JAVACREAM

Training
Consulting
Projectmanagement

GIT

Kurze Vorstellung

- Name
- Rolle im Unternehmen
- Themenbezogene Vorkenntnisse
- Aktuelle Problemstellung
- Konkrete individuelle Zielsetzung

Ausgangssituation

Grundaufgabe eines Versionsverwaltungssystems

- Quellcode, Konfigurationsdateien, … = “Werke” werden mit zusätzlichen Meta-Informationen angereichert
  - Wer hat wann warum welche Änderungen gemacht?
  - Der aktuelle Zustand mehrerer Werke wird zu einem “Stand” zusammengefasst
    - Stände repräsentieren den Fortschritt einer Software-Entwicklung
- Wiederherstellen von Ständen
- Konsolidieren von Ständen
- Zentrale Ablage auf einem Server
  - Authentifizierung und Autorisierung
- Parallelisierte Weiterentwicklung von Ständen
  - insbesondere Team-Zusammenarbeit

Realisierung mit Git

- Quellcode, Konfigurationsdateien, … = “Werke” werden mit zusätzlichen Meta-Informationen angereichert
  - Wer hat wann warum welche Änderungen gemacht?
  - Der aktuelle Zustand mehrerer Werke wird zu einem “Stand” zusammengefasst
    - Stände repräsentieren den Fortschritt einer Software-Entwicklung
- Wiederherstellen von Ständen
- Konsolidieren von Ständen
- Zentrale Ablage auf einem Server
  - Authentifizierung und Autorisierung
- Parallelisierte Weiterentwicklung von Ständen
  - insbesondere Team-Zusammenarbeit

Realisierung mit Git mit einem Git Server

- Quellcode, Konfigurationsdateien, … = “Werke” werden mit zusätzlichen Meta-Informationen angereichert
  - Wer hat wann warum welche Änderungen gemacht?
  - Der aktuelle Zustand mehrerer Werke wird zu einem “Stand” zusammengefasst
    - Stände repräsentieren den Fortschritt einer Software-Entwicklung
- Wiederherstellen von Ständen
- Konsolidieren von Ständen
- Zentrale Ablage auf einem Server
  - Authentifizierung und Autorisierung
- Parallelisierte Weiterentwicklung von Ständen
  - insbesondere Team-Zusammenarbeit

Git Server

- Separate Produkte unabhängig vom reinen Git
- Hersteller
  - Microsoft
    - **GitHub**
      - Primär eine Cloud-basierte Lösung, Server laufen in der Microsoft Cloud
  - Atlassian
    - BitBucket
      - eine Cloud-basierte Lösung
      - separater, selbst-gehosteter Server
  - Gitlab
    - GitLab
      - eine Cloud-basierte Lösung
      - separater, selbst-gehosteter Server

First Contact

Git Installation

- Executable “git.exe”
  - Vollständiges Versionsverwaltungssystem
    - Kein Command Line Interface zu einem Server!
  - Kein Hintergrund-Prozess
- Native Installation mit notwendigen Admin-Rechten ist in der Regel nicht notwendig
  - “Portable Git”
- Eine Vielzahl von Produkten (z.B. Entwicklungsumgebungen) haben eine Git-Erweiterung
  - diese nutzen aber nichts anderes als das installierte git.exe

Erste Befehle

- Elementare Konfiguration
- ~~git config server.url = http://github.com/fi/…~~
- git config --global user.name “Rainer Sawitzki”
- git config --global user.email training@rainer-sawitzki.de
  - Ablage der Konfigurationen erfolgt in einer formatierten Text-Datei (“.gitconfig”) in Ihrem user.home-Verzeichnis
- Exkurs
  - Scope der Konfiguration
    - --global ist gültig für den aktuell angemeldeten Benutzer
    - --local (= default) bezieht sich auf das aktuelle Git-Repository

Erste Befehle, Part 2

- Erzeugen eines Git-Repositories
  - Zu diesem Zeitpunkt des Seminars untypisch!
  - mkdir git_training
  - cd git_training
  - mkdir first
  - cd first
  - git init

Angelegt wurde ein normales Verzeichnis

Aus diesem “normalen” Verzeichnis wird ein Git-Projektverzeichnis und darin enthalten ist das Git-Repository
+ Unterverzeichnis “.git”

- Richtig wäre
  - Anlegen des Repositories im GitHub
  - git clone http://github.com/…

später = Morgen

Aktueller Stand

Git-Projektverzeichnis

- Workspace
  - besteht aus allen Dateien außerhalb von .git
- Git-Repository
  - das Verzeichnis .git

Ein Git-Benutzer arbeitet im Workspace, “ganz normal”
.git-Verzeichnis ist rein intern vom git-executable verwaltet
- es gibt keinen sinnvollen Arbeitsablauf, in denen ein Git-Benutzer irgendetwas in .git machen kann

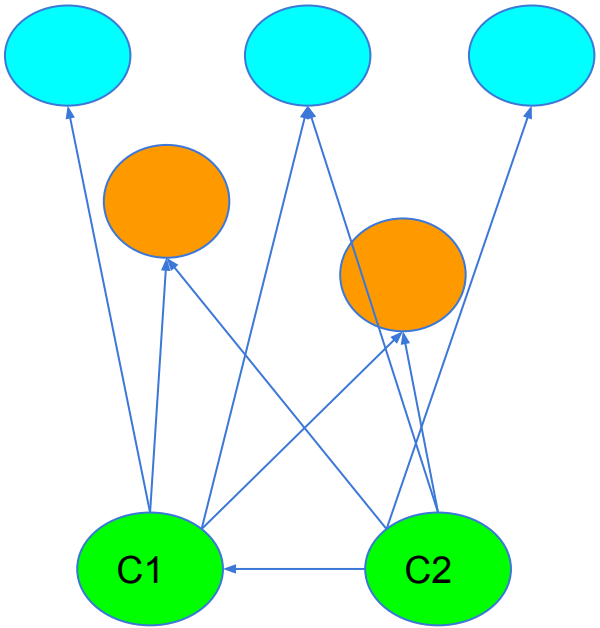
Verteilte Git-Repositories

- Dadurch, dass das Git-Repository ja “nur” ein Verzeichnis ist, kann dieses selbstverständlich kopiert und verteilt werden
- Problem
  - Wie kann verhindert werden, dass in Kopien eines Repositories Stände (Zustände von Werken sowie Meta-Informationen) manipuliert werden?
- Lösung
  - Blockchain-Technologie
  - Grundlage
    - Jede Information wird durch einen Hashwert definiert
    - Jede Änderung einer Information enthält den Hashwert des Vorgängers
- Trotz der Möglichkeit der Verteilung von Git-Repositories ist eine weltweit eindeutige Konsistenz garantiert.

Basis-Workflow mit Git

- Normales Arbeiten im Git Projektverzeichnis
- git status
  - Prüft, ob Unterschiede zwischen Workspace und Repository vorhanden sind
- git add <path>
  - z.B. git add . werden alle geänderten Informationen im Workspace in das Repository transferiert
- Hinweis: Ein add definiert keinen Stand, es ist eine vorbereitende Aktion
- git commit -m “Commit Message, Warum wurde ein neuer Stand definiert?”
  - Operiert auf den mit add hinzugefügten Informationen, NICHT auf dem Workspace

Ein Blick in das Git-Repository



BLOB = Content Object

Tree-Objekte
- \+ Dateinamen
- \+ Datei-Attribute

Commit Object
- \+ Commit Message
- \+ Committer
- \+ TimeStamp
- \+ Referenzen auf Trees und BLOBs
- \+ Vorgänger-Commit

Git arbeitet niemals mit Delta-Historien! Die Objekte sind immer vollständig

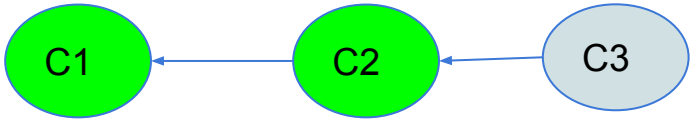
Ein pragmatischer Blick ins Repository

git log

git log --oneline --all --decorate --graph

git commit


```
git commit -m
```



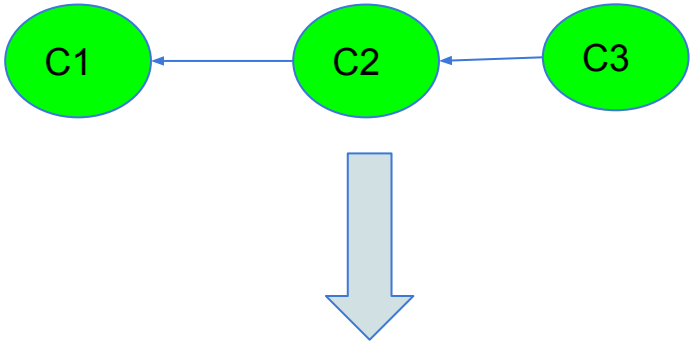
Best Practice

git status zur Überprüfung: Sind alle zu übernehmenden Änderungen bereits geadded = “grün”

Alles “rote” wird nicht übernommen

git checkout

git checkout C2



Best Practice

Bitte ein checkout nur bei unauffälligem Status!

- committen Sie Änderungen vor dem checkout
- stashen Sie Änderungen vor dem checkout
  - Ein Stash ist quasi ein Backup-Zip
  - Details: Git.PDF

Hinweis: Bitte Meldungen beim checkout / status mit dem Inhalt “detached HEAD” ignorieren, machen wir später

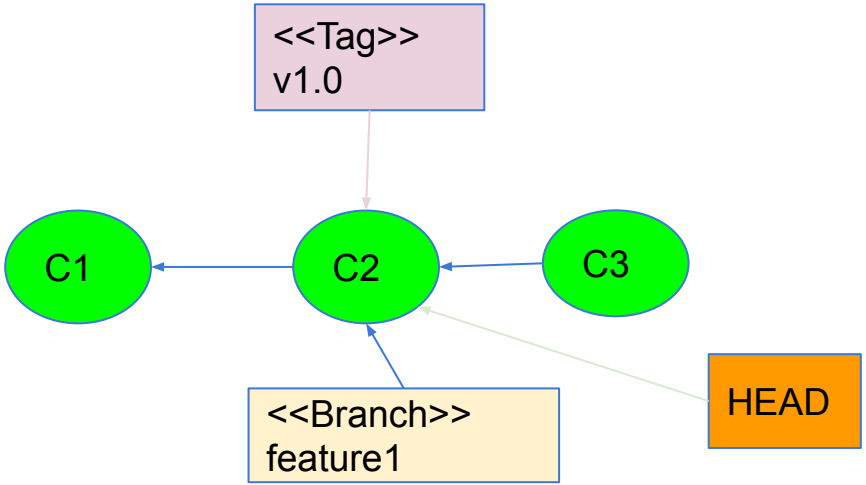
Wertung des bisherigen Arbeitsablaufs

- Für Git intern ist alles klar
- Der Git-Benutzer ist allerdings in der Situation, mit Hashwerten arbeiten zu müssen
  - “Nerd-Modus”
- Sinnvolle Erweiterung: Einführung von Alias-Namen auf Commit-Hashwerte

Alias-Namen auf Hashwerte

- WICHTIG: Alles, was im Folgenden vermittelt wird ist keine Analogie, keine Vereinfachung, sondern exakt so funktioniert Git
- Konzeptuell sind 2 Kategorien von Alias-Namen für Versionsverwaltung sinnvoll
  - Definition eines fixen, erreichten Standes
    - Release-Nummer
      - v1.4
    - Milestone oder Build-Number
    - “Heute Morgen um 8:30”
  - Agiler Stand, der eine laufende Entwicklung kennzeichnet
    - “implement webservice 42”
    - “124561” -> Jira-Issue
    - “ich probiere was aus”

Alias-Namen und Git




```
git checkout C2
```

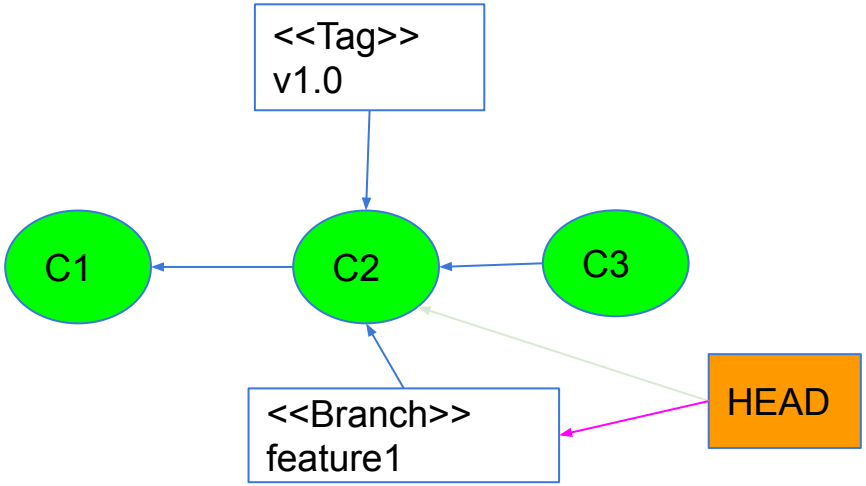
```
git tag v1.0
```

```
git branch feature1
```

WICHTIG:

Commit-Objekte haben keinen Bezug zu einem Aliasnamen!

git checkout: revisited



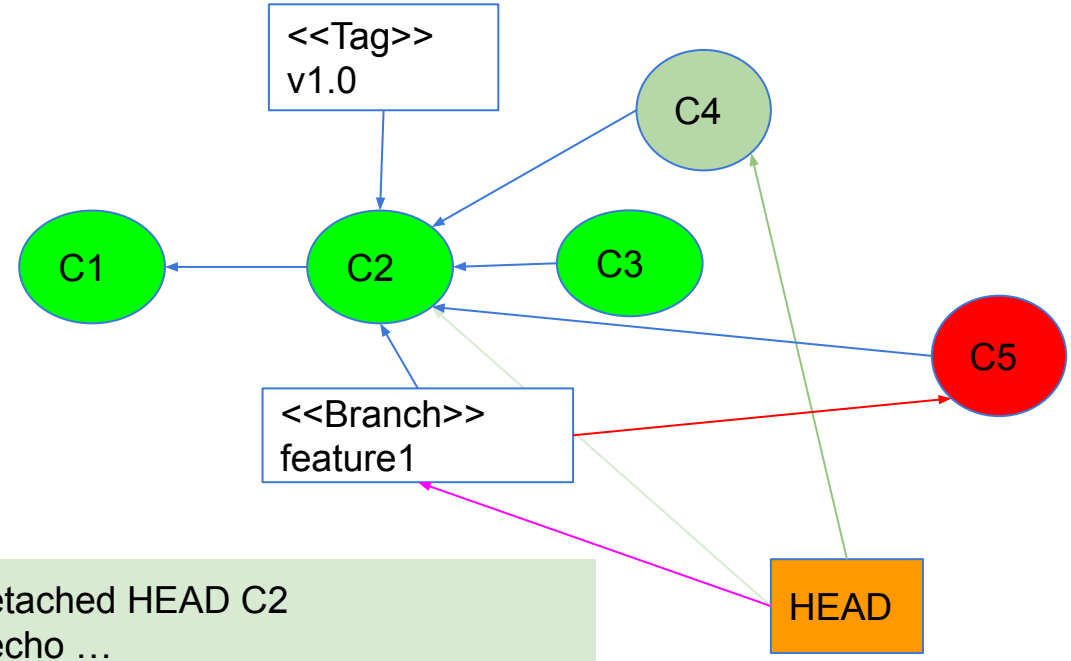
detached HEAD

attached HEAD, Standard

git checkout C2
git checkout v1.0

git checkout feature1

git commit: revisited



Ein commit im detached HEAD führt Sie in den Nerd-Modus

detached HEAD C2
//echo …
git add .
git commit -m “”

attached HEAD, Standard
/echo…
git add .
git commit -m “”

Zusammenführen von Ständen

- Merge
  - Fast Forward
  - Recursive Merge
- Rebase
- Cherry Pick
  - In der Git-Community wird die Verwendung von Cherry Pick nicht mehr als notwendig erachtet

Merge Plan

- Übersicht über die Commit-Hierarchie

```
$ git log --oneline --all --decorate --graph
* f407ae9 (feature2_part1) change content-feature2, part1
| * b5f036f (feature2_part2) change content-feature2, part2
|/
* 89e3aeb (HEAD -> feature2) add content-feature2
| * c8a1e09 (feature1) add content-feature1
|/
* 907e612 (master) change content
* ffcec38 add content
* ce66ef5 setup project
```

- Ziel: Alle Änderungen aller Feature-Branches sollen in den master übernommen werden *unter Beibehaltung der ursprünglichen Historie*
- Reihenfolge: in feature2 die beiden parts (erst 1, dann 2), dann feature1 und zum Schluss Übernahme in den master

Umsetzung

- git checkout feature2
- git merge feature2_part1
  - Git erkennt, dass die beiden Branches in einer direkten Linie verbunden sind und führt einen fast-forwardmerge aus
  - Bei einem Fast Forward-Szenario sind Merge-Konflikte unmöglich

```
Fast-forward
 content-feature2.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

Rainer Sawitzki@LAPTOP-GVSFDDCT MINGW64 ~/git_training/training_branches
 (feature2)
$ git log --oneline --all --decorate --graph
* f407ae9 (HEAD -> feature2, feature2_part1) change content-feature2, pa
rt1
| * b5f036f (feature2_part2) change content-feature2, part2
|/
* 89e3aeb add content-feature2
| * c8a1e09 (feature1) add content-feature1
|/
* 907e612 (master) change content
* ffcec38 add content
* ce66ef5 setup project
```

Umsetzung

- git merge feature2_part2
  - Git erkennt, dass die beiden Branches einen gemeinsamen Vorgänger haben, damit muss ein recursive merge ausgeführt werden
  - Bei einem Recursive Merge sind Merge-Konflikte immer möglich

```
$ git merge feature2_part2
Auto-merging content-feature2.txt
CONFLICT (content): Merge conflict in content-feature2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch feature2
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   content-feature2.txt
```

Best Practice

Machen Sie nichts anderes als den merge zu beenden oder abzubrechen!

+ git merge --abort (?)

Umsetzung

- Status: In einem Merge Conflict
  - Händisch muss der Conflict bereinigt werden

```
<<<<<<< HEAD
feature2, part1
=======
feature2, part2
>>>>>>> feature2_part2
```

```
feature2, part1 and part2
```

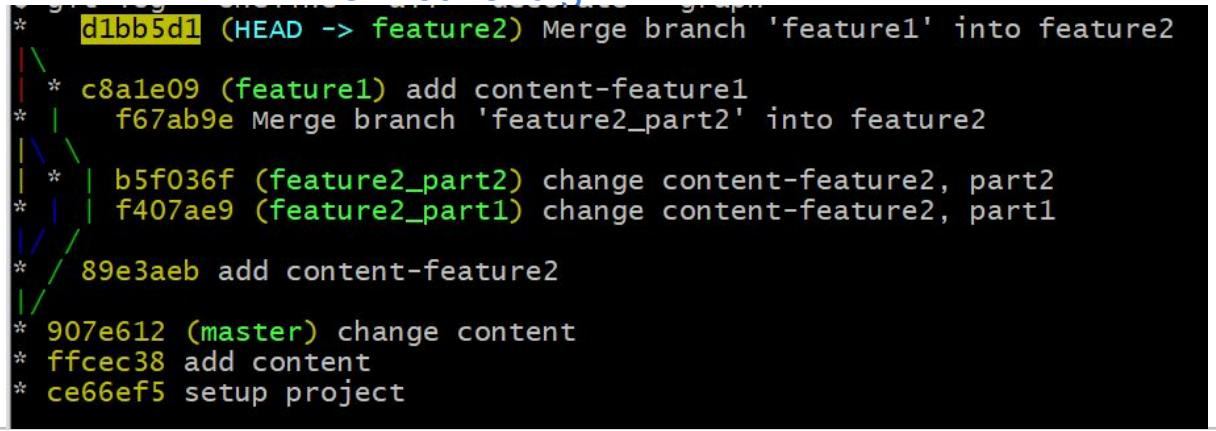
- git add .
- git commit

```
*   f67ab9e (HEAD -> feature2) Merge branch 'feature2_part2' into featur
e2
|\
| * b5f036f (feature2_part2) change content-feature2, part2
* | f407ae9 (feature2_part1) change content-feature2, part1
|/
* 89e3aeb add content-feature2
| * c8a1e09 (feature1) add content-feature1
|/
* 907e612 (master) change content
* ffcec38 add content
* ce66ef5 setup project
```

Commit mit 2 Vorgängern

Umsetzung

- git merge feature1
  - Recursive Merge
  - Merge Konflikte werden hier von Git automatisch gelöst
    - Auto Conflict Resolution: Änderungen in unterschiedlichen Dateien werden automatisch zusammengeführt
    - Vorsicht: Ein Prüfen der Konsistenz des Projektes ist nach einem Merge immer notwendig



Umsetzung

- git checkout master
- git merge feature2
  - CHECK: Das muss ein Fast Forward sein

```
*   d1bb5d1 (HEAD -> master, feature2) Merge branch 'feature1' into feat
ure2
|\
| * c8a1e09 (feature1) add content-feature1
* |   f67ab9e Merge branch 'feature2_part2' into feature2
|\ \
| * | b5f036f (feature2_part2) change content-feature2, part2
* | | f407ae9 (feature2_part1) change content-feature2, part1
|/ /
* / 89e3aeb add content-feature2
|/
* 907e612 change content
* ffcec38 add content
* ce66ef5 setup project
```

Optional: Housekeeping

- Die feature-Branches werden nicht mehr benötigt
- Tagging der Stände
  - v1.0 der ursprüngliche master
  - v1.1-Milestone1

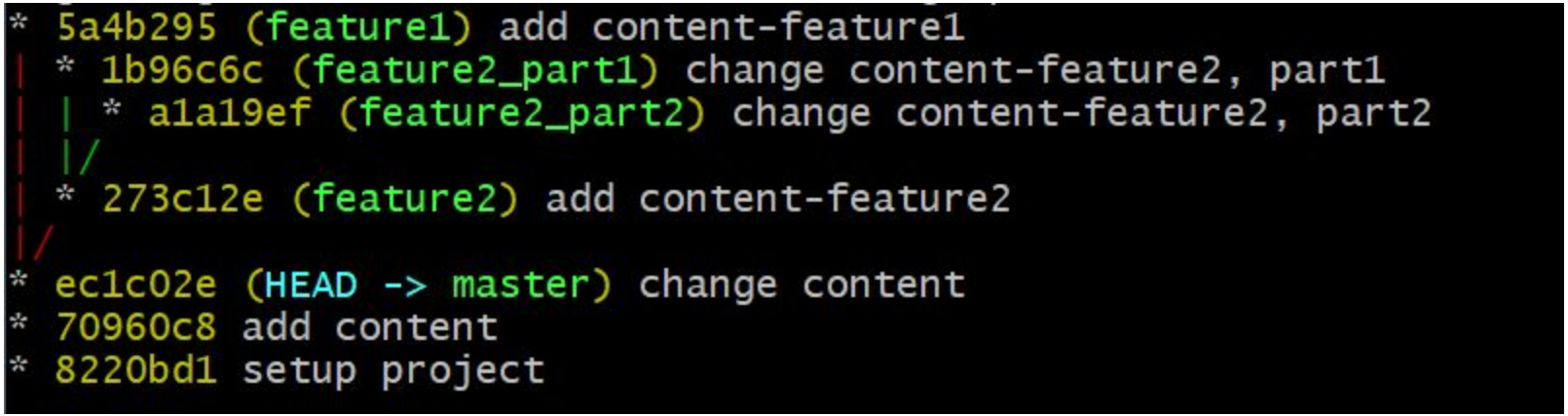
```
*   d1bb5d1 (HEAD -> master, tag: v1.1-Milestone-1) Merge branch 'featur
e1' into feature2
|\
| * c8a1e09 add content-feature1
* |   f67ab9e Merge branch 'feature2_part2' into feature2
|\ \
| * | b5f036f change content-feature2, part2
* | | f407ae9 change content-feature2, part1
|/ /
* / 89e3aeb add content-feature2
|/
* 907e612 (tag: v1.0) change content
* ffcec38 add content
* ce66ef5 setup project
```

Mit --no-ff


```
*   56296ab (HEAD -> master) Merge branch 'feature2'
|\
| *   fd87c5b (feature2) Merge branch 'feature1' into feature2
| |\
| | * d6658db (feature1) add content-feature1
| |/
|/|
| *   4bb75f1 Merge branch 'feature2_part2' into feature2
| |\
| | * f44e343 (feature2_part2) change content-feature2, part2
| * |   819a049 Merge branch 'feature2_part1' into feature2
| |\ \
| | |/
| |/|
| | * 8518807 (feature2_part1) change content-feature2, part1
| |/
| * 21768f1 add content-feature2
|/
* 9989297 change content
* d0acc8e add content
* 1aa517a setup project
```

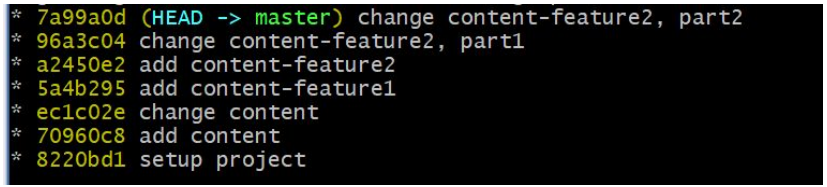
Rebase

- Plan: Konsolidieren mit Erstellung einer stringenten Dokumentation



Schritt für Schritt

- Beim Rebasing wird ein Stand “auf einen anderen nachgespielt”
- git checkout feature2
- git merge feature2_part1
  - Fast Forward
- git checkout feature2_part2
- git rebase feature2
  - Konflikt wie auch beim mergen
  - Lösen + git rebase --continue
- git rebase feature1
- git checkout master
- git merge master feature2_part2



Rebase oder Merge?

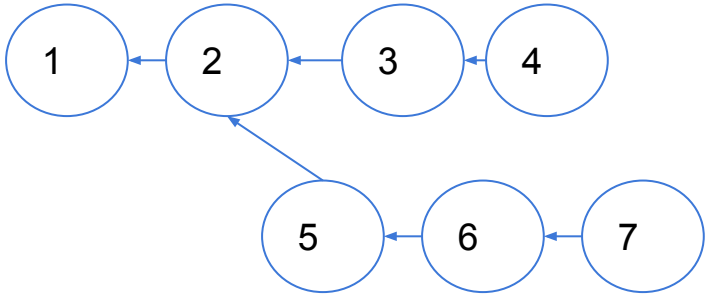
- Entscheiden Sie!
  - Beides ist möglich und kann auch in einem Repository gemacht werden
- Mischbetrieb auch sinnvoll
- Ein interaktives Rebasing ermöglicht Ihnen eine weitere Verbesserung der Dokumentation der Commit-Historie

```
* 06c6325 (HEAD -> master) add content-feature2
| * 7a99a0d (tag: savepoint) change content-feature2, part2
| * 96a3c04 change content-feature2, part1
| * a2450e2 add content-feature2
|/
* 5a4b295 add content-feature1
* ec1c02e change content
* 70960c8 add content
* 8220bd1 setup project
```

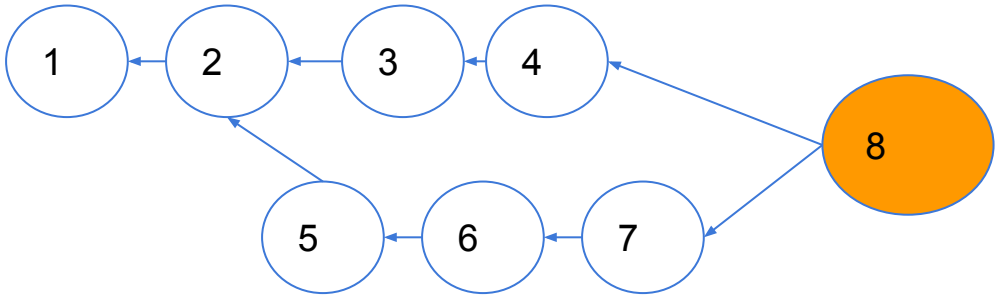
Dies und das

- Mergen erstellt die Historie “so wie sie ist”
  - Jede Parallelentwicklung, jedes fein-granulare Commit-Objekt ist enthalten
- Rebasen
  - Erstellt eine Dokumentation der Historie

Bildhafte Darstellung



Bildhafte Darstellung: Merge



Der Workspace enthält während des Mergens alle Konflikte in Dateien mit Konflikt Markern

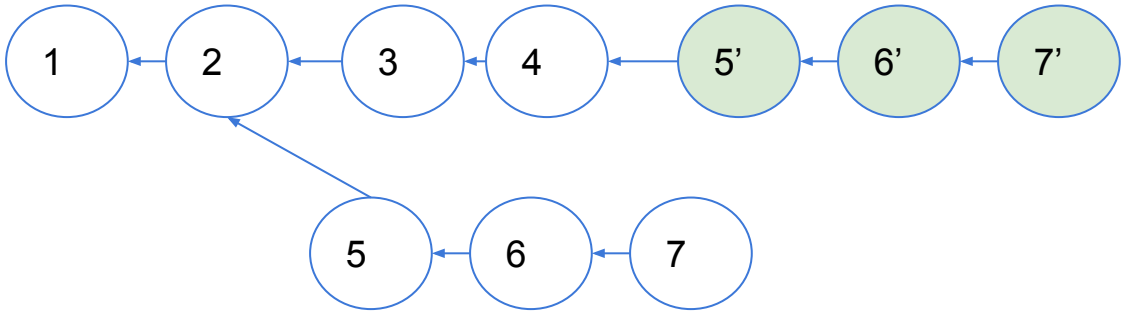
Abschluss

```
git add .
git commit
```

Abbruch

```
git merge --abort
```

Bildhafte Darstellung Rebasing



Rebase 7 on top 4

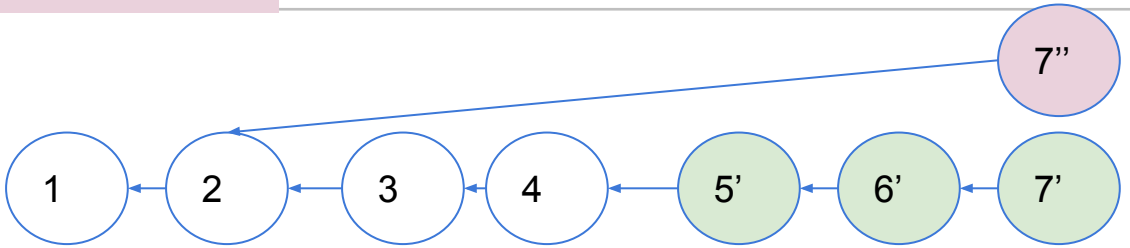
Gemeinsame Vorgänger: 2

Änderungen “5 auf 2” werden nun neu eingespielt auf 4

Bei jedem Schritt enthält der Workspace die Konflikte, die dem ursprünglichen commit entsprechen
-> Konflikte kommen “tröpfchenweise”

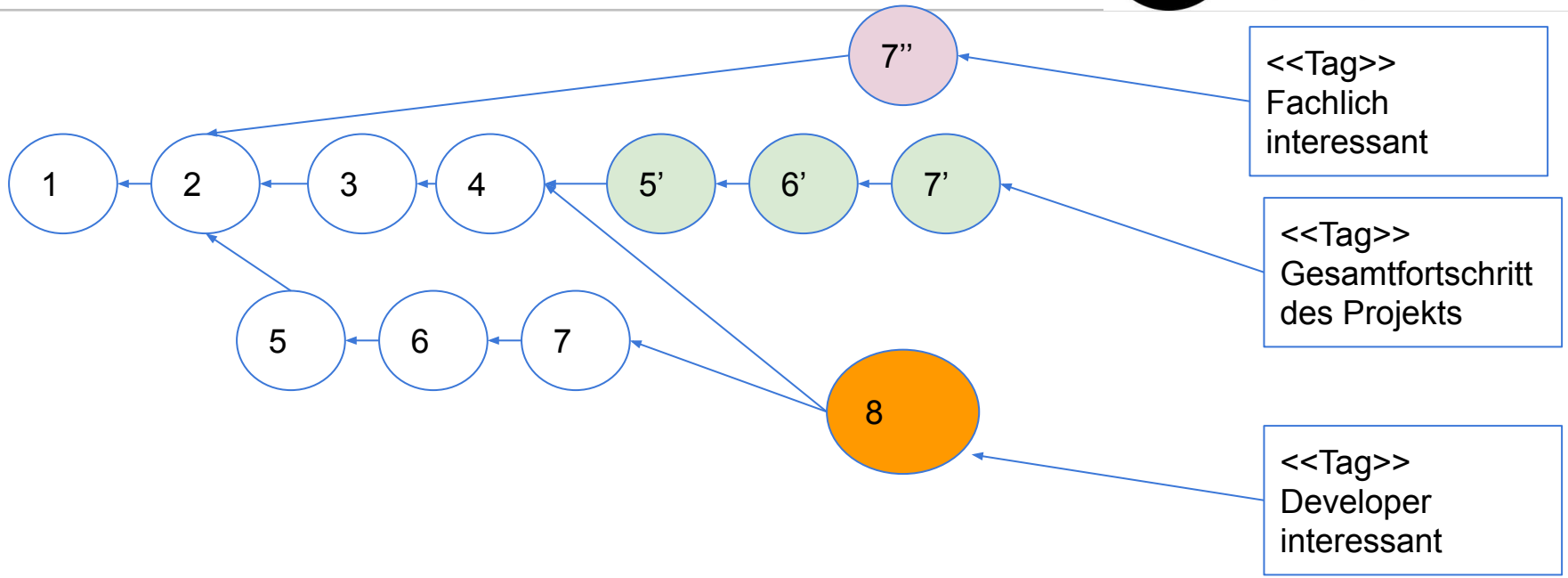
--continue

Bildhafte Darstellung Rebasing interactive



Rebase -i 7' on top 2

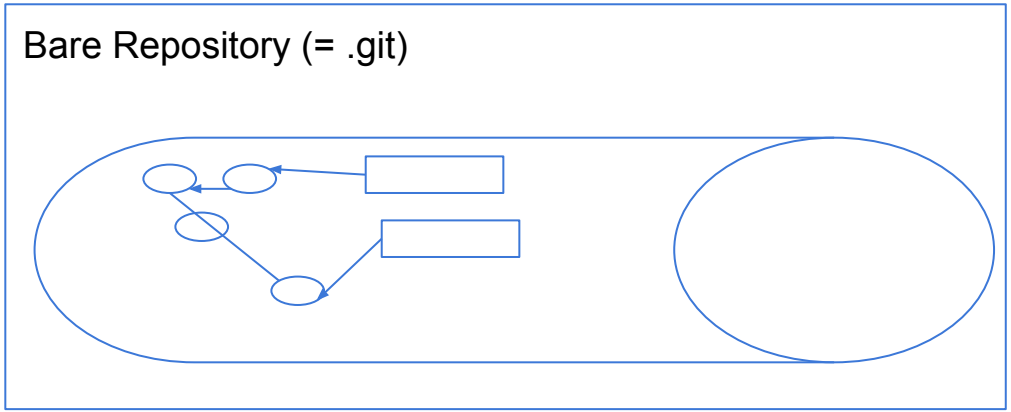
Bildhafte Darstellung



GitHub

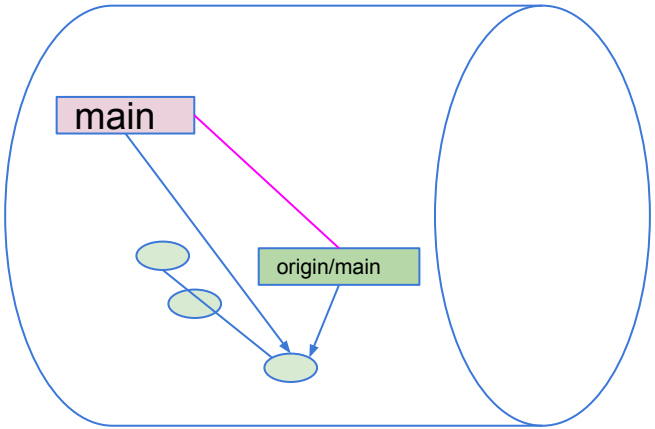
Anlegen des Projekts

- Über das Web Frontend
  - Mein Projekt: Public-Projekt, jeder GitHub-Anwender hat Zugriff auf dieses Projekt
    - Private Projekte, in denen ein Team definiert werden muss
-

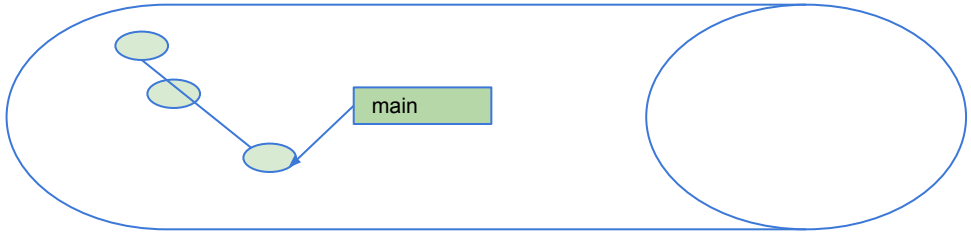


git clone

training_27.4.2023: Git Project Directory



Bare Repository (= .git)



main und origin/main sind konfigurativ verbunden

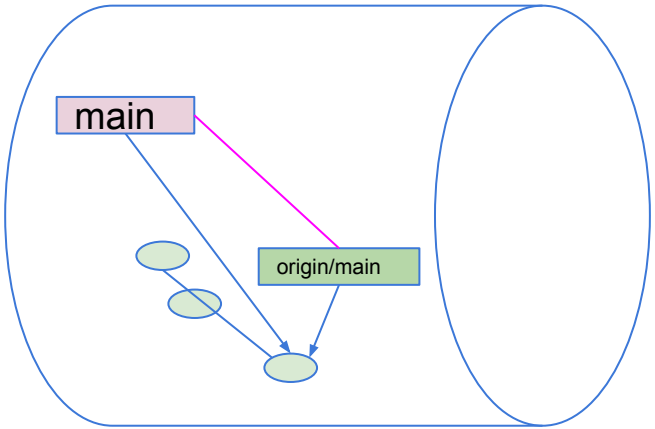
origin/main ist ein so genannter “Remote Branch”

Ein Remote Branch spiegelt exakt den Stand wieder, der auf Server-seite beim Zeitpunkt des Clones vorhanden war

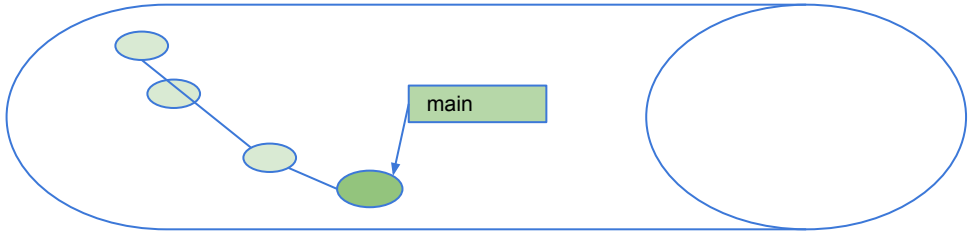
+ Remote Branches sind “Read Only”

Auf dem Server neues Commit

training_27.4.2023: Git Project Directory

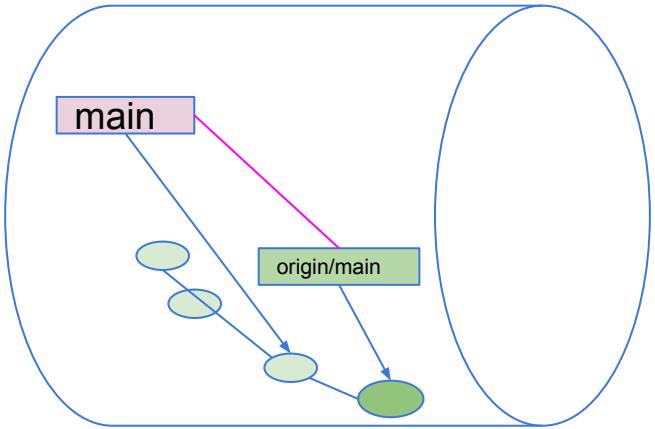


Bare Repository (= .git)

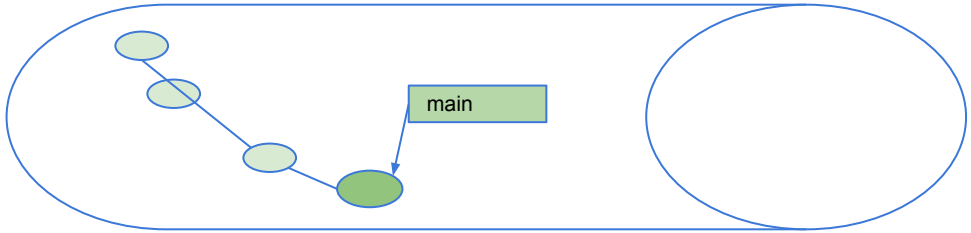


git fetch

training_27.4.2023: Git Project Directory



Bare Repository (= .git)



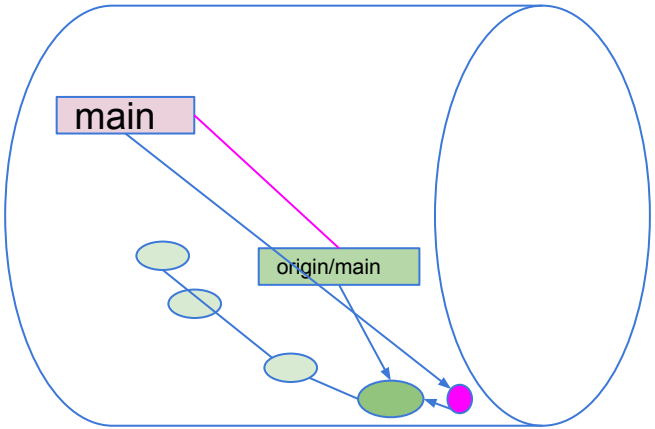
fetch bewegt einen Remote Branch so, dass er dem Server-Zustand zum Zeitpunkt des fetch entspricht

git pull

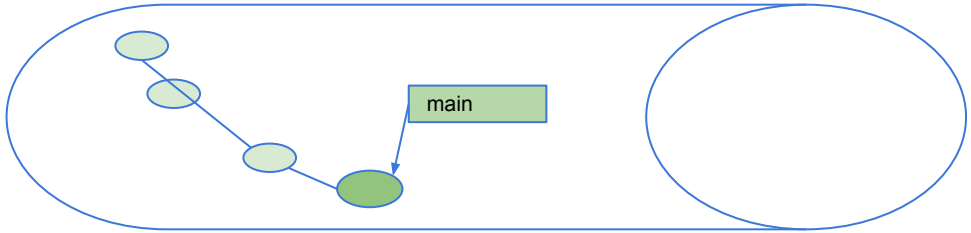
- git pull = git fetch + git merge origin/main
  - Potenziell können hier merge-Konflikte auftreten!
    - fetch ist immer garantiert ein fast forward!
- git pull --rebase = git fetch + git rebase main ontop origin/main
  - in den meisten Fällen das richtige -> später

Lokales Arbeiten im main

training_27.4.2023: Git Project Directory



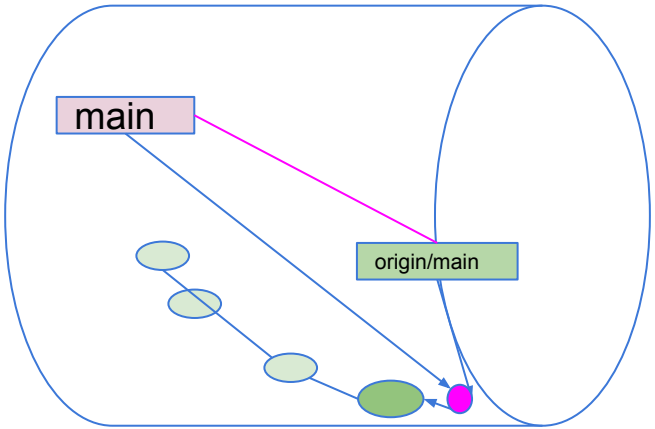
Bare Repository (= .git)



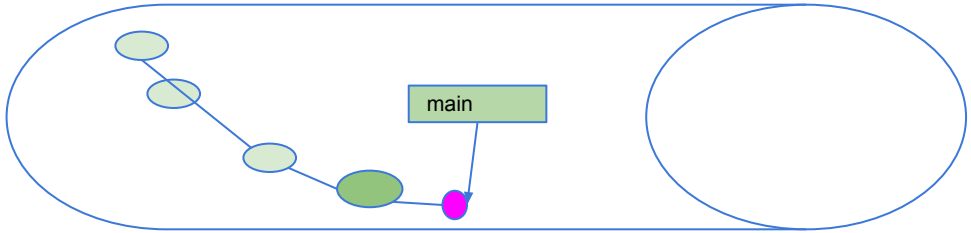
fetch bewegt einen Remote Branch so, dass er dem Server-Zustand zum Zeitpunkt des fetch entspricht

git push

training_27.4.2023: Git Project Directory



Bare Repository (= .git)



1. Übertragung der neuen Commits zum Server
2. Check: Ist ein Fast Forward möglich?
   a. Nein -> push wird abgebrochen
   b. Ja
      i. FF wird ausgeführt im Server
      ii. origin/main wird vorgezogen

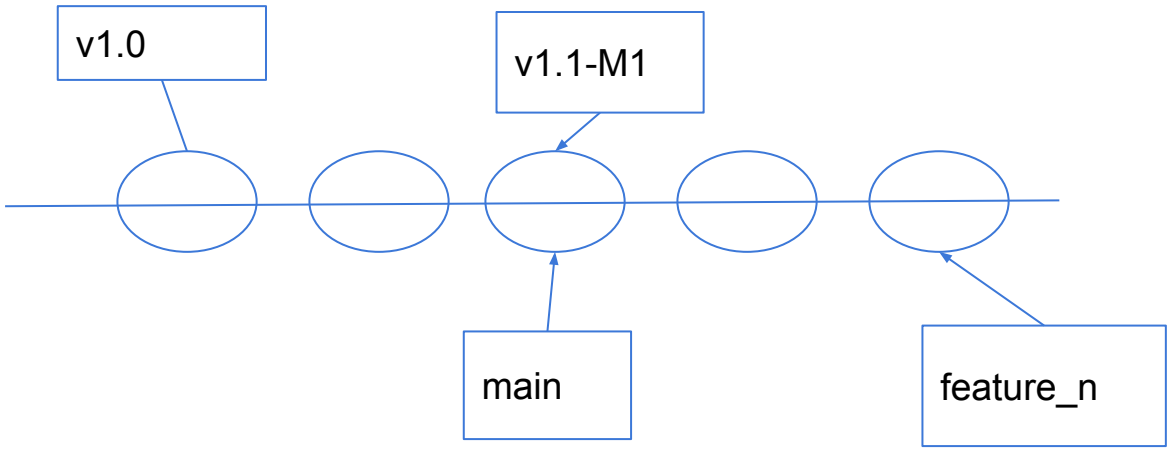
Git Flows

Allgemeine Regeln

- Branches sind laufende Aktionen
  - git branch --list
- Tags sind fixe Stände
  - git tag --list
- Es gibt (mindestens) einen lang-lebigen Branch, der einem laufenden Software-Projekt entspricht
  - main
- Jegliche Änderung / Erweiterung / BugFix wird ausschließlich in einem eigens dafür eingerichteten “Feature Branch” erledigt
  - Namenskonvention
    - feature/<name>
    - fix/<name>

Single Developer Flow

features



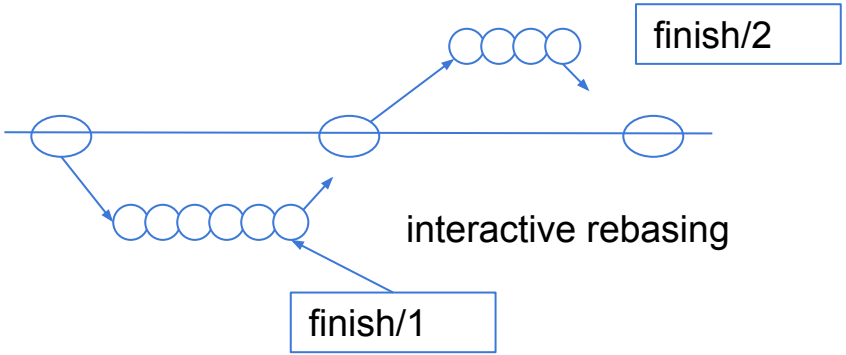
Überlegungen

+ Vor dem mergen = fast forward bietet sich ein interactiv rebasing des feature-Branches an
+ Fertigstellung eines Features wird durch ein Tag definiert

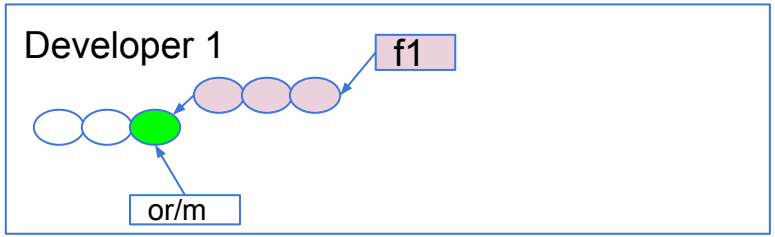
Todo

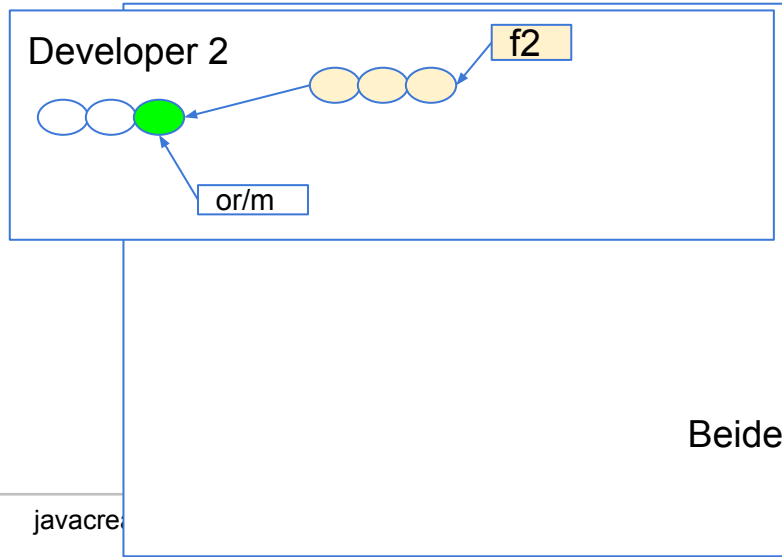
- Anlegen eines Repositories in GitHub
- Clonen
- Entwicklung
  - Editieren
  - Save
  - add
  - commit
  - push
- Gerne genutzt: “Commit & Push”
- Feature Fertig
  - feature-Branch taggen feature/feature1 -> finish/feature1
  - interactive rebase
  - main merge (fast forward)
  - main tag setzen
  - feature-Branch löschen

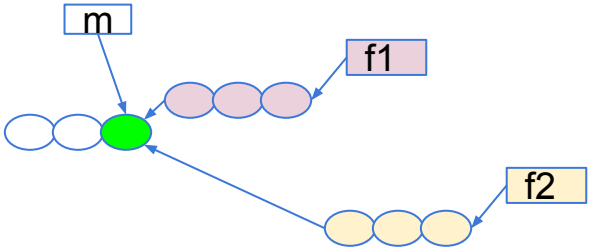
So soll es ausschauen



Team Flow: 1 Developer / Feature

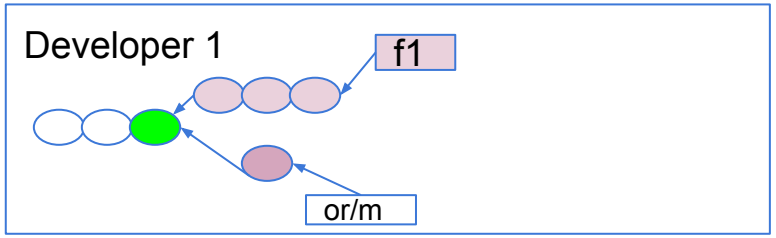


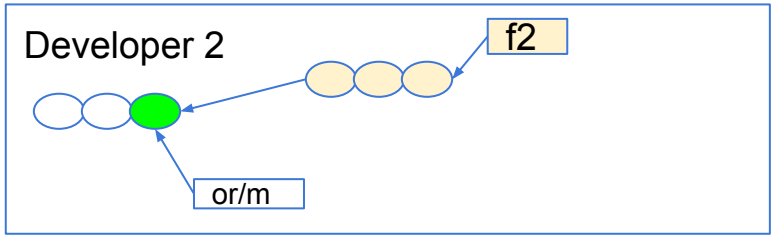


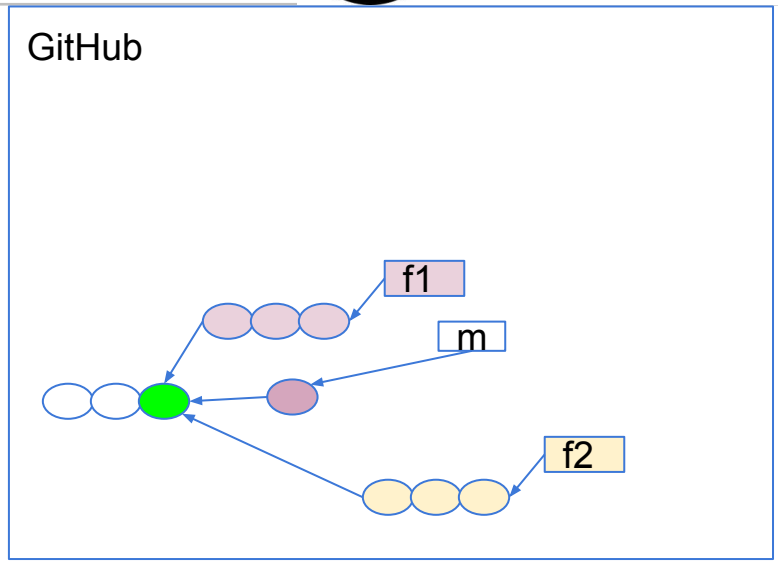


Beide Developer arbeiten aktiv

Team Flow: 1 Developer / Feature



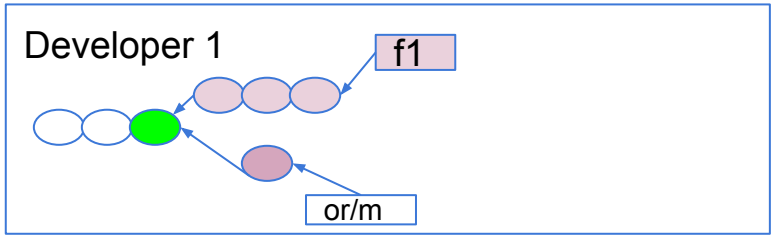


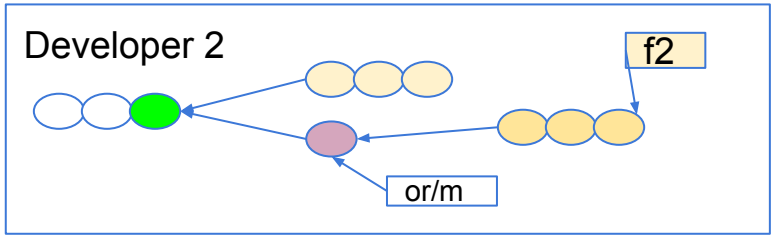


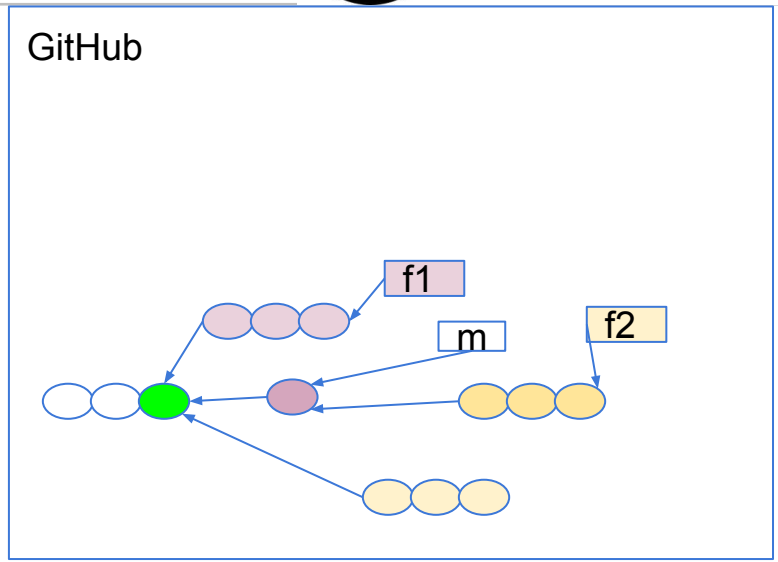
Developer 1: Bin fertig!

Als Developer 1
- CHECK: or/m = server main
- interactive rebase
- fast forward des main
- push
- KOMMUNIKATION: Es wurde im main gepushed!

Team Flow: 1 Developer / Feature





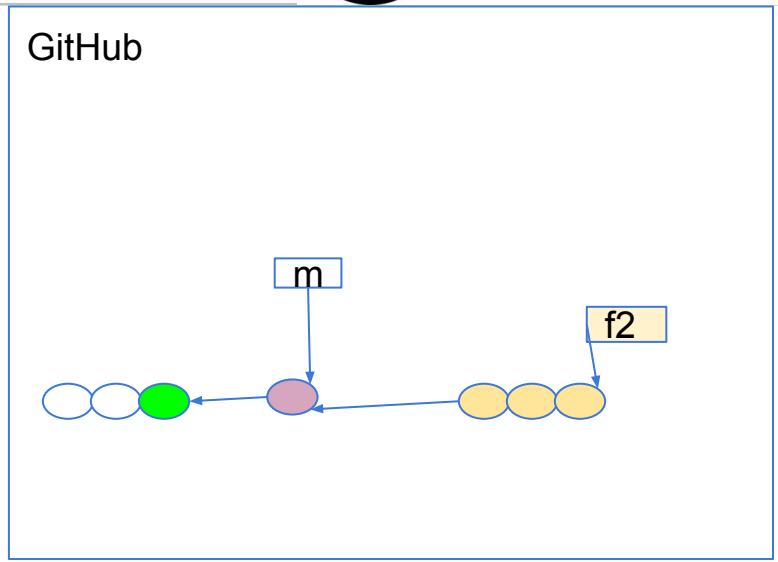


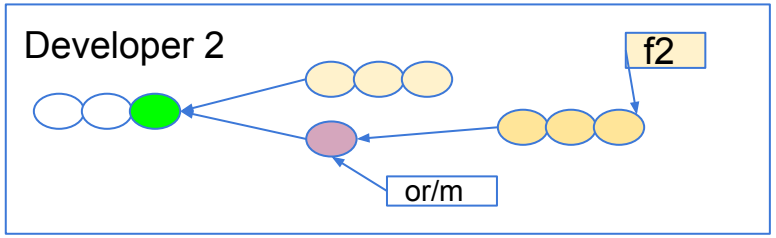
Developer 1: Bin fertig!

Als Developer 2

git pull des main (CHECK: FF!)
rebasing (Conflict Resolution)
git push -f f2

Team Flow: 1 Developer / Feature





Cleanup

- \+ Nicht mehr notwendige Elemente auf dem Server entfernen
- \+ Lokales Developer 1 Repository löschen

Developer 1: Bin fertig!

Ein paar Details

- Die Variante mit main + feature-Branches + Tags auf dem main hat den Namen “GitHub Flow”
  - Propagiert von der GitHub-Community, aber kein GitHub-Feature
- Atlassian Git Flow
  - 2 langlebige Flows
    - main
    - develop
      - Zwischenpuffer, der zumindest zeitweise instabile / fehlerhafte Commits enthalten darf
- Recherche: Wie gut ist Git Flow / GitHub Flow?
  - Ergebnis: “Alles Mist, unbrauchbar”
  - Als Grundlage sind beide Flows geeignet!

Ein paar Details…

- Team Flow, Features müssen von mehreren Team-Mitgliedern bearbeitet werden
  - Variante 1
    - Zerlegung in Feature-Parts, die dann wiederum von jeweils einem Team-Mitglied bearbeitet werden können
  - Variante 2
    - Alle arbeiten am feature-Branch
    - Push ist erfolgreich, wenn “mein” Stand der aktuelle ist
    - falls ein push nicht erfolgreich ist: “jemand anderes war schneller”
      - git pull zieht die Änderungen vom Server -> Problem: “Mein Workspace liegt in Trümmern…”
      - git fetch -> Evaluierung: Was hat sich geändert? -> git rebase feature auf origin/feature -> git push
        - “Optimistic Locking”

Pull Requests

- Feature eines Git Servers
  - Pull Request bei GitHub und BitBucket
  - Merge Request GitLab
- Auslöser für einen Pull-Request ist
  - Developer X: “Ich bin fertig mit einem Feature”
    - Nun wird ein Pull-Request erzeugt