JAVACREAM

Training
Consulting
Projectmanagement

GIT

Kurze Vorstellung

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

Einführung

Allgemein: Versionsverwaltung

- Quellcode = “Werke” werden vom Versionsverwaltungssystem mit Meta-Informationen ergänzt
  - Wer hat wann warum welche Änderungen gemacht?
- Der Stand eines Projekts wird durch ein Aggregat von Werken definiert
- Zentrale Ablage aller relevanten Informationen
  - inklusive Authentifizierung und Autorisierung
- Werkzeugen und Verfahren für die Team-Zusammenzusammenarbeit
- Werkzeuge für die Visualisierung und Konsolidierung der Arbeit

Git: Versionsverwaltung

- Quellcode = “Werke” werden vom Versionsverwaltungssystem mit Meta-Informationen ergänzt
  - Wer hat wann warum welche Änderungen gemacht?
- Der Stand eines Projekts wird durch ein Aggregat von Werken definiert
- Zentrale Ablage aller relevanten Informationen
  - inklusive Authentifizierung und Autorisierung
- Werkzeugen und Verfahren für die Team-Zusammenzusammenarbeit
  - Konzepte und Technik
- Werkzeuge für die Visualisierung und Konsolidierung der Arbeit
  - Konsolenbasiert

Git und Git Server-Lösungen: Versionsverwaltung

- Quellcode = “Werke” werden vom Versionsverwaltungssystem mit Meta-Informationen ergänzt
  - Wer hat wann warum welche Änderungen gemacht?
- Der Stand eines Projekts wird durch ein Aggregat von Werken definiert
- Zentrale Ablage aller relevanten Informationen
  - inklusive Authentifizierung und Autorisierung
  - Produkt-Lösungen, z.B. **GitHub** (Microsoft), GitLab (GitLab.com), BitBucket (Atlassian)
- Werkzeugen und Verfahren für die Team-Zusammenzusammenarbeit
  - Konzepte und Technik
  - **Pull** / Merge **Requests**
- Werkzeuge für die Visualisierung und Konsolidierung der Arbeit
  - Konsolenbasiert
  - Web Frontend

Git, Git Server-Lösungen und Client-Werkzeuge: Versionsverwaltung

- Quellcode = “Werke” werden vom Versionsverwaltungssystem mit Meta-Informationen ergänzt
  - Wer hat wann warum welche Änderungen gemacht?
- Der Stand eines Projekts wird durch ein Aggregat von Werken definiert
- Zentrale Ablage aller relevanten Informationen
  - inklusive Authentifizierung und Autorisierung
  - Produkt-Lösungen, z.B. **GitHub** (Microsoft), GitLab (GitLab.com), BitBucket (Atlassian)
- Werkzeugen und Verfahren für die Team-Zusammenzusammenarbeit
  - Konzepte und Technik
  - **Pull** / Merge **Requests**
- Werkzeuge für die Visualisierung und Konsolidierung der Arbeit
  - Konsolenbasiert
  - Web Frontend
  - Entwicklungsumgebungen (IntelliJ, Visual Studio Code) native-Git-Installationen mit Turtoise, …

Git, Git Server-Lösungen und Client-Werkzeuge: Versionsverwaltung

- Quellcode = “Werke” werden vom Versionsverwaltungssystem mit Meta-Informationen ergänzt
  - Wer hat wann warum welche Änderungen gemacht?
- Der Stand eines Projekts wird durch ein Aggregat von Werken definiert
- Zentrale Ablage aller relevanten Informationen
  - inklusive Authentifizierung und Autorisierung
  - Produkt-Lösungen, z.B. **GitHub** (Microsoft), GitLab (GitLab.com), BitBucket (Atlassian)
- Werkzeugen und Verfahren für die Team-Zusammenzusammenarbeit
  - Konzepte und Technik
  - **Pull** / Merge **Requests**
- Werkzeuge für die Visualisierung und Konsolidierung der Arbeit
  - Konsolenbasiert
  - Web Frontend
  - Entwicklungsumgebungen (IntelliJ, Visual Studio Code) native-Git-Installationen mit Turtoise, …

Seminar Tag 1 +x

Seminar Tag 2

First Contact

Installation

- Installiert wird das Executable “git”
  - Kein Client, der mit einem Server kommuniziert, sondern das komplette Core-Versionsverwaltungssystem
  - Kein Hintergrund-Dienst, Service, Daemon
    - Das Git-Executable stellt während der Kommando-Ausführung die Funktionalität eines Versionsverwaltungssystems zur Verfügung

Vorbereitung des Arbeitens mit Git

- ~~Server-Administrator legt einen Account für Sie an~~
- ~~Konfiguration: Server-URL~~

Git Core braucht keine Server

- Erstellung einer Git-Konfigurationsdatei (eine einfache Text-Datei namens .gitconfig in Ihrem User-Home)
  - Minimal:
    - user.name
    - user.email
  - git config --global user.name “Rainer Sawitzki”
  - git config --global user.email training@rainer-sawitzki.de
  -

Ein erstes Git Repository

- Ein Repository repräsentiert je nach Ihrer Projekt-Organisation
  - Ein komplettes Software-Projekt
  - ein Modul eines Software-Projektes
  - Eine Gruppe von Software-Projekten
- Auf der Ebene einer Entwickler-Maschine ist ein Git-Repository Bestandteil eines normalen Directories
- Anlegen im Seminar erst einmal komplett untypisch durch Initialisierung eines leeren Repositories
  - In der Realität: Clone eines Server-Repositories

Schritte

- mkdir first
  - Normales Arbeitsverzeichnis auf einer lokalen Maschine
- cd first
- git init
  - Legt das Git-Repository im Unterverzeichnis .git an
  - Das “normale Arbeitsverzeichnis” ist nun ein Git-Projekt-Verzeichnis
    - Alles andere als .git: “Git Workspace”
- Check
  - git status

Exkurs: Verteilte Datenverwaltung

- Problemstellung
  - Wie kann Information in einer Art und Weise verteilt werden, dass jede Kopie garantiert Informationen konsistent hält?
  - Konkrete mit den Begriffen der Versionsverwaltung
    - Wie kann garantiert werden, dass ein historischer Stand eines Software-Projektes in allen Kopien des Repositories garantiert unveränderbar (inklusive Historie) ist?
- Etablierte Lösung
  - Blockchain-Technologie
    - Basiert auf den Merkle-Trees
      - Jeder Information wird der Hashwert der Vorgänger-Information hinzugefügt und daraus ein Hashwert berechnet
    - Kollisionen von Hashwerten sind prinzipiell möglich, aber absurd unwahrscheinlich

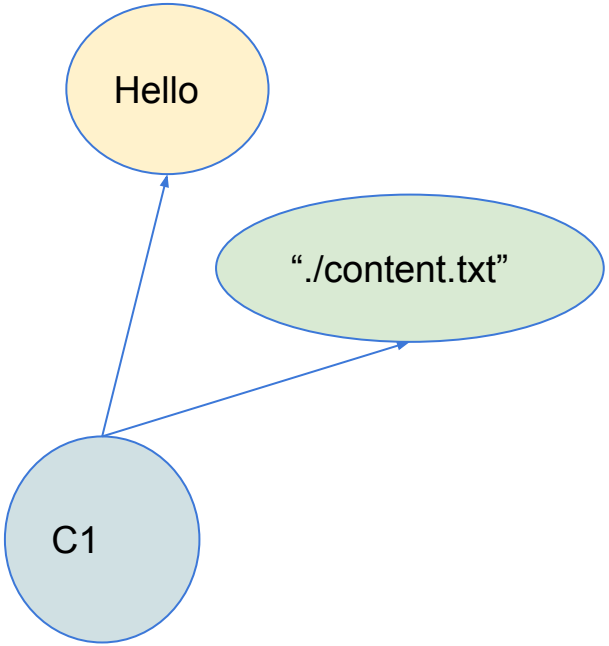
Git und Blockchain

- Git hat schon immer Blockchain-Technologie benutzt

Arbeitsweise von Git

- Alle Informationen werden in Git über berechnete Hashwerte identifizierbar gemacht
  - echo Hello > content.txt
  - git status
    - “Rote Datei”
  - git add content.txt
    - Im objects-Verzeichnis eine Datei e9/650…
  - git status
    - “Grüne Datei”
  - git commit -m “Commit Message”
  - git status
  - git log
    - Hashwert des Commits

Visualisierung (analog zum Speicher-Layout einer OOP-Sprache)



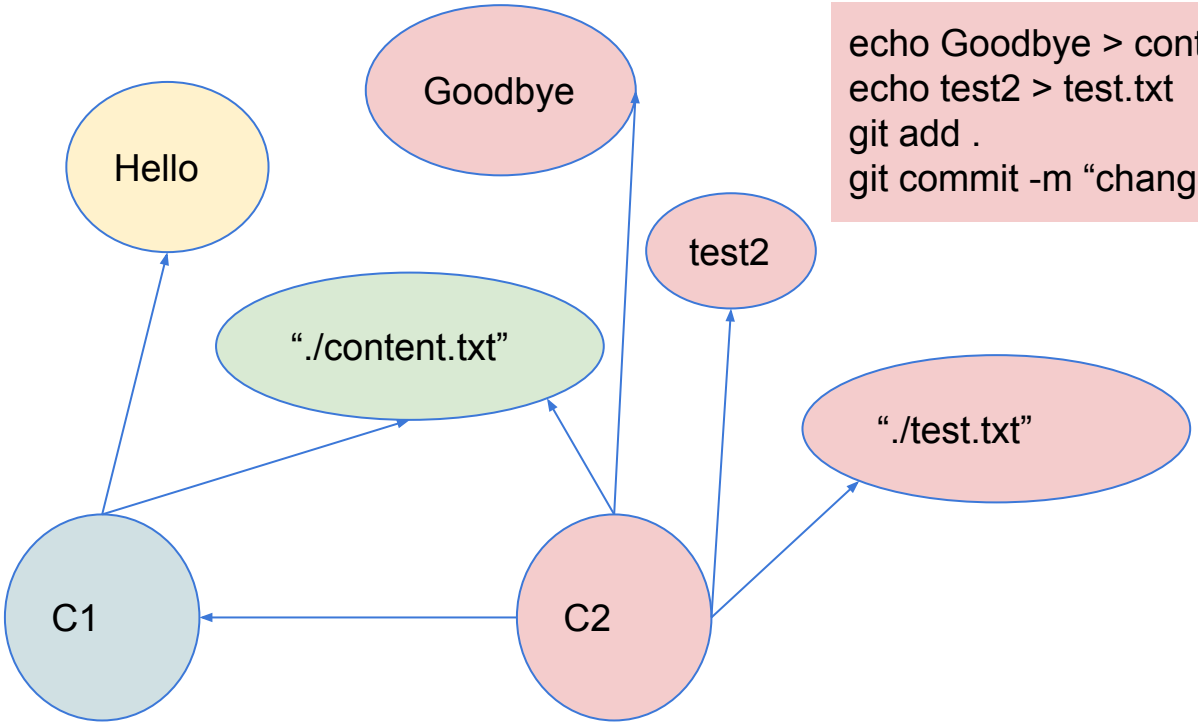
Content-Object
BLOB-Object

Tree-Object

Commit-Object

Anzeige: git log

Ein weiterer Commit




```
echo Goodbye > content.txt
echo test2 > test.txt
git add .
git commit -m "change"
```

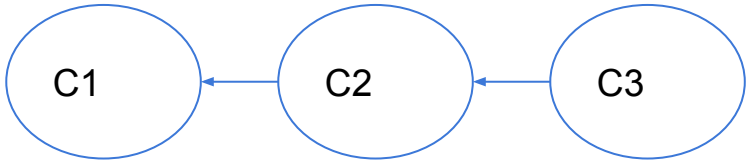
Content-Object
BLOB-Object

Tree-Object

Commit-Object

Anzeige: git log

Vereinfachte Visualisierung des Git Repository



Was macht der commit-Befehl?

- Ein commit erzeugt ein neues Commit-Objekt, das das Ausgangs-Commit-Objekt als Vorgänger enthält

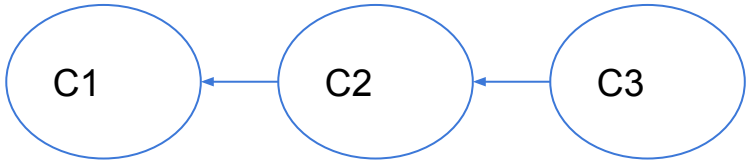
Ein paar “Details”

- git log
  - Ausgabe der Commits in einer Langform
- git log --oneline --graph --decorate --all
  - git config --global alias.pl “log --oneline --graph --decorate --all”
- git fsck --unreachable --no-reflogs
- Optionales Aufräumen (NICHT BESTANDTEIL DER NORMALEN ARBEIT)
  - git reflog expire --expire-unreachable=now
  - git gc --prune=now

Synchronisation des Workspaces mit einem Stand = Commit-Objekt

- git checkout <hash>
  - Empfehlung Sawitzki
    - “checkout nur bei unauffälligem Status”
      - “Nichts rotes, nichts grünes”
    - Falls Nein:
      - git add .
        - git commit -m “...”
        - git stash
          - Stashes sind nicht auf einen Server übertragbar
          - Verweis auf die git.pdf bzw. Online-Doku

git checkout <HASH>

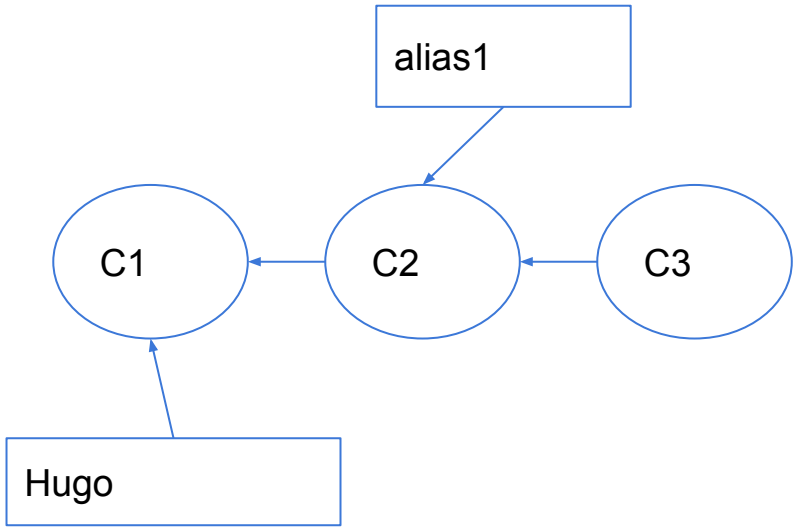


So arbeitet git intern immer

Ein Git-Anwender kann auch mit diesen Hash-Werten arbeiten -> “Nerd-Modus”

Alias-Namen auf Hash-Werte

- Statt langer Hashwerte können sprechende Alias-Namen benutzt werden



git checkout <alias>

Sinnvolle Kategorien eines Alias-Namens

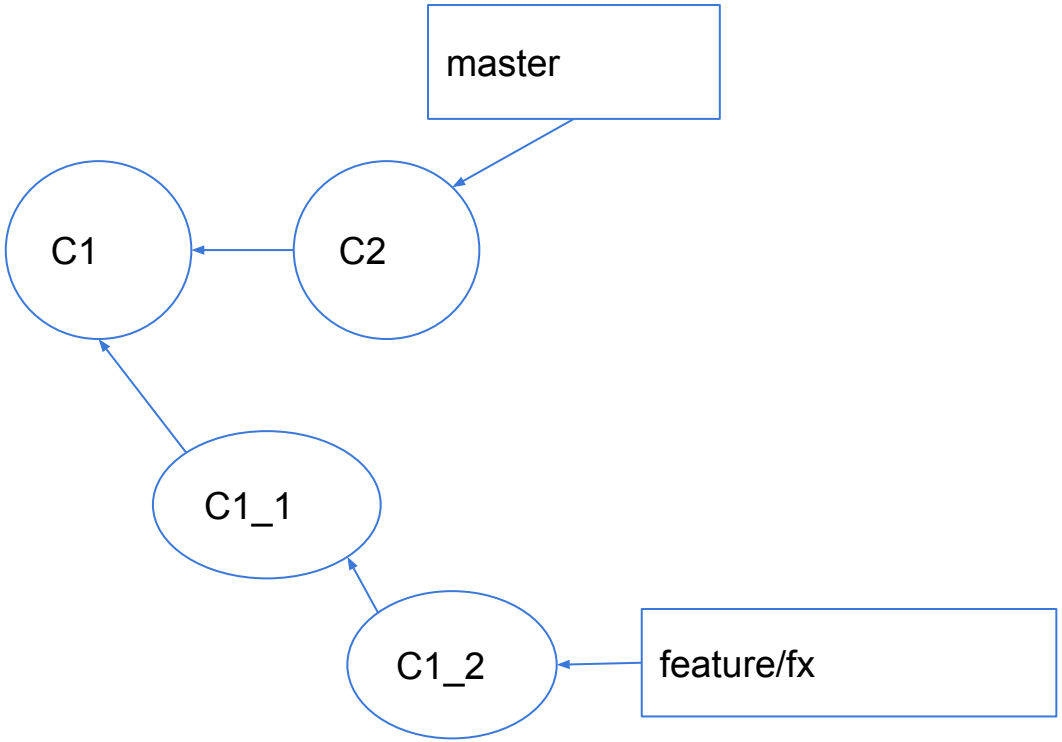
- Ein Commit-Objekt definiert einen fixen Stand
  - Beispiele
    - Release
      - v1.0
    - Milestone, Build-Nummer
    - “Heute Morgen”
  - Umsetzung mit git
    - Git Tags
      - git tag <name>
      - git tag --list
      - git tag -d <name>

Sinnvolle Kategorien eines Alias-Namens

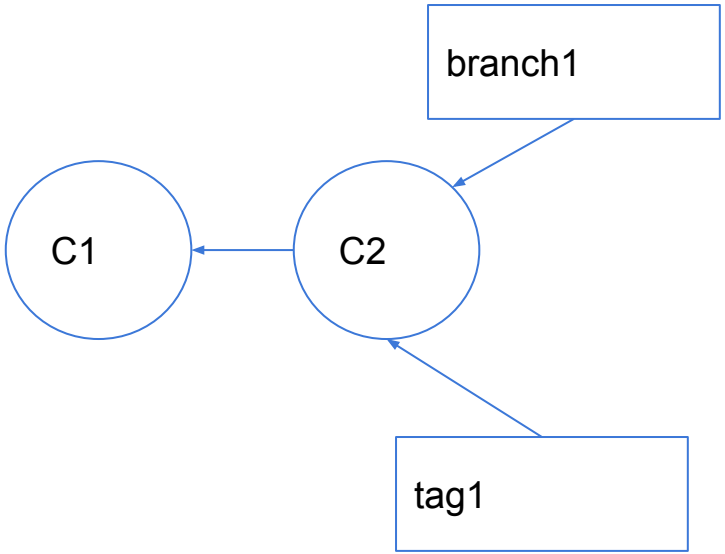
- Ein Commit-Objekt definiert einen aktuell durchgeführte Aktion
  - Beispiele
    - Entwicklung eines neuen Features
      - feature/webfrontend
    - Bugfix, Ticket-Nummers
    - “Heute Morgen”
  - Umsetzung mit git
    - Git Branches
      - git branch <name>
      - git branch --list
      - git branch -d <name>

WICHTIG

- Tags und Branches sind in Git absolut trivial
- Es sind und bleiben Alias-Namen

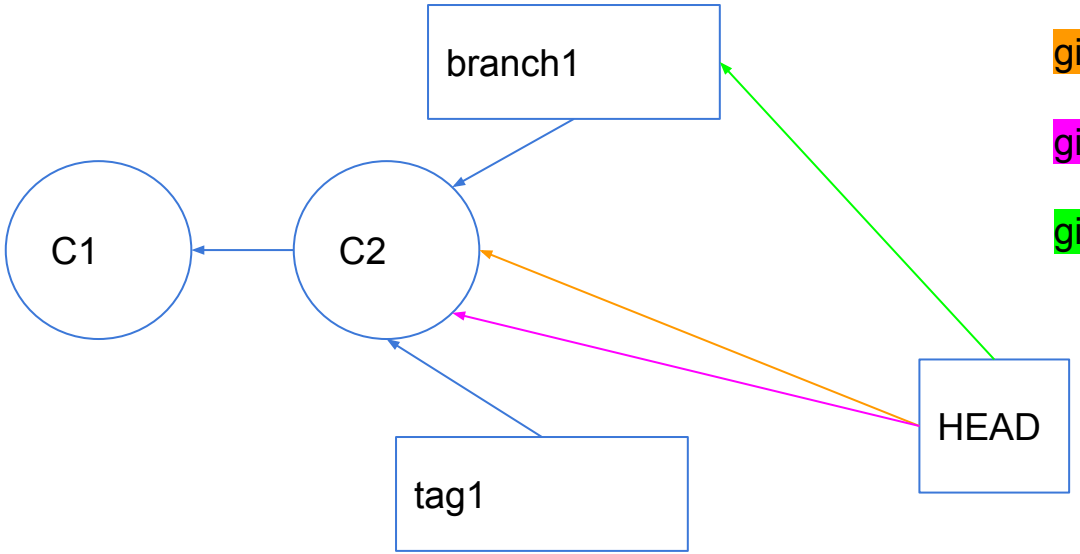

master
C1
C2
C1_1
C1_2
feature/fx

HEAD



HEAD ist ein Alias-Name, der im Geflecht der Commit-Objekte die aktuelle Position referenziert

Setzen des HEAD = git checkout



git checkout C2

git checkout tag1

git checkout branch1

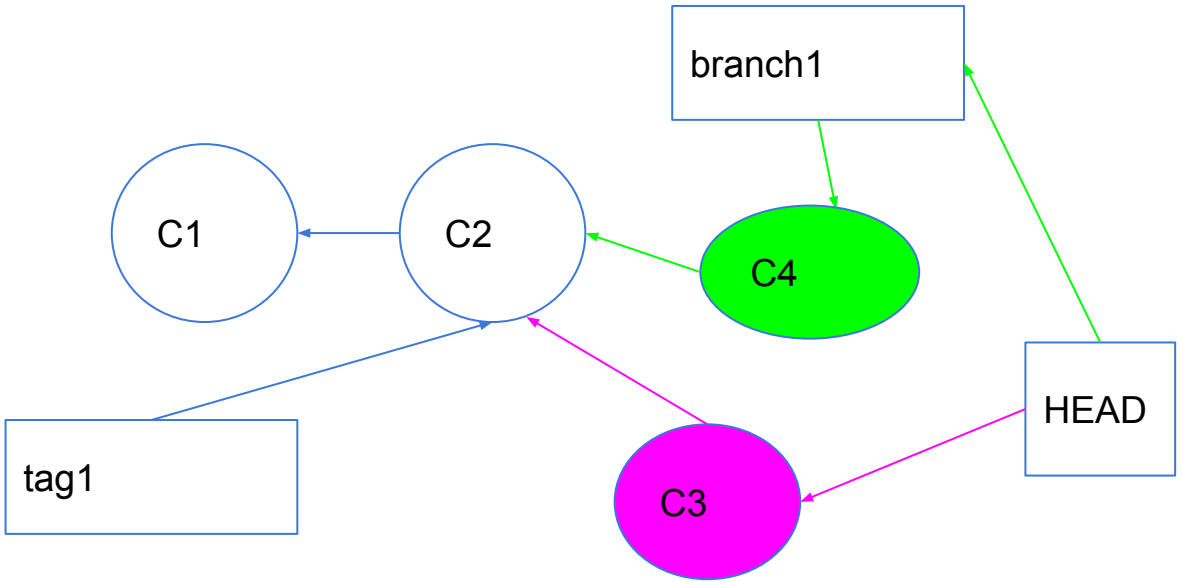
Detached HEAD

dann ist der Status auffällig

Attached HEAD

dann ist der Status unauffällig

git commit revisited




```
echo …
git add .
git commit -m “”
```

```
git checkout branch1
echo …
git add .
git commit -m “...”
```

Detached HEAD

Attached HEAD

Zusammenführen von Ständen

Ausgangssituation


```
$ git pl
* 44253d7 (feature2_part1) change content-feature2, part1
| * 05a2a0c (feature2_part2) change content-feature2, part2
|/
* c037e5a (feature2) add content-feature2
| * cdc92bc (feature1) add content-feature1
|/
* 3b74572 (HEAD -> master) change content
* f34626d add content
* f4aaddf setup project
```

Zusammenführen von Ständen

- Merge
  - Fast Forward
    - Kann nicht immer ausgeführt werden, git entscheidet
  - Recursive Merge
- Rebasing
- Interactive Rebasing
  - Kein Zusammenführen, sondern ein Zusammenfassen von Ständen
- Cherry Pick
  - “Rosinen-Picken”
  - Heute in der Git-Community als “überflüssig” gewertet

Ziel

- “Der master des Projekts soll alle Änderungen von feature1 und feature2 enthalten”
  - content.txt
  - Readme
  - content-feature1.txt
  - content-feature2.txt
- Vorhandene Konflikte durch “parallele” Änderungen an Dateien müssen aufgelöst werden
  - content-feature2.txt wird in den beiden Parts geändert
  - content-feature1.txt wurde nur im feature1-Branch hinzugefügt

Merging-Plan

- Nicht eindeutig, mehrere Wege führen zum Ziel…
- Präsentation
  - feature2 <- feature2_part1
  - feature2 <- feature2_part2
  - feature2 <- feature1
  - master <- feature2

Schritt 1

- git checkout feature2
- git merge feature2_part1
  - Direkte Linie ist vorhanden
  - ein fast forward merge ist möglich und dieser wird von git ausgeführt
  - Ein FF-Merge kann niemals einen Konflikt aufweisen

```
$ git merge feature2_part1
Updating c037e5a..44253d7
Fast-forward
 content-feature2.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

Rainer Sawitzki@LAPTOP-GVSFDDCT MINGW64 ~/git_training_fi_12.6.2023/training_
re2)
$ git pl
* 44253d7 (HEAD -> feature2, feature2_part1) change content-feature2, part1
| * 05a2a0c (feature2_part2) change content-feature2, part2
|/
* c037e5a add content-feature2
| * cdc92bc (feature1) add content-feature1
|/
* 3b74572 (master) change content
* f34626d add content
* f4aaddf setup project
```

Schritt 2: recursive merge mit Konflikt

- git merge feature_part2
  - “Vor-Zurück” statt direkter Linie
  - Damit ist nur ein recursive Merge möglich
  - mit Konflikten
    - In diesem Beispiel ein harter Konflikt: Beide parts haben eine einzige Datei geändert

```
$ 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 add .
- git commit

```
$ git pl
*   62a9d73 (HEAD -> feature2) Merge branch 'feature2_part2' into feature2
|\
| * 05a2a0c (feature2_part2) change content-feature2, part2
* | 44253d7 (feature2_part1) change content-feature2, part1
|/
* c037e5a add content-feature2
| * cdc92bc (feature1) add content-feature1
|/
* 3b74572 (master) change content
* f34626d add content
* f4aaddf setup project
```

Schritt 3

- git merge feature1
- Recursive Merge
  - mit automatischer Konflikt-Behebung
    - Änderungen in verschiedenen Dateien werden nicht als Konflikt erachtet
  - WARNUNG: In der Praxis werden Sie so etwas natürlich IMMER kontrollieren!

```
$ git pl
*   168fdf6 (HEAD -> feature2) Merge branch 'feature1' into feature2
|\
| * cdc92bc (feature1) add content-feature1
* |   62a9d73 Merge branch 'feature2_part2' into feature2
|\ \
| * | 05a2a0c (feature2_part2) change content-feature2, part2
* | | 44253d7 (feature2_part1) change content-feature2, part1
|/ /
* / c037e5a add content-feature2
|/
* 3b74572 (master) change content
* f34626d add content
* f4aaddf setup project
```

Schritt 4

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

```
$ git pl
*   168fdf6 (HEAD -> master, feature2) Merge branch 'feature1' into feature2
|\
| * cdc92bc (feature1) add content-feature1
* |   62a9d73 Merge branch 'feature2_part2' into feature2
|\ \
| * | 05a2a0c (feature2_part2) change content-feature2, part2
* | | 44253d7 (feature2_part1) change content-feature2, part1
|/ /
* / c037e5a add content-feature2
|/
* 3b74572 change content
* f34626d add content
* f4aaddf setup project
```

Zum Alias für pretty log


```
git config --global alias.pl2 "log --oneline --graph --all --decorate"
```

Ziel erweitert

- “Der master des Projekts soll alle Änderungen von feature1 und feature2 enthalten”
- Die Commit-Historie soll alle durchgeführten Aktionen dokumentieren

fast forward merge reviewed

- Best Practice
  - Zur Vollständigen Dokumentation sollte auf ein FF verzichtet werden
- git merge --no-ff

Tags und Branches revisited

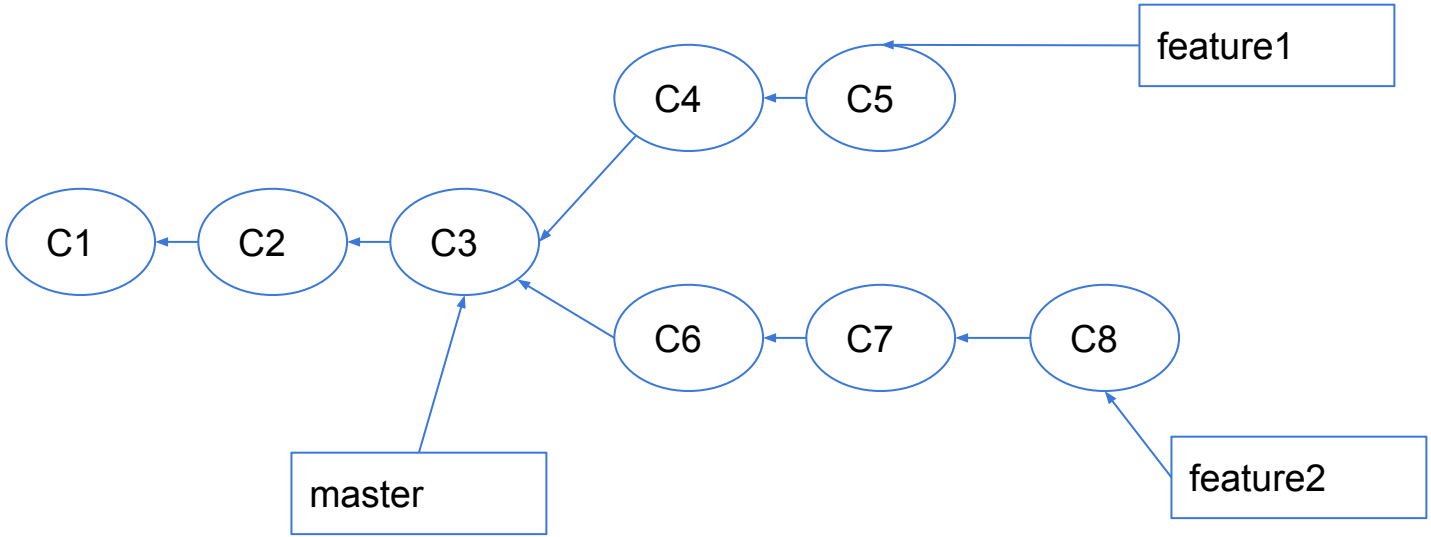
- Tag repräsentiert feste Stände, die ein Projekt erreicht hat
- Branch repräsentiert eine aktuell durchgeführte Aktion im Projekt
- Um eine Projekt-Übersicht zu erhalten, werden die Aliase aufgelistet
  - git tag --list
    - “Was haben wir erreicht?”
  - git branch --list
    - “Was machen wir gerade?”
- Best Practice
  - Fertiggestellte Feature-Branches werden gelöscht
    - eventuell: durch ein Tag ersetzt
  - Bewegungen des master-Branches werden getagged

Umdefiniertes Ziel

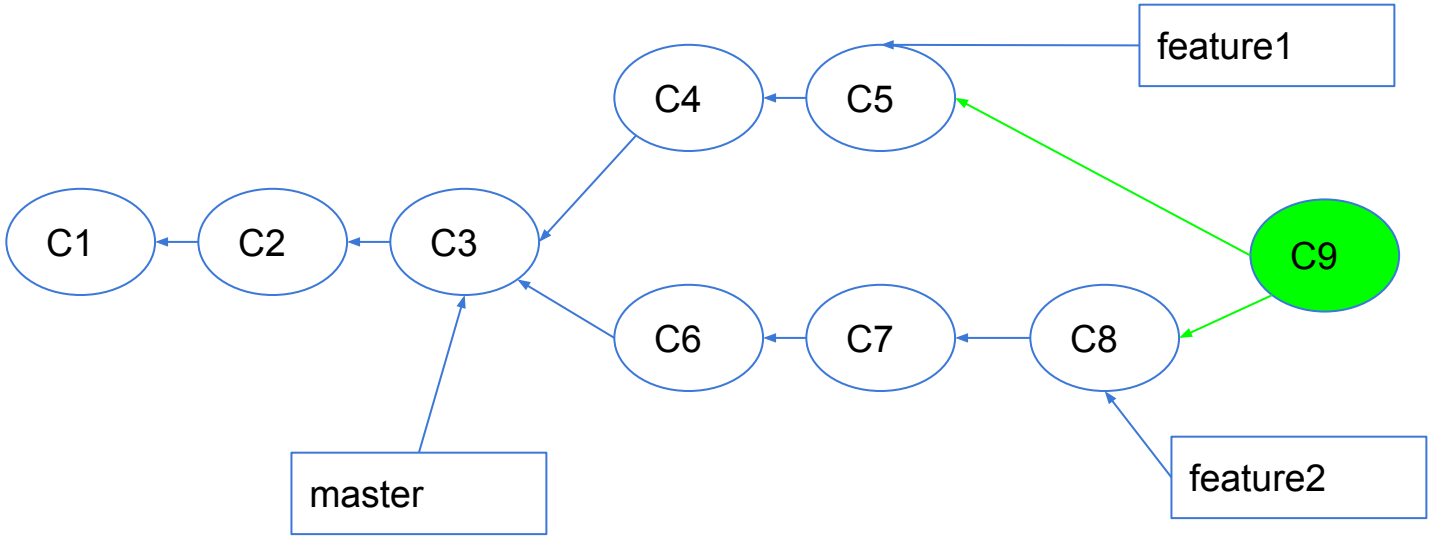
- “Der master des Projekts soll alle Änderungen von feature1 und feature2 enthalten”
- Die Commit-Historie soll eine stringente Dokumentation abbilden
  - Stringent: Sequenz von Aktionen, statt einer Parallelisierung
  - Diese Sequenz hat so aber nicht stattgefunden
    - Dokumentiert werden damit nicht die tatsächlich durchgeführten Aktionen

Umsetzung des Ziels mit Rebase

Ein Commit-Geflecht

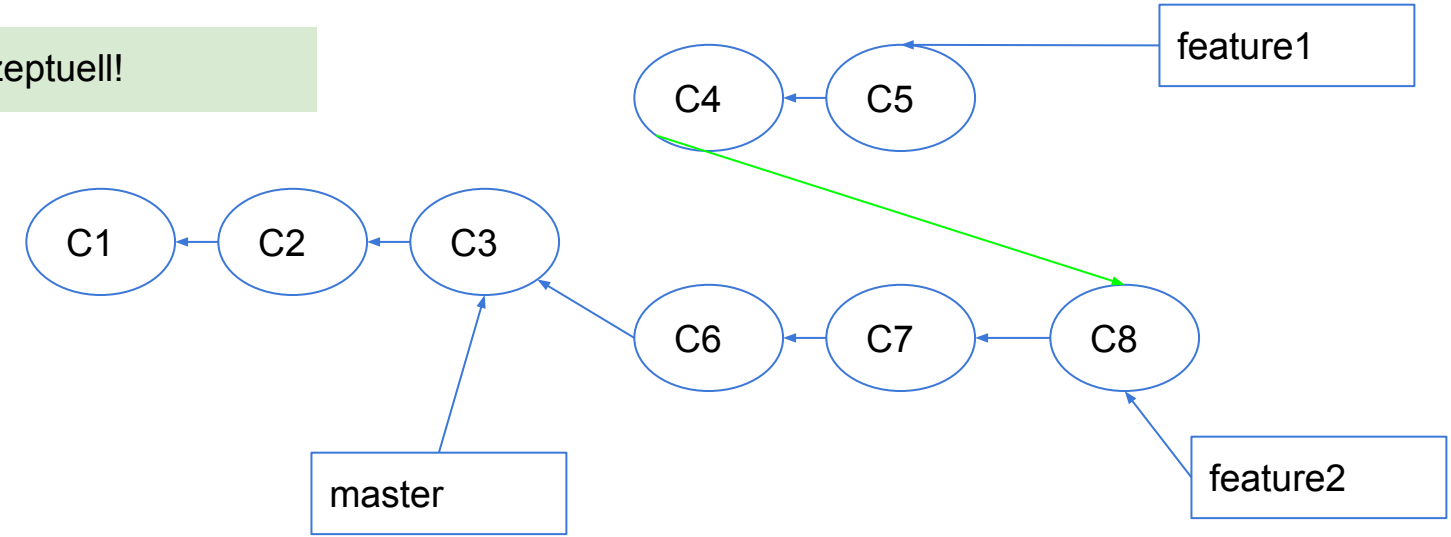


Ein merged Commit-Geflecht

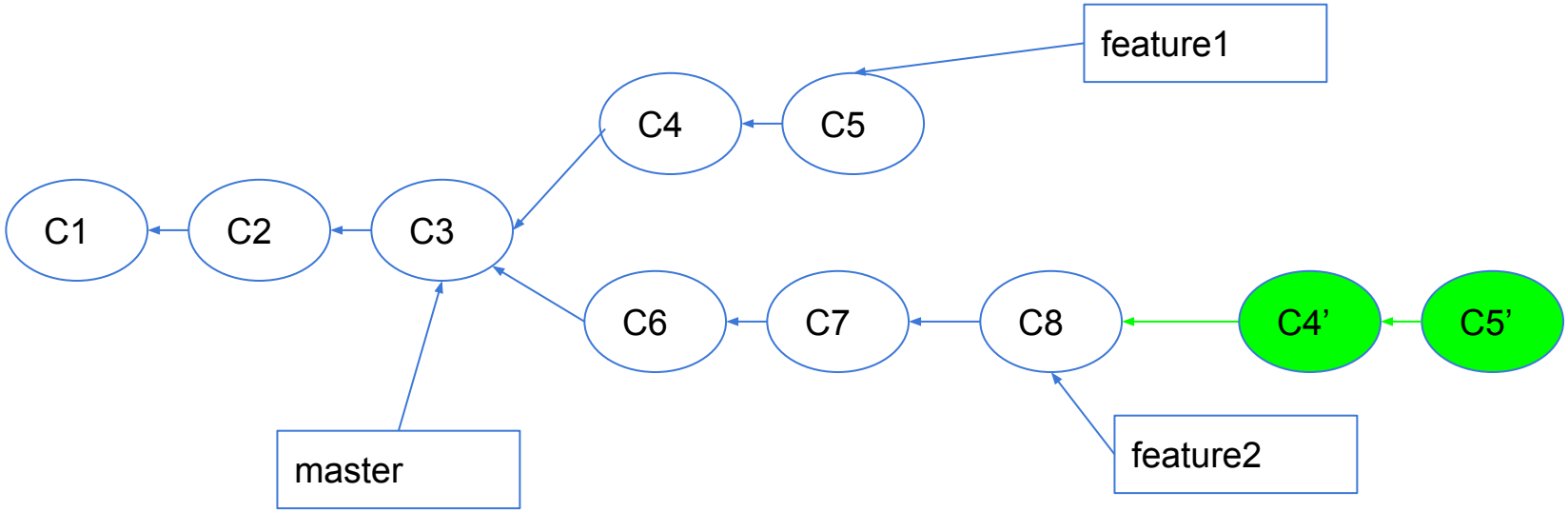


Ein rebased Commit-Geflecht

Nur konzeptuell!



Ein rebased Commit-Geflecht



Rebase Plan

- Mischung aus Rebasing & Fast Forward

Schritt 1: checkout von feature2 und ff auf feature2_part1

git checkout feature2
git merge feature2_part1
CHECK: Fast Forward Merge

```
$ git pl
* 3731feb (HEAD -> feature2, feature2_part1) change content-feature2, part1
| * c6187be (feature2_part2) change content-feature2, part2
|/
* 8570a5c add content-feature2
| * 226bf6a (feature1) add content-feature1
|/
* e46df6b (master) change content
* 2d5556b add content
* 2236e44 setup project
```

Weitere Schritte

git rebase feature2_part2

- Konflikt lösen
- git rebase --continue

git rebase feature1

git checkout master

git merge feature2

GitHub Enterprise

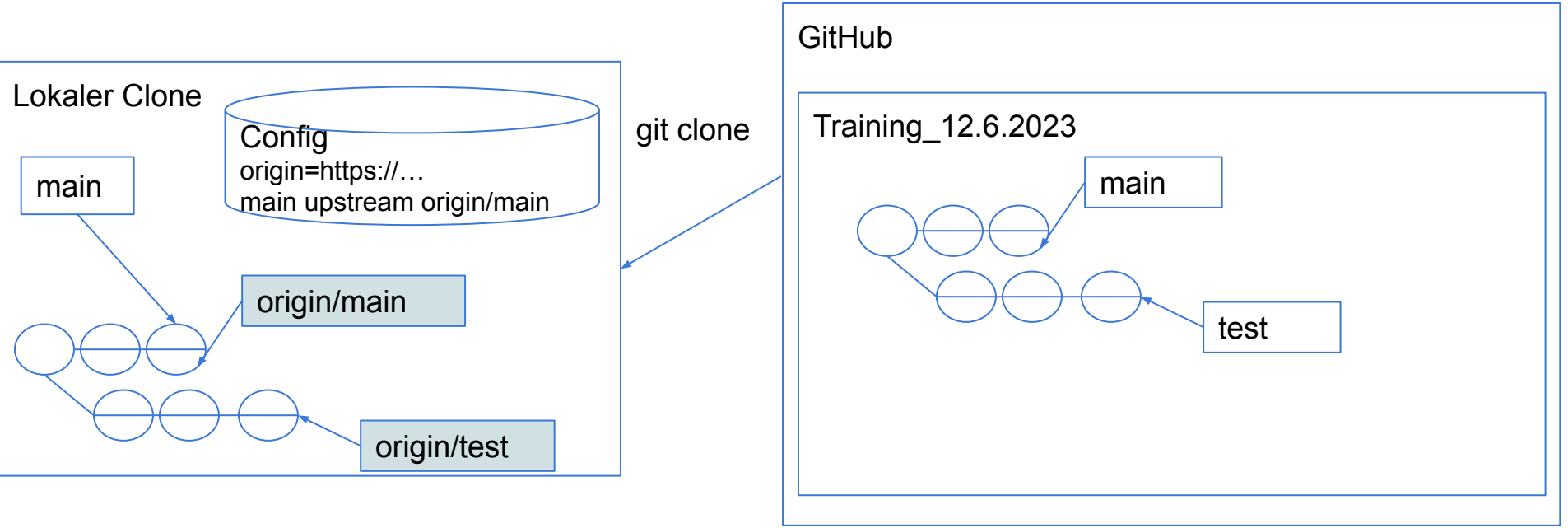
GitHub

- Verwaltung von zentralen Git Repositories
  - Anlegen eines Repositories erfolgt in der Regel auf dem Server
- Authentifizierung und Autorisierung
  - Erfolgt bei Ihnen zentral über LDAP
  - Autorisierung basiert auf einem fein-granularen Rollenkonzept
- GitHub ist nicht nur ein Git Server, sondern eine komplette Software-Development Suite
  - Wiki
  - Aufgabenverwaltung
  - Build-Maschine für CI/CD
  - Artefakt-Repository
  - Diskussionsplattform

GitHub continued

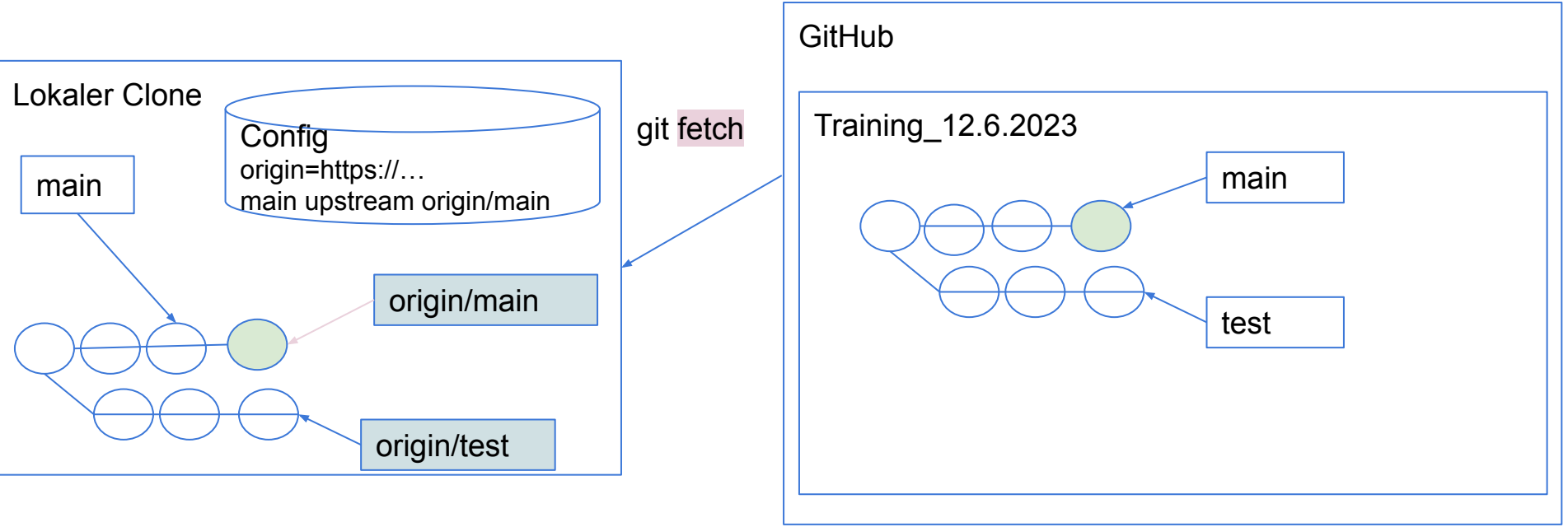
- Zugriff
  - Web Frontend
  - SSH bzw. http
    - Damit ist ein Zugriff über ein Terminal / eine Shell möglich
    - Kapselung über das Kommando git
  - REST API
    - Damit kann GitHub “gescripted” werden
    - https://www.loginradius.com/blog/engineering/github-api/#:~:text=Github%20APIs(%20or%20Github%20ReST,you%20can%20call%20the%20API.
- Authentifizierung gegen GitHub
  - Zertifikatsbasiert
  - Token-basierter Zugriff
    - Personal Access Token

Der Clone eines Repos



Remote Branches
Diese entsprechen unveränderbar dem Server-Stand zum Zeitpunkt des clone

Änderung auf Server-Seite



Remote Branches
Diese entsprechen unveränderbar dem
Server-Stand zum Zeitpunkt des letzten fetch

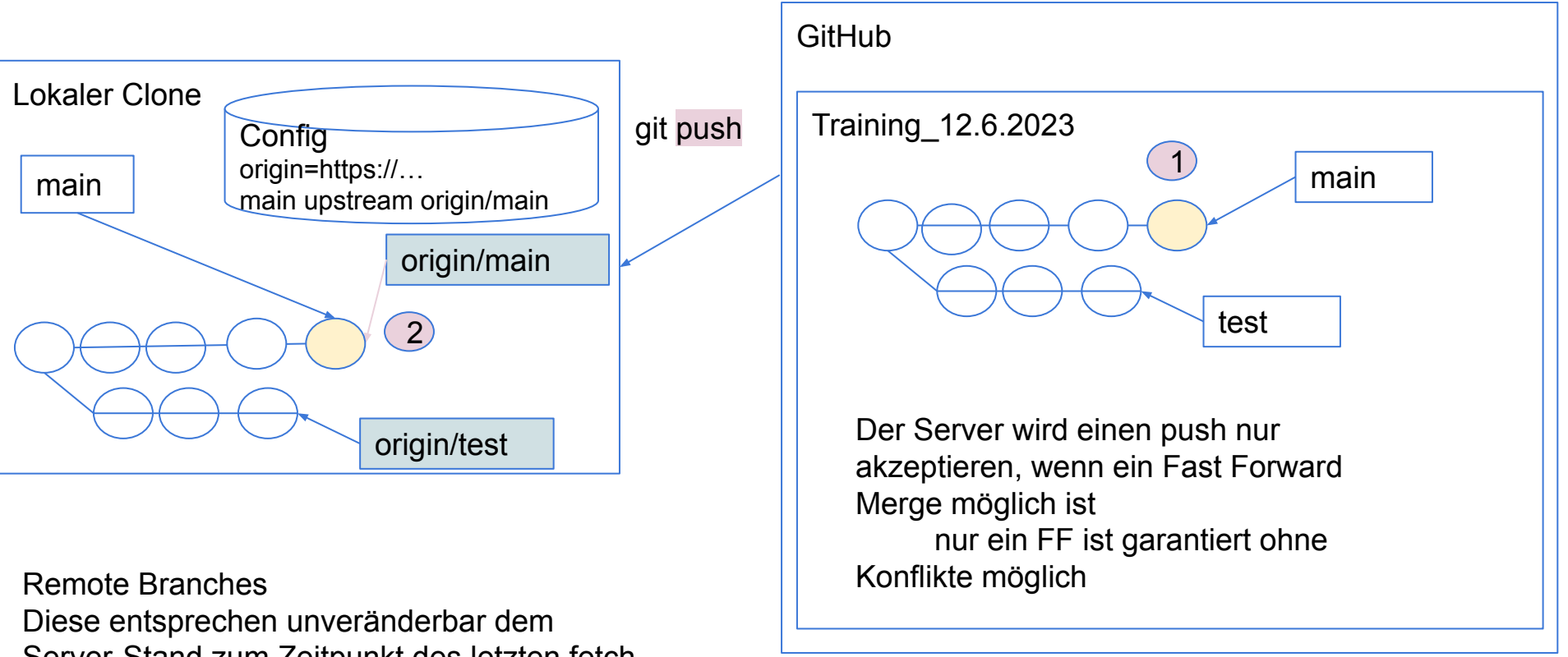
Nach dem fetch

- Mergen des lokalen Branches mit dem remote Branch
  - git fetch
  - git merge main origin/main

    git pull
    - Bei Änderungen im lokalen main natürlich mit potenziellen Konflikten und “Schleifen” in der Historie
- Rebasen des lokalen Branches auf den remote Branch
  - git fetch
  - git rebase main origin/main

    git pull --rebase
    - Bei Änderungen im lokalen main natürlich mit potenziellen Konflikten
    - “Ich spiele meine lokalen Änderungen im main auf den aktualisierten Server-Zustand nach”
-

Änderung auf Client-Seite



Remote Branches
Diese entsprechen unveränderbar dem Server-Stand zum Zeitpunkt des letzten fetch

Git Workflows

2 Haupt-Ausprägungen

- GitHub Flow
  - Ein von der GitHub-Community favorisierter Flow
- Git Flow
  - Atlassian
- Beiden gemeinsam ist die “Goldene Regel” von git
  - “Jegliche Änderung in einem Software-Projekt wird ausschließlich in einem Feature-Branch durchgeführt”
  - Ein Feature-Branch ist “kurzlebig”, wird nach Fertigstellung gelöscht
    - Langlebig ist
      - GitHub Flow: main
      - Git Flow: master/main ergänzt um einen develop-Branch
        - optional: “preprod”, “test&qs”

Präsentation

- Feature “stars” -> developer1
- Feature “planets” -> developer2

Beispiel: GitHub Flow

- Jeder Feature-Entwickler cloned das Repository mit dem aktuellen main
- Jeder Feature-Entwickler erzeugt sich lokal einen Feature-Branch und (optional)commit&push-Strategie
- Bei Fertigstellung eines Features
  - Optional:
    - Interactive Rebase und push -f oder
    - Erstellen eines finish/feature mit interactive rebase git push –set-upstream
  - Erstellen eines Pull-Requests
    - Voraussetzung: main-Branch ist aus Sicht eines Developers read-only, protected
- Pull Request durchgeführt
  - Developer des fertiggestellten Features löscht sein Repository
  - Die restlichen Developer fetchen main und rebasen ihr feature

Link-Sammlung der FI

- Git Introduction
  - https://www.youtube.com/watch?v=MaQnpCaiop0
- Git Pull Requests
  - https://www.youtube.com/watch?v=MoXxF3aWW8k
- IntelliJ
  - https://www.jetbrains.com/de-de/idea/features/#version-control
- Liste der Schulungsvideos (FI intern)
  - https://wiki.intern/display/admingit/Schulungs-Videos