der zusätzliche Entwickler benötigt die bisherigen Projektdaten. Wie lässt sich dies nun bewerkstelligen? Ein simpler Ansatz wäre, die erstellten Java-Dateien aus dem Arbeitsverzeichnis des ersten Teampartners in das des neuen Entwicklers zu kopieren. Dann müsste der zweite Entwickler das Repository mit dem init-Befehl neu erzeugen. Der bisherige Stand des ersten Entwicklers wäre der Anfangsstand des zweiten Entwicklers. Die Dateien beider Entwickler sind komplett identisch, jedoch erzählen die beiden Repositorys zwei verschiedene Geschichten. Im Idealfall arbeitet der neu hinzugekommene Entwickler mit demselben Stand des ersten Entwicklers weiter und übernimmt noch den Stand des Repositorys. Das Arbeitsverzeichnis des ersten Entwicklers wird also samt

Repository kopiert. In Abbildung 18 ist die Ausführung des clone-Befehls zu sehen.

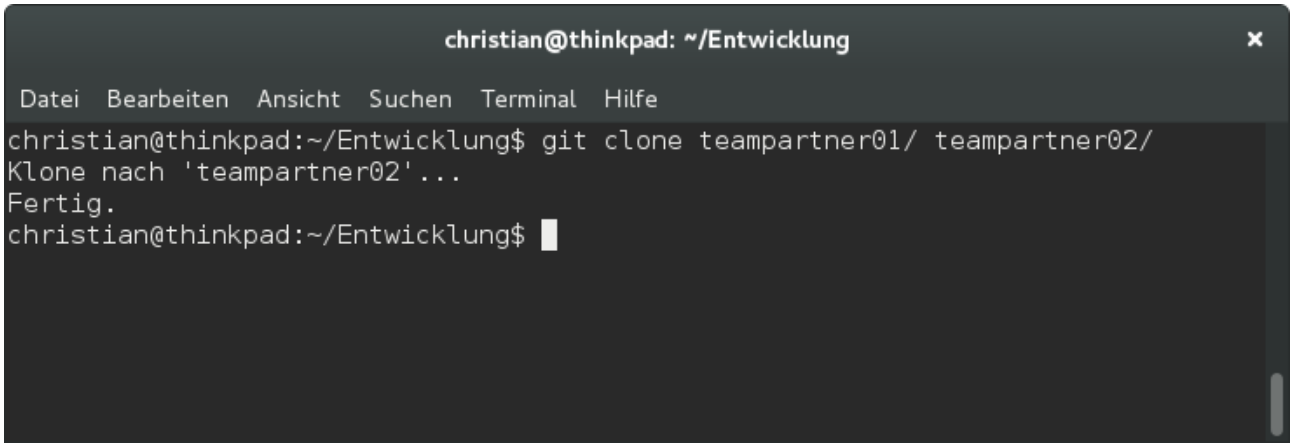

Abbildung 18: Die Historie des Original Repositorys mit dem clone-Befehl kopieren

Wie In Abbildung 19 zu erkennen, sieht das (geklonte) Arbeitsverzeichnis des zweiten Entwicklers im Endergebnis genau so aus wie das Arbeitsverzeichnis des ersten Entwicklers.

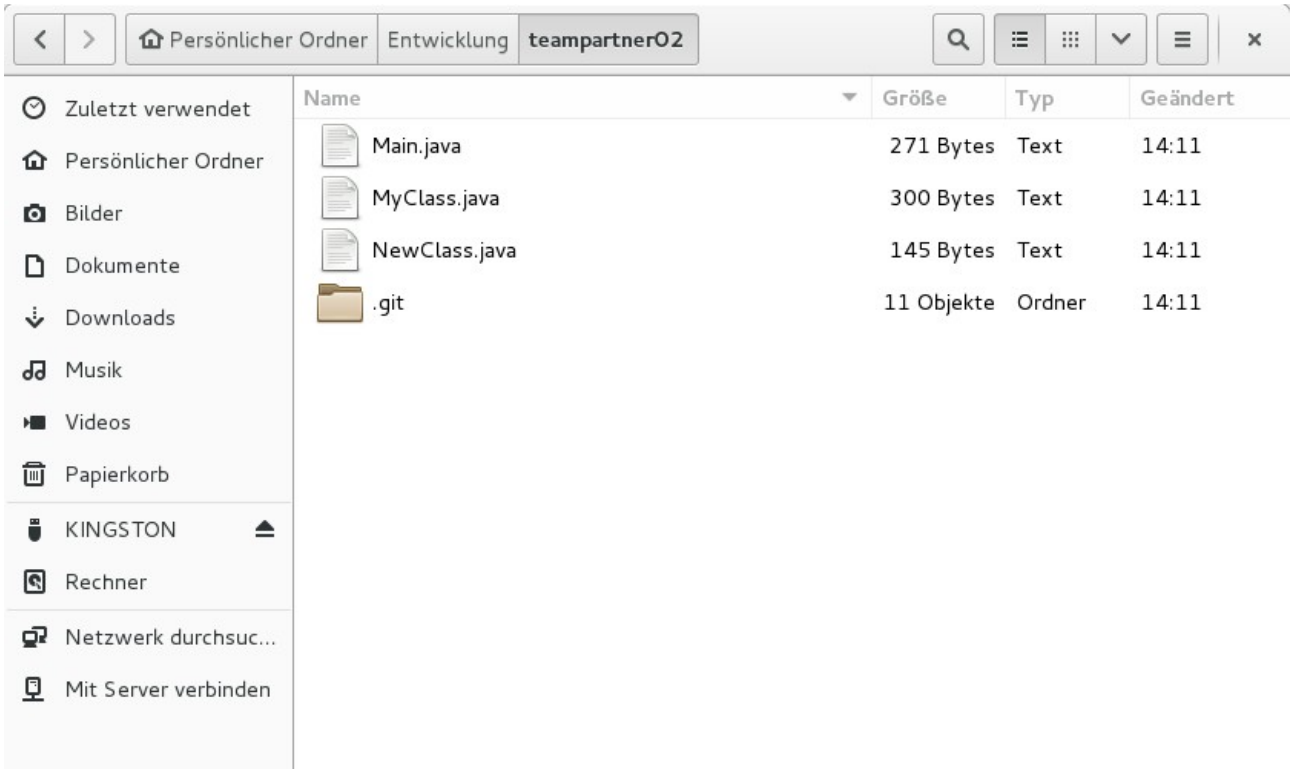

Abbildung 19: Geklontes Repository in einem Dateimanager

Auch das Repository im versteckten Ordner .git ist vorhanden. Es gibt noch einen weiteren

Grund, weshalb diese Vorgehensweise vorzuziehen ist: Das geklonte Repository hat eine Verknüpfung zu seinem Ursprungsverzeichnis. Dies wird sich später noch als nützlich erweisen.

Als nächstes wird die Datei MyClass.java im Original-Repository geändert. Die Änderungen werden wie gewohnt angemeldet und das Commit wird durchgeführt. Das gleiche wird mit der Datei NewClass.java im geklonten Repository gemacht. In Abbildung 20 wird der Vorgang mit den dafür erforderlichen Befehlen durchgeführt.

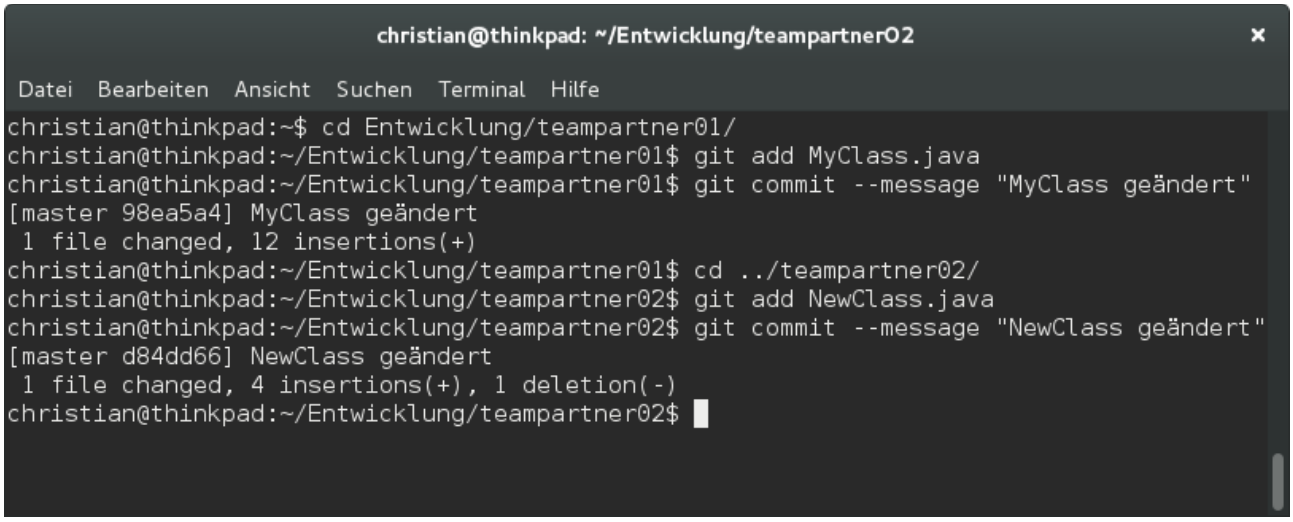

Abbildung 20: In beiden Repositorys entstehen neue Commits

Im Endergebnis befinden sich in den beiden Repositorys zwei gemeinsame Commits und ein jeweils neues Commit. Dies wirft automatisch die Frage auf, wie man die Änderungen vom einen ins andere Repository bekommt. Dafür steht der pull-Befehl zur Vefügung. Nehmen wir an, der zweite Entwickler möchte die getätigten Änderungen von seinem Partner importieren. Dabei kommt es einem zugute, dass beim Klonen der Pfad zum Original-Repository im Klon hinterlegt worden ist. „Der pull-Befehl weiß also, wo er neue Commits abholen soll“[51]. Konkret heißt das, dass keine zusätzlichen Angaben in Form von Optionen dem pull-Befehl mitgegeben werden müssen, um die Aktion auszuführen zu können.[52] Für ein besseres Verständnis sowie mehr Übersichtlichkeit sind in den Abbildungen 21 und 23 die Ausgaben des log-Befehls vor und nach dem Merge-Commit zu sehen. Der log-Befehl gibt in diesem Fall die Historie des lokalen Repositorys des zweiten Entwicklers aus. Fer-

51 Preißel R./Stachmann B., Git, 2012, S. 13.
52 Vgl. Preißel R./Stachmann B., Git, 2012, S. 14.

ner ist in Abbildung 21 die Ausführung des Merge-Commits mit dem Pull-Befehl zu sehen.

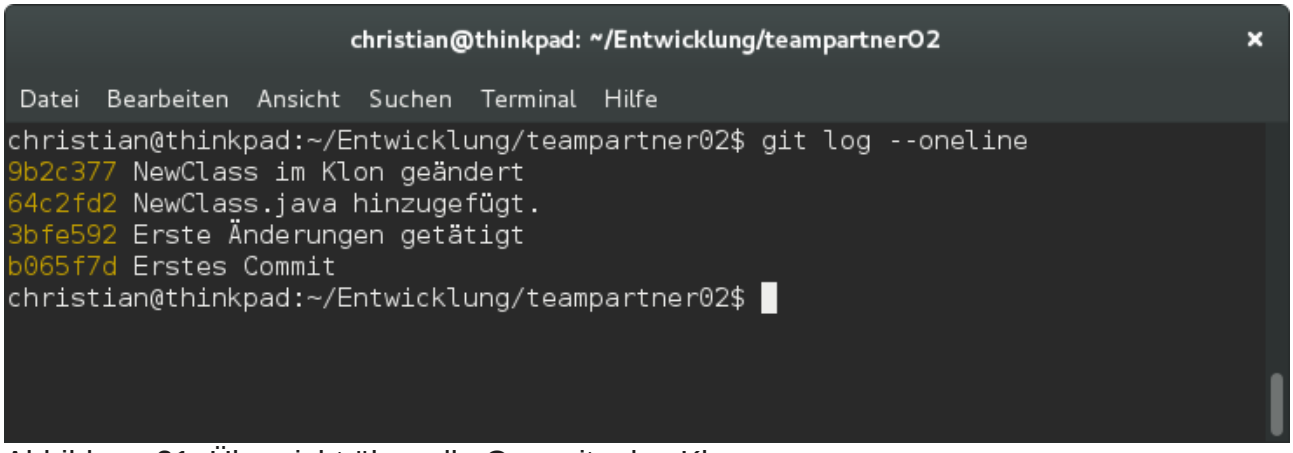

Abbildung 21: Übersicht über alle Commits des Klon

Der log-Befehl gibt in diesem Fall chronologisch die Historie des lokalen Repositorys von Entwickler zwei in verkürztem Ausgabeformat aus. Beim Vergleich der beiden Abbildungen 21 und 23 kommen die Änderungen vor und nach dem Merge-Commit zum Vorschein.

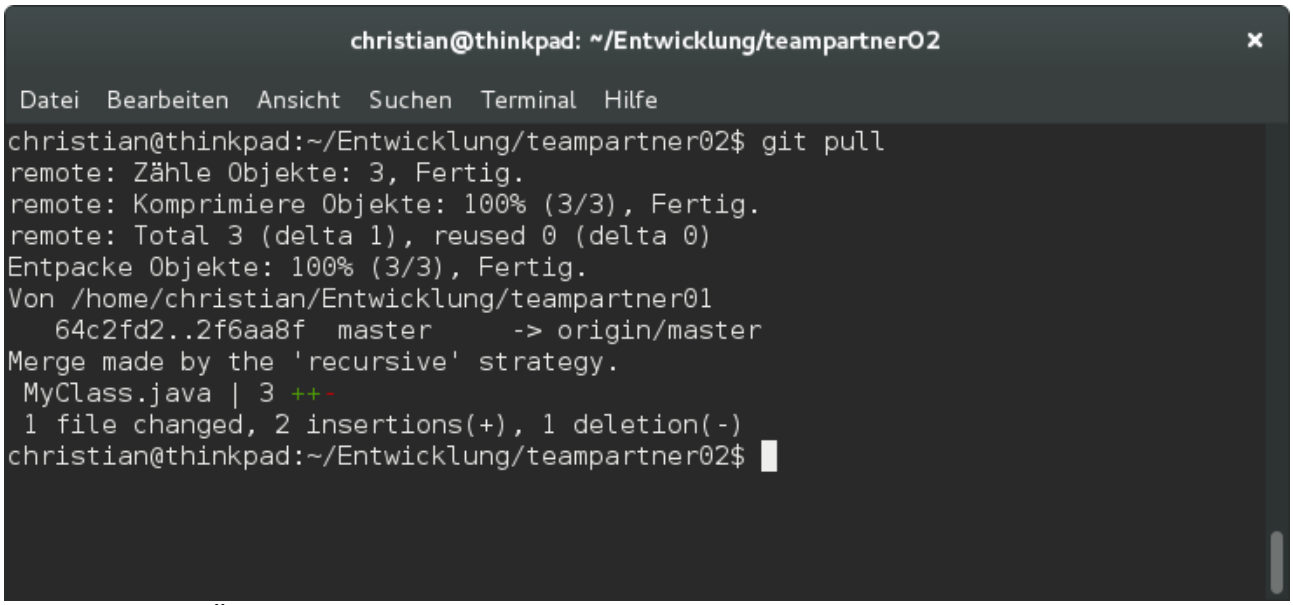

Abbildung 22: Änderungen mit dem pull-Befehl aus dem Original-Repository abholen

Die Änderungen aus dem Original-Repository wurden abgeholt und Datei für Datei abgeglichen. Danach werden die Änderungen zusammengeführt. Man nennt dies einen Merge. Das Abholen und Zusammenführen von Änderungen aus einem Original-Repository ge-

staltet sich aufgrund der Verknüpfung zum Original sehr einfach. Allerdings kommt es des Öfteren vor, dass Änderungen aus einem beliebigen Repository abgeholt werden sollen.

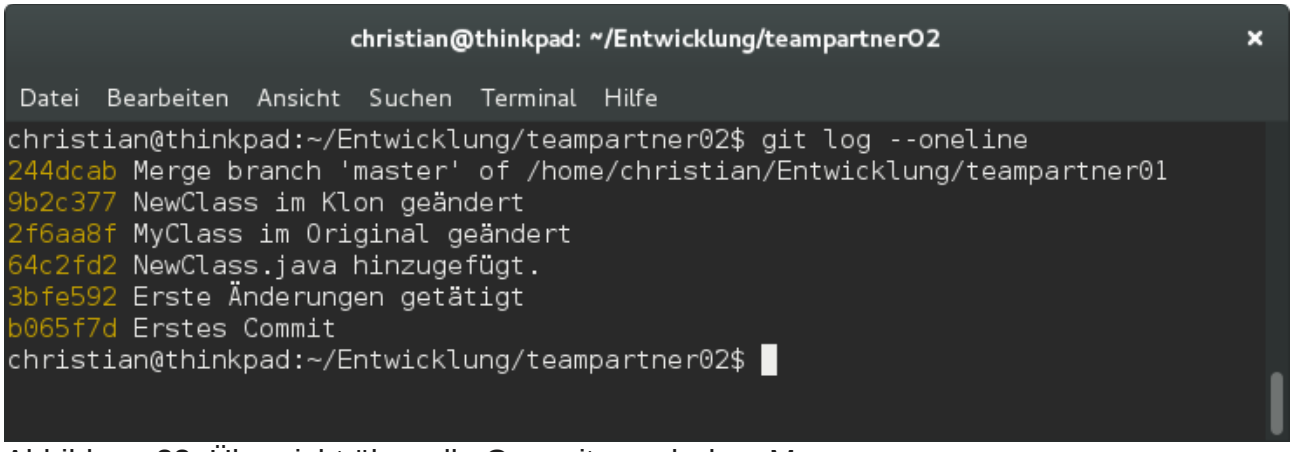

Abbildung 23: Übersicht über alle Commits nach dem Merge

Auch dafür ist der pull-Befehl vorgesehen. Um Änderungen aus einem beliebigen Repository abzuholen, gibt man ihm einfach den Branch (Entwicklungszweig) als Parameter mit.[53]

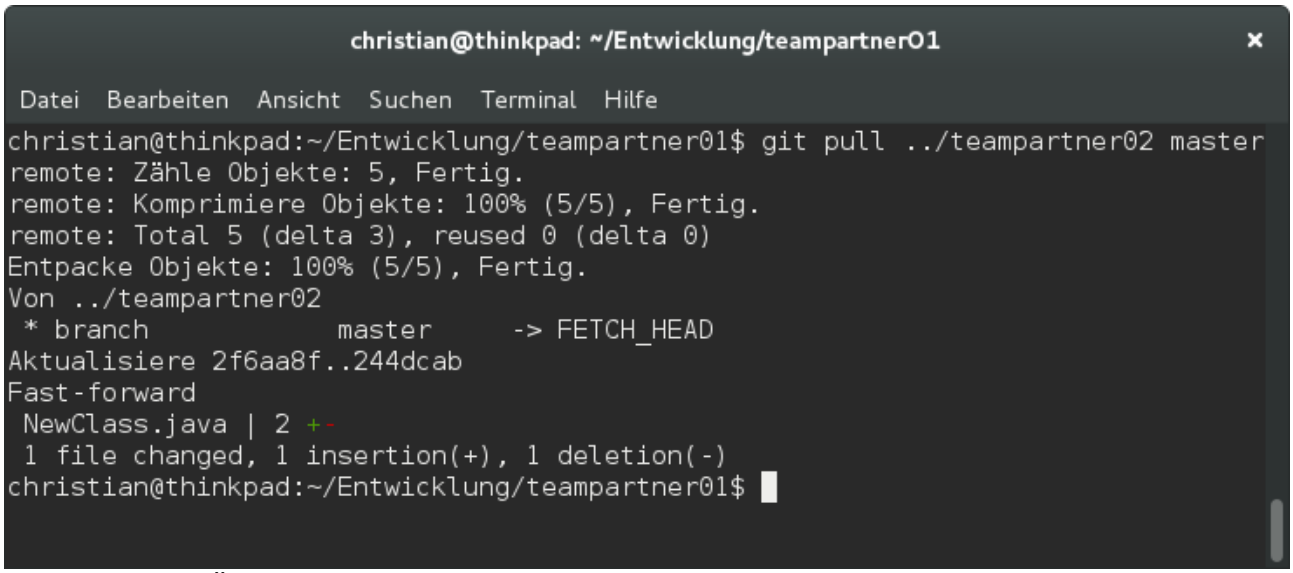

Abbildung 24: Änderungen mit dem pull-Befehl abholen

Nach der Ausführung des pull-Befehls ist das Ursprungs-Repository auf dem Stand von seinem Klon. Wie man wahrscheinlich erahnen kann, gibt es neben dem pull-Befehl, der für das Importieren zuständig ist, einen weiteren Befehl zum Exportieren. Zum Exportieren

53 Vgl. Preißel R./Stachmann B., Git, 2012, S. 15.

benötigt man den Befehl push. Wie Abbildung 24 zu entnehmen ist, lege ich zur Demonstration dieses Befehls ein sog. Austausch-Repository an. In diesem Zusammenhang weisen Preißel und Stachmann explizit darauf hin, dass der push-Befehl „nur auf Repositorys angewendet werden [sollte], auf denen gerade kein Entwickler arbeitet“.[54] Es empfiehlt sich, das Austausch-Repository ohne einen workspace drumherum zu erzeugen. Im Fachjargon spricht man auch von einem Bare-Repository. In der Realität werden solche Repositorys als zentrale Anlaufstelle verwendet. Jeder Entwickler innerhalb eines Teams holt sich die Änderungen der anderen Kollegen ab und exportiert seine vorgenommenen Modifikationen dorthin. Ein solches Repositoy wird durch Nutzung der Option --bare erzeugt[55].

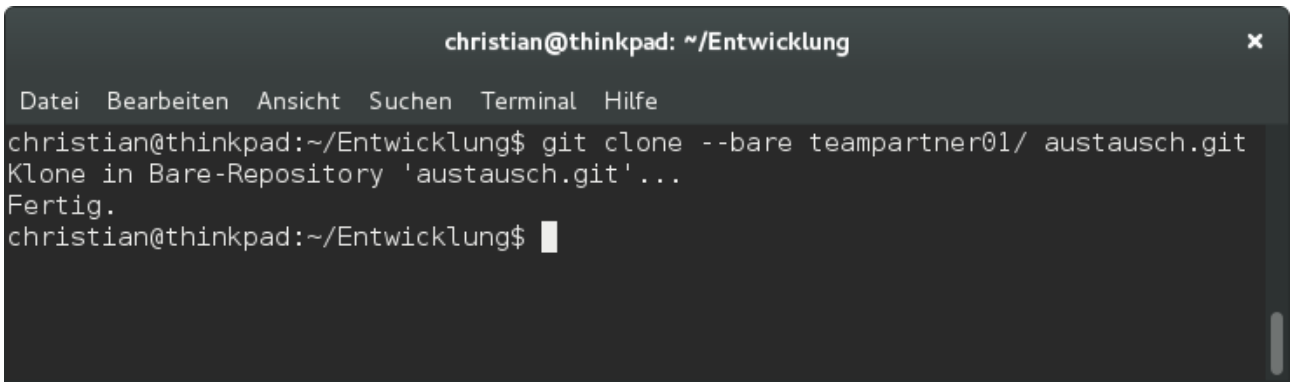

Abbildung 25: Repository für den Austausch erstellen

Zur Demonstration des push-Befehls wird die Klasse MyClass.java im lokalen Repository des ersten Teampartners ein drittes Mal geändert und anschließend ein Commit erstellt. Die Anmeldung der geänderten Datei MyClass.java und die Durchführung des Commits ist in Abbildung 25 zu sehen.

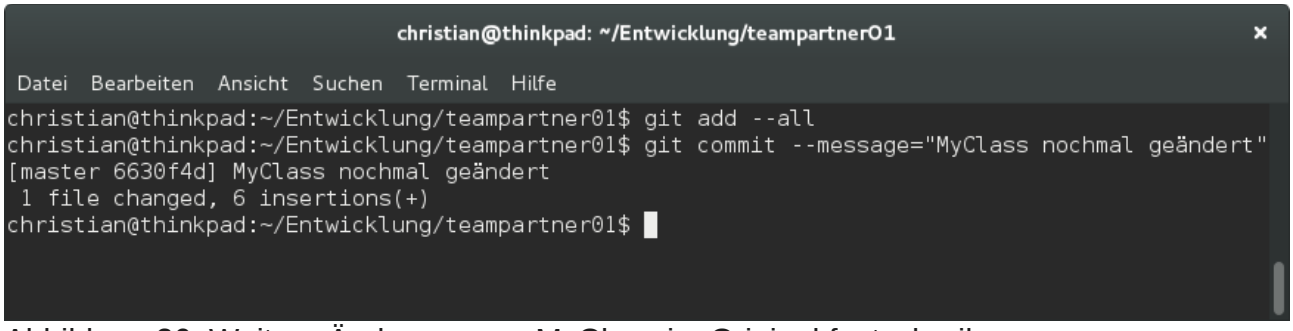

Abbildung 26: Weitere Änderung von MyClass im Original festschreiben

54 Preißel R./Stachmann B., Git, 2012, S. 16.
55 Vgl. Preißel R./Stachmann B., Git, 2012, S. 16.

Das neue Commit des ersten Teampartners soll mit dem push-Befehl ins zentrale Repository übertragen werden. Dazu gibt man dem push-Befehl den Pfad zum Repository und den zu benutzenden Branch als Parameter mit.[56] In Abbildung 27 ist die Durchführung der Übertragung des neuen Commits in das Austausch-Repository zu sehen.

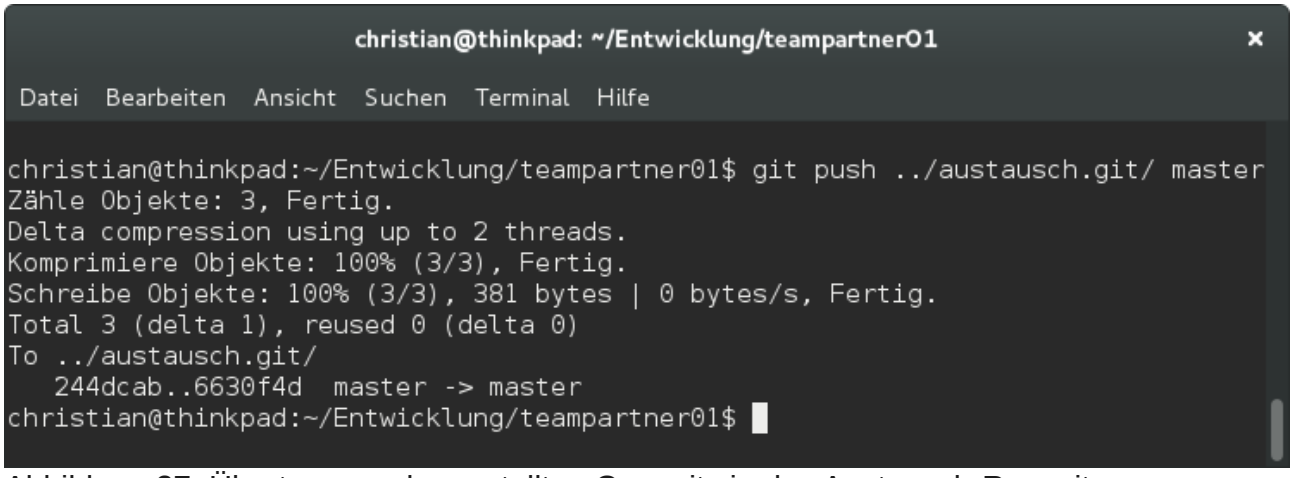

Abbildung 27: Übertragung des erstellten Commits in das Austausch-Repository

Damit der zweite Entwickler auch auf dem aktuellen Stand bleibt, muss dieser sein lokales Repository auf den Stand des zentralen Repositorys bringen. Wie Abbildung 28 zu entnehmen ist, geschieht dies wieder mit dem pull-Befehl mit Angabe des Pfads zum Austausch-Repository und dem Branch.[57]

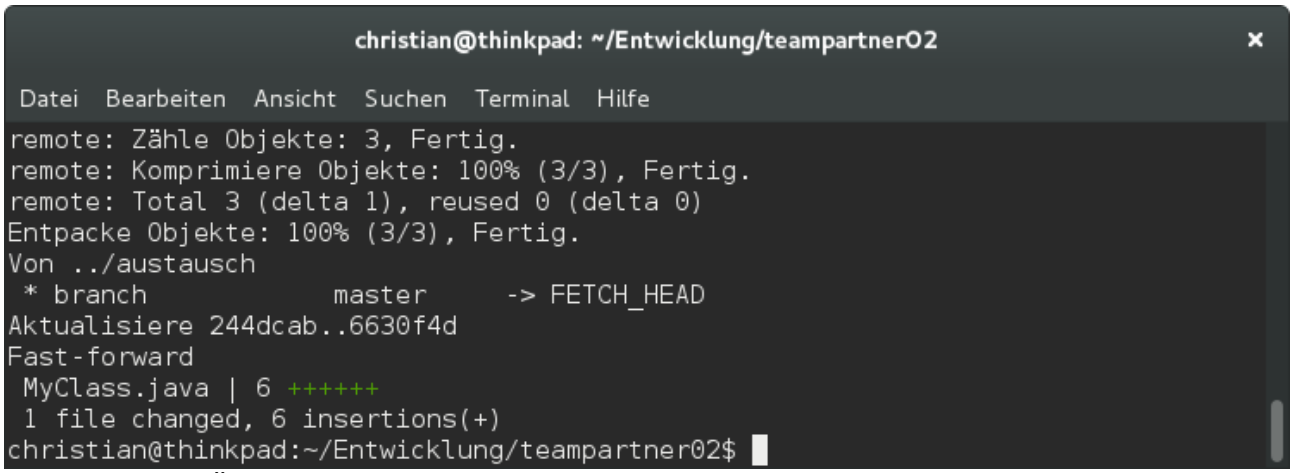

Abbildung 28: Änderungen anderer Entwickler aus dem Austausch-Repository abholen

56 Vgl. Preißel R./Stachmann B., Git, 2012, S. 17.
57 Vgl. Preißel R./Stachmann B., Git, 2012, S. 18.

5. GitHub

„Github ist Dienstleister, um Git-Repositorys zu verwalten und im Internet verfügbar zu machen. Github bietet einen SSH- und einen HTTPS-Zugang zu den Repositorys an.[58]“ Das Anlegen von öffentlichen Repositorys ist kostenlos, hingegen von privaten Repositorys kostenpflichtig. Interessenten an einem Git-Repository müssen dazu einfach die Webseite von Github aufrufen und dort einen Benutzerkonto einrichten. Mittlerweile wird Github ausgiebig von vielen Open-Source-Projekten als Entwicklungszentrale genutzt. „Dazu kann man vorhandenen Github-Repositorys innerhalb von Github klonen. Diese serverseitigen Klone werden Forks genannt. Anschließend wird ganz normal auf dem eigenen Fork-Repository gearbeitet. Es wird ein lokaler Klon angelegt und Änderungen werden via push-Befehl zurückgeschrieben. Möchte man seine Änderungen in das Ausgangs-Repository zurückübertragen, kann man über Github einen Pull-Request an den Eigentümer dieses Repositorys schicken. Dieser kann jetzt mittels pull-Befehl die Änderungen holen und zusammenführen“.[59]

6. Fazit

Git ist ein ausgereiftes dezentrales Versionsverwaltungssystem und erweist sich durch und durch als praxistauglich. Besonders in Punkto Sicherheit und Schnelligkeit punktet Git. Die Tatsache, dass man Unterwegs ohne Internetanbindung weiterarbeiten kann, erweist sich praktisch und ist somit der Hauptvorteil von Git gegenüber der Konkurrenz. Das erzeugen eines Commits sowie das Zusammenführen einer oder mehrerer Dateien lässt sich mit Git schneller und einfacher bewerkstelligen als beispielsweise mit Subversion. Da Geld gleich Zeit ist, hat auch hier Git die Nase vorn. Git haftet der schlechte Ruf an, es sei schwerer zu bedienen als seine Kontrahenten. Dies ist nicht haltbar. Eine marginale Hürde für Umsteiger stellt die Bedienung mit Konsolenbefehlen dar. Und selbst dies ist kein muss, denn es stehen ja zahlreiche Oberflächen zu Verfügung. Nach aktuellem Stand werden verteilte Versionierungssystem zukünftig immer eine größere Rolle in Softwareprojekten spielen. Aus diesem Grund ist es abzusehen, dass der Marktanteil der zentralen System immer mehr zurückgehen wird. Der in den letzten Jahren um Git stattfindende Hype könnte der Tod von Subversion und anderer zentraler Versionsverwaltungen bedeuten.

58 Preißel R./Stachmann B., Git, 2012, S. 237.
59 Preißel R./Stachmann B., Git, 2012, S. 238.

7. Literaturverzeichnis

Literatur:

Preißel, René/Stachmann, Bjørn (2012): Git 1. Auflage, Heidelberg

Jurzik, Heike (2011): Debian/GNU Linux 4. Auflage (aktual.), Bonn

Fischer, Markus (2010): Ubuntu/GNU Linux 5. Auflage (aktual.), Bonn

Hafner, Ullrich (2013): Versionsverwaltung mit Git und Subversion 1. Auflage, München

Internetquellen:

Herkomer, Günter: Die Trends beim Versionsmanagement [Online]. Verfügbar unter: http://www.computer-automation.de/steuerungsebene/steuernregeln/fachwissen/article/79313/1/Die_Trends_beim_Versionsmanagement/ [Zugriff am 24.04.2013]

ubuntu Deutschland e. V.: Grafische Oberflächen für Git [Online]. Verfügbar unter: http://wiki.ubuntuusers.de/Grafische_Oberflächen_für_Git [Zugriff am 18.04.2013]

ubuntu Deutschland e. V.: Versionsverwaltung [Online]. Verfügbar unter: http://wiki.ubuntuusers.de/Versionsverwaltung [Zugriff am 19.04.2013]

o. V.: Los geht’s – Git installieren [Online]. Verfügbar unter: http://git-scm.com/book/de/v1/Los-gehts's-Git-installieren [Zugriff am 26.04.2013]

Wikipedia Foundation: KISS-Prinzip [Online]. Verfügbar unter: http://de.wikipedia.org/wiki/KISS-Prinzip [Zugriff am 21.04.2013]

Abbildungsverzeichnis

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Seminararbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Christian Ondreka