JAVACREAM

Training
Consulting
Projectmanagement

GIT

Kurze Vorstellung

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

Git - Ein verteiltes Versionsverwaltungssystem

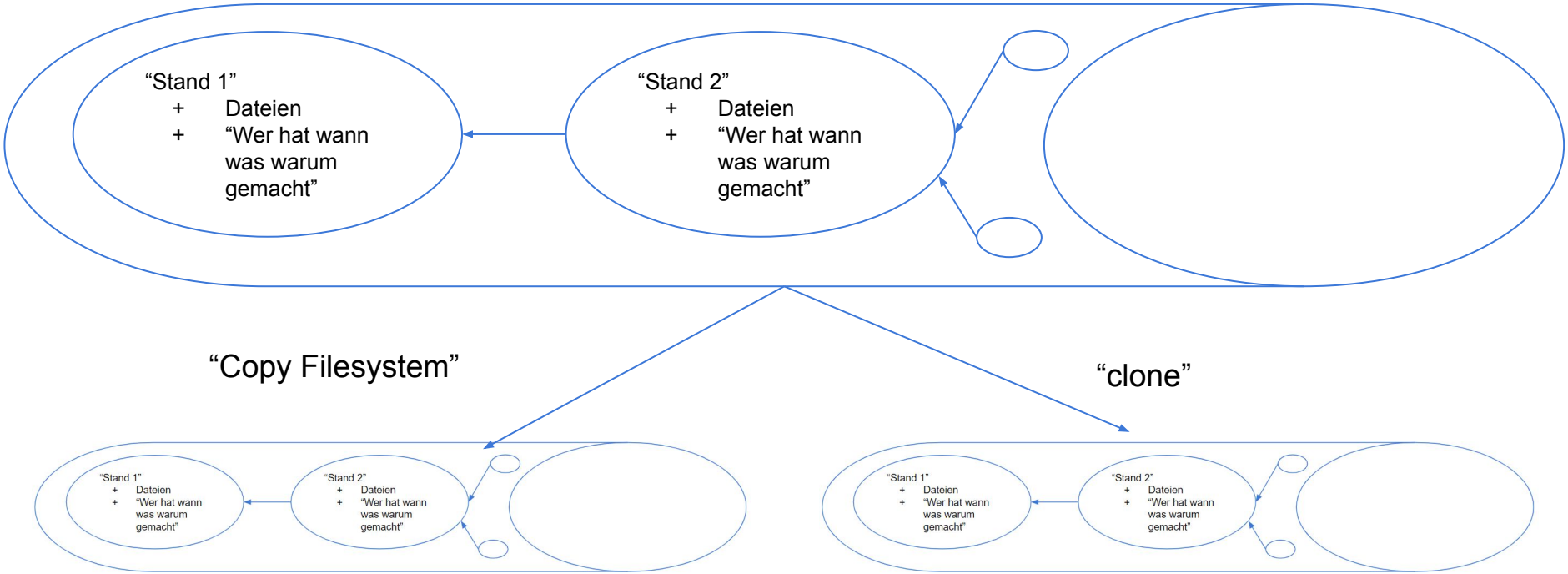
Git-Installation

- Das git-Kommando implementiert eine vollständige Umgebung
  - Ich brauche keinen Git-Server (!)
- Synchronisation mit einem Server ist möglich, aber nicht permanent nötig
- Authentifizierung / Autorisierung ist nicht vorgesehen
  - git config --global user.name = “...”
  - git config --global user.email = “...”
    - Diese Informationen werden in ihrem User-Profil abgelegt .gitconfig
- Team-Zusammenarbeit ist mit dem eigentlichen Git nicht vorgesehen

Rolle des eines Git Servers

- Produkte
  - GitHub (Microsoft)
    - Bei Ihnen **GitHub Enterprise** gehosted auf Server der FI
  - GitLab (GitLab)
  - Bitbucket (Atlassian)
- Aufgaben
  - Echte Authentifizierung und Autorisierung
  - Zentrale Ablage der Informationen
  - Team-Zusammenarbeit
    - Team-Git-Flows zum effizienten Arbeiten
      - “Pull Request”

Git-Repository



Konsequenz

- Es muss gewährleistet sein, dass die Informationsablage im Repository konsistent (“fälschungssicher”) abgelegt ist
- Lösung
  - Unabhängig von Git
  - Merkle Trees
    - Aus Informationen wird ein Hashwert berechnet
    - Historie von Informationen wird dadurch berücksichtigt, dass der Hashwert des Vorgänger-Standes in die Information mit aufgenommen wird
  - Blockchain-Technologie
- Git arbeitet seit Version 1.0 mit Blockchain

Exkurs

- Ist die Fälschungssicherheit garantiert
  - Nein
- 1000 Mitarbeitende, die pro Tag 100 Stände neu definieren und das über mindestens 100 Jahre -> Warhscheinlichkeit einer Kollision im unteren Prozentbereich
  - Für reale Softwareentwicklung absurd Unwahrscheinlich

Git First Contact

Erzeugen eines Repositories

JAVACREAM
Training
Consulting
Projectmanagement

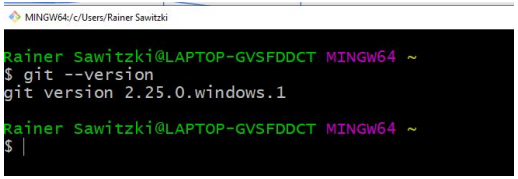
Developer

clone

GitHub Enterprise
+ (Administratives) Anlegen eines Repositories
+ Definition des Teams

sauber

Developer



training

mkdir git_first_contact
cd git_first_contact
git init

Begriffe

~~Directory~~ Git Project Directory

git init

.git
= Git-Repository

- Verzeichnisse und Dateien
  - “Workspace”

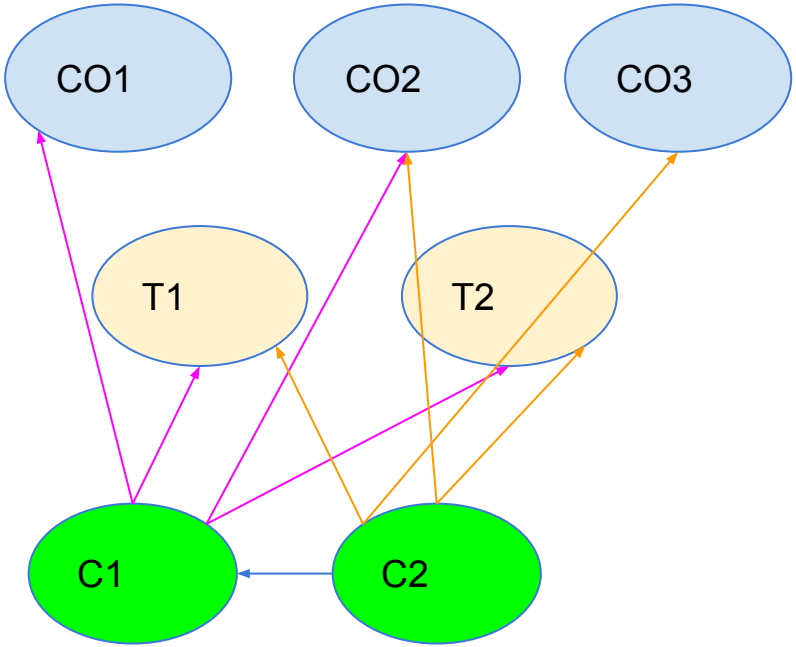
Wichtig
Eine Git-Installation enthält keinerlei Hintergrund-Dienste, Domain-Prozesse oder ähnliches

Versionsverwaltungs-Funktionen laufen nur, solange das Git-Kommando aktiv ist

Very Basic Workflow

- Anlegen / Ändern einer Datei im Workspace
  - echo Hello > content.txt
    - CHECK: ls -> content.txt
- Hinzufügen zu Git
  - git add .
    - CHECK: objects/e9/…
- Definieren eines neuen Standes
  - “Wer hat wann was warum gemacht”
    - git commit -m “”
      - CHECK
        - objects/xy, objects/ab
        - git log -> Ausgabe des commit-Hashwertes

Objekt-Typen in Git



Content-Objekte (BLOB)
- \+ Datei-Inhalt
- \+ git add

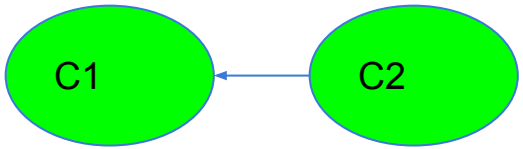
Tree-Objekte
- \+ Pfad und Datei-Attribute
- \+ git commit

Commit-Objekt
- \+ Meta-Daten
- \+ Referenzen

Exkurs

- Git arbeitet NIE mit Deltas!
  - Alle Objekte in Git sind immer atomar vollständig
- Konsequenzen
  - Braucht mehr Speicherplatz
  - Interpretation eines Objektes ist einfach bestimmbar

Ab jetzt



Bis jetzt

- git commit definiert immer ein neues Commit-Objekt mit einem Vorgänger-Commit
- Commits werden durch den Hashwert eindeutig identifiziert
- Die Information “auf welchem Stand bin ich gerade” wird durch die Angabe des Hashwertes beantwortet
  - Benutzer: “Nerd-Modus”

Alias-Namen auf Hashwerte

- Statt Nerd-Modus sprechende Namen
- 2 Situationen
  - Ein bestimmter Stand ist fix
    - v1.1
    - savepoint
    - “Heute Morgen um 9:00”
  - Eine gerade laufende Aktion
    - implement/feature1
    - development
    - Ticket-Nummer
    - “experimentiere”

Alias-Namen in Git

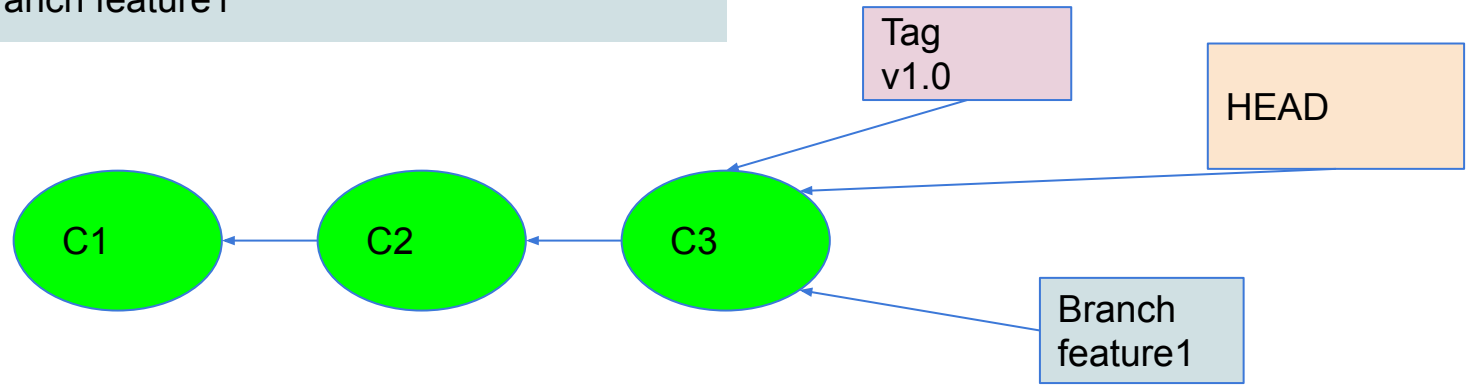
Alles folgende sind und bleiben ausschließlich Alias-Namen

Alias-Name für die aktuelle Position: HEAD

Tags und Branches sind total leichtgewichtig

git tag v1.0

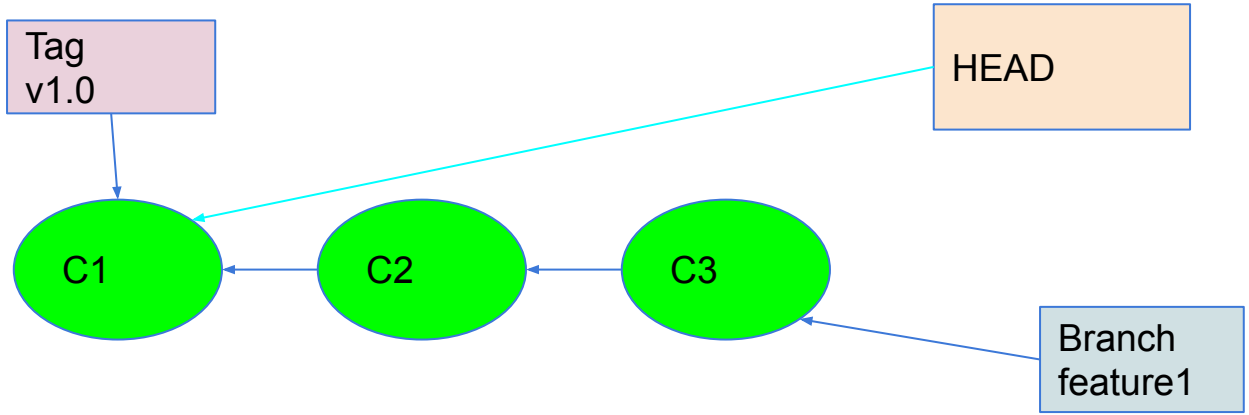
git branch feature1



Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout C1

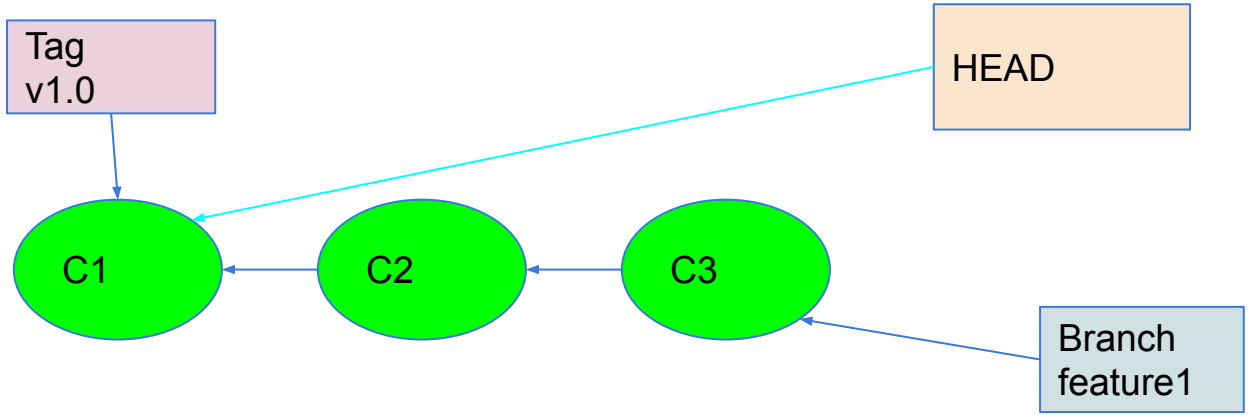


git status
+ DETACHED HEAD

Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout v1.0

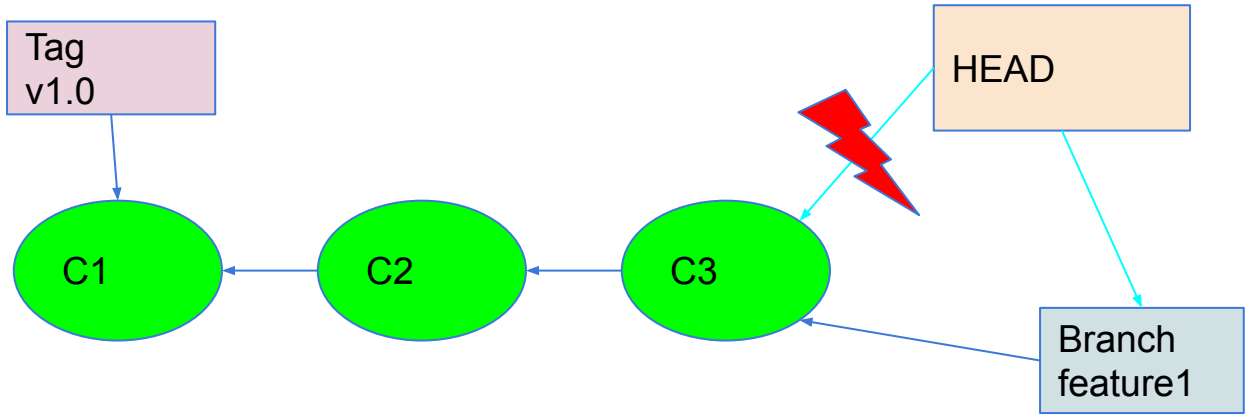


git status
+ DETACHED HEAD

Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout feature1



git status
+ Unauffällig
+ ATTACHED HEAD

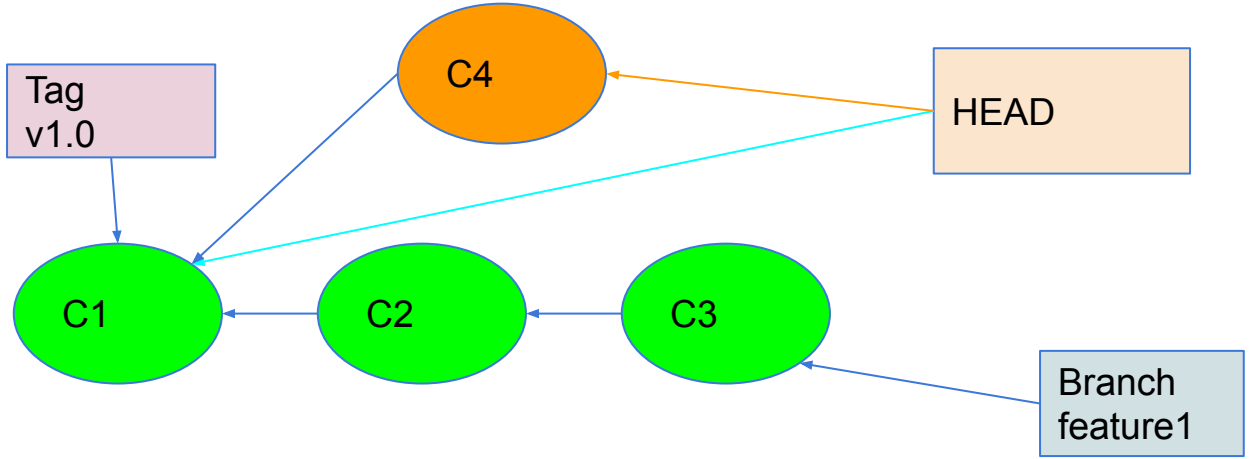
Bemerkungen zum checkout

- Ein checkout sollte nur gemacht werden, wenn git status unauffällig ist
  - Anlegen eines neuen Branches und commit
  - add + stashing -> später, Git.pdf, Online-Dokumentation
- git checkout -b new_branch …

Commit im Detached HEAD

git checkout v1.0 oder C1

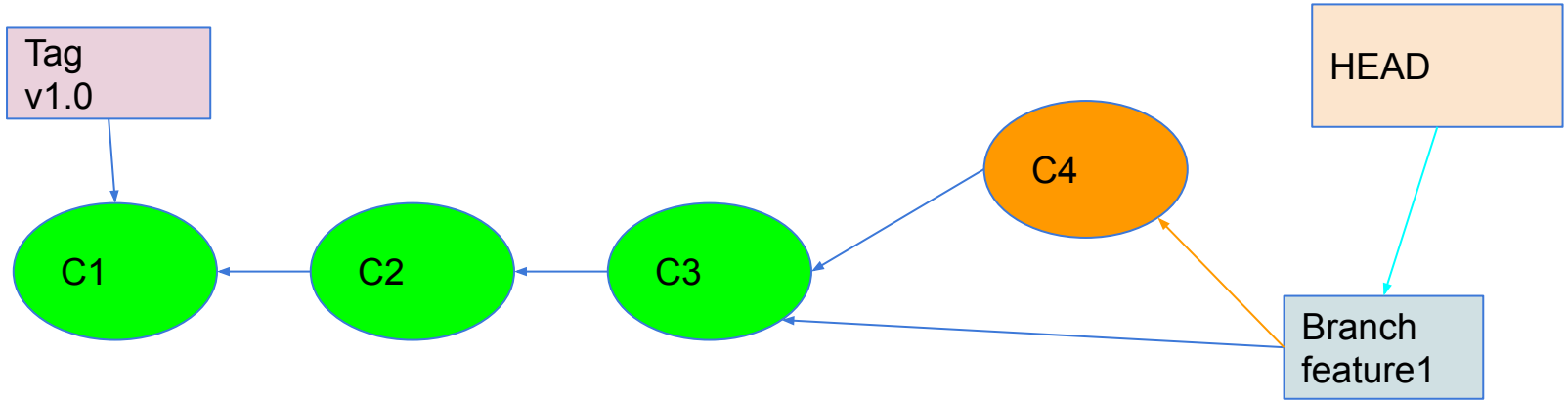
```
//Änderungen
git add .
git commit -m “”
```



Commit im Attached HEAD


```
git checkout feature1
```

```
//Änderungen
git add .
git commit -m “”
```



Dangling Objects

- Ein “Dangling Object” ist ein Objekt, das nicht über den HEAD, einen Branch oder ein Tag direkt oder indirekt referenzierbar ist
- Anzeigen
  - git fsck --unreachable --no-reflogs
- Garbage Collection
  - git reflog expire --expire-unreachable=now --all
  - git gc --prune=now

Konsolidieren von Ständen / Commit-Objekte

Strategien

- Merge
  - fast-forward
  - recursive
- Rebase
- Interactive Rebasing
- Cherry Pick
  - Aktuell in der Git Community eher als “deprecated” betrachtet
- Patching
  - Problemsituation: Internes Repo -> Kunden -> Kunden-Repo
  - Erstellen eines Patches und Einspielen in der Zielumgebung

Saubere Ausgangsbasis für die Übungen

- setup-branches.sh auf Referenten-GitHub

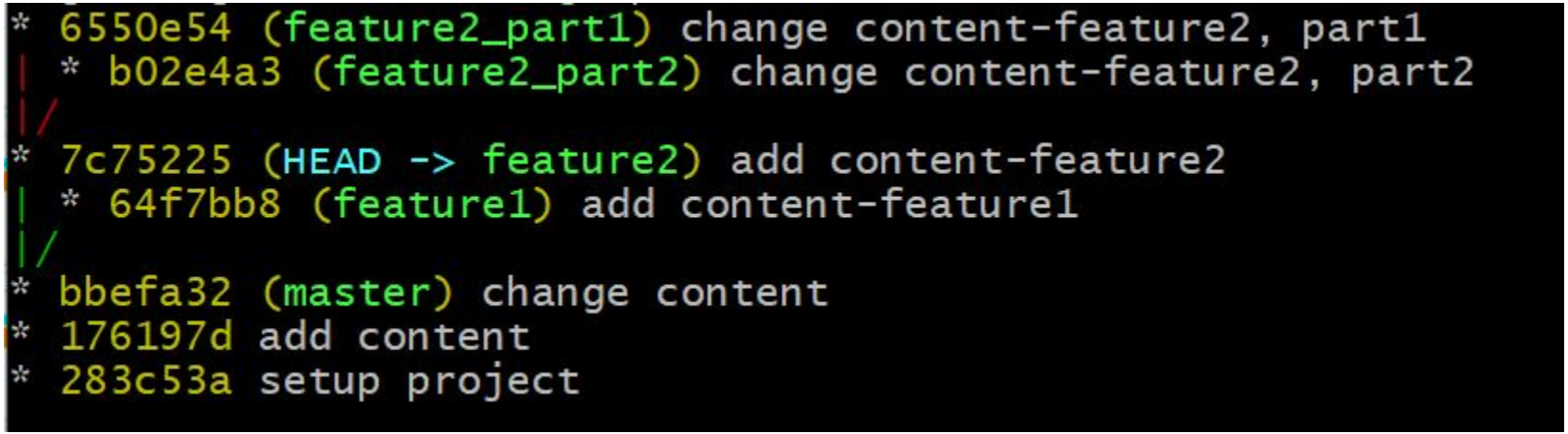
Merge-Plan

- Ziel
  - Integration von feature1 und feature2 in den master und damit ein Fortschritt der allgemeinen Entwicklung im Projekt
  - Es muss alles so nachvollziehbar sein, wie es geschehen ist
- Plan (subjektiv)
  - feature2 wird konsolidiert mit den beiden parts
  - anschließend kommt feature1 dazu
  - zum schluss dann der master

git status

- muss unauffällig

git checkout feature2



git merge feature2_part1

JAVACREAM
Training
Consulting
Projectmanagement

```
Updating 7c75225..6550e54
Fast-forward
 content-feature2.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
* 6550e54 (HEAD -> feature2, feature2_part1) change content-feature2, part1
| * b02e4a3 (feature2_part2) change content-feature2, part2
|/
* 7c75225 add content-feature2
| * 64f7bb8 (feature1) add content-feature1
|/
* bbefa32 (master) change content
* 176197d add content
* 283c53a setup project
```

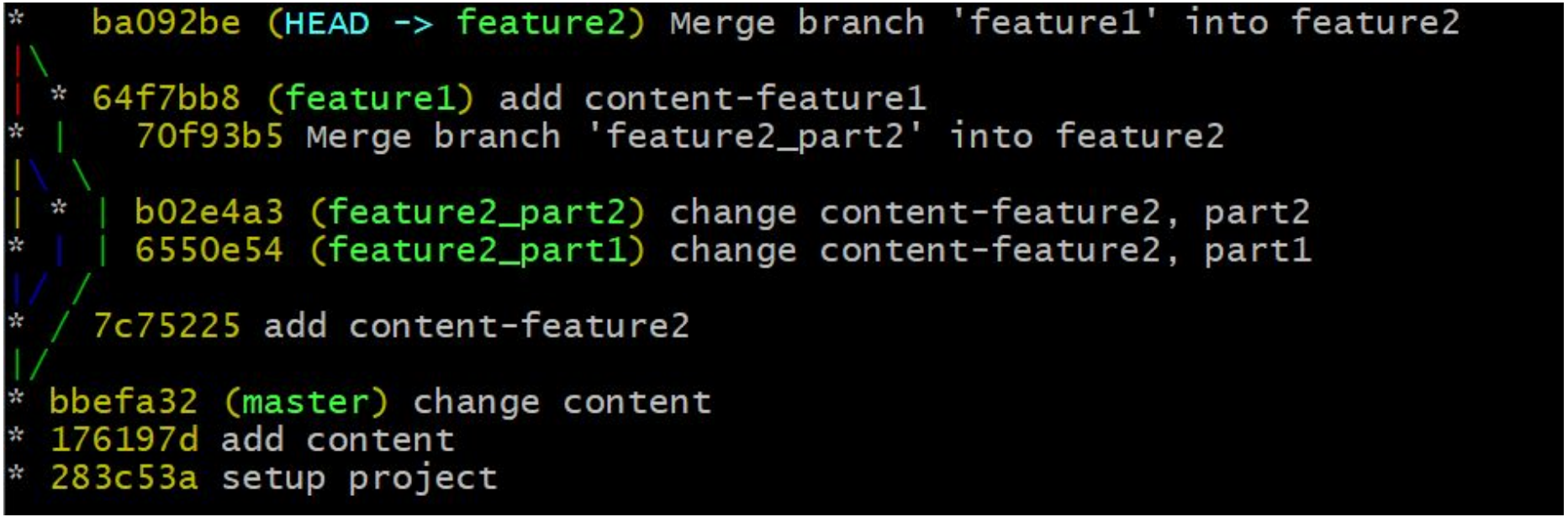
git merge feature2_part2

JAVACREAM
Training
Consulting
Projectmanagement

```
$ 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.
```

```
*   70f93b5 (HEAD -> feature2) Merge branch 'feature2_part2' into feature2
|\
| * b02e4a3 (feature2_part2) change content-feature2, part2
* | 6550e54 (feature2_part1) change content-feature2, part1
|/
* 7c75225 add content-feature2
| * 64f7bb8 (feature1) add content-feature1
|/
* bbefa32 (master) change content
* 176197d add content
* 283c53a setup project
```

git merge feature1



git checkout master git merge --no-ff feature2


```
*   9eec2ba (HEAD -> master) Merge branch 'feature2'
|\
| *   ba092be (feature2) Merge branch 'feature1' into feature2
| |\
| | * 64f7bb8 (feature1) add content-feature1
| |/
|/|
| *   70f93b5 Merge branch 'feature2_part2' into feature2
| |\
| | * b02e4a3 (feature2_part2) change content-feature2, part2
| * | 6550e54 (feature2_part1) change content-feature2, part1
| |/
| * 7c75225 add content-feature2
|/
* bbefa32 change content
* 176197d add content
* 283c53a setup project
```

Details

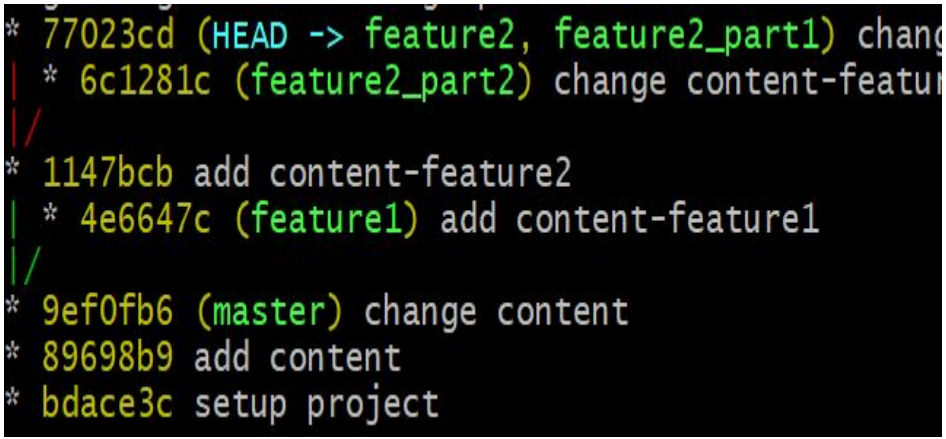
- Ich persönlich würde diesen merge-Plan etwas modifizieren, und das merging auf einem temporären Branch durchführen
- merge immer mit no-ff
  - Empfehlung der Git-Community
- Es fehlte im Beispiel das Arbeiten mit Tags
- Cleanup fehlt
  - feature-Branches löschen
  - Ersetzen durch Tags
    - feature1-Branch -> finish/feature1

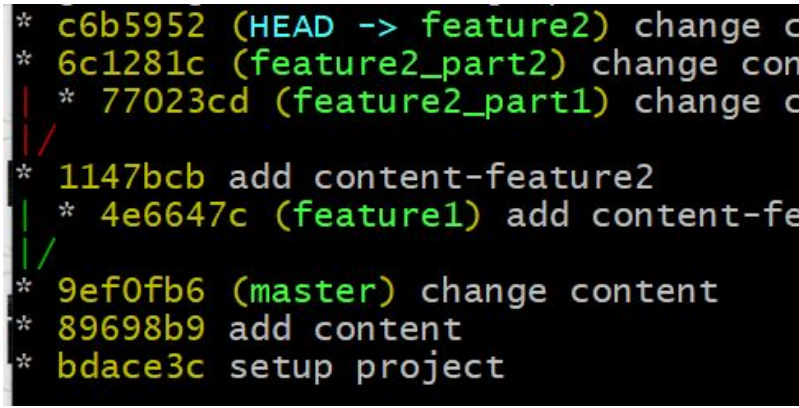
Rebase-Plan

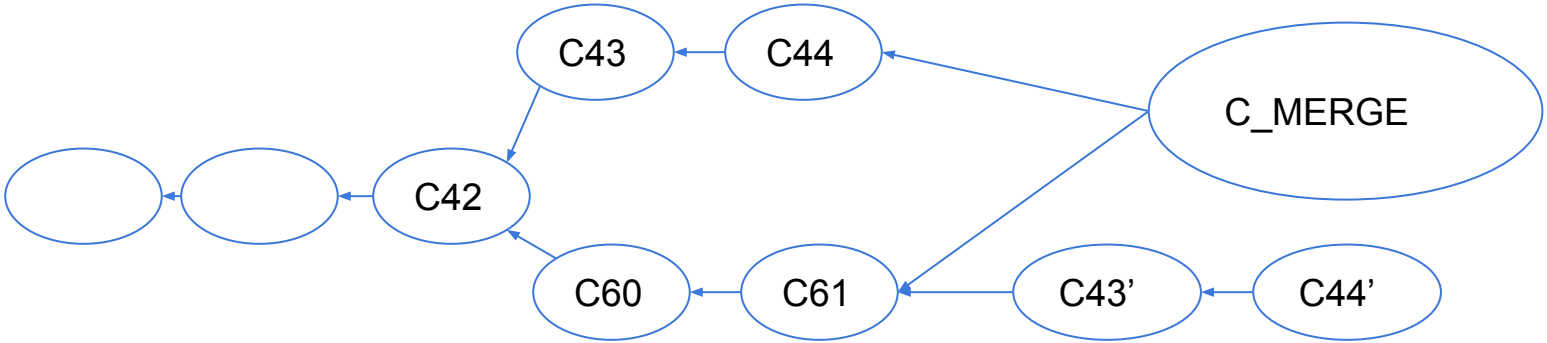
- Ziel
  - Integration von feature1 und feature2 in den master und damit ein Fortschritt der allgemeinen Entwicklung im Projekt
  - ~~Es muss alles so nachvollziehbar sein, wie es geschehen ist~~
  - es soll eine stringente, nachvollziehbare sequentielle Historie entstehen
    - Geschichtsfälschung
- Plan (subjektiv)
  - feature2 wird konsolidiert mit den beiden parts
  - anschließend kommt feature1 dazu
  - zum schluss dann der master

Step by step

- git checkout feature2
- git merge feature2_part1
  - fast forward
- git rebase feature2_part2
-






C43
C44
C_MERGE
C42
C60
C61
C43’
C44’

Step by step continued

- git rebase feature1
- git checkout master
- git merge feature2
- Cleanup

```
$ git log --oneline --graph --all
* 5f205d5 (HEAD -> master) change content-feature2, part1
* ad7c40d change content-feature2, part2
* 3d89739 add content-feature2
* 4e6647c add content-feature1
* 9ef0fb6 change content
* 89698b9 add content
* bdace3c setup project
```

Interactive Rebasing

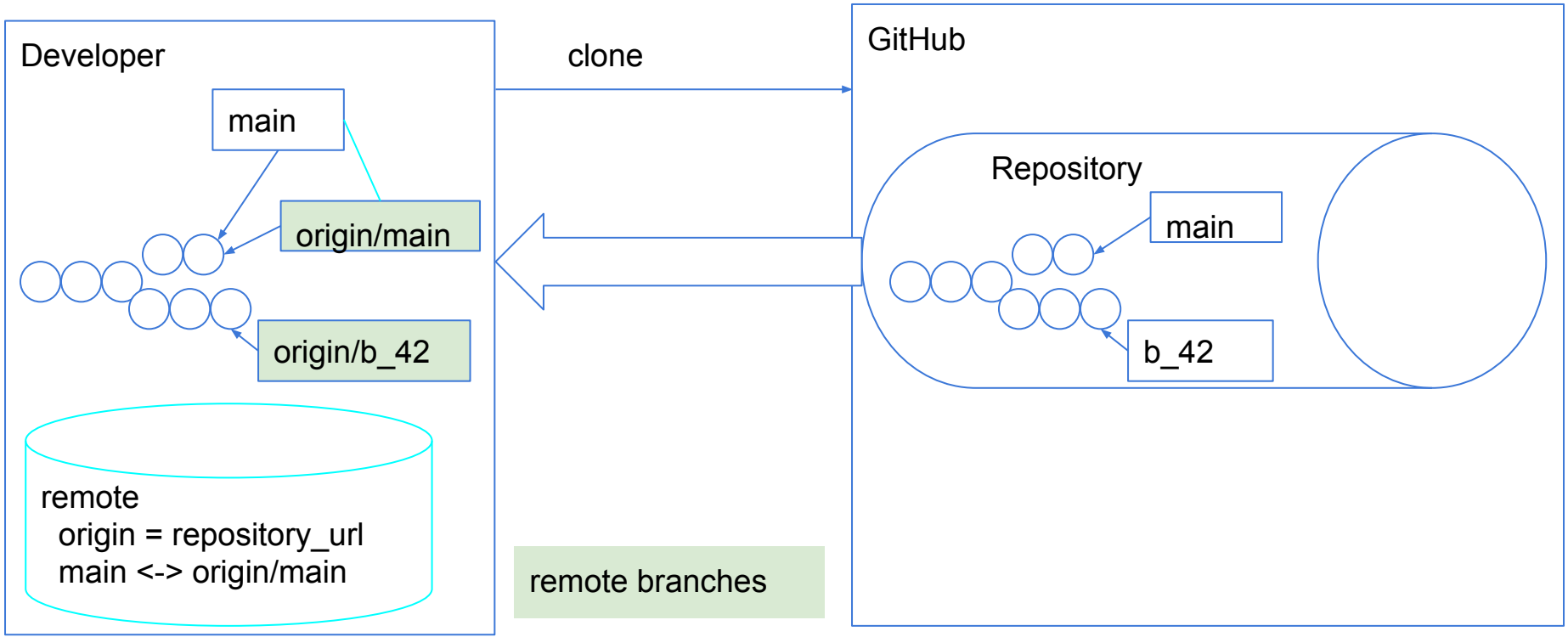
- git rebase -i <hash>

```
pick 89698b9 add content
pick 9ef0fb6 change content
pick 4e6647c add content-feature1
pick 3d89739 add content-feature2
pick ad7c40d change content-feature2, part2
pick 5f205d5 change content-feature2, part1

# Rebase bdace3c..5f205d5 onto 3d89739 (6 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
```

Remote Repositories

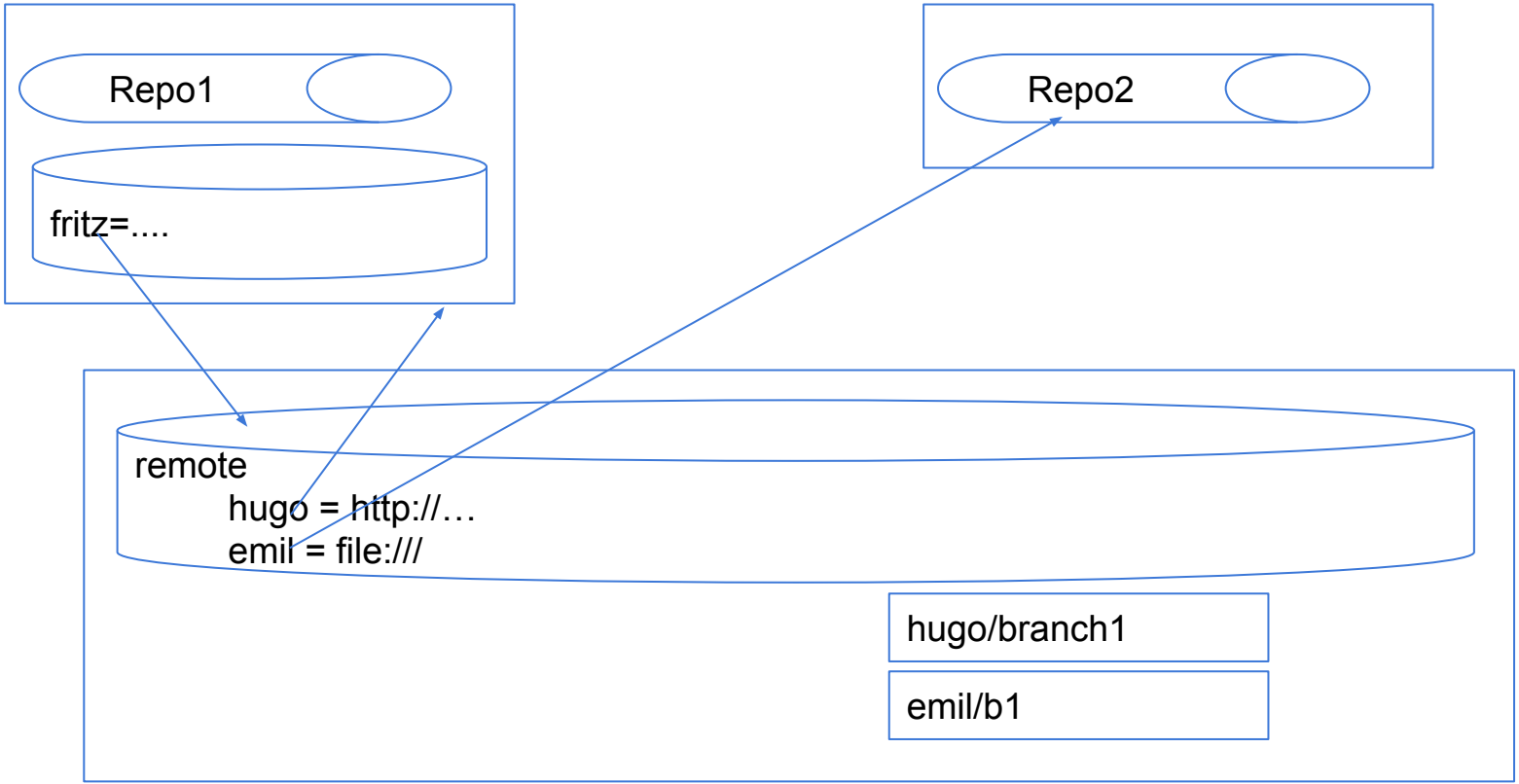
Cloning


Developer
main
origin/main
origin/b_42
remote
origin = repository_url
main <-> origin/main
clone
GitHub
Repository
main
b_42
remote branches

Remote Branches

- Haben einen Prefix, der einem Alias-Namen auf die Remote Konfiguration entspricht
- Ein Remote Branch muss garantiert einem auf dem Server-Repository vorhandenen Zustand entsprechen
  - Remote Branch entspricht garantiert dem Zustand, der beim Clone auf Server-Seite vorhanden war
  - Aus Sicht des Developer-Repositories ist ein Remote Branch immer “read-only”
    - Dies wird in Git umgesetzt durch

Remote oder Distributed?



ToDo

- Verifizieren Sie den Zugriff auf den FI-GitHub-Server intern

fetch

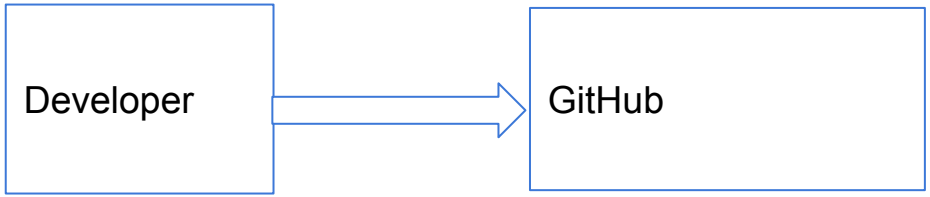
- Bringt zur Ausführungszeit einen remote branch wieder in sync mit dem Server-Branch
  - fetching bewegt einen remote branch
  - fetch geht immer
    - fast-forward merge im remote branch

pull

- fetch
- anschließend gleich ein merge mit dem konfigurierten lokalen Branch
  - Wichtig: Das kann natürlich knallen = merge Konflikte
- Meinung der Git Community
  - Eigentlich sollte hier der pull --rebase der Standard sein

git push

- Änderungen vom Developer zum GitHub



- Ein push schickt die Daten zum GitHub (stimmt)
  - ~~These: Detaillierter~~
    - ~~der lokale branch aktualisiert den remote branch~~
    - ~~die Änderungen werden zum GitHub geschickt~~
    - ~~dort werden die Änderungen dann den Server-Branch bewegen~~
  - Alle Änderungen des lokalen branches werden zum server gesendet
  - Server prüft: Ist ein fast -forward auf dem Server möglich?
    - Ja: Server Branch wird bewegt, dann wird der Erfolg zum Developer gesendet und zum Schluss der Remote-Branch bewegt

Git Flow: “Single Developer Flow”

- Anlegen eines Repositories auf dem Server
  - Solche Repositories können auch private sein…
- main-Branch
- Clone des Repositories
- Arbeiten auf dem main
  - das ist ungewöhnlich, aber in diesem Flow pragmatisch und OK
- Auf der Developer-Maschine
  - save, add ., commit -m “” andauernd
  - Commit & Push
    - Vorteil: Alle Commits sind sofort auf dem Server abgelegt
    - Nachteil: Historie ist schon sehr detailliert
      - rebase -i -Kandidat

ToDo

- Spielen Sie mit dem “Single Developer Flow”
-

Konsequenz von rebase, interactive rebase

- Folgende Situation
- Sie pushen einen Branch auf GitHub
  - Das ist eine Veröffentlichung
  - und damit in der Regel für alle sichtbar / clonebar

ToDo

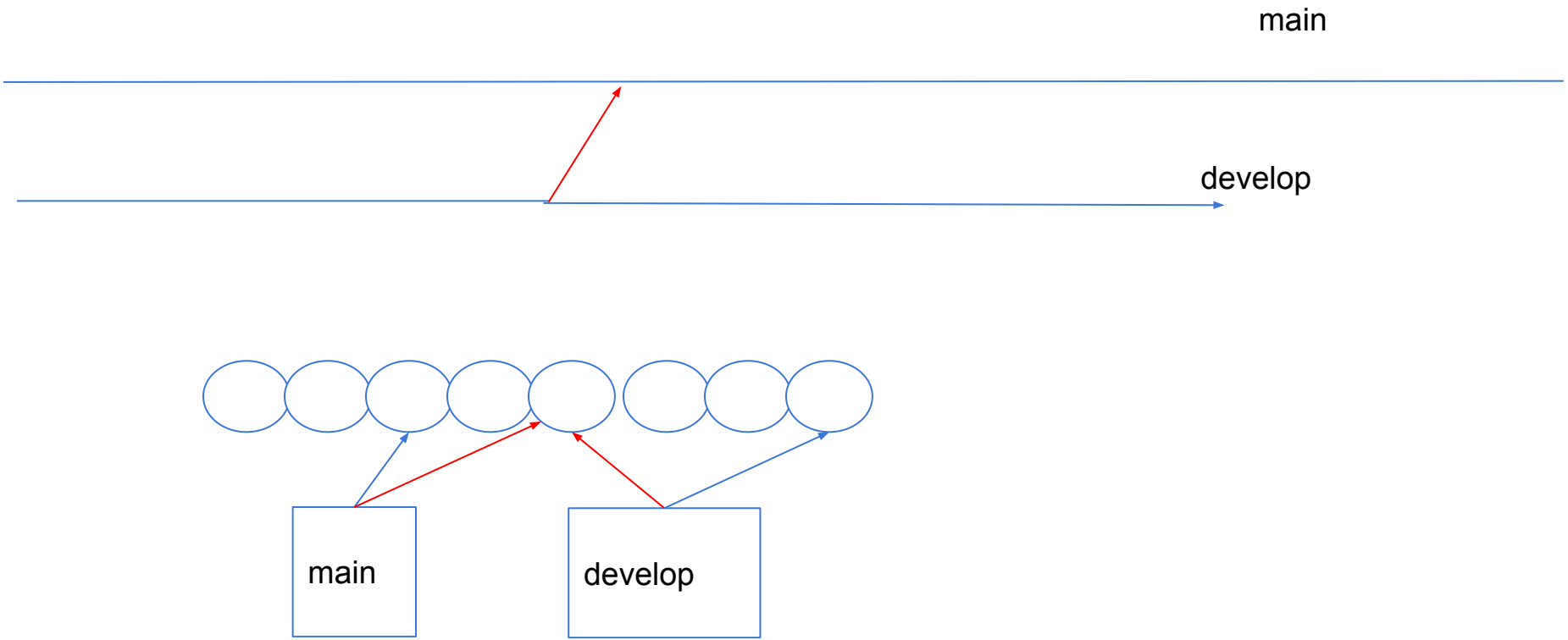
- Kombinieren Sie den push mit rebase
  - rebasing auf Developer erfordert ein push -f
  - Machen Sie sich die Konsequenzen klar

Team-Zusammenarbeit

Team-fähige Git Flows

- GitHub Flow
- Atlassian Git Flow
- …
- Hinweis:
  - Eine Recherche nach diesen Flows im Internet wird immer wieder Treffer liefern, “XYZ ist völliger Unsinn”
  - Nehmen Sie doch diese Flows als Grundlage für ihre eigenen, angepassten Flows
- Eine Gemeinsamkeit aller Team-Flows
  - Es gibt einen “langlebigen Branch”
  - Änderungen eines Developers erfolgen **immer unverhandelbar** in einem sogenannten feature-Branch
    - Seien Sie gewiß: der langlebiege Branch auf dem Server ist protected

Zu “langlebiegen Branches”



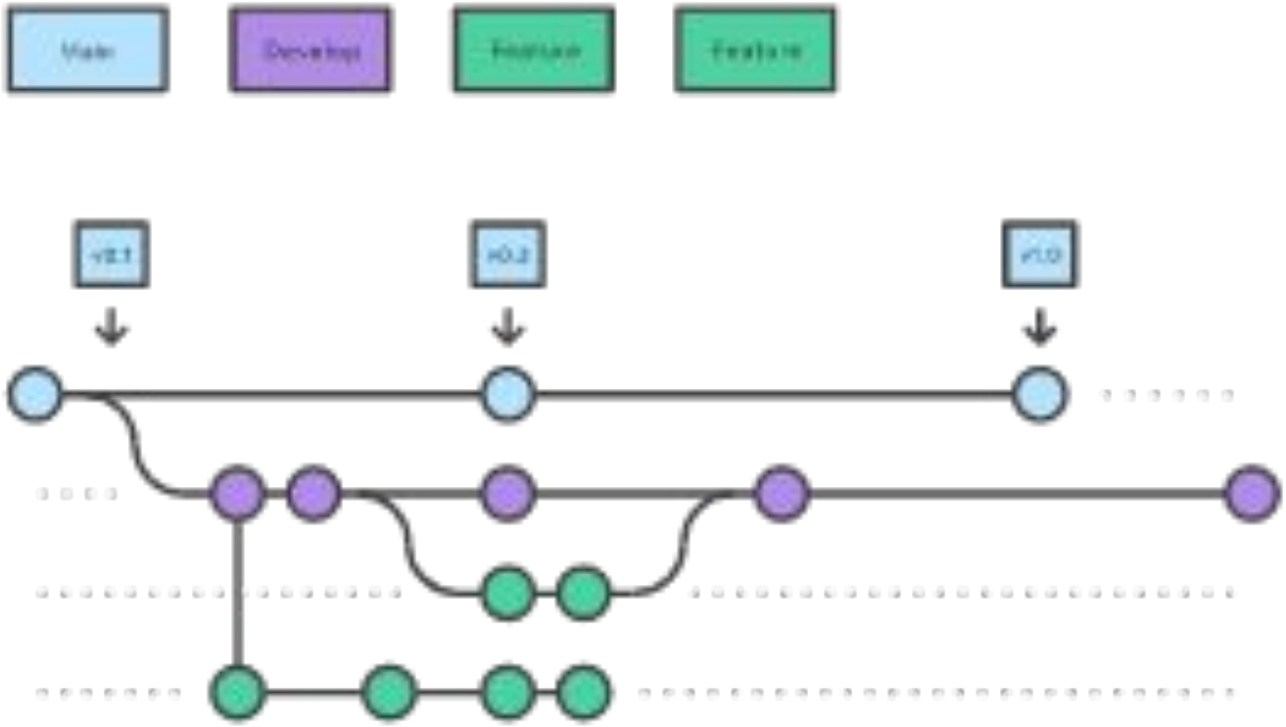
Zum GitHub Flow

- Jetzt erst mal ohne Pull Request
- n Developer und n feature-Branches
- developer m ist fertig -> Mail ans Team: “Bin fertig”
- Integrator
  - clone, check, rebase push master
  - Meldung ans Team: main hat sich geändert
- developer m
  - löscht Repo, ist ja fertig
- Andere developer
  - checkout main, pull, was hat sich geändert
  - checkout feature, rebase main, Konflikte beheben
  - git push -f
- Zum Schluss (alle features fertig: Tag auf main mit einer Versionsnummer)

Zum GitHub Flow

- Mit Pull Request
- n Developer und n feature-Branches
- developer m ist fertig -> Erzeugt einen Pull Request
- Integrator
  - clone, check, rebase push master
  - Meldung ans Team: main hat sich geändert
- developer m
  - löscht Repo, ist ja fertig
- Andere developer
  - checkout main, pull, was hat sich geändert
  - checkout feature, rebase main, Konflikte beheben
  - git push -f
- Zum Schluss (alle features fertig: Tag auf main mit einer Versionsnummer)

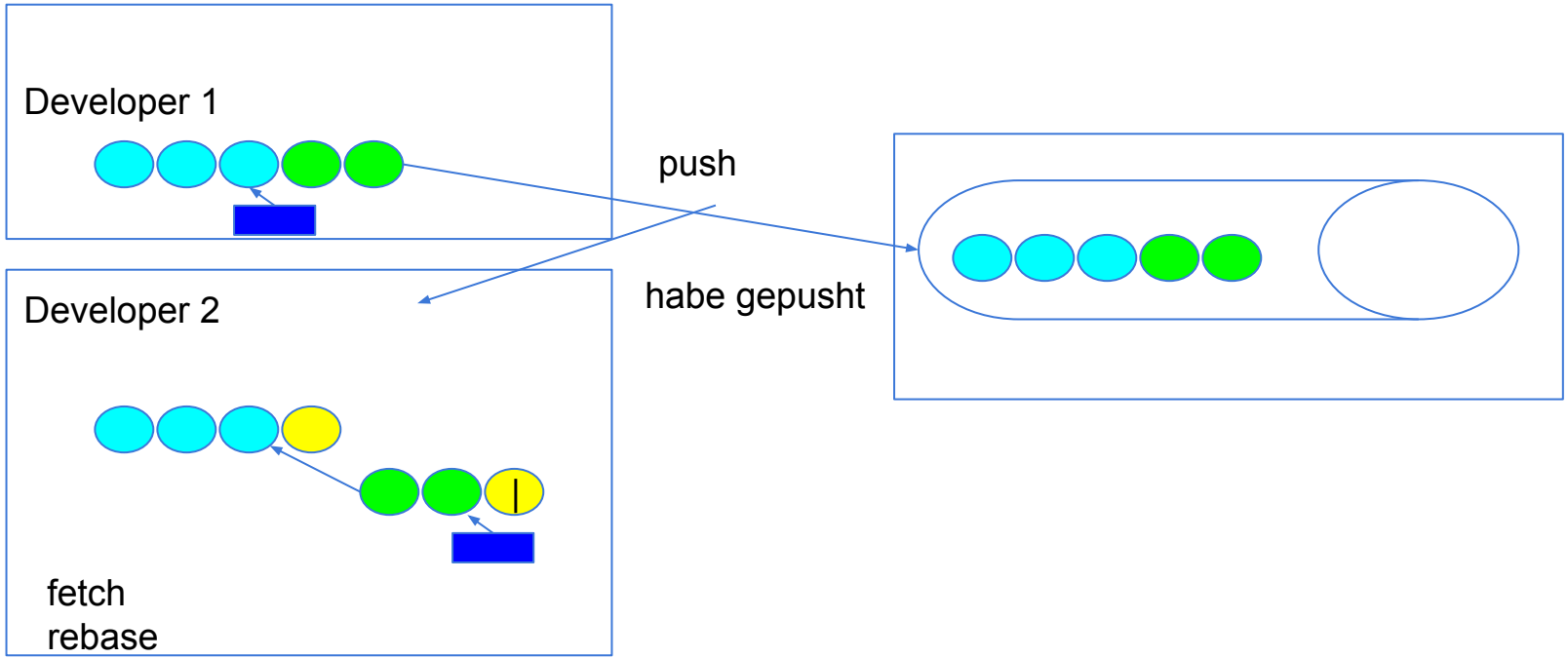
Atlassian Git Flow



Feature-Entwicklung im Team

- Was passiert, wenn ein fachliches Issue = Feature von mehreren Team-Mitgliedern gleichzeitig bearbeitet werden muss?
- Lösung 1:
  - Wir machen Sub-Feature–Feature-Requests
  - Das Team hat einen “Feature-Integrator”
  - Pull-Requests können benutzt werden
- Lösung 2:
  - Vorausetzung: Die Kommunikation der beteiligten Team-Mitgleider ist sehr direkt
  - Der einzelne feature-Request genügt

Exemplarisch



Exemplarisch