



**JAVACREAM**

*Training  
Consulting  
Projectmanagement*

# GIT

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

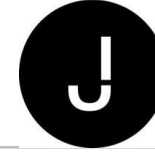


## Ausgangssituation

- Definition eines Standes eines Projekts bestehend aus Dateien
  - angereichert um Meta-Informationen: “Wer hat wann warum welche Änderungen gemacht?”
- Parallele Fortentwicklung von verschiedenen Ständen
- Konsistentes Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage der gesamten Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Team-Zusammenarbeit
- Werkzeugunterstützung zum effizienten Arbeiten

- Definition eines Standes eines Projekts bestehend aus Dateien
  - angereichert um Meta-Informationen: “Wer hat wann warum welche Änderungen gemacht?”
- Parallele Fortentwicklung von verschiedenen Ständen
- Konsistentes Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage der gesamten Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Team-Zusammenarbeit
- Werkzeugunterstützung zum effizienten Arbeiten

- Definition eines Standes eines Projekts bestehend aus Dateien
  - angereichert um Meta-Informationen: “Wer hat wann warum welche Änderungen gemacht?”
- Parallele Fortentwicklung von verschiedenen Ständen
- Konsistentes Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage der gesamten Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Team-Zusammenarbeit
  - Git Flows mit Pull- bzw. Merge-Requests
- Werkzeugunterstützung zum effizienten Arbeiten
  - Web Frontend



- BitBucket
  - Atlassian
- GitLab
  - GitLab.com
- **GitHub**
  - Microsoft

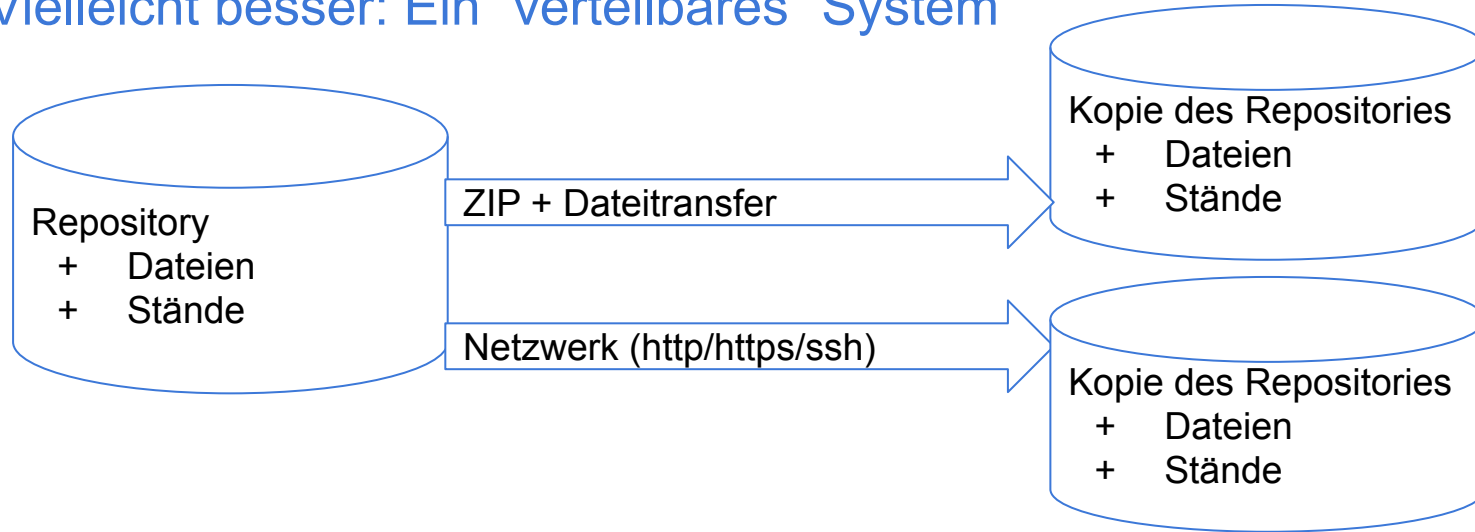
- Definition eines Standes eines Projekts bestehend aus Dateien
  - angereichert um Meta-Informationen: “Wer hat wann warum welche Änderungen gemacht?”
- Parallele Fortentwicklung von verschiedenen Ständen
- Konsistentes Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage der gesamten Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Team-Zusammenarbeit
  - Git Flows mit Pull- bzw. Merge-Requests
- Werkzeugunterstützung zum effizienten Arbeiten
  - Web Frontend

Im Seminar  
+ Tag 1 + Tag 2  
erste Session  
+ Rest

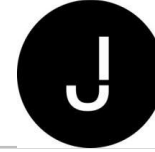


# Git ist ein “verteiltes Versionsverwaltungssystem”

- Vielleicht besser: Ein “verteilbares” System

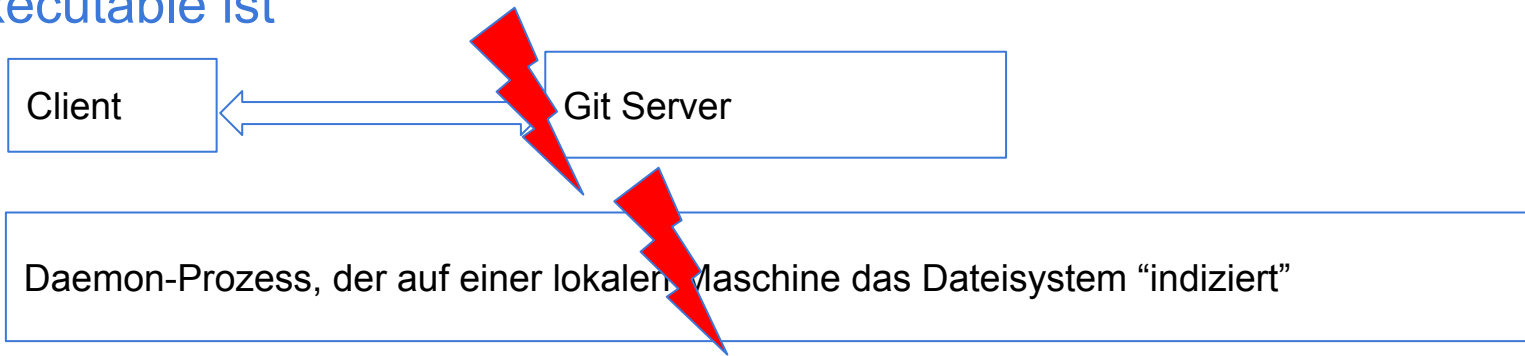


Unbedingt nötige Konsistenz = “Fälschungssicherheit” wird erreicht durch den Einsatz von Merkle-Trees (Jeder Stand bekommt einen Hashwert, und jeder Nachfolger enthält den Hashwert des Vorgängers) = Blockchain-Technologie




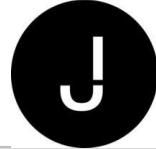
## First Contact

- Git ist auf Ihren Maschinen installiert
  - `git --version` ist erfolgreich
- Die Git-Installation installiert ein `git-executable`
- `git-executable` ist



- Ein Kommando, das während der Ausführung eines Git-Kommandos die Funktionen eines Versionsverwaltungssystem bereitstellt

- 
- Einrichten eines Users auf dem Git Server
  - Lokal die Angabe der Server-URL
  - Lokale Konfiguration eines user.name und einer user.email
    - `git config --global user.name "Rainer Sawitzki"`
    - `git config --global user.email rainer.sawitzki@gmail.com`
    - `git config --help`
    - `git config --get user.name`



- Jetzt im Seminar total unüblich
  - Initialisieren eines neuen, lokalen Repositories mit `git init`
  - Didaktisch notwendig
- richtig wäre -> später
  - Repository wird auf GitHub eingerichtet
  - und auf die lokale Maschine gecloned



- Anlegen eines neuen Verzeichnisses

- mkdir training
- cd training

training ist ein ganz normales Verzeichnis

- Initialisieren des Repositories

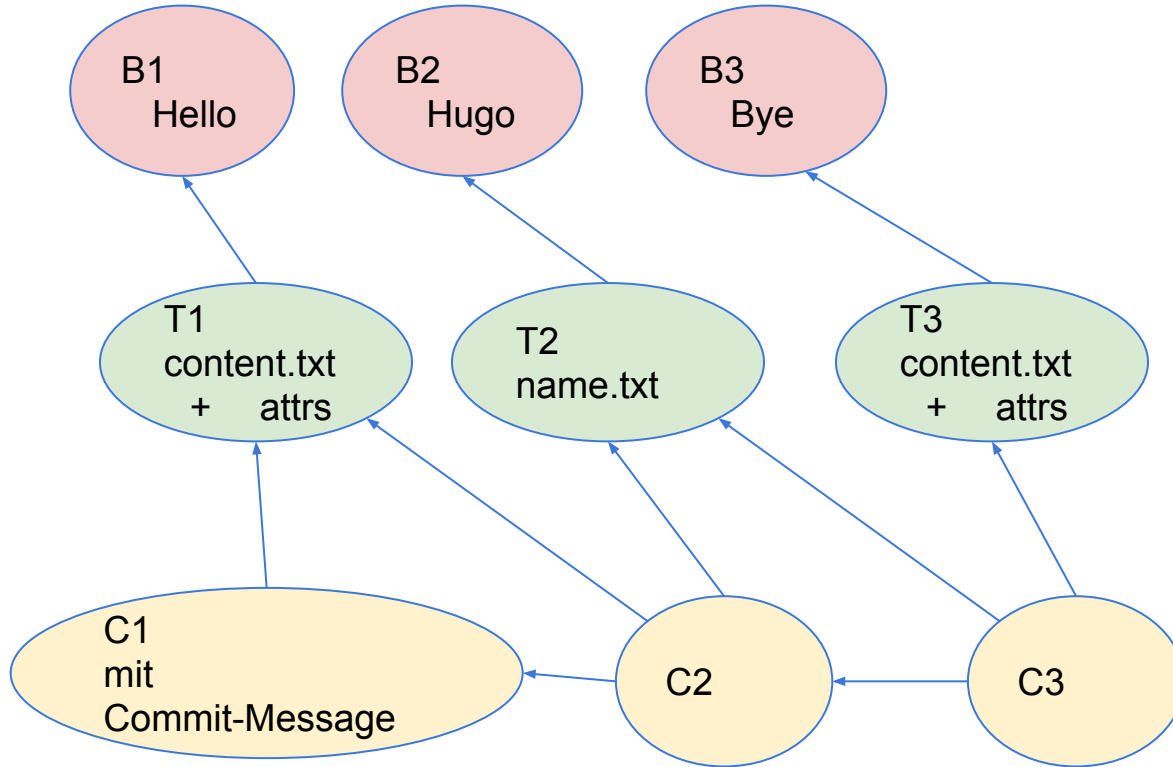
- git init
  - check: git status

training ist ein ganz normales Verzeichnis, aber nun genannt als Git Projekt-Verzeichnis

Das eigentliche Repository ist das Unterverzeichnis .git  
Der Rest des Git-Projekts wird als Git-workspace bezeichnet

content.txt ist Bestandteil des Workspaces, aber dem Repository völlig unbekannt

- Erstellen einer Datei
  - echo Hello > content.txt
    - check: git status mit einer “roten” Datei
- Bekanntmachen der Datei durch Hinzufügen zum Repository
  - genauer: Hinzufügen der Datei zur Staging-Area des Repositories
  - git add content.txt
    - check: git status mit einer “grünen” Datei
    - Hinweis: Das Hinzufügen zur Staging-Area ist keine Stand-Definition!
    - check: ls .git/objects/e9 mit der Datei `65047ad7c57865823c7d992b1d046ea66edf78`
- Definition des Standes mit git commit -m “commit message”
  - Vorsicht: Wenn Sie -m vergessen, öffnet sich ein Linux-Editor (vim, i -> Eingabemodus, ESC zurück zum Befehlsmodus, :wq zum schreiben und beenden)
    - git config --global core.editor <path\_to\_editor>
    - check: git status ist unauffällig, git log mit Ausgabe des Commits inklusive Commit-Hash

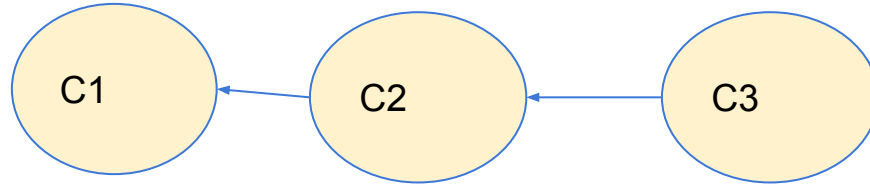


Content-Objects  
oder  
BLOBs

Tree-Objects

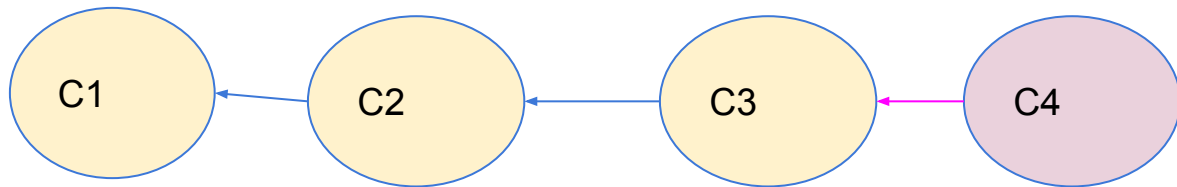
Commit-Objects





Commit-Objects

- Der alte Stand=Commit-Objekt + Staging-Area werden zu einem neuen Stand=Commit-Object zusammengeführt



Bisher: `git log` -> Standard-Darstellung, relativ verbose

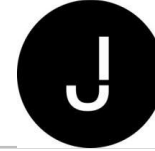
Jetzt: `git log --oneline --graph --all --decorate`

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

# Wiederherstellung des Workspaces mit einem vorhandenen Commit Object



- `git checkout <hash>`
- Hinweise
  - Das Arbeiten mit dem hash-Wert ist natürlich sehr gewöhnungsbedürftig -> “Nerd-Modus”
  - In den meisten Fällen genügen bei der Angabe des Hash die ersten 7 Stellen
  - Dringende Empfehlung “Sawitzki”
    - checkout nur bei unauffälligem Status
    - Falls Status auffällig
      - `git add . + git commit -m`
      - `git add . + git stash (-> git.pdf bzw. Online-Dokumentation)`
  - Der Status nach dem checkout spricht von einem “detached HEAD” -> etwas später



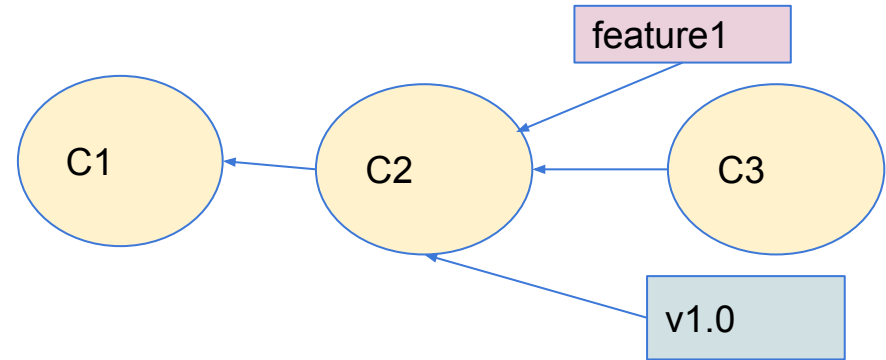
- `--global`
  - Für den angemeldeten Benutzer, `user.home .gitconfig`
- `--system`
  - für diese Git-Installation
- `--local`
  - gültig für das aktuelle Repository



## Alias-Namen auf vorhandene Commit-Objekte

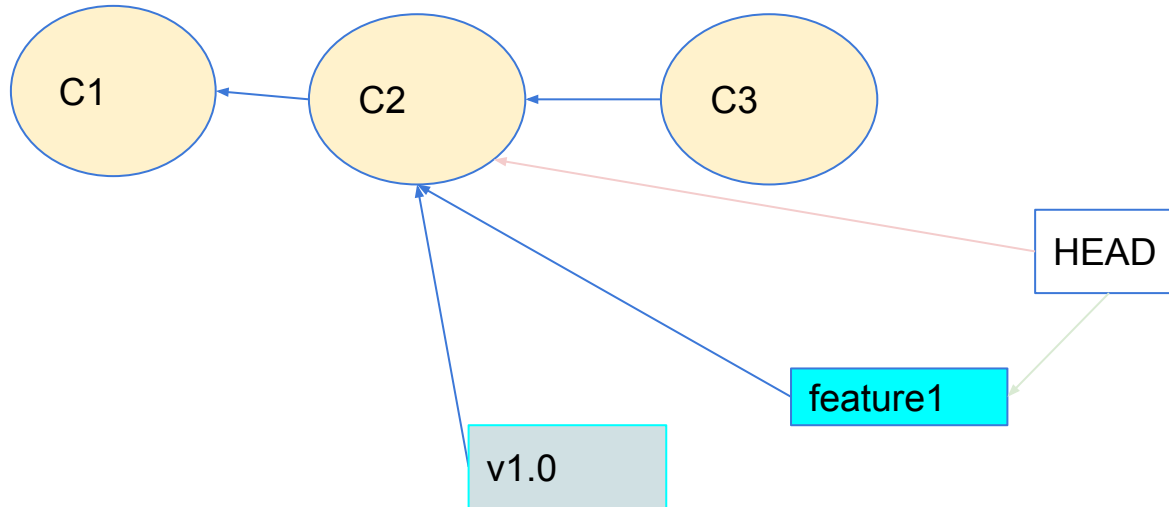
- Statt Nerd-Modus benutzen wir sprechende Namen
- 2 Einsatzbereiche im Kontext Versionsverwaltung
  - Definition eines fixen Standes
    - Versionsnummer, Release, Milestone
      - v1.0
    - Savepoint1
    - Heute Morgen
  - Definition einer aktuell fortschreitenden Entwicklung
    - implement/feature1
    - jira-issue-4711
    - experiment
    -

- Fixe Stände sind Tags
  - `git tag <tag_name>`
  - `git tag --list`
  - `git tag -d <tag_name>`
- Weiterentwicklung
  - `git branch <branch_name>`
  - `git branch --list`
  - `git branch -d <branch_name>`



```
git checkout C2
git tag v1.0
git branch feature1
```

HEAD, ein intern gepflegter  
Alias-Name auf den aktuell  
benutzten Stand



git checkout C2

Detached HEAD

git checkout v1.0

git checkout feature1

Attached HEAD



# git commit revisited

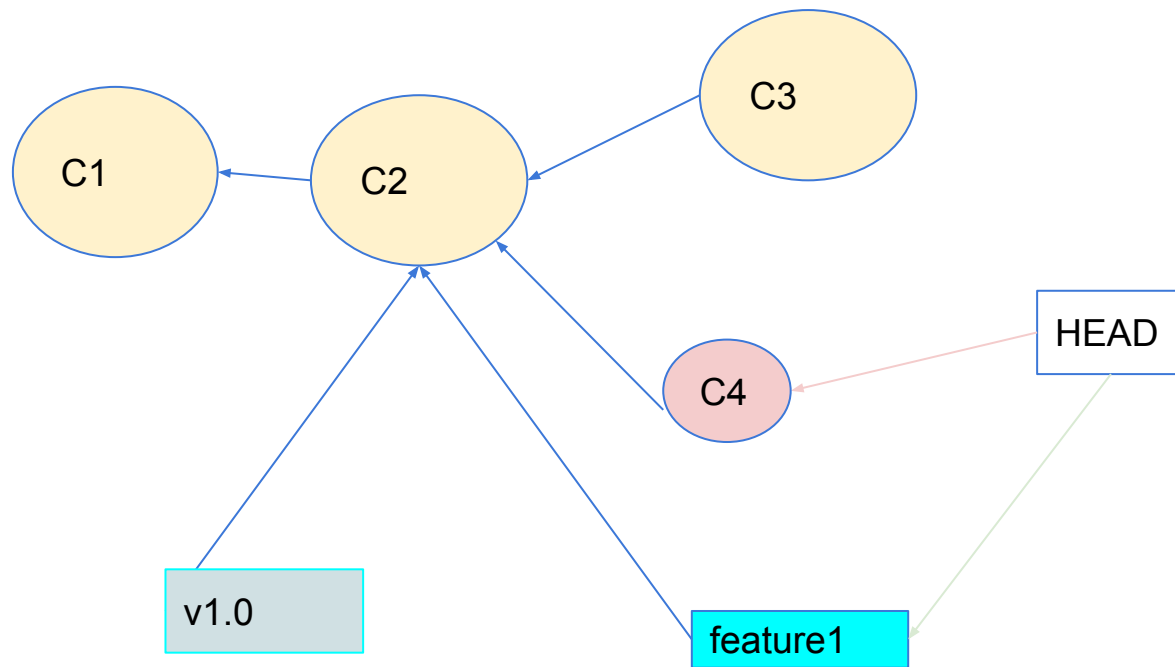


JAVACREAM

Training  
Consulting  
Projectmanagement

Detached HEAD

Attached HEAD



```
git checkout C2  
echo ... > ...  
git add .  
git commit -m
```

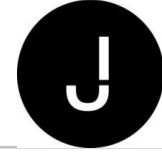
```
git checkout feature1
```

git plog

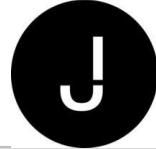
```
* d1fad6f (feature/planet_part2) add Jupiter
* a218784 add Uranus
| * 77181b3 (feature/planet_part1) add Mercury
|/
* 0255cb0 (feature/planet) add venus
| * 4b89178 (feature/star) change to So1
| * e2f3bea add vega
|/
* d12bccb (HEAD -> master) setup project
```



## Konsolidieren von Ständen



- merge
  - Begriffe
    - Fast Forward Merge
    - Recursive Merge
- rebase
  - cherry-pick
    - In der Git-Community mittlerweile als unnötiges Feature bezeichnet
- interactive rebase



- Fachlich: Der Haupt-Branch ( master oder main) soll die Änderungen der beiden Features star / planet enthalten
  - README.txt, star.txt (Sol), planet.txt (Mercury and Jupiter)
- Die Historie der Commit-Objekte soll exakt dokumentieren, wie das Projekt entwickelt wurde



- Konsolidierung des Features planet
- Danach konsolidieren mit star
- Zum Abschluss “Vorziehen” des master

- git checkout feature/planet

```
* fc153f0 (feature/planet_part1) add Mercury
| * 8511519 (feature/planet_part2) add Jupiter
| * 481f70c add Uranus
|/
* 9b765e9 (HEAD -> feature/planet) add venus
| * 8c5df9e (feature/star) change to Sol
| * 9f92324 add vega
|/
* 4502a2c (master) setup project
```

- Mergen von planet mit planet\_part1
- `git merge feature/planet_part1`
  - git erkennt die direkte Verbindung und führt einen Fast Forward Merge aus
  - Fast Farward ist IMMER Konflikt-frei möglich

```
Updating 9b765e9..fc153f0
Fast-forward
 planet.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

Rainer Sawitzki@LAPTOP-GVSFDDCT MINGW64 /c/training/training_branches (feature/planet)
$ git pl
* fc153f0 (HEAD -> feature/planet, feature/planet_part1) add Mercury
* 8511519 (feature/planet_part2) add Jupiter
* 481f70c add Uranus
/
* 9b765e9 add venus
* 8c5df9e (feature/star) change to Sol
* 9f92324 add vega
/
* 4502a2c (master) setup project
```

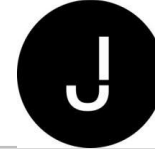


- `git merge feature/planet_part2`
- Hier: Keine direkte Verbindung -> Fast Forward nicht möglich, es wird ein Recursive Merge ausgeführt
  - hier sind Konflikte jederzeit möglich

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

- Konflikte
  - Beheben durch Entfernen der Marker und fachlich konsistente Erstellung der Dateien
  - `git add`
  - `git commit`

## Step 3



```
* 5a8ad1b (HEAD -> feature/planet) Merge branch 'feature/planet_part2'
into feature/planet solve conflict in planet.txt, ...
| \
| * 8511519 (feature/planet_part2) add Jupiter
| * 481f70c add Uranus
* | fc153f0 (feature/planet_part1) add Mercury
| /
* 9b765e9 add venus
| * 8c5df9e (feature/star) change to Sol
| * 9f92324 add vega
| /
* 4502a2c (master) setup project
```

- Konsolidieren mit feature/star
- git merge feature/star
  - Das muss ein recursive merge sein
  - Konflikte: Änderungen in planet.txt und star.txt
  - Diese Konflikte werden aber im Standard-Merge von Git über “autoconflict resolution” gelöst
    - Die beiden Dateien werden einfach beide übernommen
    - VORSICHT: Das kann fachlich falsch sein!

```
* 9702252 (HEAD -> feature/planet) Merge branch 'feature/star' into fe
ature/planet
|
| * 8c5df9e (feature/star) change to So1
| * 9f92324 add vega
| * | 5a8ad1b Merge branch 'feature/planet_part2' into feature/planet so
|ve conflict in planet.txt, ...
|
| * | 8511519 (feature/planet_part2) add Jupiter
| * | 481f70c add Uranus
| * | fc153f0 (feature/planet_part1) add Mercury
|
| * 9b765e9 add venus
|
| * 4502a2c (master) setup project
```

- Vorziehen des master
- git checkout master
- git merge feature/planet
  - CHECK: Das muss ein Fast Forward Merge sein

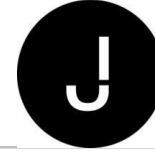
```
* 9702252 (HEAD -> master, feature/planet) Merge branch 'feature/star'
into feature/planet
|
| * 8c5df9e (feature/star) change to Sol
| * 9f92324 add vega
| * | 5a8ad1b Merge branch 'feature/planet_part2' into feature/planet so
|ve conflict in planet.txt, ...
|
| * | 8511519 (feature/planet_part2) add Jupiter
| * | 481f70c add Uranus
| * | fc153f0 (feature/planet_part1) add Mercury
|
| * 9b765e9 add venus
|
| * 4502a2c setup project
```

- Taggen
- Löschen des Feature-Banches

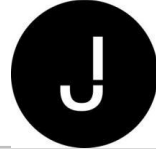
```
* 9702252 (HEAD -> master, tag: v1.0) Merge branch 'feature/star' into
feature/planet
|
| * 8c5df9e change to Sol
| * 9f92324 add vega
| * | 5a8ad1b Merge branch 'feature/planet_part2' into feature/planet so
|ve conflixt in planet.txt, ...
| |
| | * | 8511519 add Jupiter
| | * | 481f70c add Uranus
| * | fc153f0 add Mercury
| |
| * 9b765e9 add venus
|
| * 4502a2c (tag: v0.0) setup project
```

- Eine Option von merge
- Es wird immer ein recursive merge (ort-merge in neuesten Git-Versionen durchgeführt)

```
* d7ac0b2 (HEAD -> main) Merge branch 'feature/planet'
* 49e6463 (feature/planet) Merge branch 'feature/star' into feature/planet
* 220db52 (feature/star) change to Sol
* f737369 add vega
* 2c6ace9 Merge branch 'feature/planet_part2' into feature/planet
* 498fbff (feature/planet_part2) add Jupiter
* 7b3ba65 add Uranus
* c03d128 Merge branch 'feature/planet_part1' into feature/planet
* b02c848 (feature/planet_part1) add Mercury
* 4624890 add venus
* bb2d785 setup project
```



- Kategorien von Tags
  - Transiente Tags
    - Tag nur als Alias-Name
  - Documented Tags
    - Diese enthalten zusätzlich eine beschreibende Nachricht
  - Signed Tags
    - Enthalten eine Signatur, die geprüft werden kann

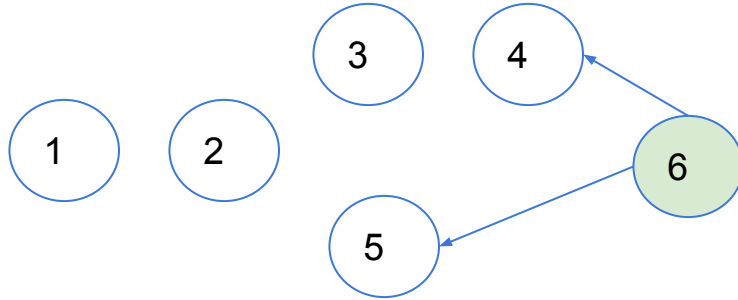


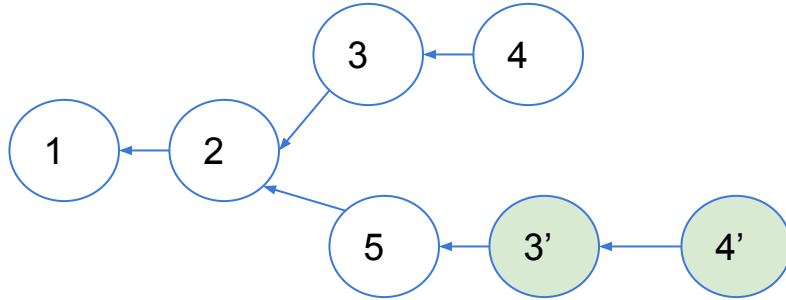
- Fachlich: Der Haupt-Branch ( master oder main) soll die Änderungen der beiden Features star / planet enthalten
  - README.txt, star.txt (Sol), planet.txt (Mercury and Jupiter)
- Die Historie der Commit-Objekte soll stringent und nachvollziehbar die Projekt-Weiterentwicklung dokumentieren



# Recap: Merging

git merge 4 5





`git rebase 4 5`

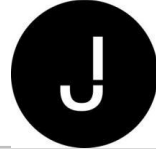
## Arbeitsweise

- + Gehe zurück zum gemeinsamen Vorgänger = 2
- + Nimm 3 und spiele diese Änderungen nach auf 5
  - + Konflikt-Potenzial
- + Nimm 4 und spiele diese Änderungen auf 3' nach
  - + Konflikt-Potenzial

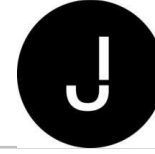
## Bei Konflikten

- + Datei mit Marker
- + Lösen
- + `git add .`
- + `git rebase --continue`

- `git rebase --interactive to_rebase`
  - `git rebase -i to_rebase`
- Ablauf
  - Es öffnet sich ein Editor-Fenster, in dem das Rebase-Skript angezeigt wird
  - Commit-Message kann in einem weiteren sich öffnenden Fenster editiert werden
  - Final kommt die Commit-Nachricht des erstellten endgültigen Commit-Objekts



<https://stackoverflow.com/questions/1967370/git-replacing-lf-with-crlf>



## Git Workflows

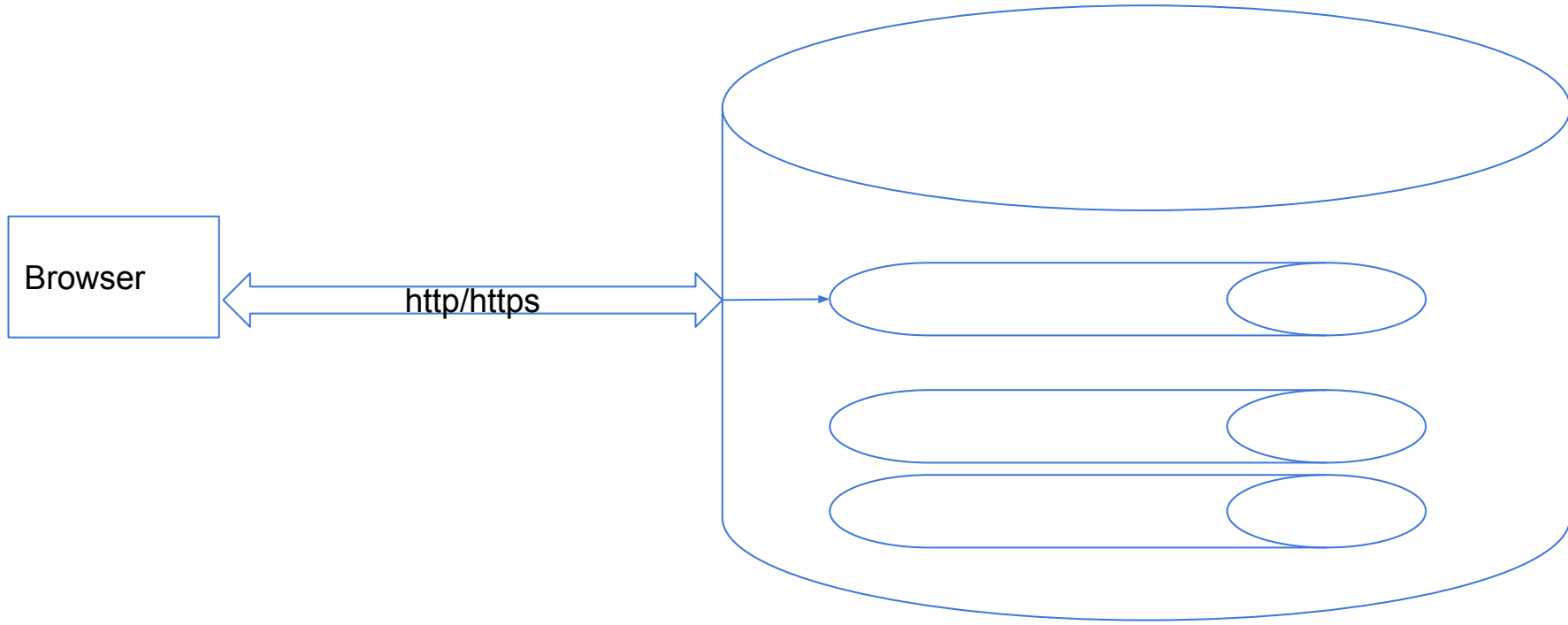
- Etablierte Arbeitsweisen zum Arbeiten mit Git
  - Vorsicht:
    - Diese Flows werden bei Recherchen häufig als “alles Mist” bezeichnet
    - Sie sind aber klar als anzupassende Vorlagen zu sehen
- Heute die beiden quasi Standards
  - Git Flow propagiert von der Atlassian Community
  - GitHub Flow propagiert von der GitHub Community
  - Hinweis: Die Flows haben nichts mit den Produkten zu tun
    - z.B. der GitHub Flow kann auch in einem lokalen Repo verwendet
    - “Ich mache GitHub Flow mit BitBucket”

- “Es gibt einen langlebigen Branch, der das Fortschreiten eines Projektes definiert”
- Es gibt einen Satz von kurzlebigen Branches
  - Feature-Branche für Weiterentwicklungen
  - HotFix / BugFix
  - ...
- Unverhandelbare Best Practice
  - Änderungen / Weiterentwicklungen werden niemals direkt im main-Branch durchgeführt!

- Durchgeführt von einem einzelnen Developer an einem lokalen Repository



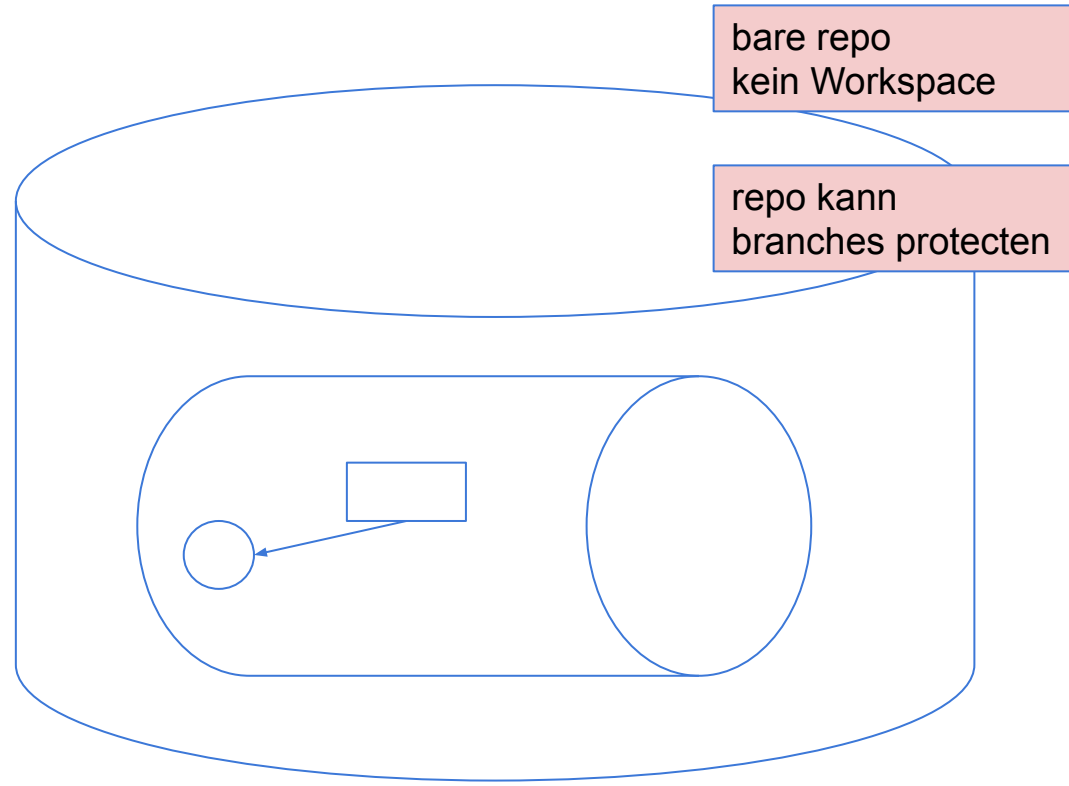
# Einschub: Git Server am Beispiel GitHub Enterprise



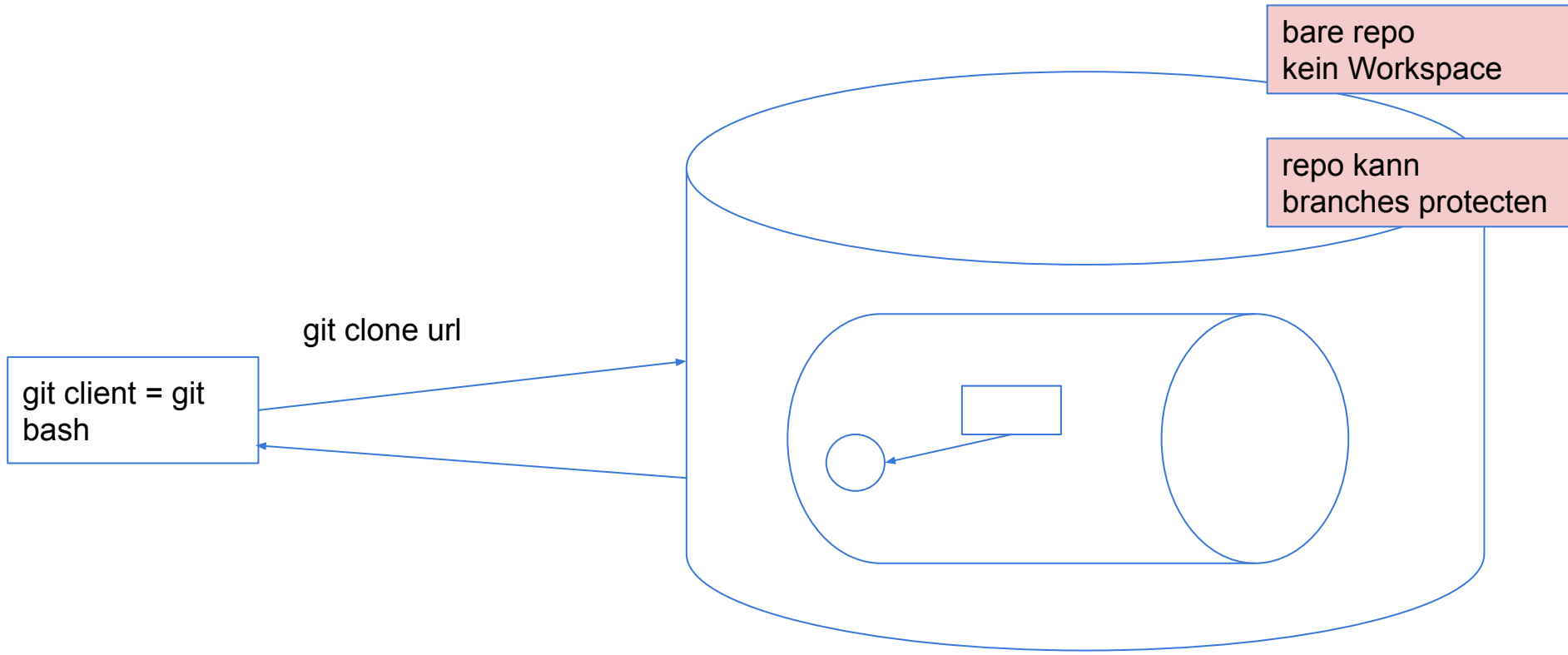


- Ein Server für Git Repositories
- Pipelines für CI/CD (Continuous Integration / Continuous Deployment/Delivery)
- Issue Management
- Wiki
- Security-Scans

# Ein Repository im GitHub Enterprise



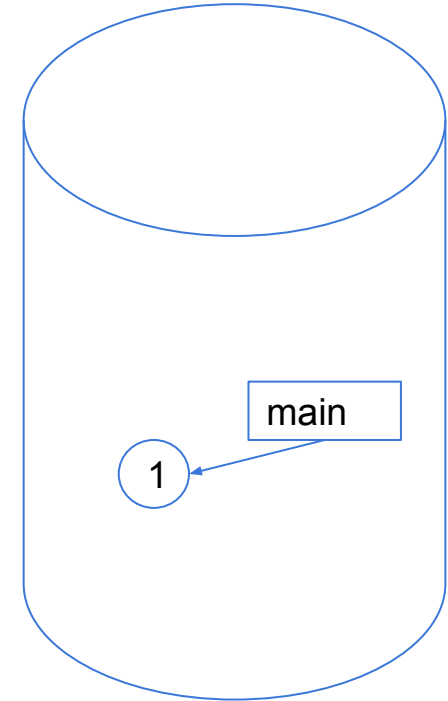
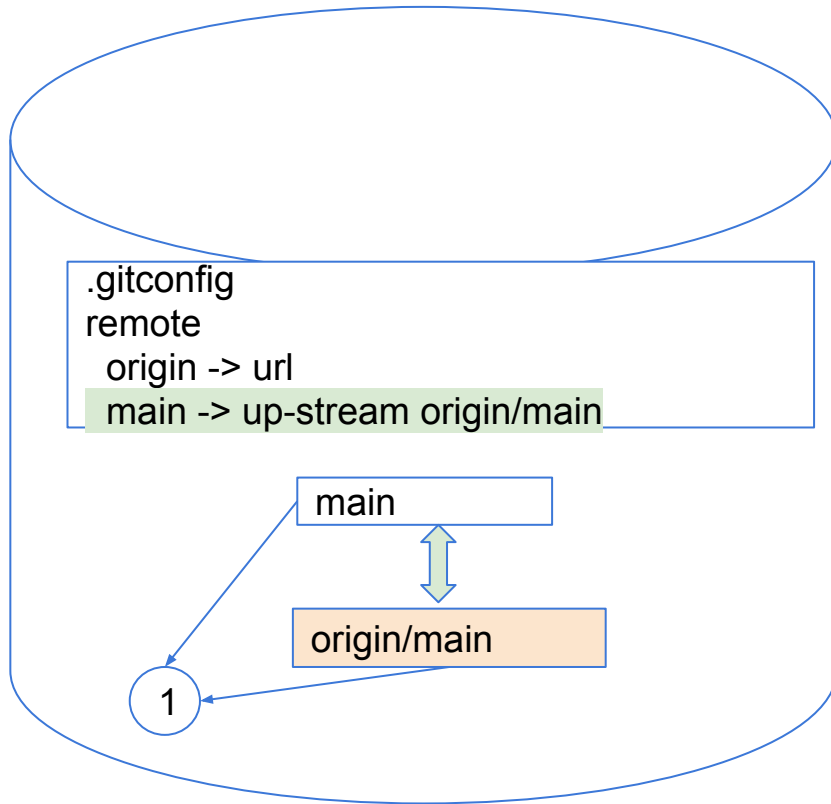
# git clone



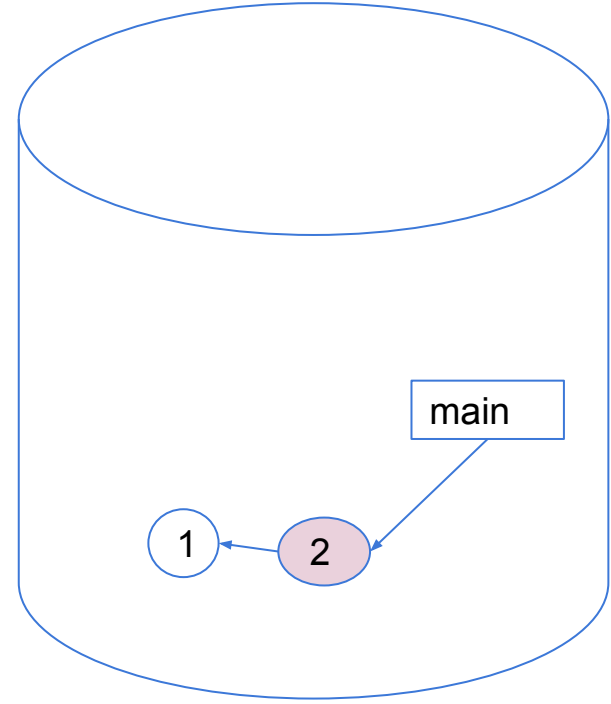
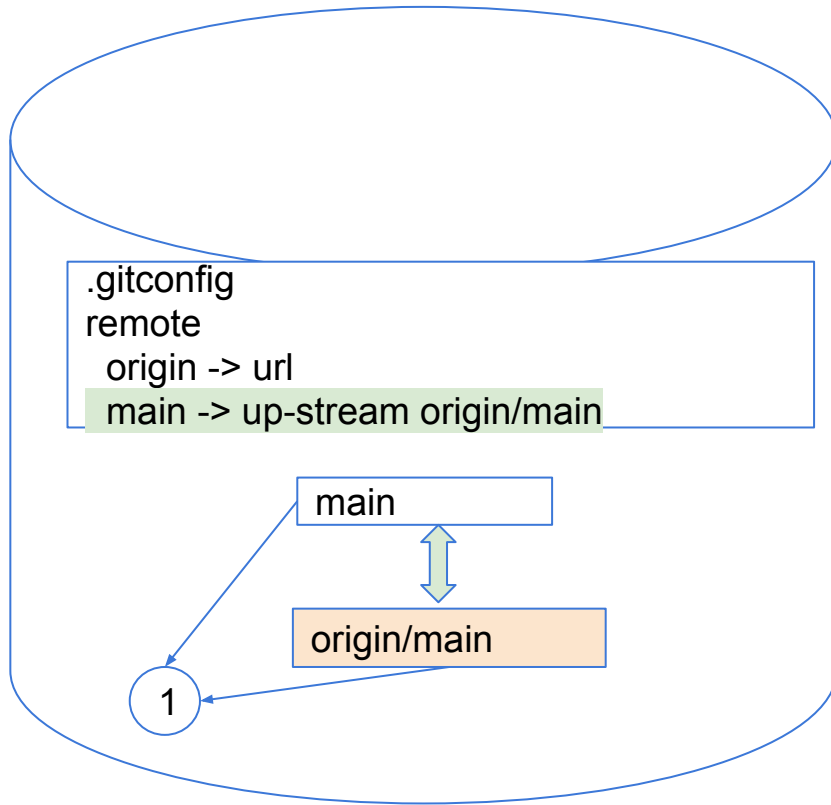
- existieren im Clone eines remote repositories
- Sind nur konfiguratv gemapped auf die url des Servers
- Ein remote branch ist immer in dem Zustand, in dem der zugehörige Branch auf Server-Seite zum Zeitpunkt des clonens war
  - Andere Formulierung:
    - Aus Sicht des Client-Repositories sind remote branches Read-Only
    - Ein checkout führt zu detached HEAD

# Was macht “clone”

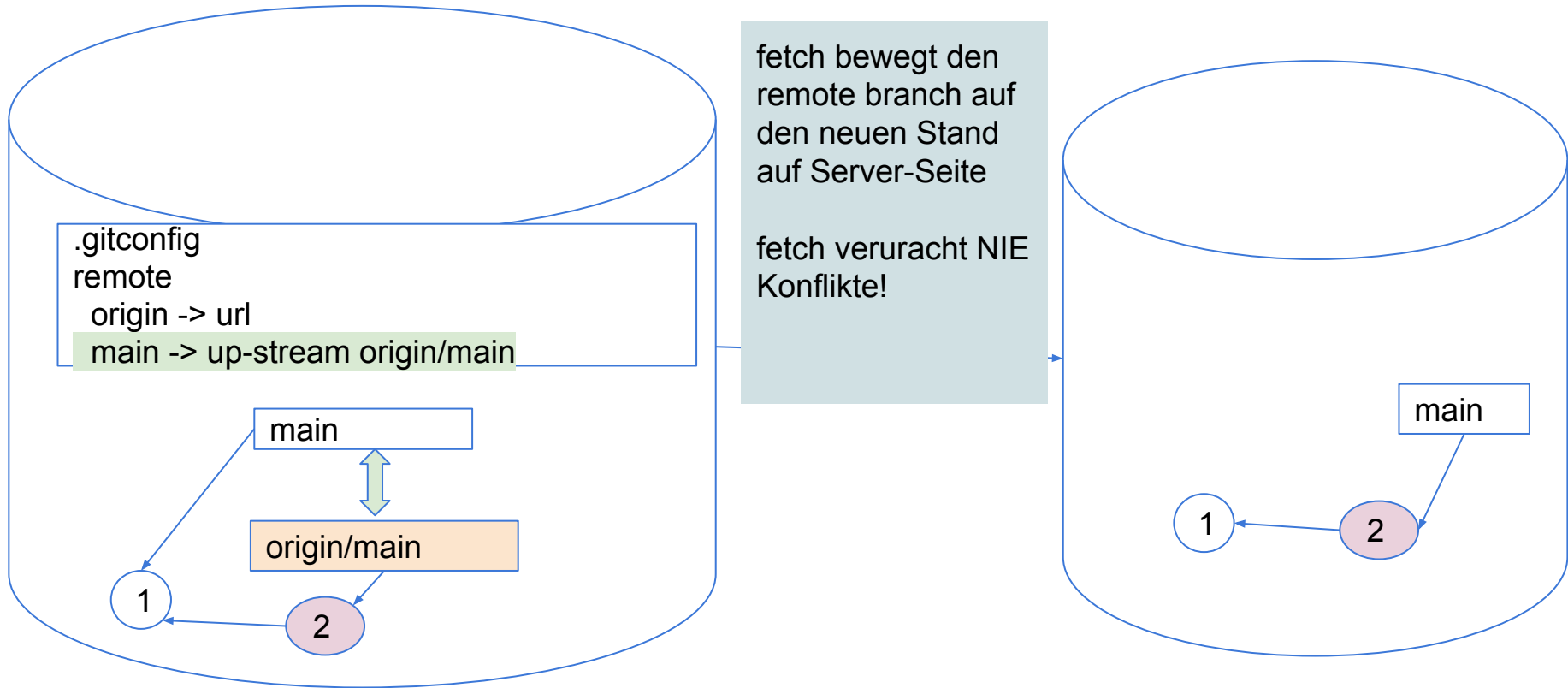
remote branch



# Änderung auf Server-Seite



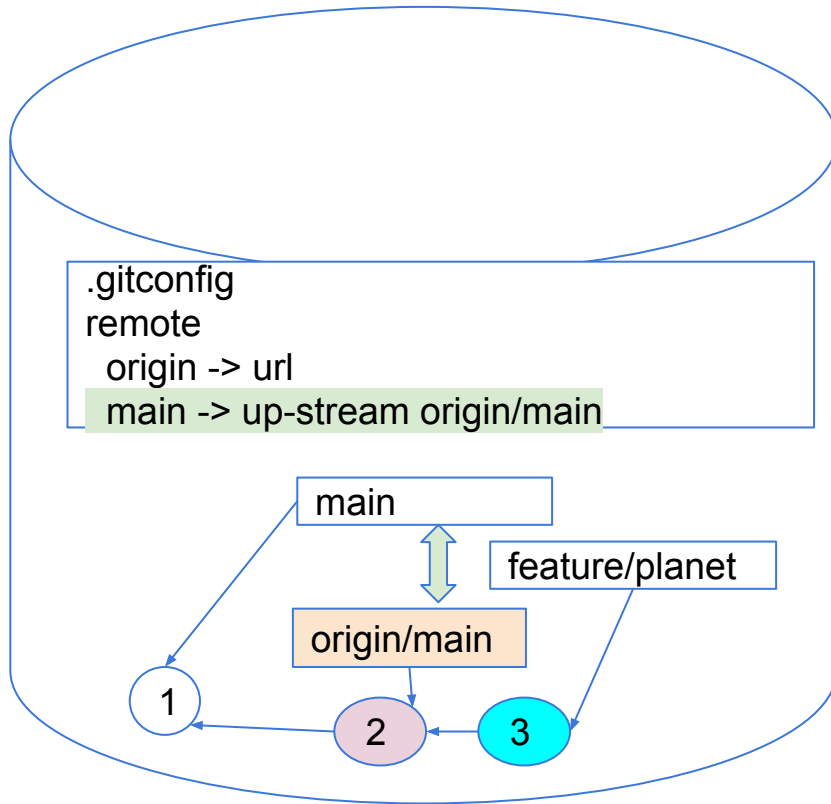
# Änderung auf Server-Seite mit fetch





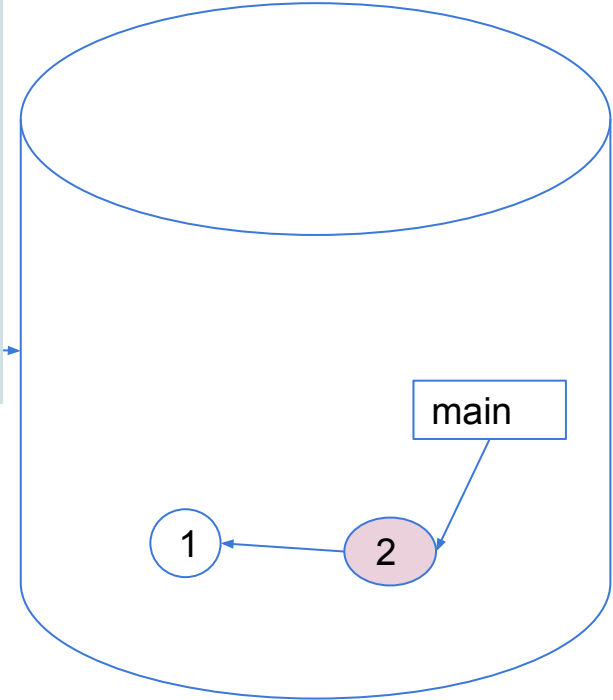
- Abkürzung für
  - `git fetch`
  - `git merge main origin/main`
- `git pull --rebase`
  - `git fetch`
  - `git rebase main origin/main`
- **VORSICHT:** Ein `git pull` kann selbstverständlich zu Konflikten führen!
- Hinweis
  - In den meisten Fällen ist ein rebase die bessere Idee...

# Änderung im Clone

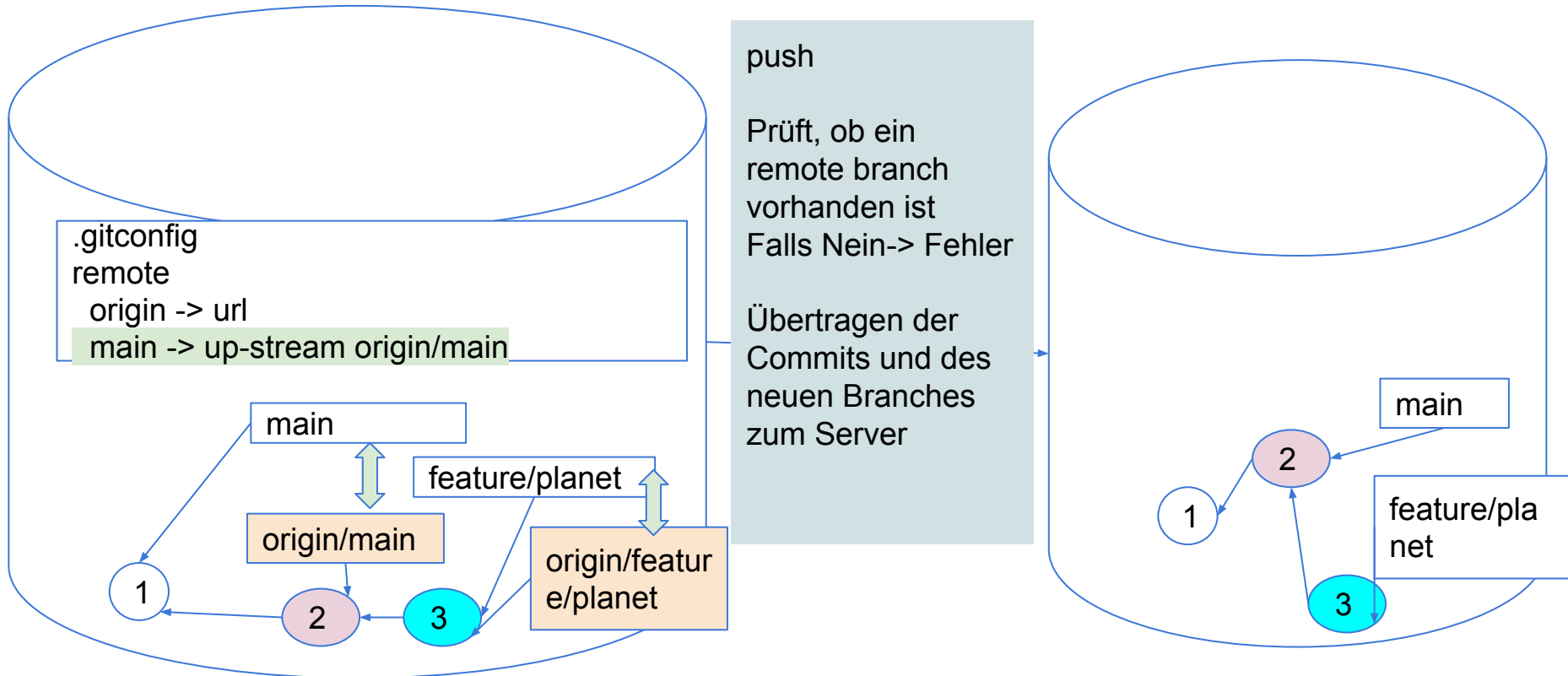


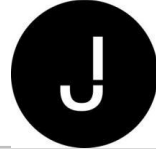
fetch bewegt den  
remote branch auf  
den neuen Stand  
auf Server-Seite

fetch verursacht NIE  
Konflikte!

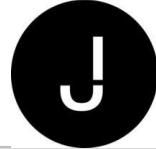


# Änderung im Clone zum Server mit push





- Starten eines Prozesses
  - Diskussion der Team-Mitglieder
  - Review
  - Wenn alles OK -> Integration in den main
- Git Server stellen hierzu eine Funktionalität bereits
  - Pull Request
    - Andere Produkte: Merge Request
  - Warum dieser Name
    - Zur Integration in den main muss ein Verantwortlicher das Repository clonen bzw. mit pull aktualisieren



- release
  - langlebig
  - enthält nur hochwertige Commits, die einem Release-Stand entsprechen
- develop
  - langlebig
  - enthält hochwertige Commits, aber auch Milestones, instabile Commits bei fehlerhaften Pull Requests
- feature-Branches
  - kurzlebig



- Fokussiert auf der Ablage von “großen Dateien”
- Installation einer git-extension, git lfs install
- Ab jetzt
  - Alle Files einer bestimmten Dateigröße werden als “groß” erkannt
  - git lfs track “\*.pdf”
  - Ganz normal weiterarbeiten mit git...
- Hintergrund
  - Referenzen auf “Large BLOBs” werden durch Proxies ersetzt
    - Diese werden erst aufgelöst, wenn ein zugehöriger Commit ausgecheckt wird