Projektmanagement

9 Versionsmanagement

Vorlesung für AI/MI 4

Prof. Dr. Thomas Allweyer

thomas.allweyer@hs-kl.de

Warum Versionsmanagement?

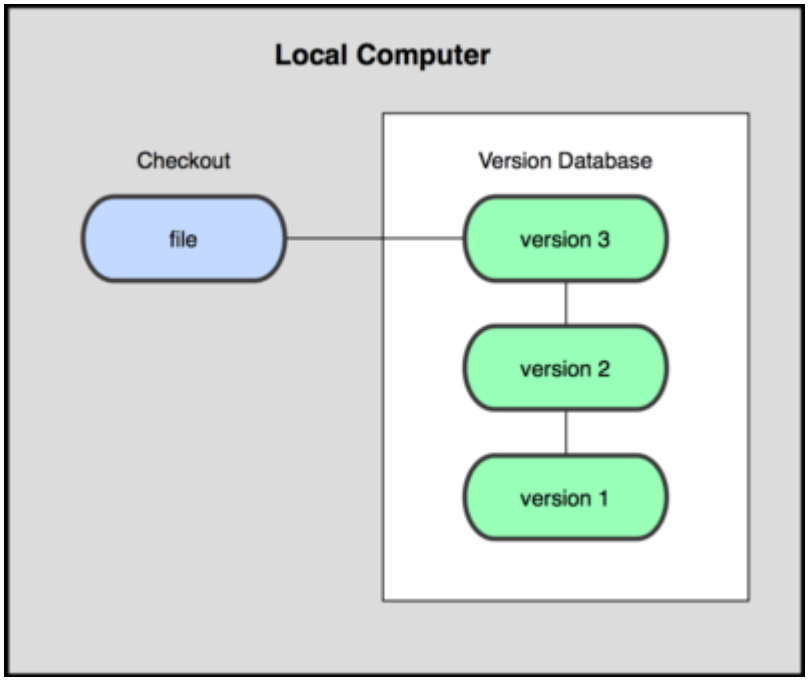
- **Änderungen nachvollziehen**
  - Man hat etwas im Code geändert, möchte dann aber wieder einen früheren Inhalt wiederherstellen
- **Zusammengehörige Dateien verwalten**
  - Alle Quelltextdateien, Bibliotheken, Hilfsdateien usw.
  - Jeweils in den Versionen, die zusammengehören
- **Zusammenarbeit im Team**
  - Alle sollen jeweils mit den aktuellen Dateien der Kollegen arbeiten
  - Geänderte Dateien sollen den anderen wieder zur Verfügung gestellt werden
  - Lösung von Konflikten: 2 Leute haben die gleiche Datei geändert
- **Wartung verschiedener Versionen**
  - Kunde erhält eine stabile Version (Release)
  - Entwicklung arbeitet an der aktuellen Version weiter
  - Bugfix muss in die stabile Version einfließen

Software-Elemente

- **Eindeutig identifizierbare (Zwischen-)Ergebnisse, die im Laufe der Software-Entwicklung entstehen**
- **Beispiele**
  - Pflichtenheft
  - Qualitätsplan
  - Quelltext-Dateien
  - Ausführbare Datei
- **Unterscheidung nach Erzeugungsart:**
  - Quellelement
    - Durch manuelle Eingaben erzeugt, z. B. Pflichtenheft, Quelltext
  - Abgeleitetes Element
    - Z. B. ausführbare Datei
- **Software-Elemente müssen eindeutig bezeichnet werden**
  - Namenskonventionen

Lokale Versionskontrolle

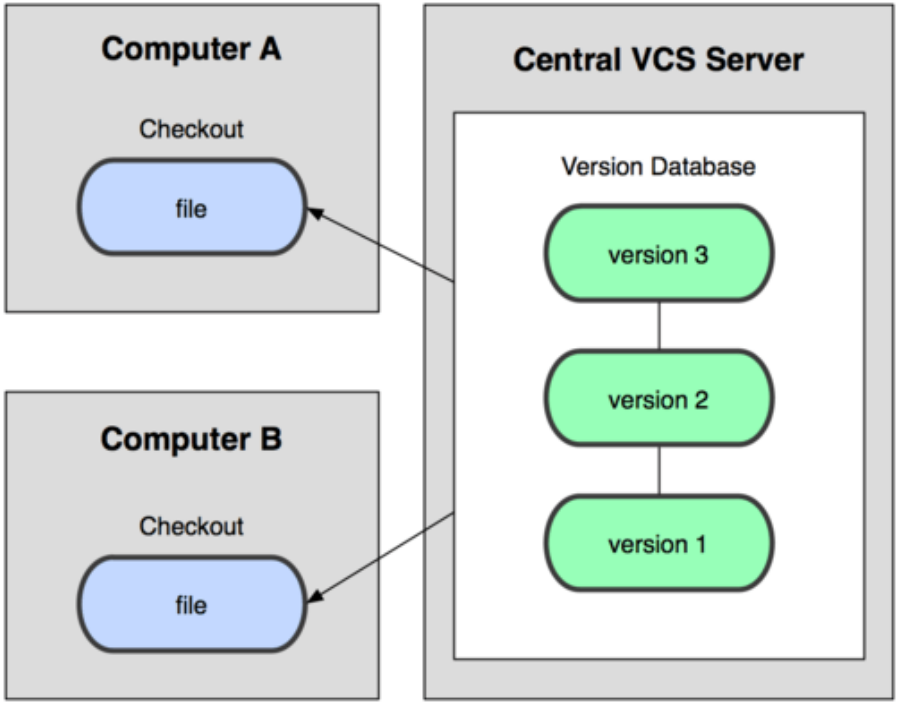
- **Wie kann man Änderungen ggf. rückgängig machen?**
  - Ggf. zu früherem Stand zurückkehren
- **Lokale Versionskontrolle**
  - Z. B. von manchen Betriebssystemen angeboten



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Zentrale Versionskontrolle

- **Wie schafft man es, das alle im Team immer auf dem gleichen Stand arbeiten?**
- **Zentrale Versions-kontrollsysteme**
  - Jeder lädt regelmäßig die aktuellen Versionen herunter (Checkout) und lädt seine Änderungen wieder hoch
  - Z. B. CVS, Subversion, Perforce
  - Problem: Abhängigkeit von einem zentralen Server



VCS = Version Control System

Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Verwaltung von Konfigurationen

- **Wie verwaltet man zusammengehörende Dateien?**
  - In einer älteren Version der Software ist ein Fehler aufgetreten, für den ein Bugfix erstellt werden soll
  - Wie findet man heraus, welche Versionen welcher Quelltext-Dateien zu der betreffenden Version gehört haben?
- **Versionsmanagement-Systeme verwalten Konfigurationen**
  - „Schnappschüsse“ eines gesamten Software-Projekts zu einem bestimmten Entwicklungsstand

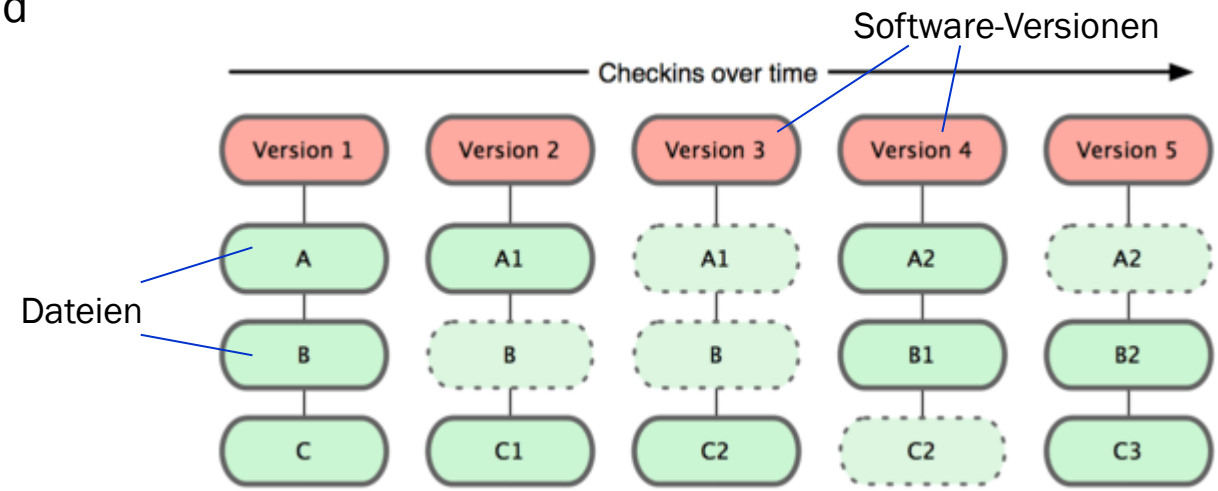
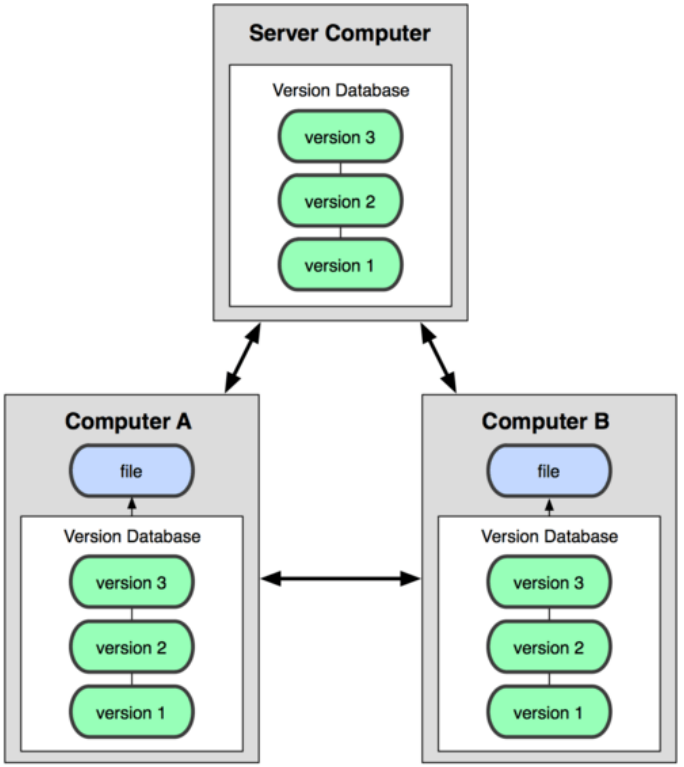


Abbildung: Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1

Verteilte Versionskontrolle

- **Wie kann man auch ohne Verfügbarkeit eines zentralen Servers arbeiten?**

- **Verteilte Versionskontrollsysteme**
  - Z. B. Git, Mercurial
  - Jeder Nutzer erhält eine vollständige Kopie des Repositories
  - Bei Beschädigung ist Zurückspielen von anderen Repository-Kopien möglich
  - Abgleich nicht nur mit zentralem Server, sondern mit anderen dezentralen Repository-Nutzern möglich.
    - Erlaubt z. B. Abbildung von hierarchischen Teamstrukturen



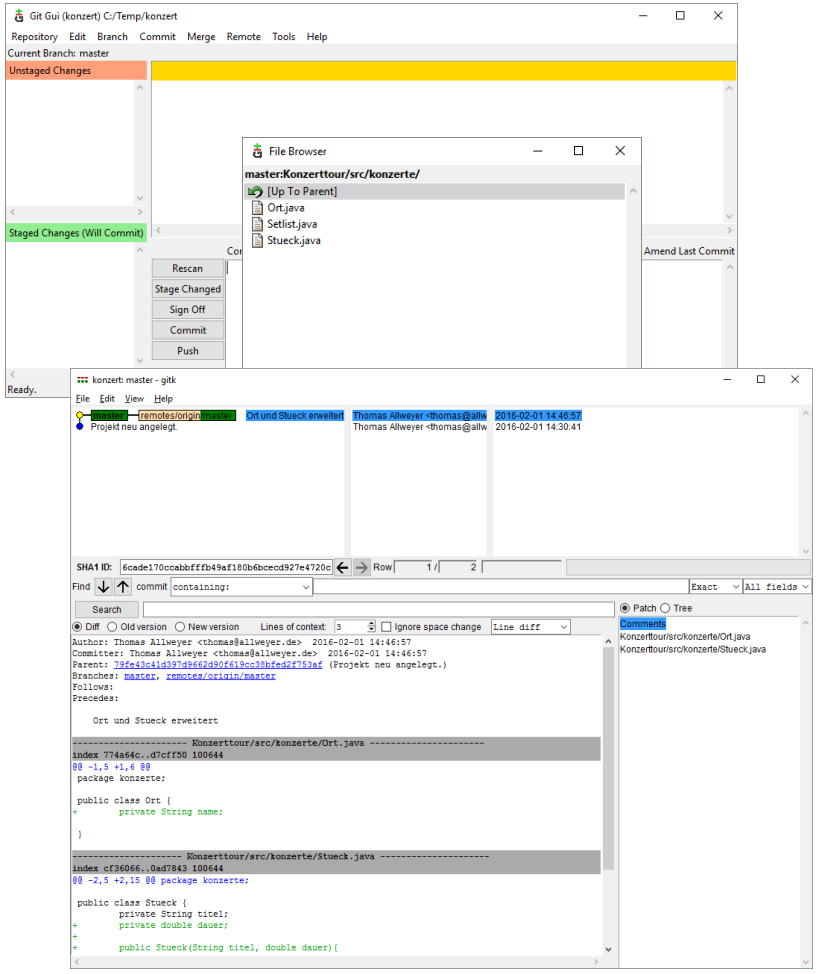
Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Git

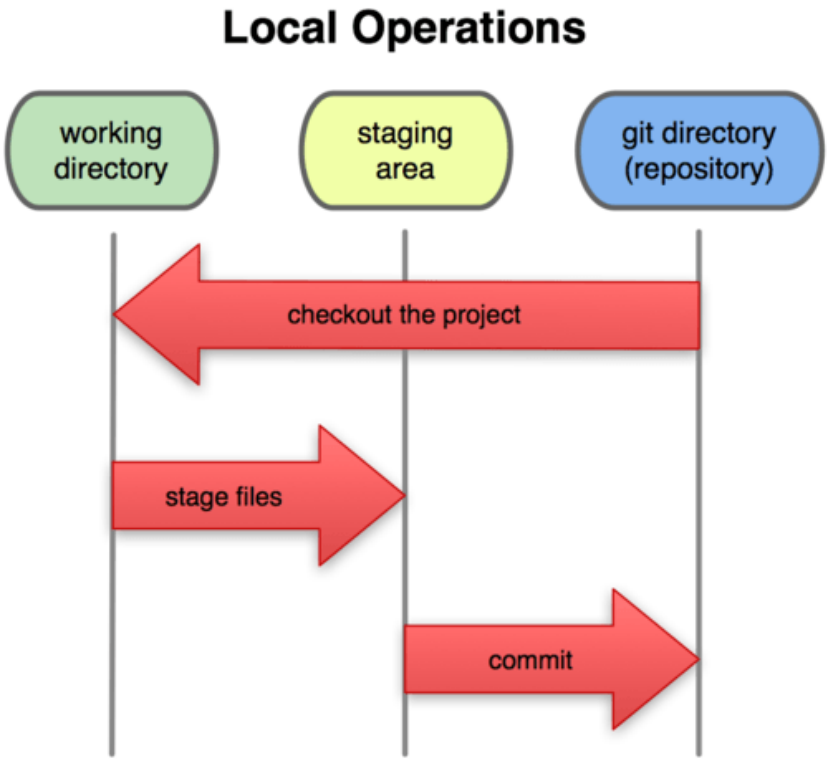
- **Open Source**
- **Seit 2005 aus der Linux Community heraus entwickelt**
- **Von vielen bekannten Open Source-Projekten genutzt, z. B.**
  - Linux, Android, Eclipse, JUnit, LibreOffice, PHP, Qt, Ruby on Rails, VLC Media Player

Download und Installation

- **Download von der Homepage (Installer)**
  - https://git-scm.com
- **Bedienung**
  - Kommandozeile
  - GitGUI (enthalten)
  - Zudem gibt es verschiedene Clients, die die Bedienung komfortabler machen, z. T. mit Integration in Windows Explorer
- **Integration in Entwicklungs-umgebungen**
  - Z. B. EGit für Eclipse
  - In neueren Eclipse-Installationen bereits integriert
  - Wird im Folgenden und in den Übungen verwendet



Git: Drei Zustände



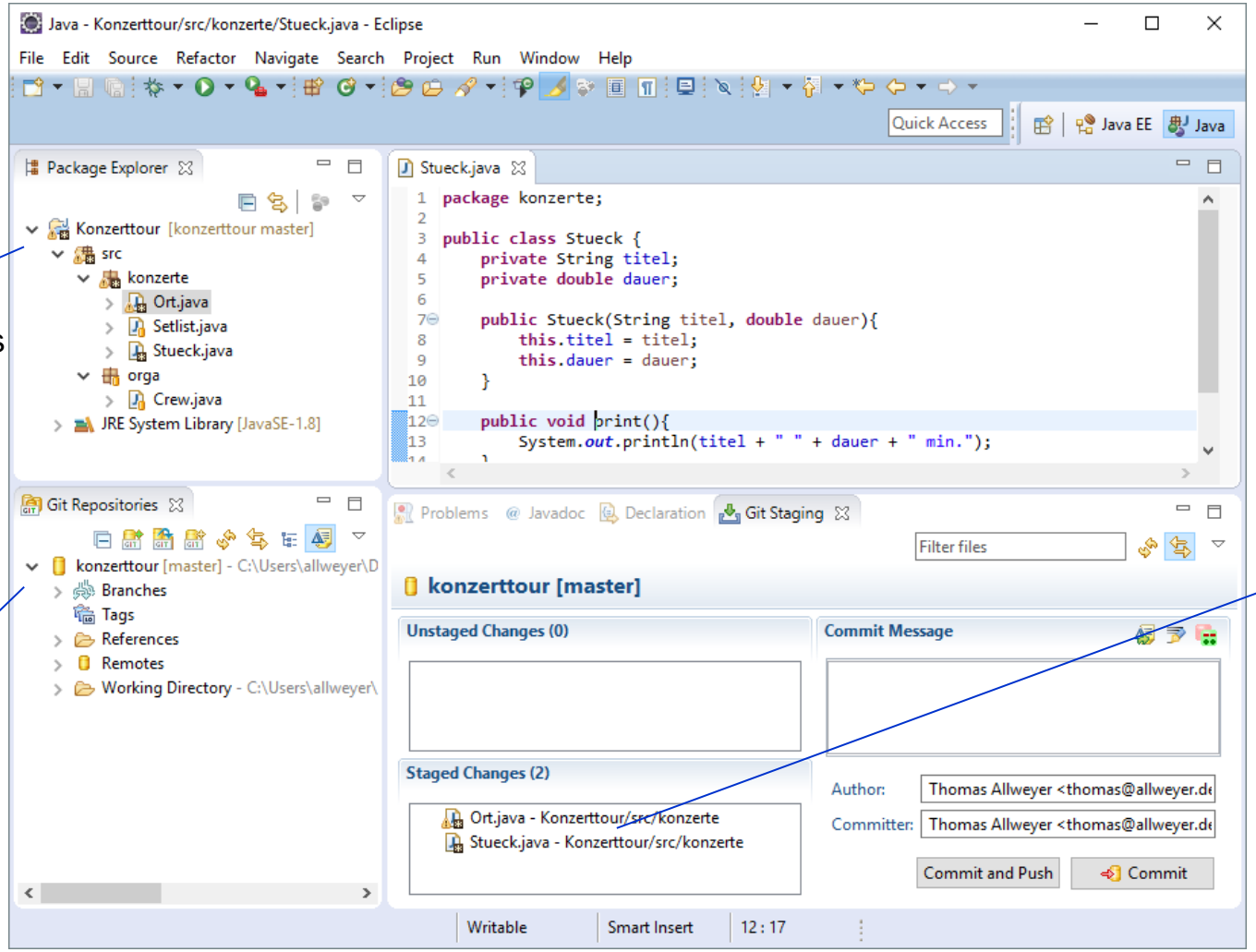
Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

- **Repository (auf lokalem Rechner)**
  - Die von Git verwalteten Projekt-daten mit allen Änderungen etc.
- **Working Directory**
  - Mit diesen Dateien arbeiten Sie (wie gewohnt)
  - Checkout: Daten aus dem Repository ins Working Directory laden
- **Staging Area (oder „Index“)**
  - Hier werden geänderte Daten für die Speicherung im nächsten Commit vorgemerkt
  - Commit: Vorgemerkte Daten ins Repository übernehmen

Drei Zustände in Eclipse (EGit)

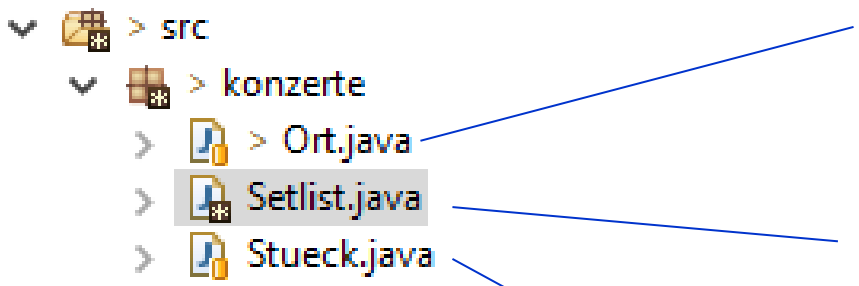
Working Directory (gewöhnliches Java-Projekt in Eclipse)

Lokales Repository



In Staging Area vorgemerkte geänderte Dateien

Zustände der Dateien



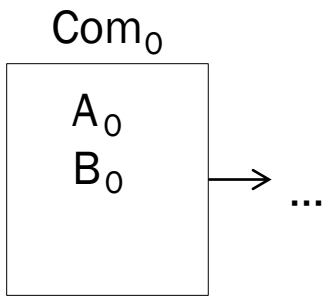
- **Modified**
  - Geändert, aber noch nicht vorgemerkt
- **Staged**
  - Für den nächsten Commit vorgemerkt
- **Commited**
  - Liegt so im Repository, wurde im Workspace nicht verändert.

Arbeit mit Git

- **Grundlegender Arbeitsablauf**
  1. Dateien im Working Directory bearbeiten
  2. Gewünschte Dateien für nächsten Commit vormerken
  3. Commit durchführen
     - Damit ist der Schnappschuss aus der Staging Area heraus dauerhaft in lokalem Repository gespeichert.
- **Vereinfachung**
  - Häufig will man den augenblicklichen Zustand des Working Directory direkt committen
  - Hierfür lassen sich Schritt 2 und 3 in einen Commit-Befehl zusammenfassen

Aufgabe

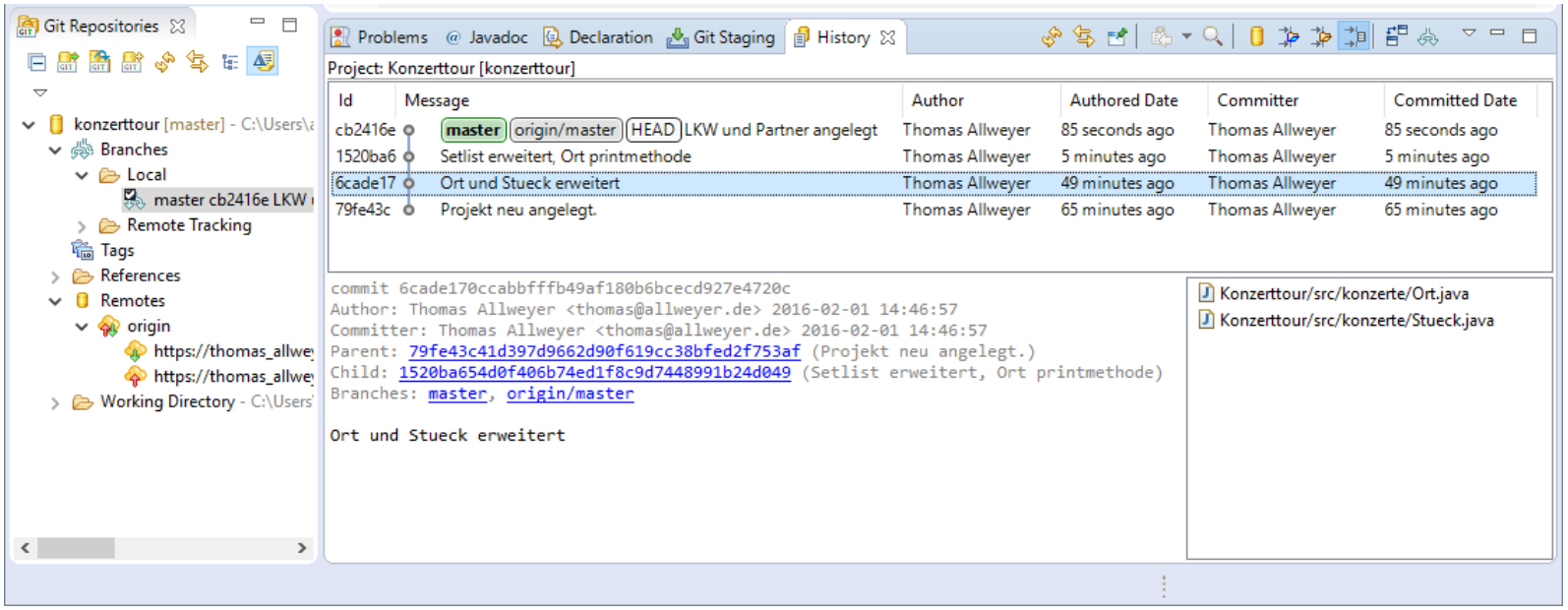
- **Bei Ihrer Entwicklung geschieht Folgendes:**
  - Sie legen zwei Dateien A und B an und führen einen Commit aus
  - Dann ändern sie die Datei A, legen eine weitere Datei C an und führen einen weiteren Commit aus
  - Sie ändern A und C und führen einen weiteren Commit aus
  - Sie ändern A und führen einen weiteren Commit aus
  - Sie ändern A und B und führen einen weiteren Commit aus
- **Stellen Sie die Historie Ihres Projektes grafisch dar (s. u.)**
  - Nummerieren Sie die Commits und die Versionen der Dateien durch, beginnend mit 0



Git Repository anlegen

- **Zwei Möglichkeiten**
  - Neues Repository lokal anlegen und vorhandene Dateien des Projekts hineinlegen (mittels Commit)
  - Ein existierendes Repository klonen
    - Von einem Server (z. B. mittels https oder ssh)
    - oder von einem gemeinsam genutzten Laufwerk
    - Kopiert die gesamten Daten inkl. Historie in ein neues lokales Repository
    - Änderungen des lokalen Repositories können in das entfernte Verzeichnis zurückgespielt werden („Push“)

Historie


Git Repositories
konzerttour [master] - C:\Users\a
Branches
Local
master cb2416e LKW
Remote Tracking
Tags
References
Remotes
origin
https://thomas_allwe
https://thomas_allwe
Working Directory - C:\Users
Problems
Javadoc
Declaration
Git Staging
History
Project: Konzerttour [konzerttour]
Id
Message
Author
Authored Date
Committer
Committed Date
cb2416e
master
origin/master
HEAD
LKW und Partner angelegt
Thomas Allweyer
85 seconds ago
Thomas Allweyer
85 seconds ago
1520ba6
Setlist erweitert, Ort printmethode
Thomas Allweyer
5 minutes ago
Thomas Allweyer
5 minutes ago
6cade17
Ort und Stueck erweitert
Thomas Allweyer
49 minutes ago
Thomas Allweyer
49 minutes ago
79fe43c
Projekt neu angelegt.
Thomas Allweyer
65 minutes ago
Thomas Allweyer
65 minutes ago
commit 6cade170ccabbfffb49af180b6bcecd927e4720c
Author: Thomas Allweyer <thomas@allweyer.de> 2016-02-01 14:46:57
Committer: Thomas Allweyer <thomas@allweyer.de> 2016-02-01 14:46:57
Parent: 79fe43c41d397d9662d90f619cc38bfed2f753af (Projekt neu angelegt.)
Child: 1520ba654d0f406b74ed1f8c9d7448991b24d049 (Setlist erweitert, Ort printmethode)
Branches: master, origin/master
Ort und Stueck erweitert
Konzerttour/src/konzerte/Ort.java
Konzerttour/src/konzerte/Stueck.java

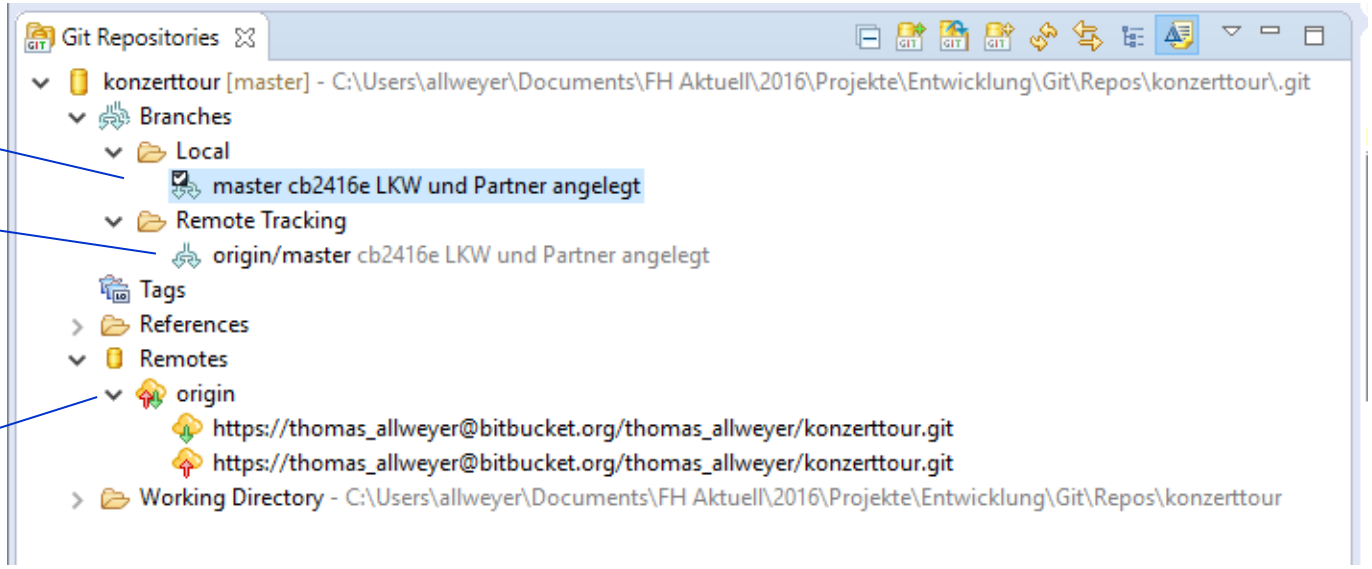
Remote Repositories

- **Ein lokales Repository kann mit entfernten (remote) Repositories verbunden sein**
  - Verbindung wird beim Klonen automatisch angelegt
  - Auch nachträglich kann eine Verbindung hergestellt werden
  - Änderungen können zwischen lokalem und entferntem Repository abgeglichen werden.

Lokaler Entwicklungs-Strang (Branch)

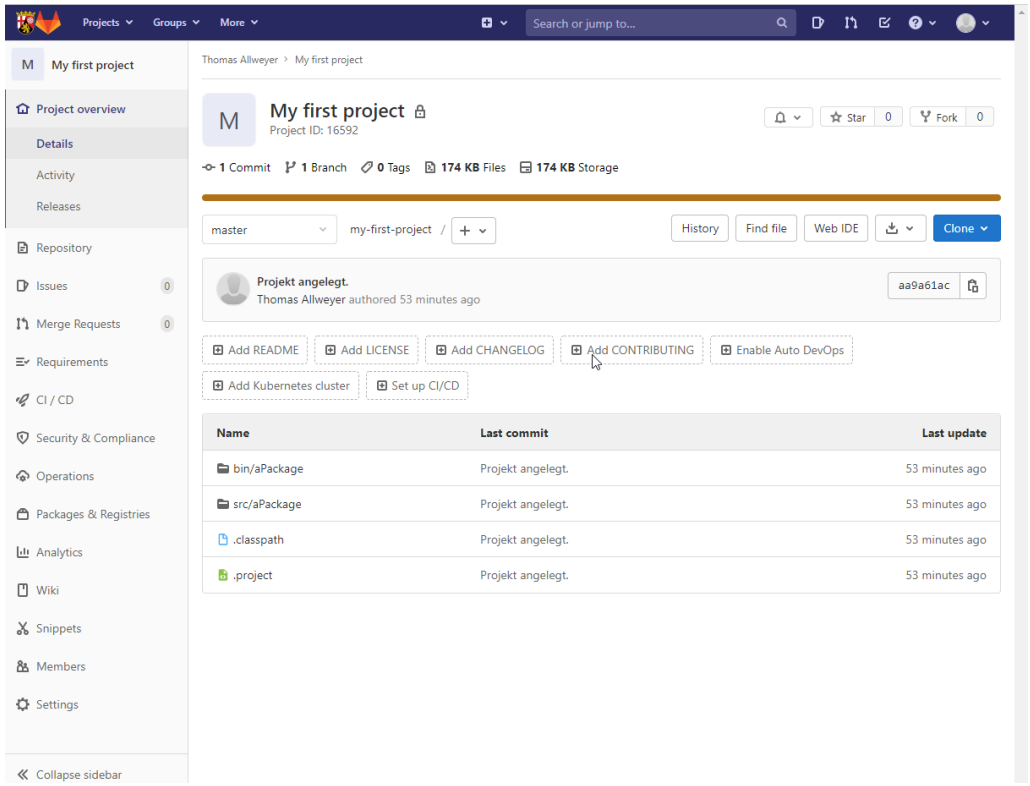
Entwicklungsstrang (Branch) im entfernten Repository

Entferntes Repository



Remote Repositories - Optionen

- Gemeinsam genutztes Verzeichnis
- Eigener Server
- Spezielle Cloud-Server für Git-Repositories
  - Z. B. GitHub, Bitbucket, GitLab
  - Teilweise kostenlose Angebote für kleine Teams
- GitLab-Server der Hochschulen in RLP
  - https://gitlab.rlp.net/
  - Nutzung mit Studierendenaccount



Abgleich mit entferntem Repository – Daten herunterladen

- **„Fetch“**
  - Holt den augenblicklichen Stand eines Branch (Entwicklungsstrang) vom entfernten Repository, führt ihn aber noch nicht mit dem lokalen Strang zusammen.

- **„Merge“**
  - Führt den mittels „Fetch“ geholten Branch mit dem lokalen Branch zusammen.
  - Wurden sowohl im remote Repository als auch im lokalen Repository Änderungen durchgeführt, so versucht Git, diese automatisch zusammenzuführen.
  - Falls das nicht möglich ist, werden die Unterschiede in den betroffenen Dateien angezeigt und der User muss sie manuell zusammenführen.

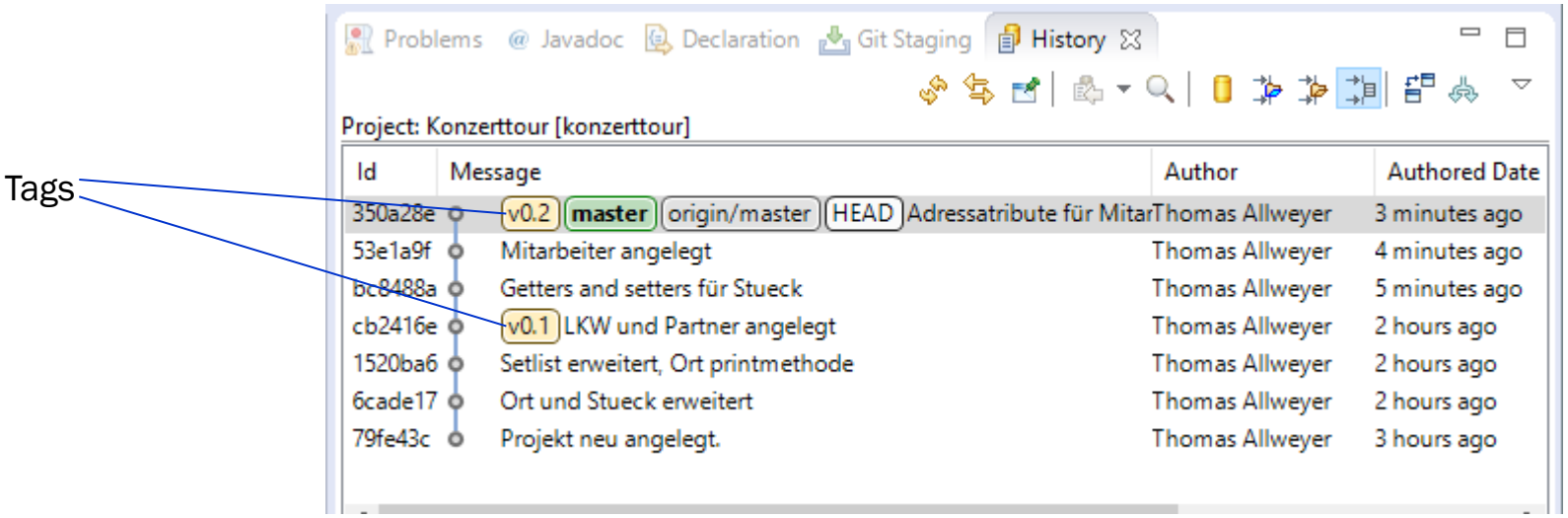
- **„Pull“**
  - Fetch und Merge in einem Schritt

Abgleich mit entferntem Repository – Daten hochladen

- **„Push“**
  - Man muss Schreibrechte für das remote Repository haben
  - Es darf zwischenzeitlich keine Änderung im remote Repository stattgefunden haben
  - Falls doch Änderungen stattgefunden haben:
    - Zunächst Änderungen herunterladen und zusammenführen (fetch und merge bzw. pull)
    - Anschließend erneut hochladen

Tagging

- **Markierung besonders wichtiger Schnappschüsse des Projektes, insbesondere für**
  - Release
    - zum Kunden ausgeliefert
  - Baseline
    - „internes Release“, definierter, getesteter Stand

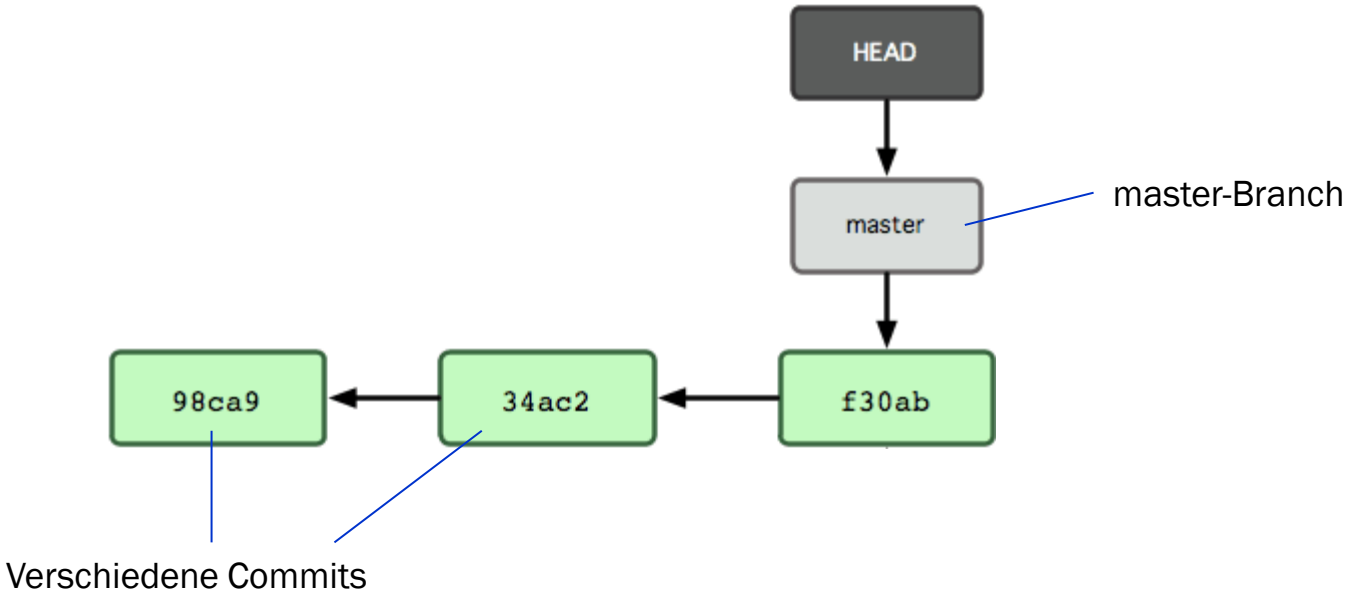


Branching (Verzweigung)

- **Ein Branch ist eine Verzweigung vom Hauptentwicklungsstrang**
- **Beispiel:**
  - Das Software-Entwicklungsteam liefert Version 1.0 der Software aus
    - Als „V1.0“ getaggt
  - Anschließend wird weiter entwickelt, um eine Version 2.0 zu erstellen
  - Nun tritt ein Fehler in Version 1.0 auf
  - Da der Fehler schwerwiegend ist, genügt es nicht, ihn in der nächsten Version zu beheben.
  - Daher wird ein (Branch) Zweig angelegt
    - Zweigt bei V1.0 vom Branch „master“ (Hauptentwicklungsstrang) ab
    - Hier wird die Fehlerbehebung durchgeführt und eine verbesserte Version V1.1 erstellt, die an die Kunden ausgeliefert wird
  - Der Fehler soll natürlich auch im master-Branch für die Version 2.0 behoben werden
    - Hierzu wird der Fehlerbehebungs-Branch mit dem master-Branch zusammengeführt („Merge“)

Branching (1)

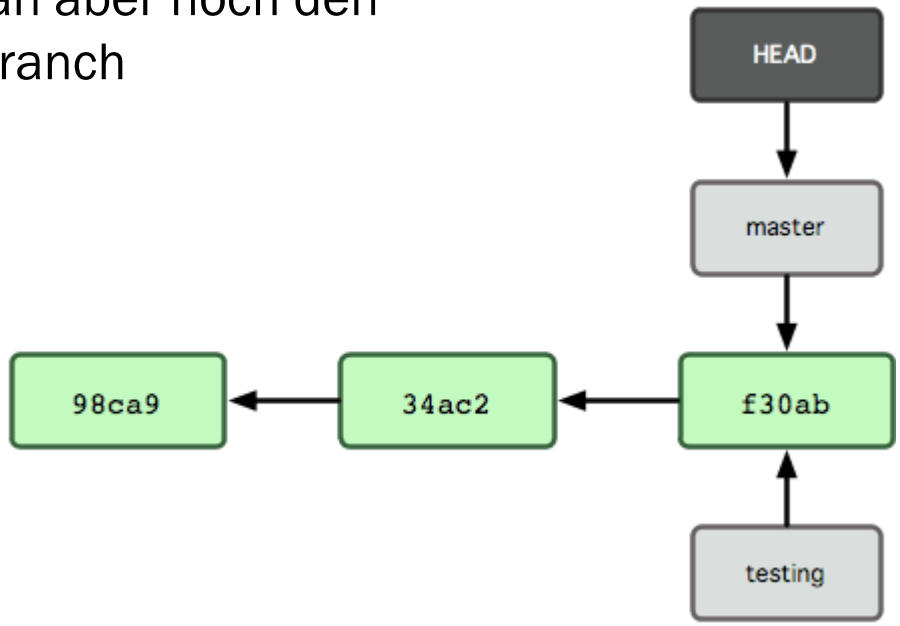
- Ohne Verzweigung: Es gibt nur einen Branch, master
- HEAD ist ein Zeiger auf den gerade ausgecheckten Zustand im Working Directory



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Branching (2)

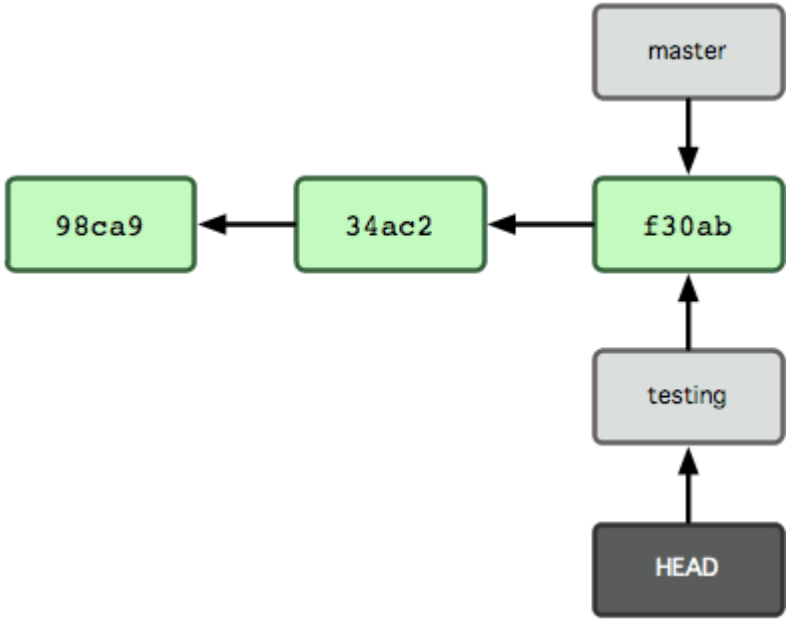
- **Branch „testing“ anlegen**
  - Das Working Directory enthält momentan aber noch den master-Branch



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Branching (3)

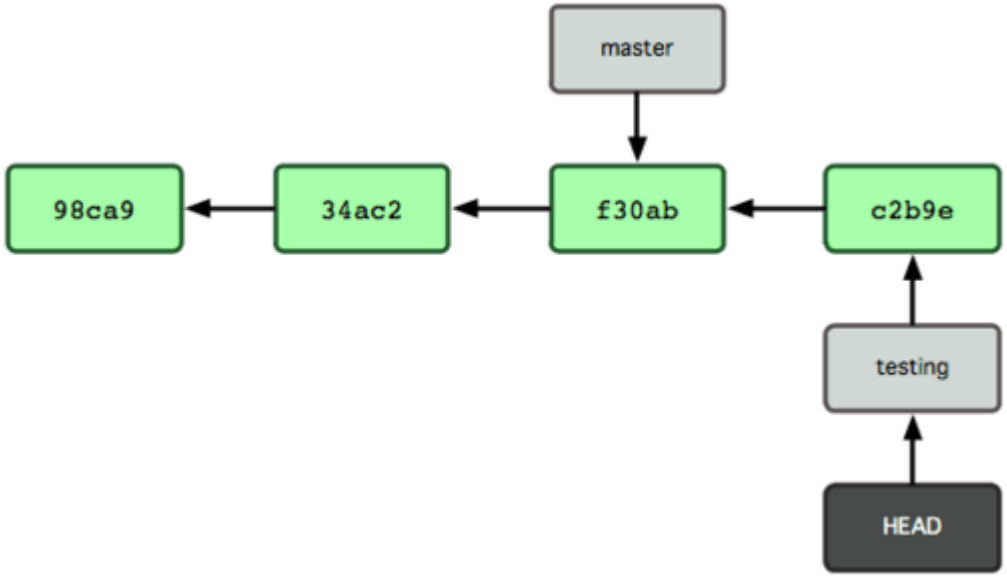
- **Branch „testing“ auschecken**
  - Jetzt arbeitet man in dem Branch



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Branching (4)

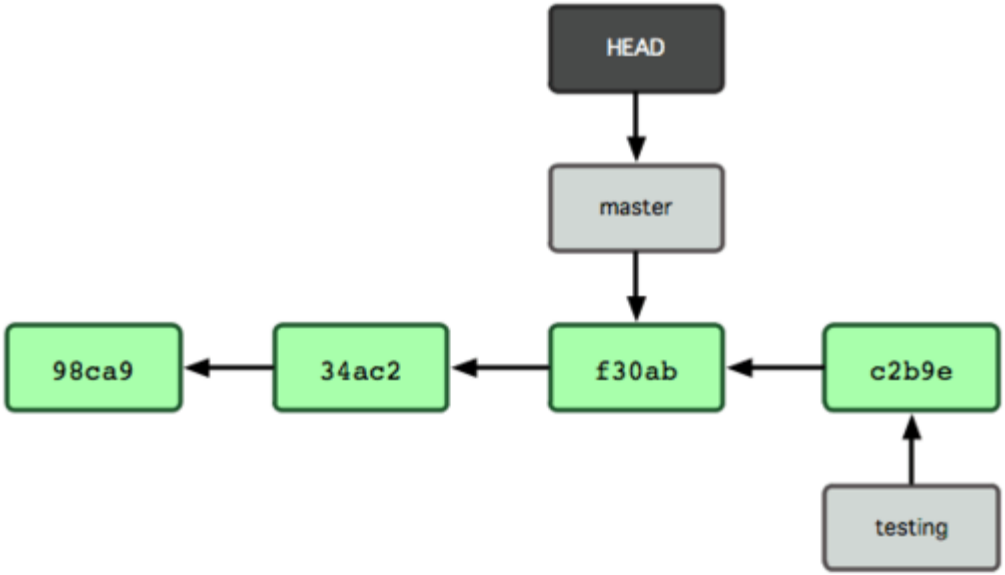
- **Commit durchführen**
  - Der Commit gehört zu dem Branch „testing“
  - Der master-Branch ist unverändert



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Branching (5)

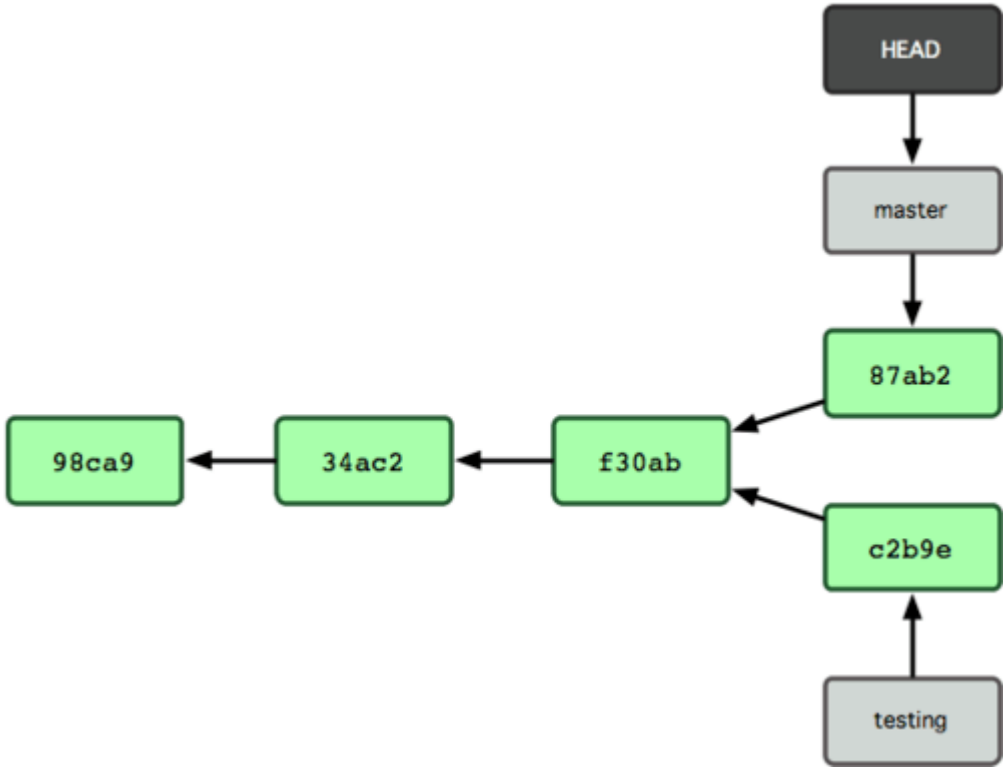
- **Zurück zu „master“ wechseln („master“ auschecken)**
  - Es wird also wieder im Master gearbeitet



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Branching (6)

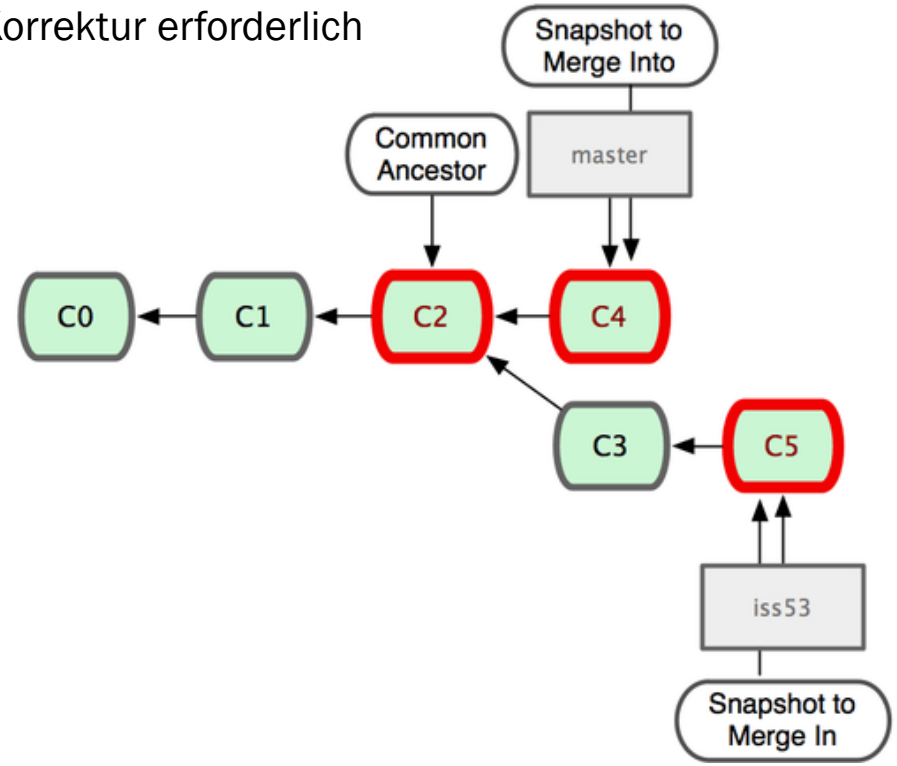
- Weiterer Commit im Master



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Merging (1)

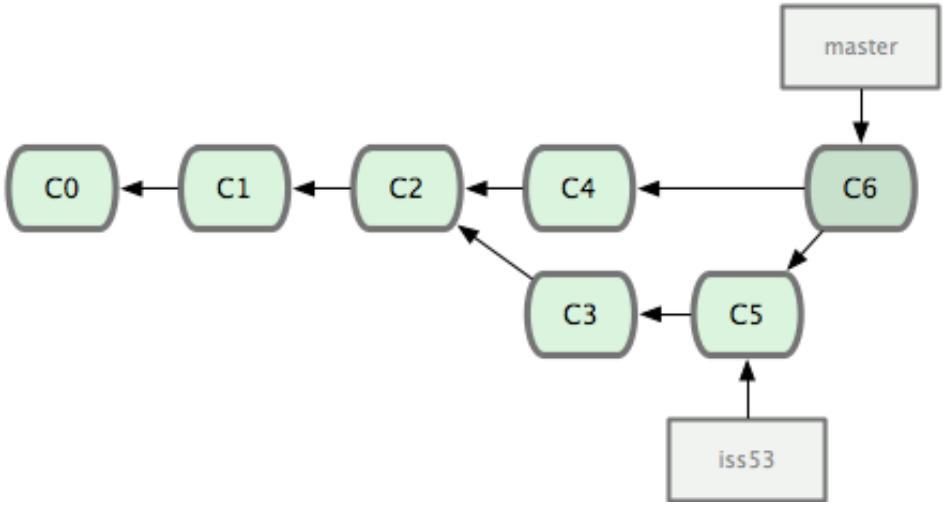
- Git ermittelt den gemeinsamen Vorfahren der beiden Versionen (C2)
- Es ermittelt die Unterschiede von C4 und C5 zu diesem Vorfahren
- Es versucht, diese Unterschiede zusammenzuführen
  - Bei Konflikten manuelle Korrektur erforderlich



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Merging (2)

- Ergebnis:



Inhalt nach Chacon/Straub: Pro Git. https://git-scm.com/book/de/v1
Abbildungsquelle: ebda.

Aufgabe

- **Bei Ihrer Entwicklung geschieht weiterhin Folgendes:**
  - Sie taggen den 3. Commit (Commit Nr. 2, wenn man bei der Nummerierung mit 0 beginnt) als „Version1.0“ und den zuletzt committeten Stand als „Version2.0“
  - Sie erstellen einen Branch „bugfix“, ausgehend von dem mit Version1.0 getaggten Commit.
  - In dem Branch ändern Sie Datei A und C und führen einen Commit durch
  - Sie wechseln in den Master, ändern dort Datei A und B und führen einen Commit durch.
  - Sie wechseln wieder in den Branch, ändern dort die Datei A und B und führen einen Commit durch.
  - Sie wechseln wieder in den Master, mergen den Branch in den Master und führen erneut einen Commit durch.
- **Erweitern Sie die grafische Darstellung der Historie Ihres Projektes**

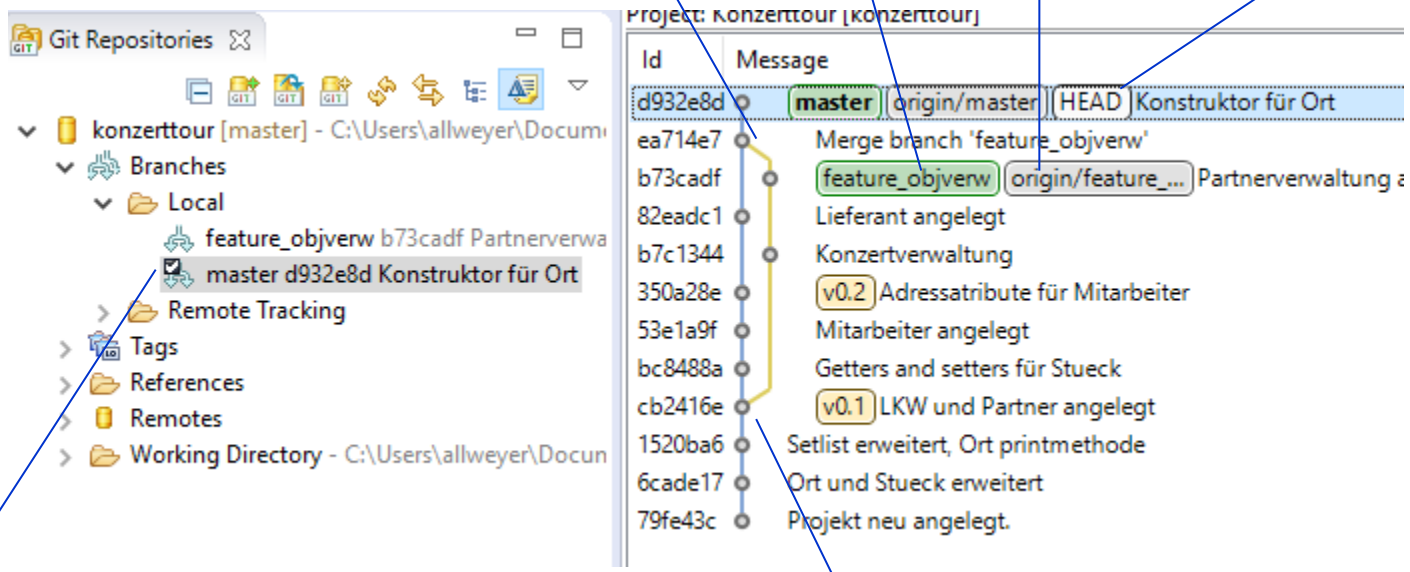
Branching and Merging

Name des Branches

Name des Branches im verbundenen Remote Repository

HEAD

Mergen des Branches in den master-Branch

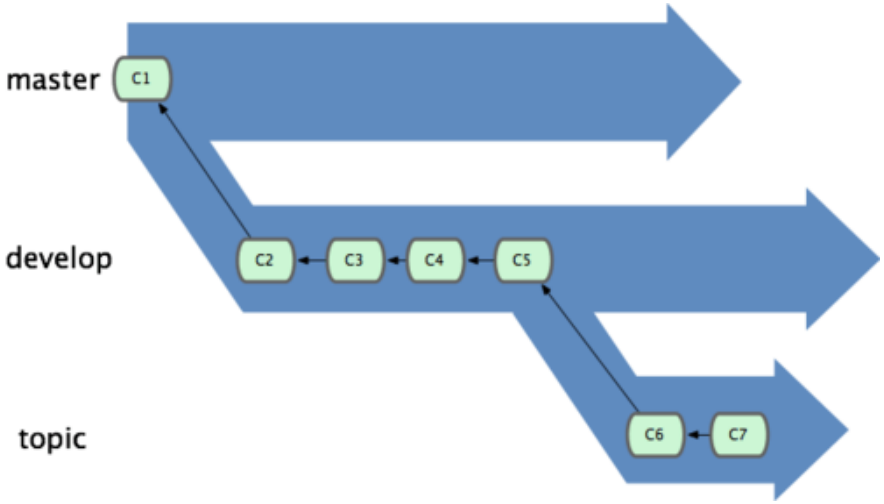


Dieser Branch ist gerade ausgecheckt, d. h. HEAD verweist darauf und im Working Directory wird mit diesem Branch gearbeitet

Verzweigung

Mehr zu Branching

- **Auch Branches können in ein entferntes Repository gepusht werden**
  - Backup
  - Gemeinsame Arbeit an einem Branch
- **Auch Branches von Branches sind möglich**



Abbildungsquelle Chacon/Straub: Pro Git.
https://git-scm.com/book/de/v1

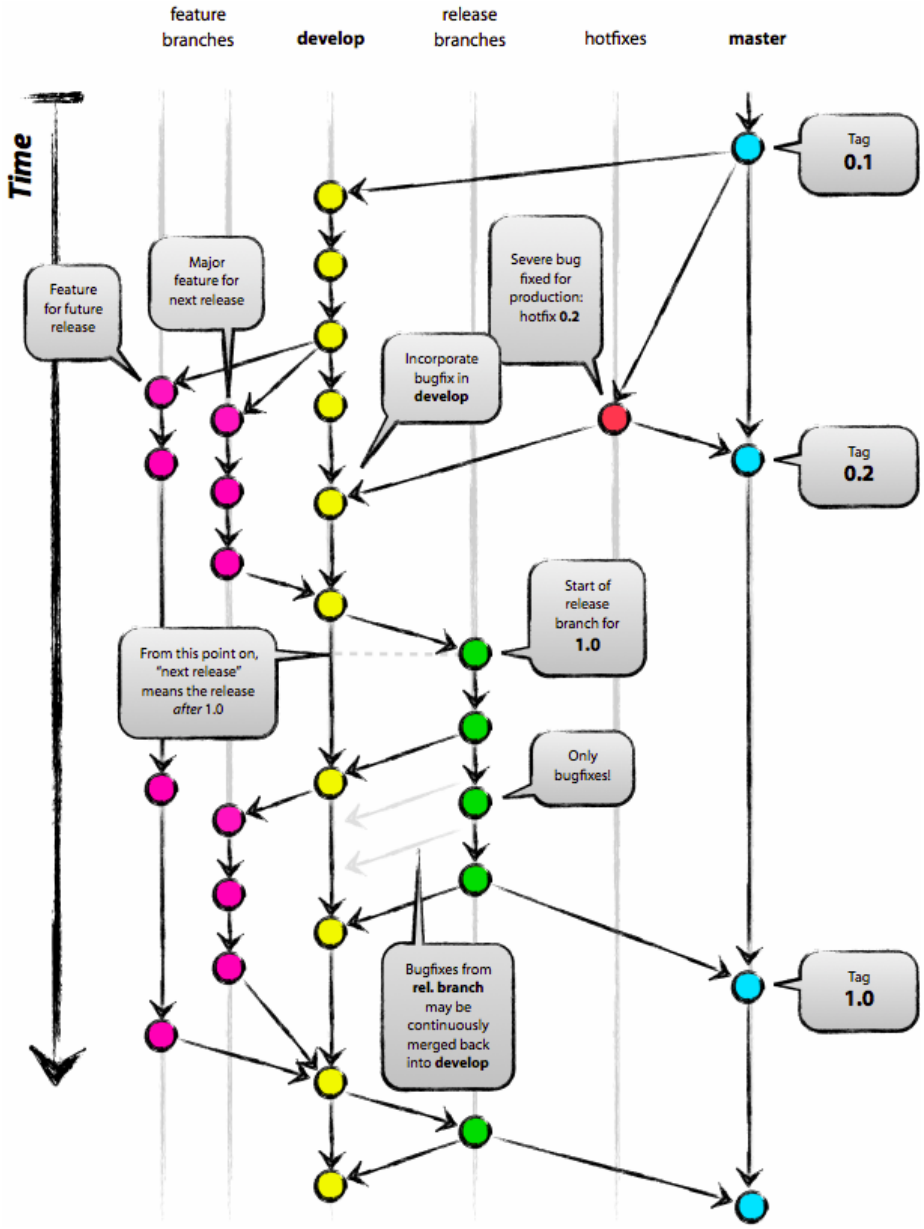
Einsatz von Branching

- **Wann sollte man einen Branch erstellen?**
  - Wenn man Bugfixes für eine ältere Version entwickeln möchte (s. o.)
  - Wenn man eine größere Änderung oder Erweiterung entwickeln möchte
  - Wenn man etwas entwickelt, von dem noch nicht sicher ist, ob es verwendet wird
  - Wenn man experimentieren möchte und sich nicht sicher ist, ob es auch so funktioniert
  - Wenn man gebeten wird, seine aktuelle Arbeit in einen Branch auszulagern, weil grundlegende Änderungen im Master anstehen
- **Änderungen des master-Branchs regelmäßig in den eigenen Branch mergen**
  - Damit wird sichergestellt, dass die eigenen Änderungen mit den Änderungen im master zusammenpassen
  - Reduziert Probleme beim späteren Merge des Branchs in den master

Nach http://programmers.stackexchange.com/questions/165725/git-branching-and-tagging-best-practices

Beispiel

- Beispiel eines Branching-Modells aus einer realen Entwicklungsorganisation

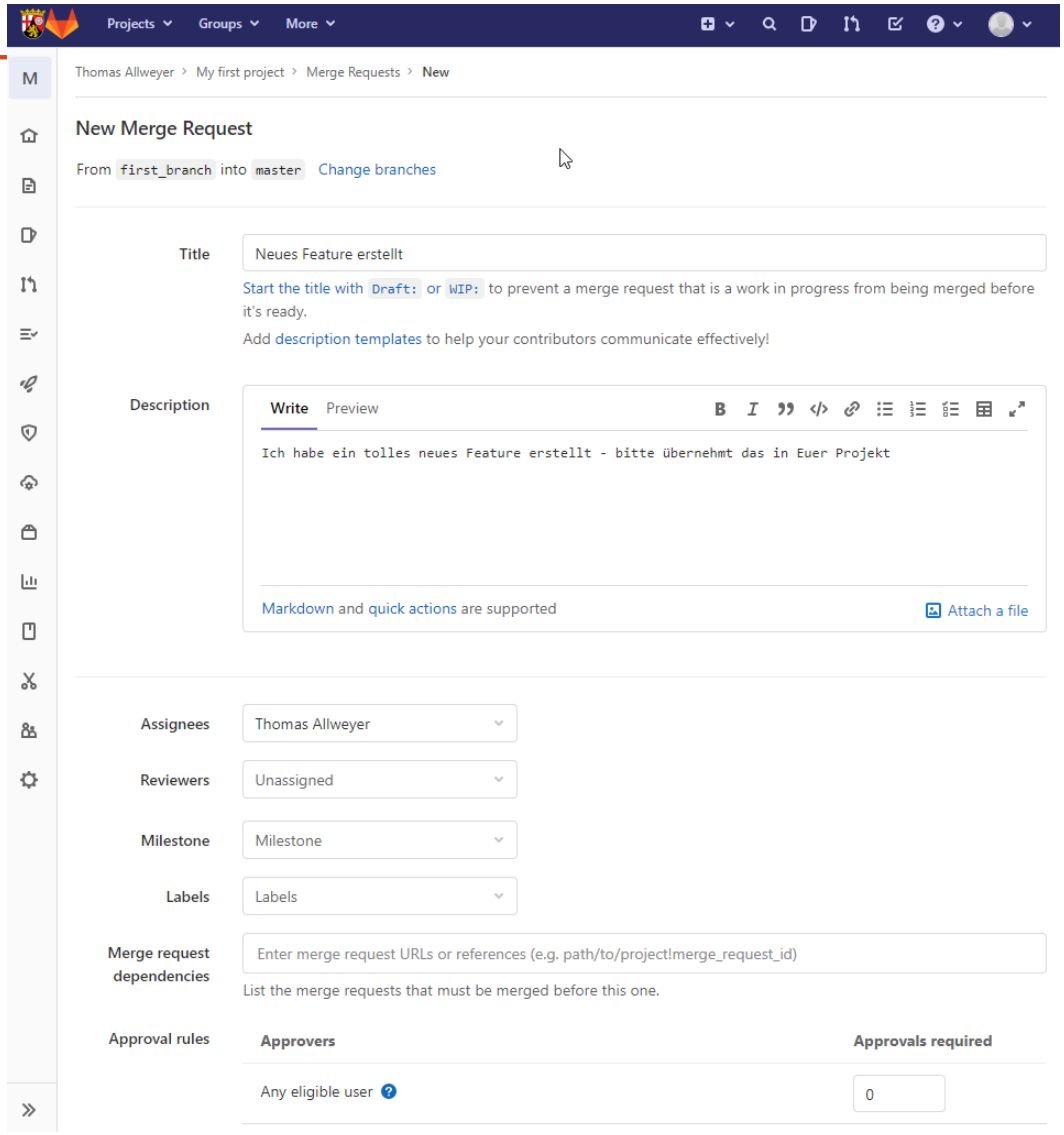


Quelle: http://nvie.com/posts/a-successful-git-branching-model/

Merge-Request

- **Änderungen durch Entwickler ohne Schreibrechte**
  - Z. B. abgestuftes Berechtigungskonzept in großen Projekten
  - Open Source-Projekte – kontrolliertes Einbringen von Beiträgen verschiedenster Entwickler
- **Vorgehen:**
  - Der Entwickler erstellt einen Klon des Repositories
  - Erstellt darin einen Branch, in dem er seine Entwicklung durchführt
  - Anschließend stellt er seinen Klon dem Administrator des Hauptrepositories zur Verfügung
  - Dieser prüft die Änderungen und mergt sie in das Hauptrepository
- **Bei Git-Repository-Hostern kann man Merge-Requests erstellen, mit denen der jeweilige Repository-Eigentümer informiert wird**
  - Z. T. auch als „Pull-Request“ bezeichnet

Merge-Request



Weitere Infos zu Git

- **Git Pro**
  - Buch zu Git, auch auf Deutsch, komplett online verfügbar
  - https://git-scm.com/book/de/v2
- **Git – Der einfache Einstieg**
  - https://rogerdudler.github.io/git-guide/index.de.html
- **Egit Tutorials**
  - http://eclipsesource.com/blogs/tutorials/egit-tutorial/
  - http://www.vogella.com/tutorials/EclipseGit/article.html