Git-Schulung

–Firma–

Valentin Hänel, Julius Plenz

–Datum–

Ablaufplan: Tag 1

- **Tag 1, vormittags:** Einführung in Git
  - Ein Repository erstellen
  - Die wichtigsten Git-Kommandos
  - Der Index
  - Git-Interna
- **Tag 1, nachmittags:** Mit Branches arbeiten
  - Branches erstellen und wieder zusammenführen
  - Änderungen rückgängig machen
  - Merge-Konflikte lösen

Ablaufplan: Tag 2

- **Tag 2, vormittags:** Kollaboration
  - Rebase
  - Parallele Entwicklung mit Git
  - Workflows
  - Praktischer Teil: Zusammen ein Projekt erstellen
- **Tag 2, nachmittags:** Erweitertes Git
  - Hilfreiche Git-Kommandos
  - Fehlersuche
  - Automatisierung
  - Evtl. Auffangbecken für noch nicht geklärte Fragen

Übersicht

Session 3: Rebase & Remotes

- Rebase
- Workflows
- Remotes verwalten
- Übung: Gemeinsam ein Projekt erstellen

Übersicht

Rebase: Auf eine neue Basis stellen

- **Rebase**: Einen Branch auf eine »neue Basis« stellen.

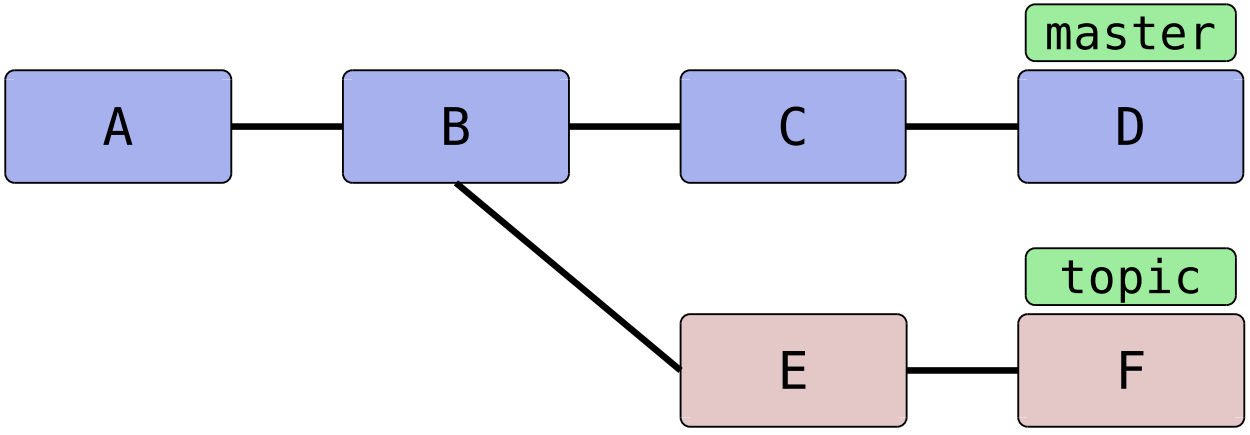
`master` als neue Basis für *`topic`*

```
git checkout topic
git rebase master
```

Alternativ

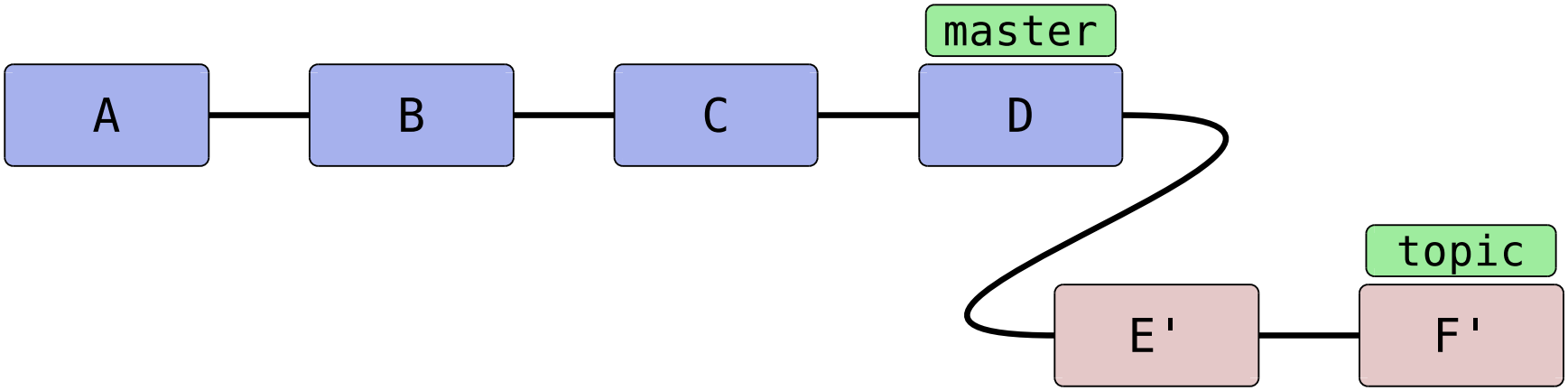
```
git rebase master topic
```

Vor dem Rebase



- *topic* soll auf der neusten Version von master basieren

Nach dem Rebase



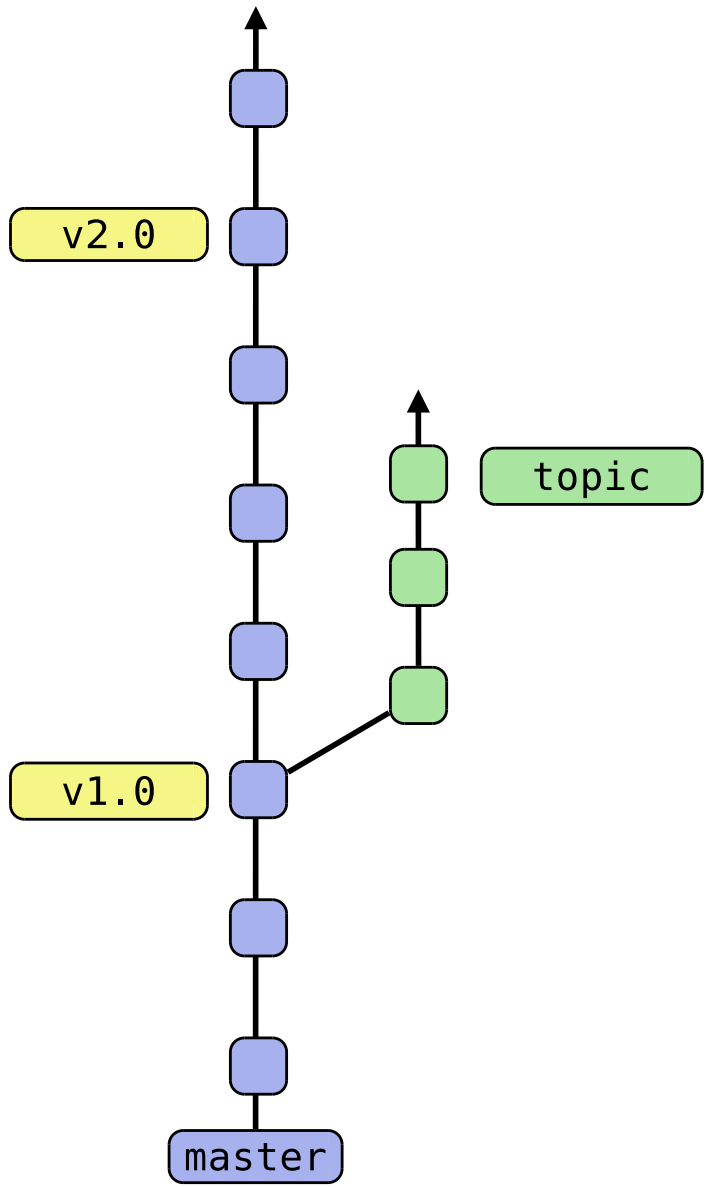
- `git rebase master` *topic*

Rebase: Wozu?

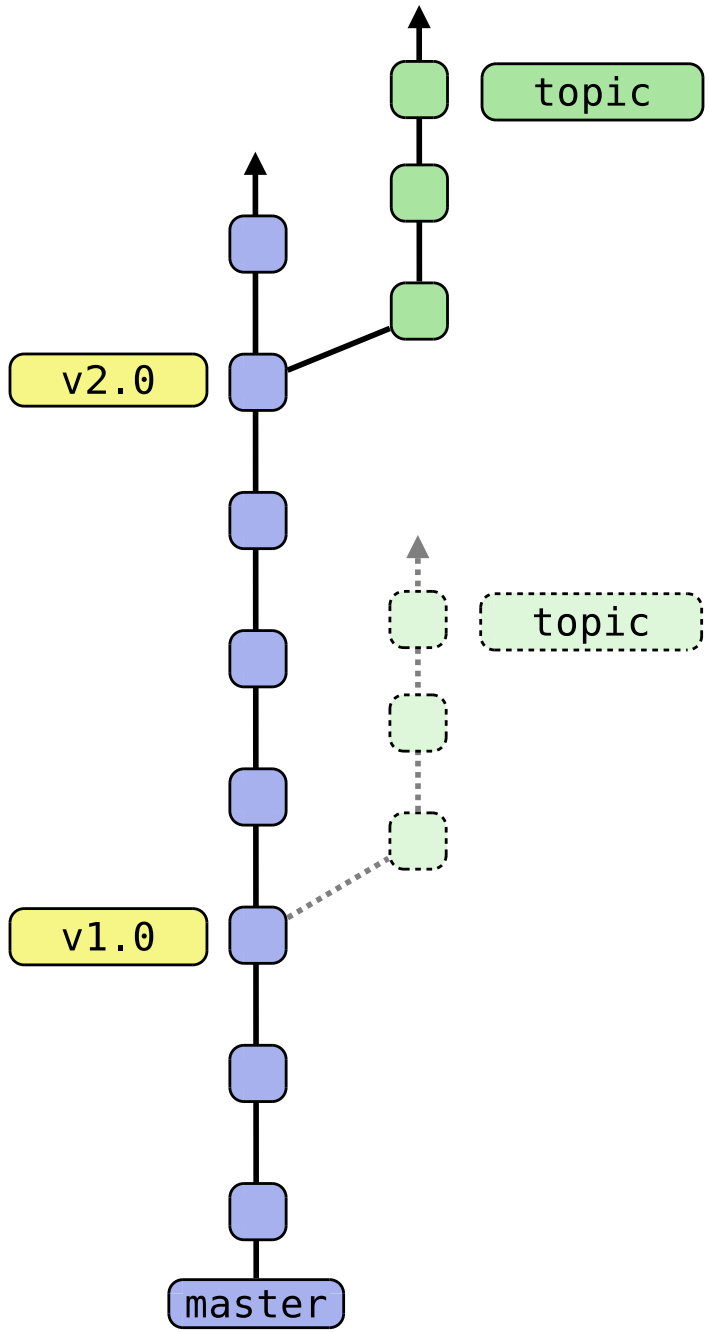
Mit einem Rebase kann man »die Geschichte umschreiben« – aber wozu?

- Kontinuierliche, parallele Entwicklung
- Entwicklungsgeschichte linearer und übersichtlicher machen
- Einen Teil der Entwicklung auf einen anderen Branch transplantieren
- Patch-Stacks verwalten

Anwendungsfall visualisiert



Anwendungsfall visualisiert



Rebase: Kontinuierliche Entwicklung

1. Die Entwicklung eines Features wird begonnen
2. Teile des Features werden in das Release übernommen
3. Feature-Branch soll nun wieder auf dem aktuellsten Release basieren
4. Entwicklung geht weiter, das Feature wird fertig gestellt und released

1. Die Entwicklung eines Features wird begonnen

Januar (`v1.0`): Das Ausgabe-Handling soll umgeschrieben werden

Neuen Branch erstellen, in dem das Feature entwickelt wird

```
git checkout -b rewrite-io v1.0
```

2. Teile des Features werden in das Release übernommen

Februar (`v1.1`): Funktionen wurden umbenannt und vereinheitlicht, Interna sind im Wesentlichen gleich geblieben

Teile der Commits werden übernommen nach `v1.1`

```
git merge/cherry-pick ...
```

3. Feature-Branch soll nun wieder auf dem aktuellsten Release basieren

Die geplanten neuen Ausgabe-Routinen benötigen eine Funktionalität, die erst in der neusten Version implementiert wurden

Der Feature-Branch wird auf eine neue Basis gebracht

```
git rebase v1.1 rewrite-io
```

4. Entwicklung geht weiter, das Feature wird fertig gestellt und released

März (`v1.2`): Die Ausgabe-Routinen sind fertig und werden eingebunden

Der Feature-Branch wird gemerged

```
git merge rewrite-io
```

Rebase: Entwicklungsgeschichte linearisieren

- Parallel stattfindende Entwicklung muss nicht notwendigerweise als solche aufgezeichnet werden
- Für Fehlersuche sowie Code-Review sollte die Versionsgeschichte logisch »gegliedert« sein
  - Erst den Protokoll-Stack schreiben, dann Funktionen, die ihn verwenden (selbst wenn beides gleichzeitig entwickelt wird)
- Achtung: Sinnlose Linearisierung ist nicht wünschenswert!
  - Zusammenhängende (aber zeitlich weit auseinanderliegende) Commits sind nicht mehr einfach als solche zu identifizieren

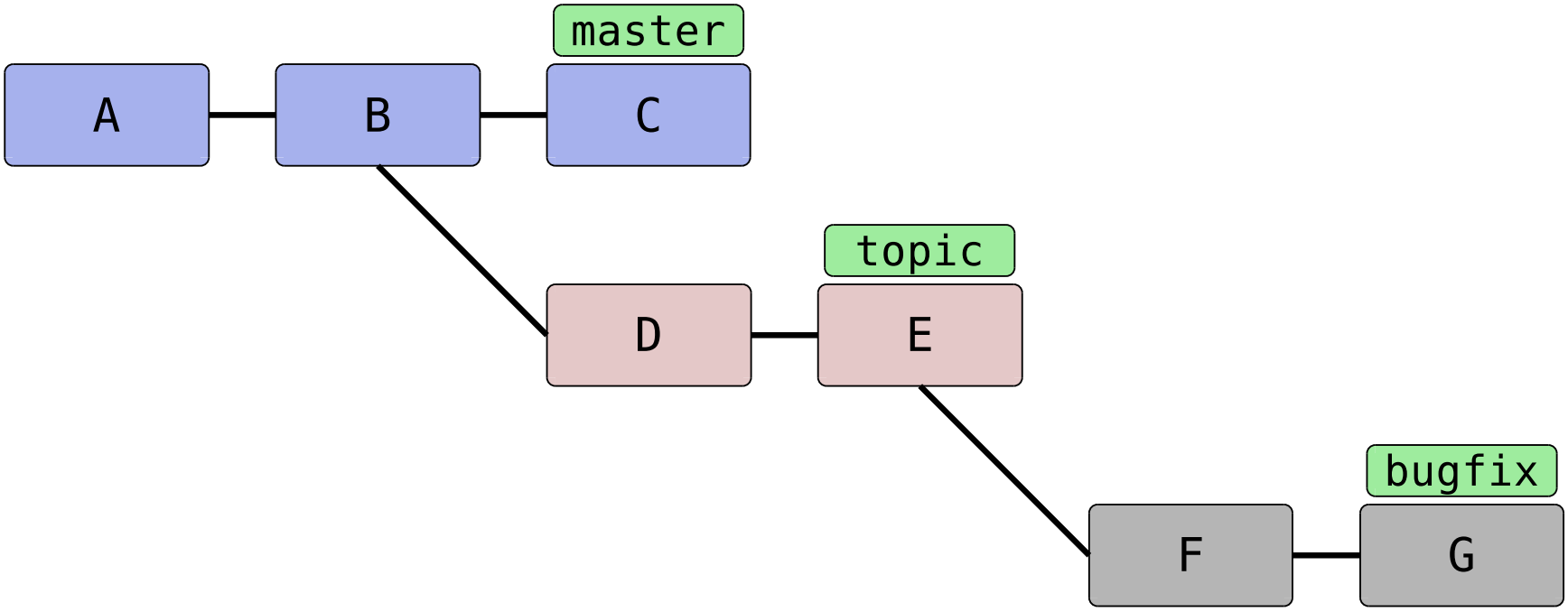
Rebase Onto: Commits »transplantieren«

- Aufbau der Branches: `master` ← `topic` ← `bugfix`
- `bugfix` soll nun direkt auf `master` basieren, ohne dass die Commits von `topic` dupliziert werden
- Lösung: `git rebase --onto`

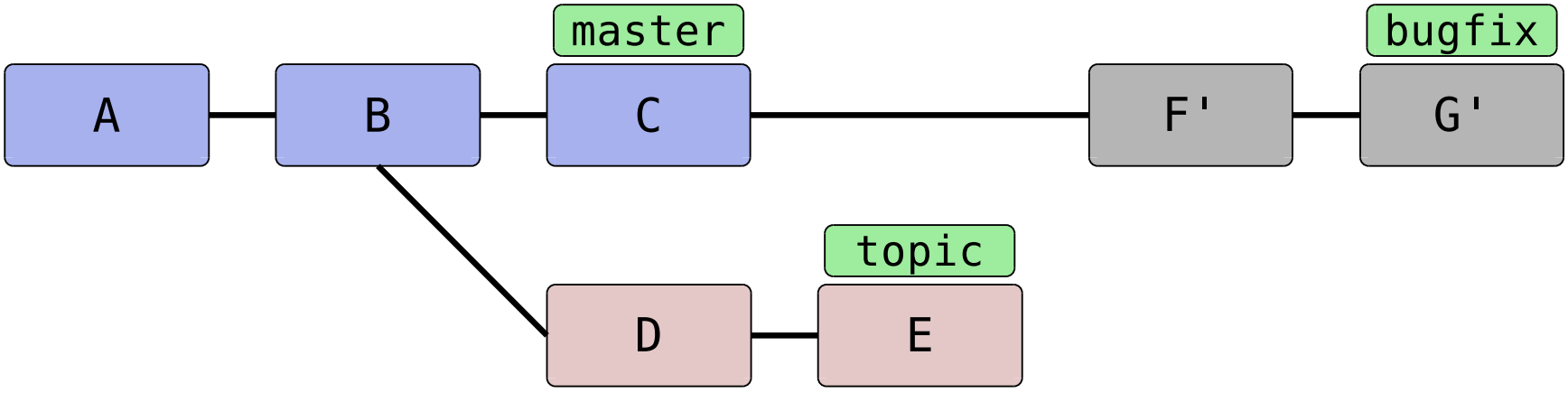
Teile eines Branches »transplantieren«

```
git rebase --onto master topic bugfix
```

Rebase Onto – Vorher



Rebase Onto – Nachher



Upstream Rebase

- **Wichtig:** Sie sollten ***niemals*** Commits aus einem bereits veröffentlichten Branch durch `git rebase` verändern!
  - Branches (z. B. anderer Entwickler), die darauf basieren, stehen nun »elternlos« da – so etwas zu reparieren ist teilweise mühselig.
- **Daher: Nur unveröffentlichten Code rebasen!**
  - `git rebase origin/master`
  - `git rebase v1.1.23`
- Eine Ausnahme bilden natürlich Vereinbarungen zwischen den Entwicklern
  - »Auf die Branches `test/*` soll niemand seine Arbeit aufbauen«

Übung: Rebase

1. Erstellen Sie auf einem Branch, der von `master` abgeht, einige Commits. Erstellen Sie auf `master` Commits. Bauen Sie den anderen Branch von neuem auf `master` auf (*rebase*)
2. Schaffen Sie eine geeignete Situation, in der `rebase --onto` hilfreich ist
3. Transplantieren Sie die drei neusten Commits auf Ihrem aktuellen Branch in einen anderen Branch (*rebase onto*)
4. Erstellen Sie mehrere Commits auf mehreren Branches, und linearisieren Sie alle Commits aller Branches
5. Übernehmen Sie aus einem Branch einen Commit per `cherry-pick` in den `master` und bauen Sie den Branch dann neu auf `master` auf. Was passiert mit dem Commit, den Sie übernommen hatten?

Rebase: Interaktiv

Mit einem interaktiven Rebase kann man die Entwicklungsgeschichte nach Belieben anpassen (»rewrite history«).

- Der Rebase-Prozess wird zwischendurch angehalten
- Commits können dann beliebig verändert werden (`edit`)
  - Reihenfolge der Commits vertauschen
  - Commits verwerfen
  - Commit-Message abändern (`reword`)
  - Zwei oder mehr Commits zusammenfassen (`squash`, `fixup`)
  - Einen Commit in kleinere aufteilen
- Danach wird der Rebase-Prozess fortgesetzt

Rebase-Kommandos

Interaktives Rebase starten

```
git rebase -i basis
git rebase -i HEAD~7
git rebase -i master
git rebase -i origin/master
```

Interaktives Rebase abbrechen

```
git rebase --abort
```

Nach einer Änderung fortfahren

```
git rebase --continue
```

Commit überspringen

```
git rebase --skip
```

Interaktiver Rebase: Beispiel

In einem Commit (1069e2e) steckt ein Passwort, was nicht ins Repository wandern sollte.

Commit abändern

`git rebase -i 1069e2e^`
... beim Commit `pick` durch `edit` ersetzen, `rebase` hält dort an
`vim` *datei* $\rightarrow$ Passwort entfernen!
`git add` *datei*
`git commit --amend -C HEAD`
`git rebase --continue`

- Da nun 1069e2e geändert wurde, ist die SHA1-ID nicht mehr die gleiche
- Als Folge sind auch die Commit-IDs aller darauf aufbauenden Commits verändert, weil sie nun die geänderte ID referenzieren (»ripple effect«)

Einen Commit in mehrere aufteilen

Interaktives Rebase starten

```
git rebase -i master
```

... beim gewünschten Commit `pick` durch `edit` ersetzen

Commit aufteilen

```
git reset HEAD^
git add -p
git commit -m 'Erster Teil'
git add ...
git commit -m 'Zweiter Teil'
```

Rebase weiterlaufen lassen

```
git rebase --continue
```

»Rebase early, rebase often«

- Fehler passieren immer – spätere Commits beheben das Problem
  - Haupt-Commit und Verbesserungen hintereinander anordnen und per `fixup` zu einem Commit verschmelzen
- So erhält man eine sinnvolle Entwicklungsgeschichte ohne ablenkende Commits (fünf mal „einen kleinen Fehler behoben“)
- Ziel ist es, die Entwicklung schlüssig darzustellen
  - Nicht zu viele mikroskopisch kleine Commits
  - Nicht zu viele riesengroße Commits
- Erst dann sollten die Commits veröffentlicht werden

Achtung! Auch hier gilt wieder: Nur unveröffentlichte Commits verändern!

Übung: Rebase interaktiv

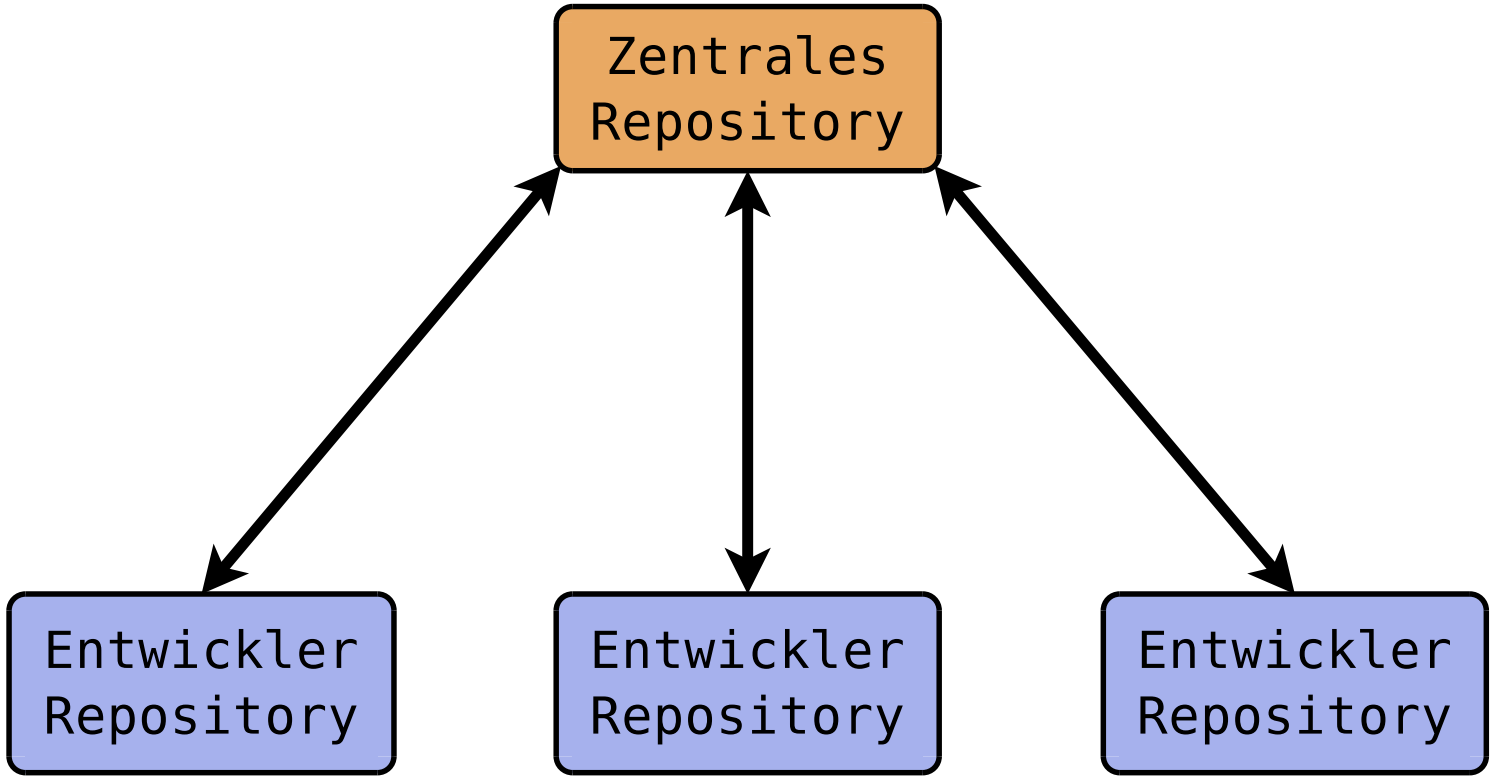
1. Ändern Sie die Commit-Nachricht mehrerer Commits
2. Fassen Sie mehrere Commits zu einem zusammen
3. Löschen Sie einen Commit aus der Versionsgeschichte
4. Wählen Sie einen Commit aus dem aktuellen Branch (nicht den neusten!) und teilen Sie ihn in mehrere kleinere auf

Übersicht

Hinaus in die weite Welt!

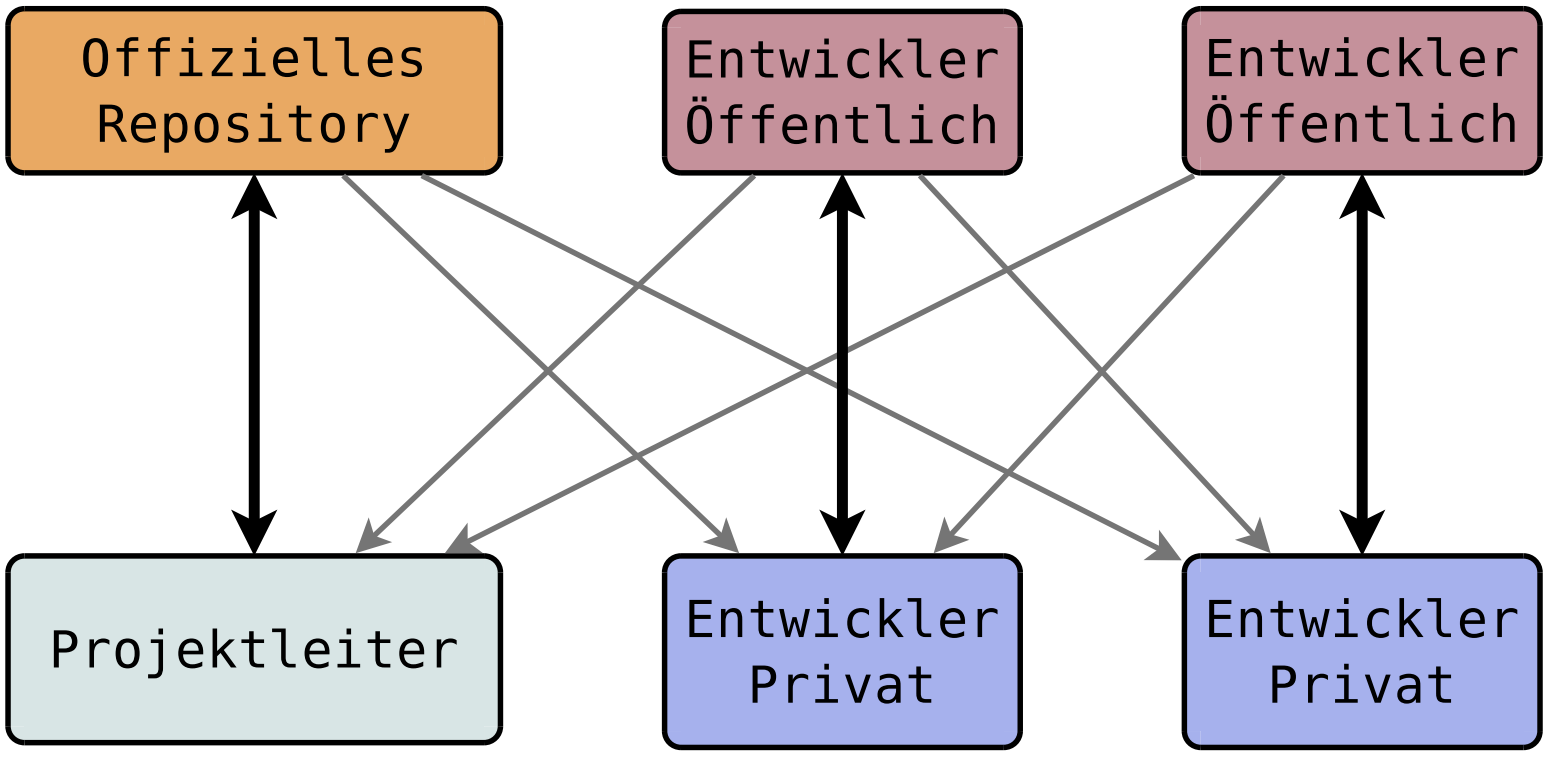
- Wir wollen unsere Arbeit mit der anderer Entwickler austauschen!
- Durch die verteilte Architektur von `git` braucht es keinen *zentralen* Server zu geben.
- Das Entwicklerteam muss sich auf einen *Workflow* einigen:
  - Zentralisiert
  - Öffentliche Entwickler-Repositories
  - Patch-Queue per E-Mail
  - ... oder auch alles durcheinandergemixt

Hybrid-Zentralisiert



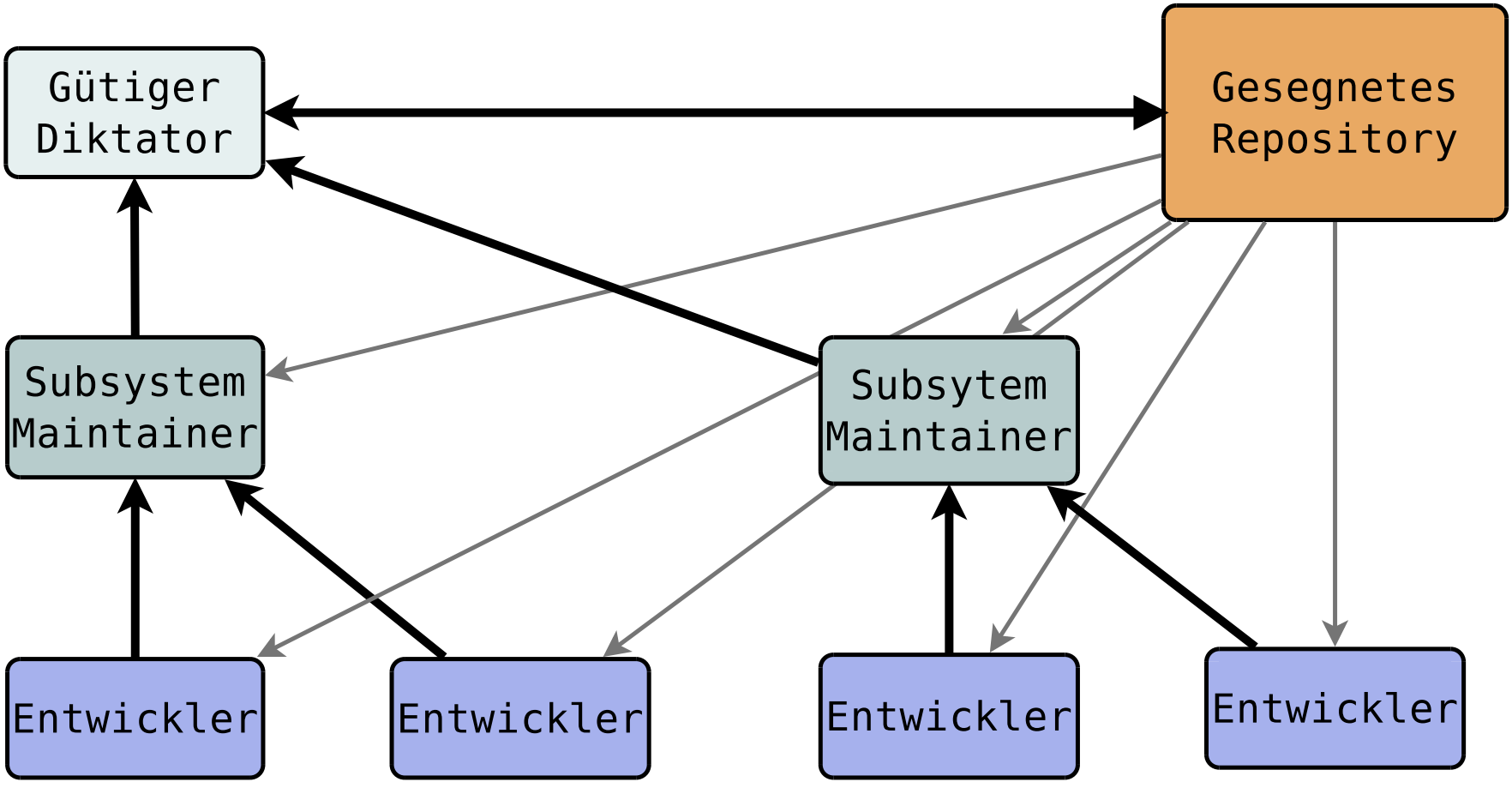
- Ein einziges zentrales Repository
- Alle Entwickler haben Schreibzugriff

Öffentliche Entwickler-Repositories



- Ein öffentliches Repository pro Entwickler
- Der Projektleiter integriert Verbesserungen

Patch-Queue per E-mail



- Stark vom Kernel und Git selbst verwendet

Übersicht

Remote-Repositories

- Alle »anderen« Repositories heißen bei Git *Remote Repository*
  - Das zentrale Repository
  - Repositories von anderen Entwicklern
  - Kopien (Klone) des Repositories
- Kurzbezeichnung: **Remotes**

- Im simpelsten Fall (nach einem `git clone`) ist nur ein Remote namens `origin` eingetragen

Bestehendes Projekt *klonen*

```
git clone git://gitschulung.de/git-test
git clone tn01@gitschulung.de:/repos/git-test
```

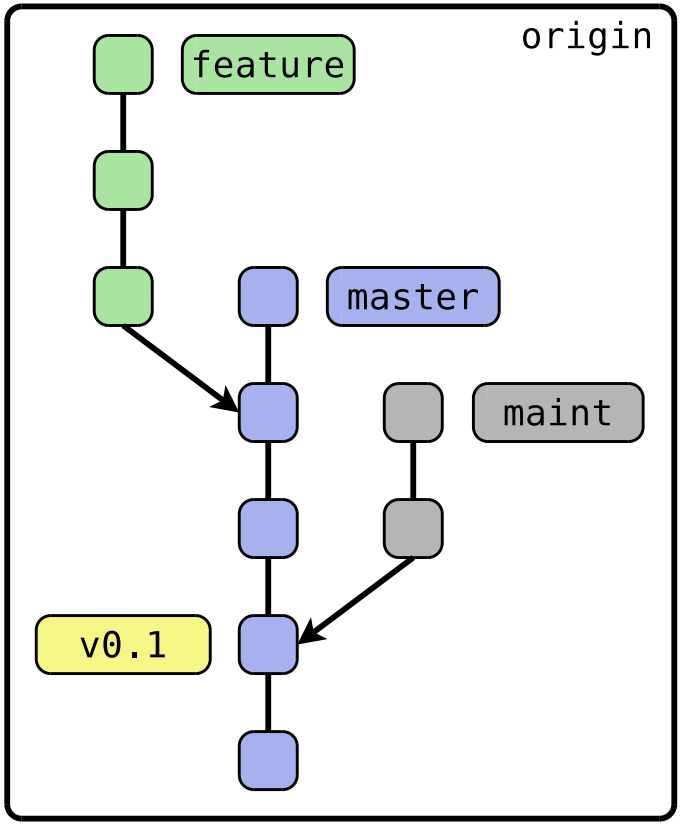
Remote-Tracking Branches

- Zur Erinnerung: Branches sind nur *Zeiger* in den Graphen
- Remote-tracking-branches sind spezielle Branches
- Git merkt sich damit den Zustand auf der Remote-Seite
- Können nicht vom User verwendet werden
- Werden beim Fetch aktualisiert

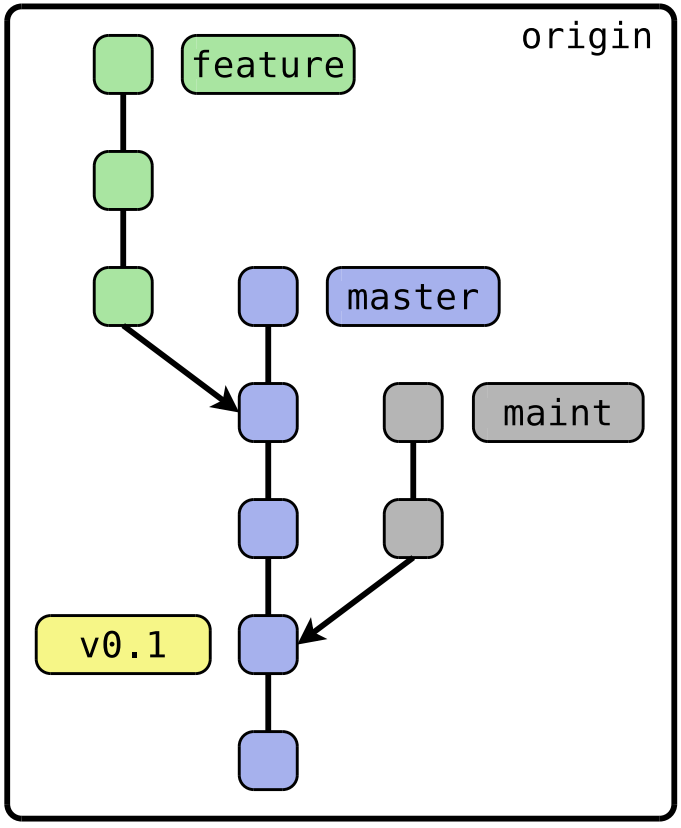
`origin/master`

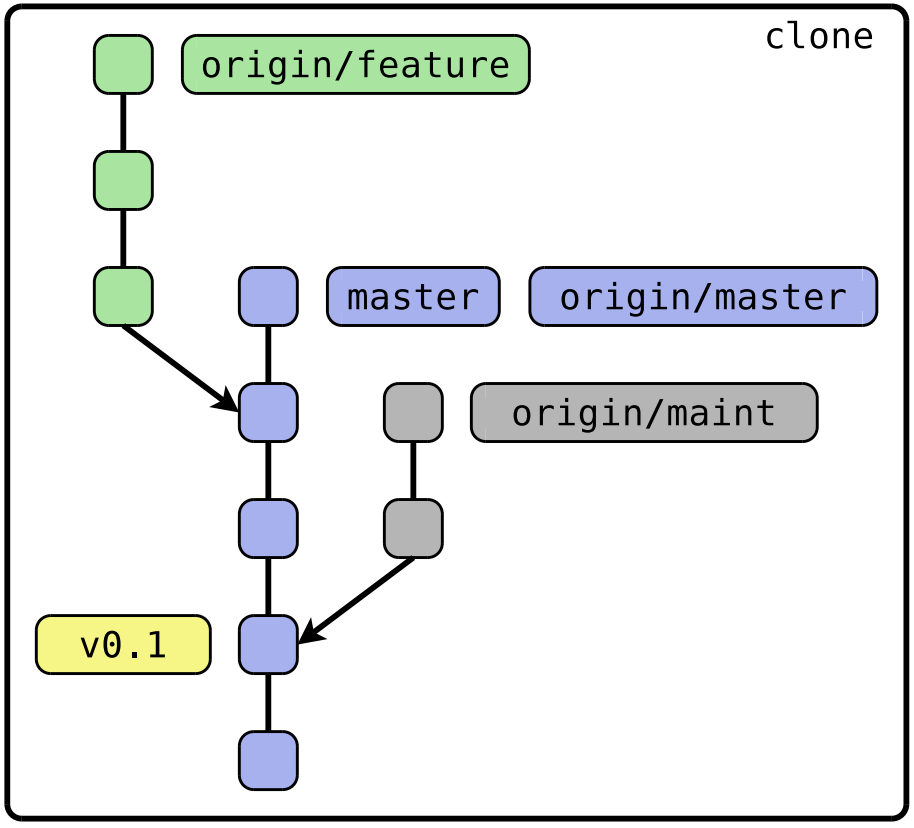
`origin/master` ist der *Remote-Tracking-Branch* des Branches `master` aus dem Remote `origin`

Vor `git clone`



Nach `git clone`





`git push` – Änderungen hochladen

Änderungen im *branch* nach *remote* hochladen

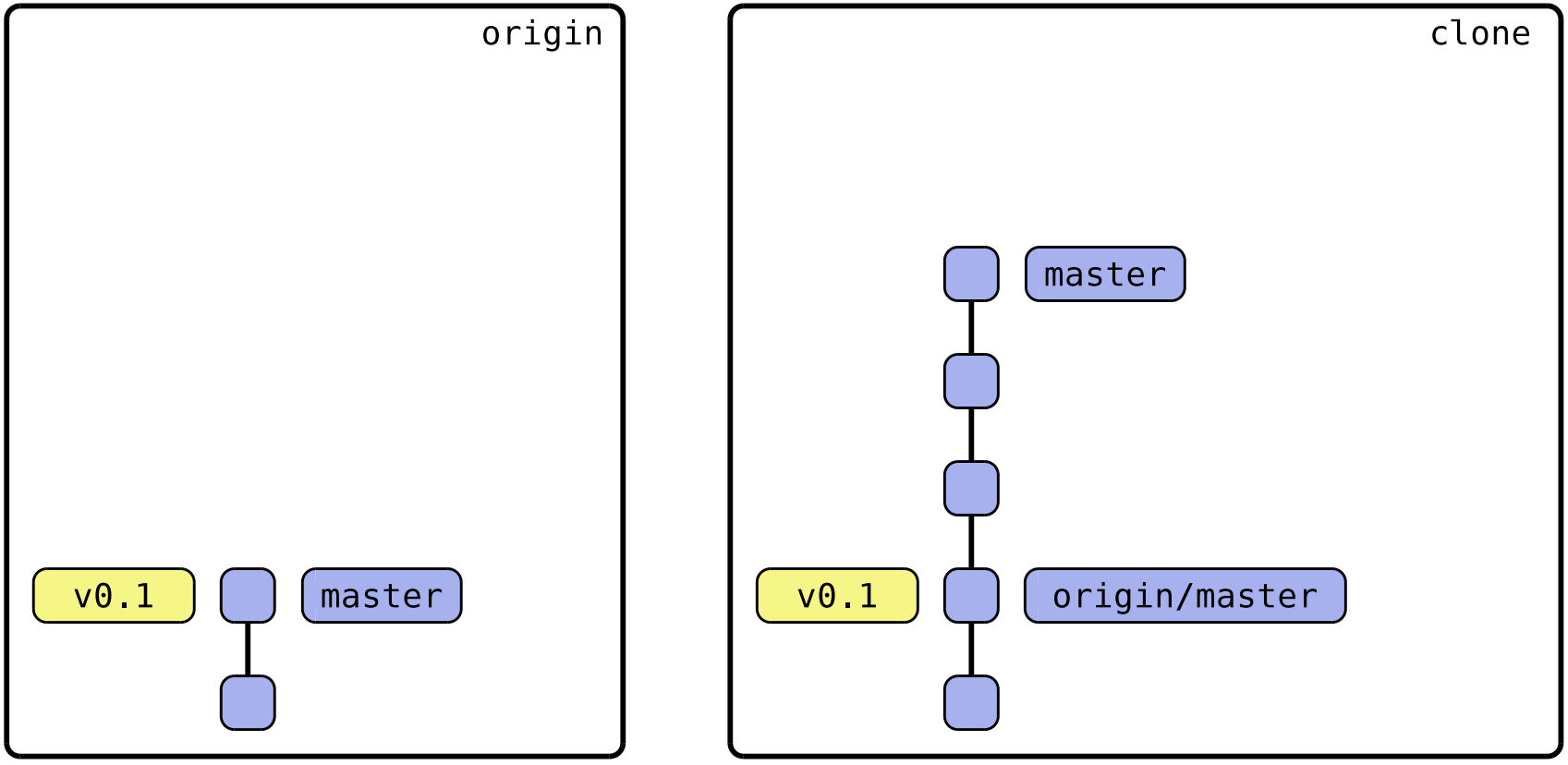
```
git push remote branch
git push origin master
```

Soll der Branch im Remote anders heißen

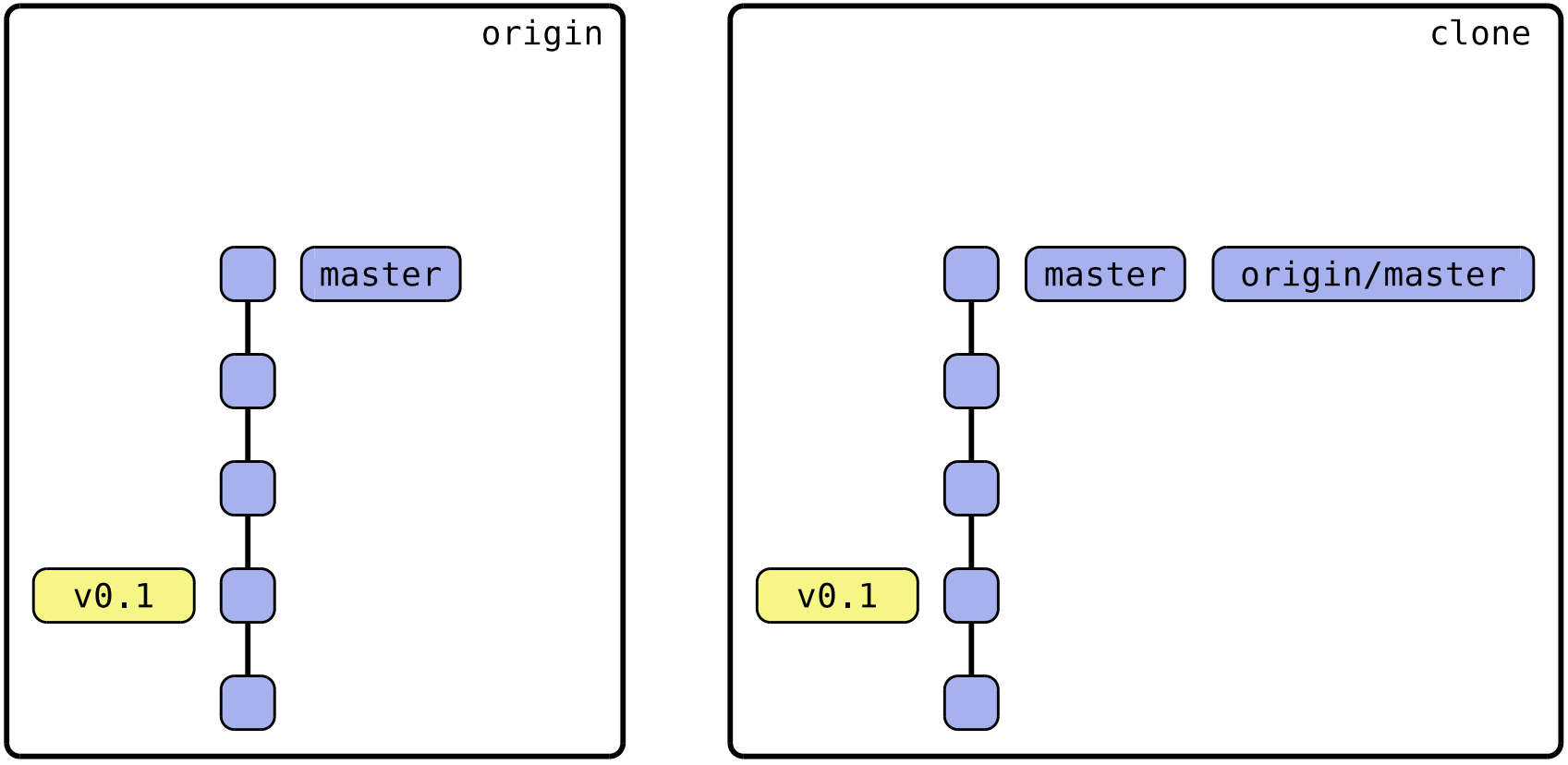
```
git push remote branch-lokal:branch-remote
```

- Existiert der Branch bereits, versucht Git, ihn »weiterzurücken« (Fast-Forward)
- Geht das nicht, gib Git eine Fehlermeldung aus
- Git erstellt den Branch, falls er nicht existiert

Vor `git push`



Nach `git push`



`git pull` – Änderungen herunterladen

Änderungen aus dem *branch* im *remote* herunterladen

```
git pull remote branch
git pull origin master
```

- Änderungen werden in den aktuellen Branch gemerged

`git fetch` – Remote-Tracking-Branches aktualisieren

- Lokales Repository mit einem Remote synchronisieren
  1. Veränderungen herunterladen
  2. Remote-Tracking-Branches werden automatisch angepasst

Veränderungen aus einem einzelnen Repository herunterladen

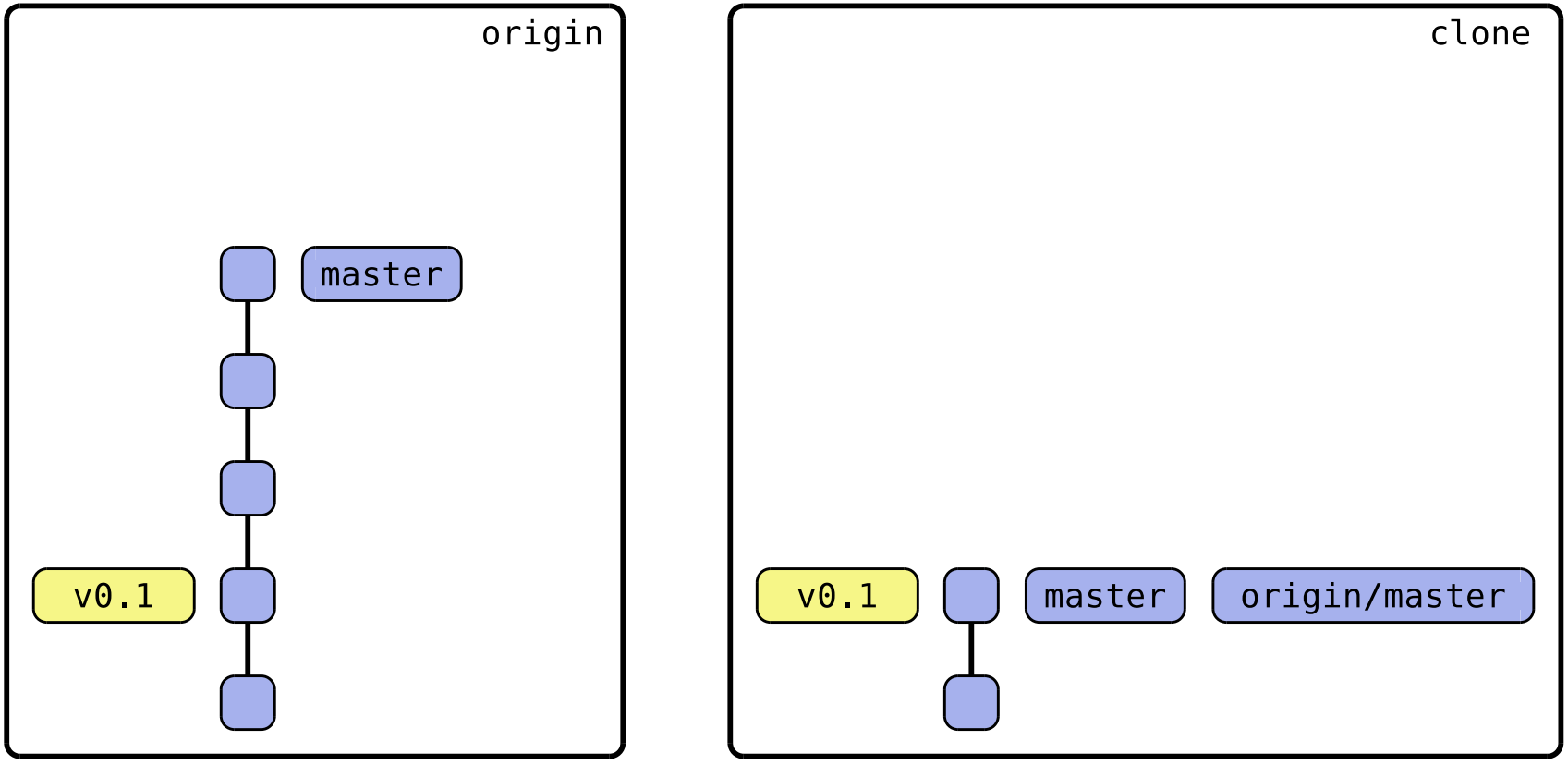
```
git fetch remote
```

Veränderungen aus allen Remotes herunterladen

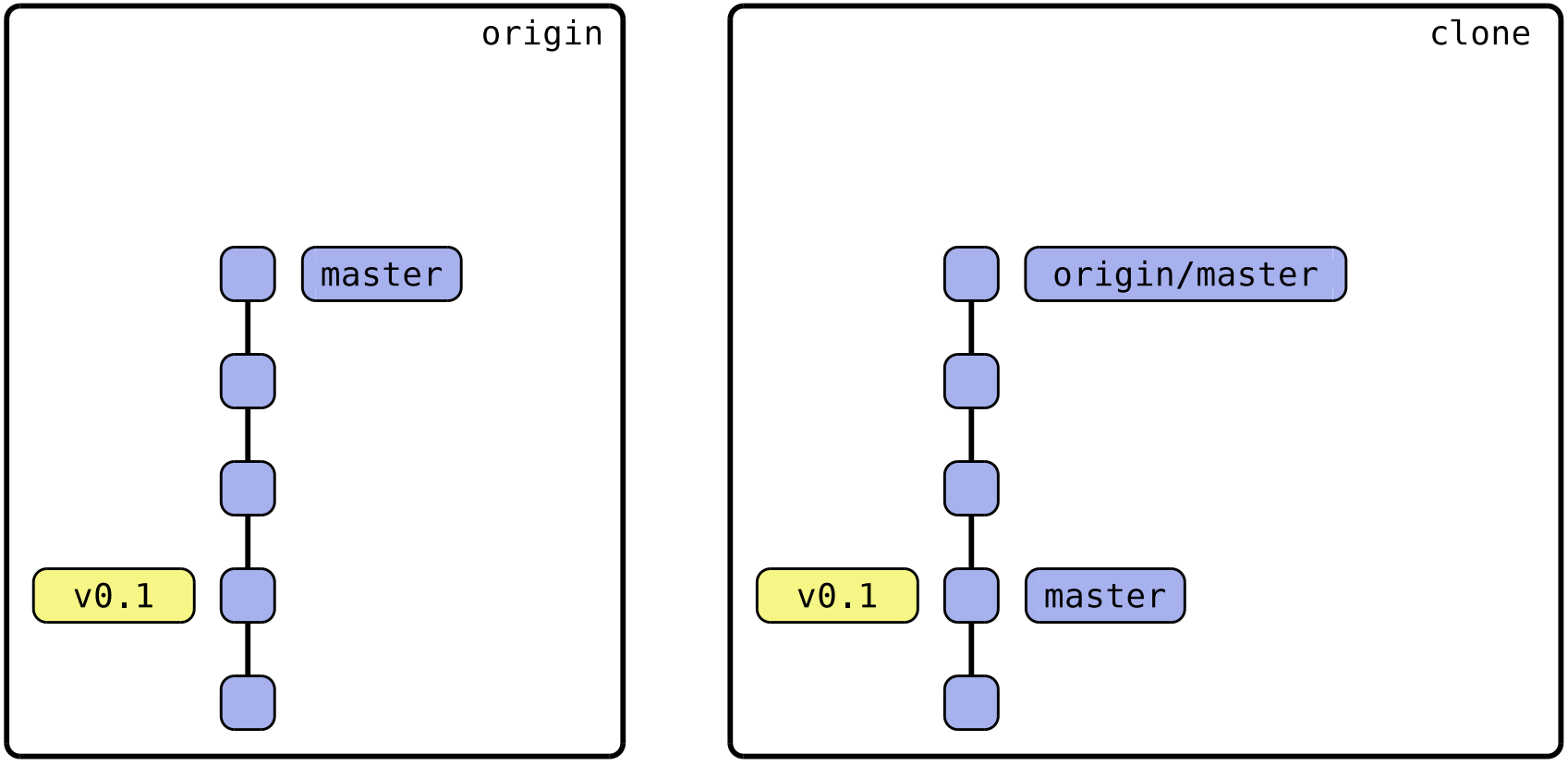
```
git remote update   (oder alternativ)
git fetch --all
```

- Nach dem Update müssen Änderungen eingepflegt werden
  - → `git merge` oder `git rebase`

Vor `git fetch`



Nach `git fetch`



Remote-Tracking-Branches Auslauben

- Die Remote-Tracking-Branches verschwinden nicht automatisch
- Zum Beispiel: die Feature-Branches der Kollegen

Während des fetch

```
git fetch --prune
```

Immer

```
git config --global fetch.prune true
```

`git pull = fetch + X`

`git pull` verbindet zwei Kommandos:

1. Änderungen herunterladen, Tracking-Branches aktualisieren
   - `git fetch`
2. Tracking-Branch integrieren
   - `git merge` oder `git rebase`

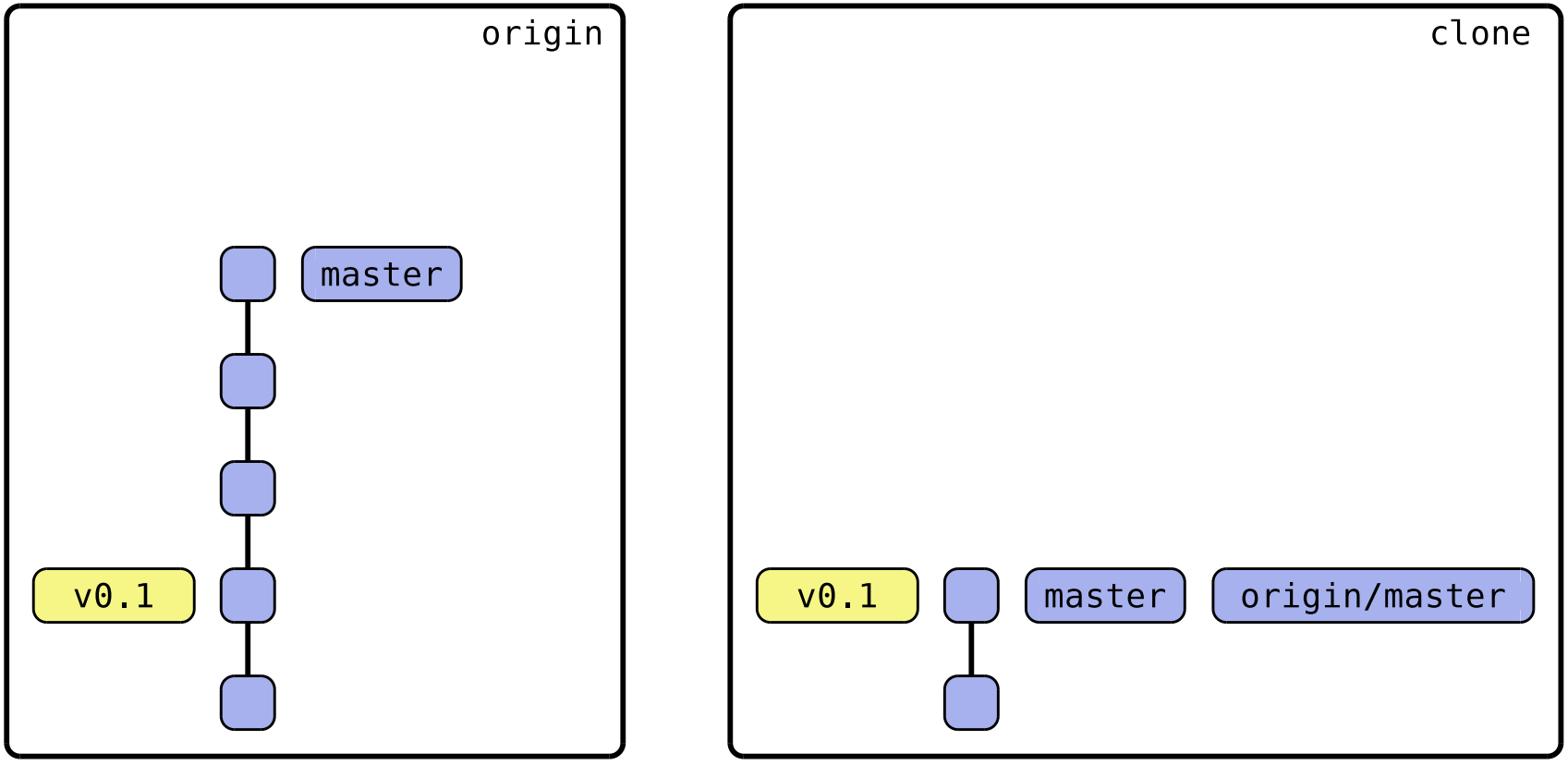
Fetch und Merge

```
git pull
```

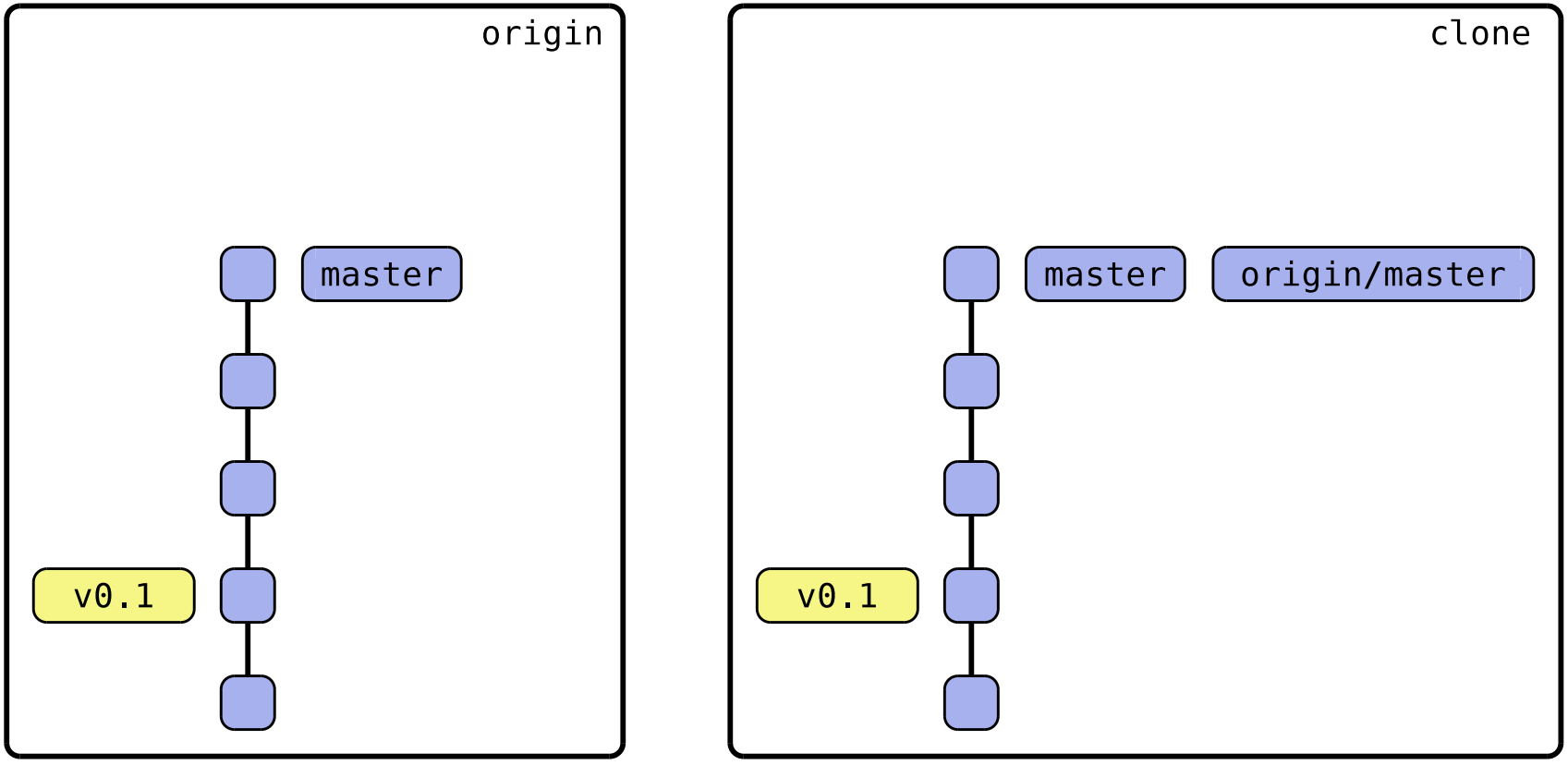
Fetch und Rebase

```
git pull --rebase
```

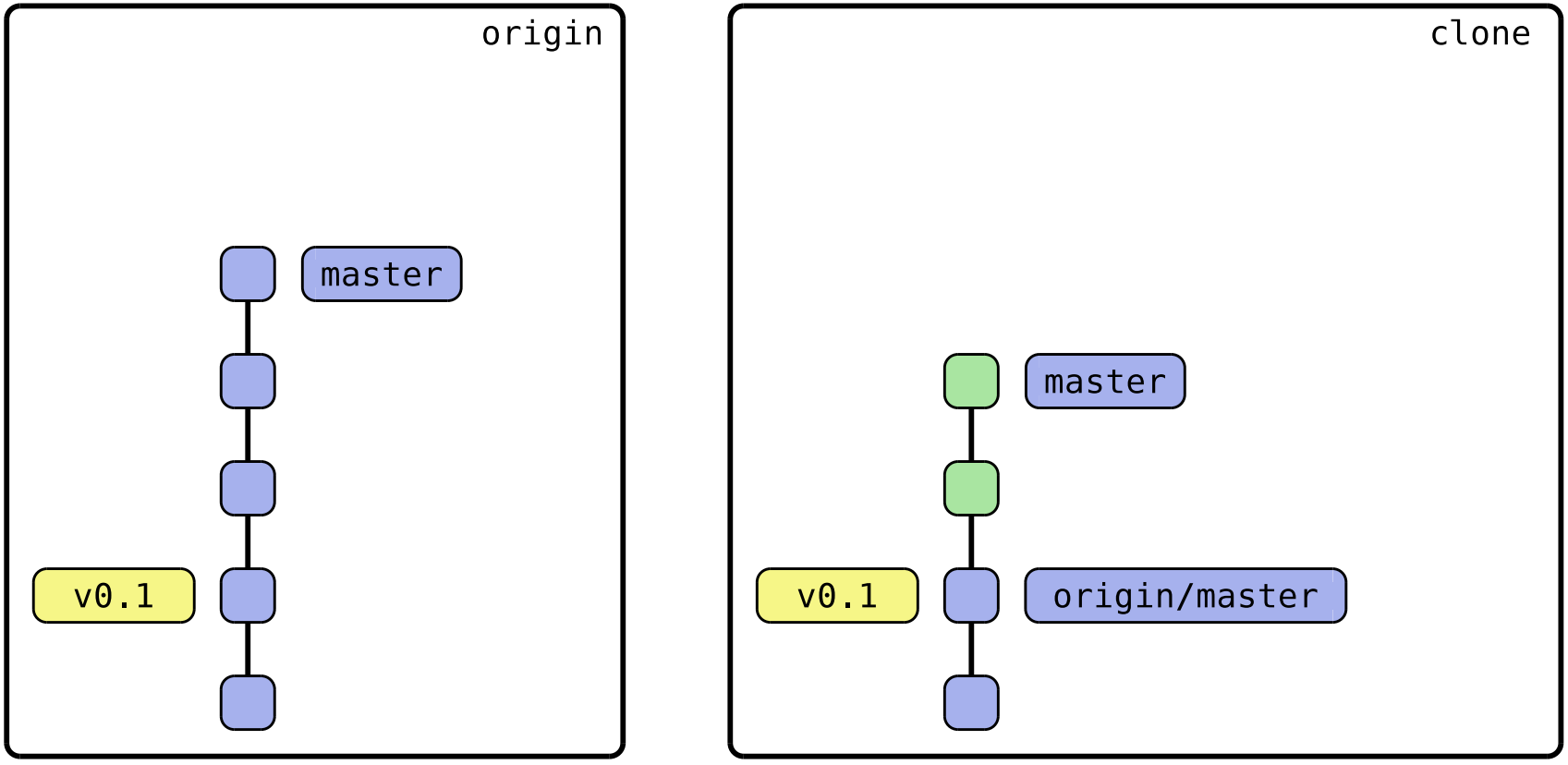
Vor einem Pull mit Fast-Forward



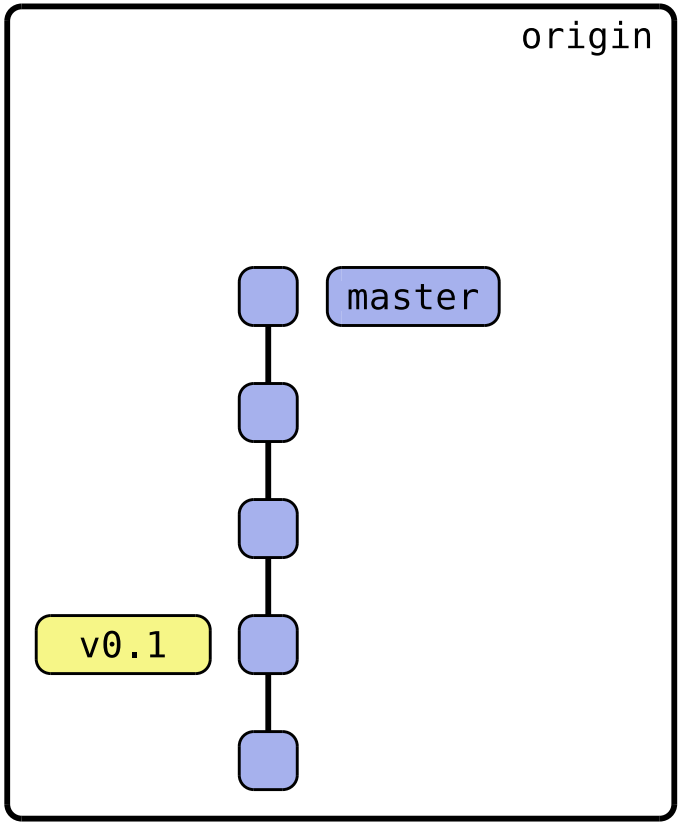
Nach einem Pull mit Fast-Forward

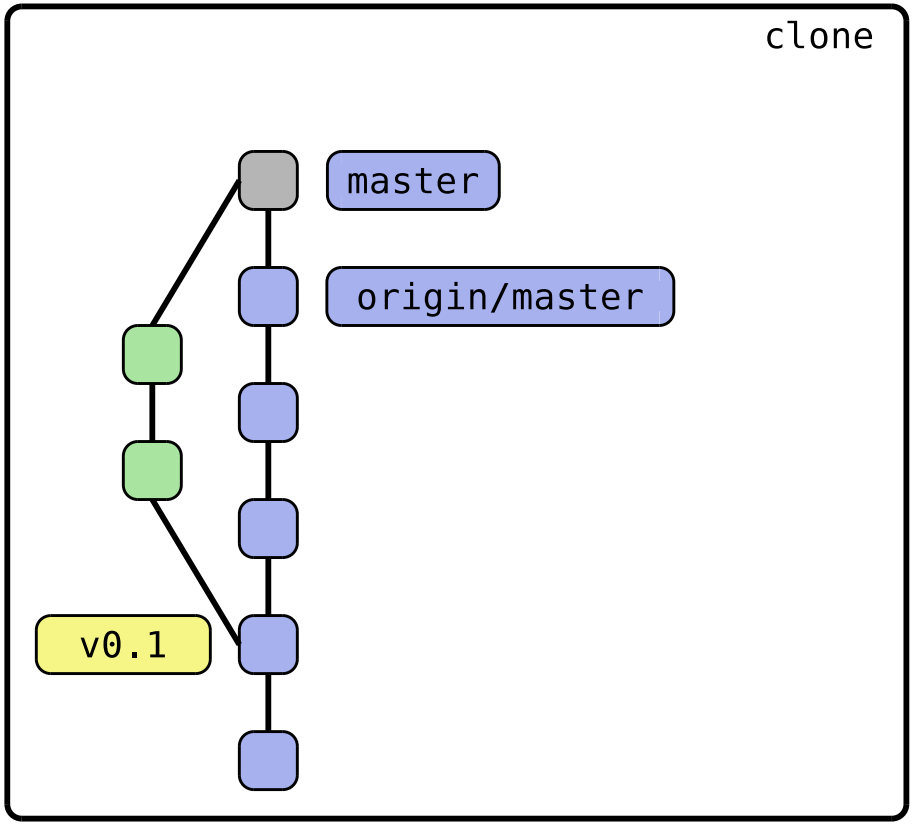


Vor einem Pull mit Merge

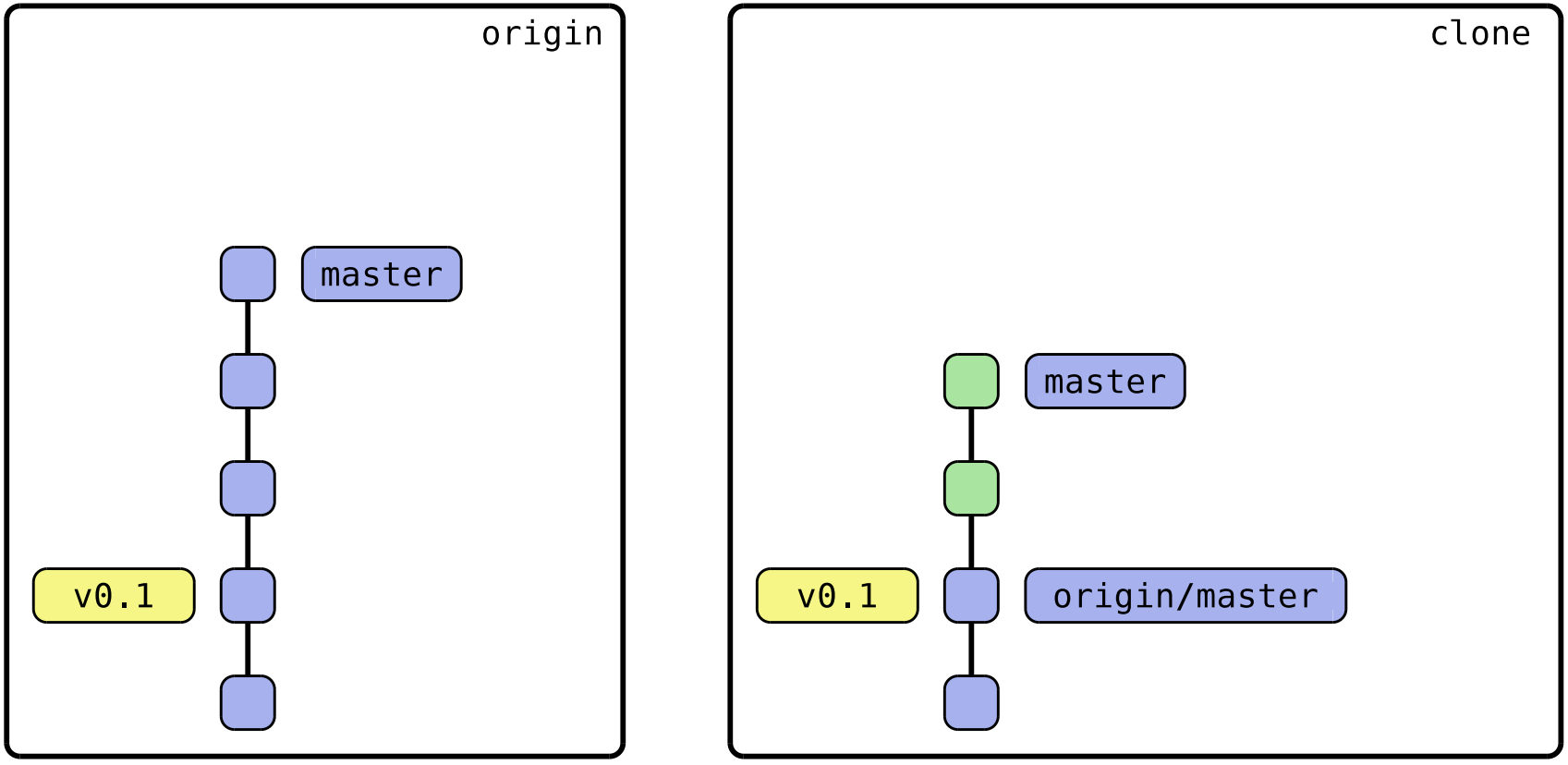


Nach einem Pull mit Merge

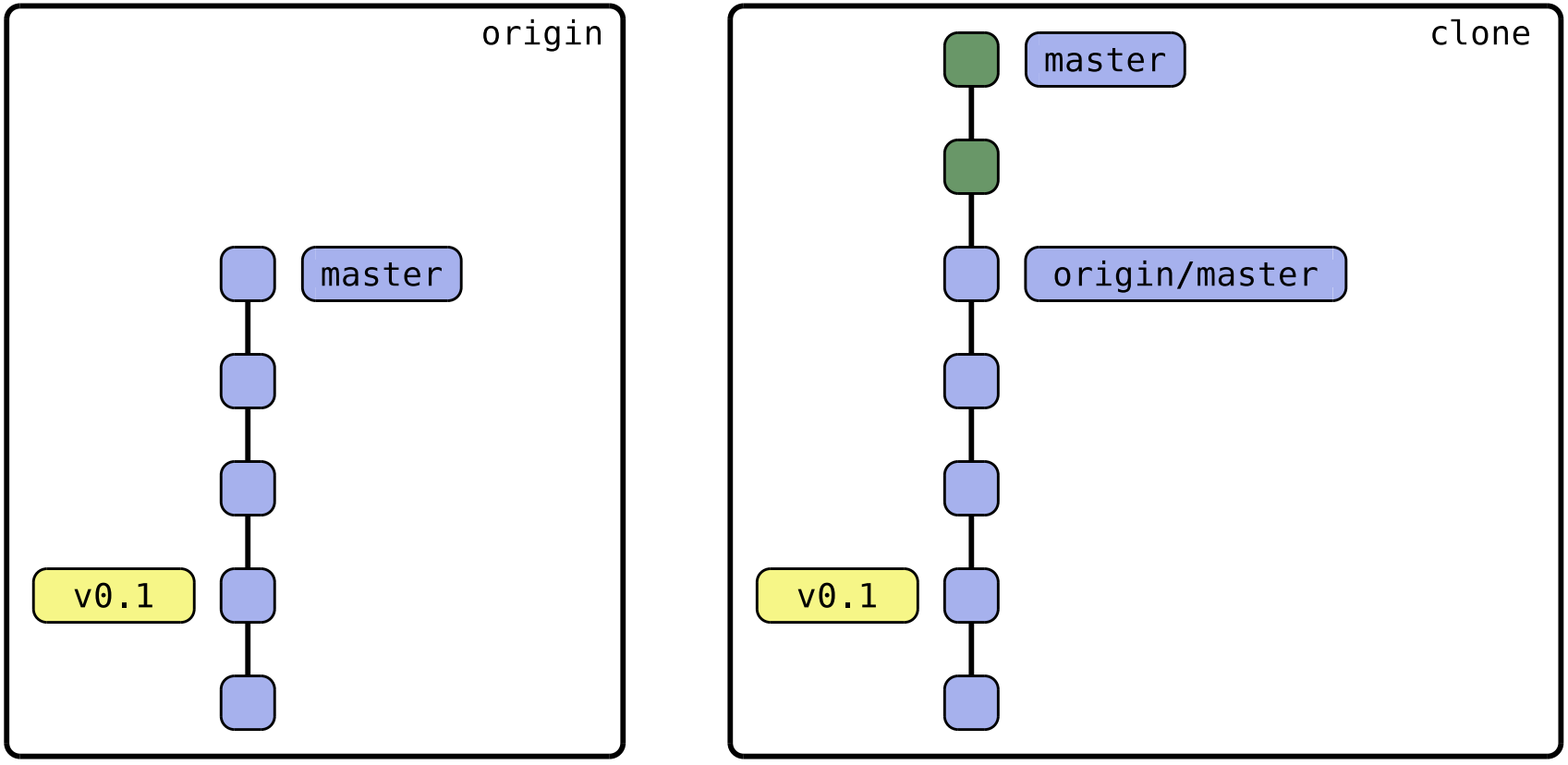




Vor einem Pull mit Rebase



Nach einem Pull mit Rebase



Übung: Remotes

1. Klonen Sie das `git-test` Repository:
   `git clone` *`user`*`@gitschulung.de:/repos/git-test`
2. Erstellen Sie einen Commit in `master` und laden Sie diesen hoch (`git push origin master`)
   - 2.1 Es kann sein, das dies fehlschlägt – warum?
   - 2.2 Was können Sie tun, um die Situation zu beheben? (Mehrere Möglichkeiten)
3. Erstellen Sie auf einem neuen lokalen Branch mit beliebigem Namen neue Commits und laden Sie den Branch dann in das Remote hoch
4. Warten Sie, bis Ihr Nachbar einen Branch hochgeladen hat – führen Sie dann `git fetch origin` aus, um Ihre Remote-Tracking-Branches zu aktualisieren
5. Betrachten Sie alle Remote-Tracking-Branches mit `git branch -r` und `gitk --all`

Local-Tracking-Branches

- Wir wollen einen Remote-Branch modifizieren
- Wir erstellen einen lokalen Branch mit dem gleichen Namen wie der Branch im Remote
  - `origin/feature` → `feature`
  - `johndoe/fix-typos` → `fix-typos`
- `origin/master` ist auch als sog. *upstream-branch* `master` bekannt

Einen lokalen Branch zum Arbeiten anlegen

Local-Tracking-Branch erstellen

```
git checkout -b branch-name remote-name/branch-name
git checkout -b feature origin/feature
```

Ist *branch-name* eindeutig, reicht

```
git checkout branch-name
git checkout feature
```

Upstream-Configuration

- Die Beziehung der lokalen Branches zu denen in Remotes wird in der `.git/config` gespeichert
- Dies ist die sog. *upstream-config*

Upstream-Config

```
[branch "branch-name"]
    remote = remote-name
    merge = refs/heads/branch-name
```

Beispiel

```
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

Abfragen der Tracking-Beziehung

```
git branch -vv
```

Upstream-Config verwenden

- Andere Kommandos nutzen diese Informationen
- Voraussetzung: der aktuelle Branch hat eine Upstream-Config
  - `master` ist ausgecheckt und trackt `origin/master`
- Git-Kommandos `fetch`, `pull` und `push` können ohne Argumente aufgerufen werden

- `git fetch`
  - $\rightarrow$ Ziel-Remote ist bekannt
- `git pull`
  - $\rightarrow$ Ziel-Remote ist bekannt
  - $\rightarrow$ Remote-Tracking-Branch zum mergen ist bekannt
- `git push`
  - $\rightarrow$ Ziel-Remote ist bekannt
  - $\rightarrow$ Remote-Branch ist bekannt

Push ohne Argumente

- Seit git 2.0 wird nur der aktuell ausgecheckte Branch gepushed wenn:
  - Er eine Upstream-Configuration hat
  - der Branch im Remote den gleichen Namen hat
- Ausgezeichnete Einstellung für Anfänger
- Vorher: alle Branches die einen Branch des selben Namens im Remote haben werden gepushed
- Verhalten ist Konfigurierbar
  - `push.default = simple`: Git 2.0 default
  - `push.default = matching`: alte Einstellung

Remote-Branches Löschen

Remote-Branches löschen

```
git push remote-name --delete branch-name
git push origin --delete feature
```

Alternative Syntax

```
git push remote-name :branch-name
git push origin :feature
```

Remotes anzeigen

Auflistung aller Remotes

```
git remote
```

Gleiche Auflistung mit mehr Einzelheiten

```
git remote -vv
```

Alle verfügbaren Infos zu *einem* Remote ausgeben

```
git remote show remote-name
```

Remotes verwalten

Remote hinzufügen

```
git remote add remote-name URL
```

Remote umbenennen

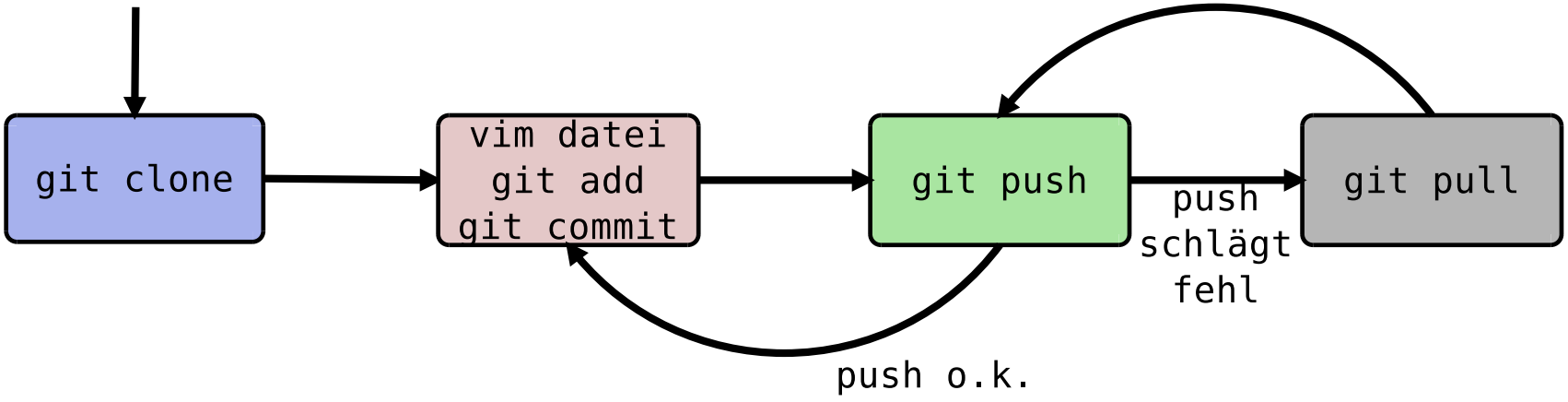
```
git remote rename alt neu
```

Remote löschen

```
git remote rm remote-name
```

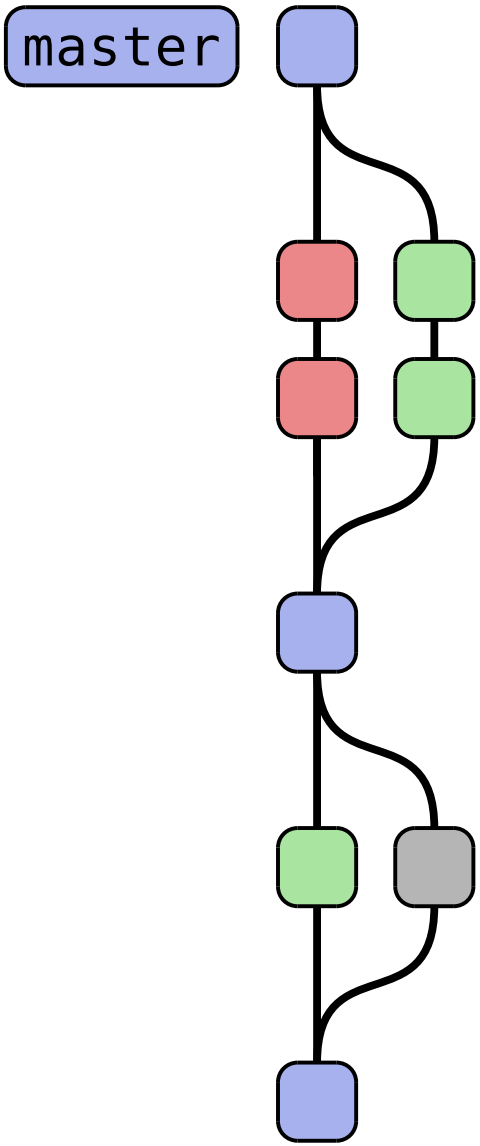
Übersicht

Push'n'pull Workflow



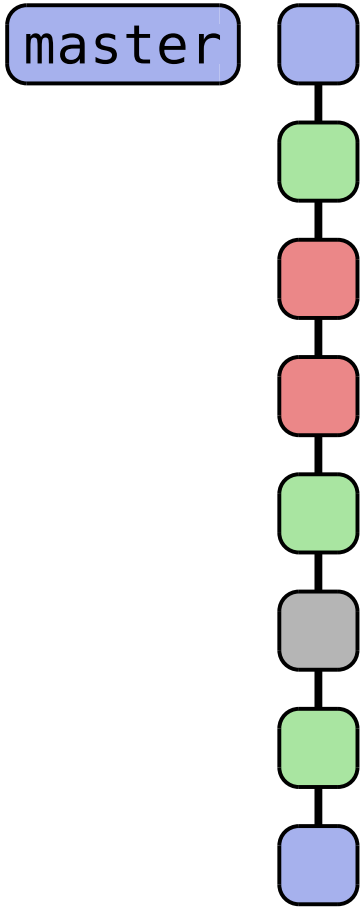
1. Lokale Änderungen
   - `vim` *`datei`*
   - `git add` *`datei`*
   - `git commit -m "`*`msg`*`"`
2. Änderungen veröffentlichen
   - `git push`
   - Wenn `push` fehlschlägt
   - `git pull`, dann `git push`

Push'n'pull Workflow Resultat



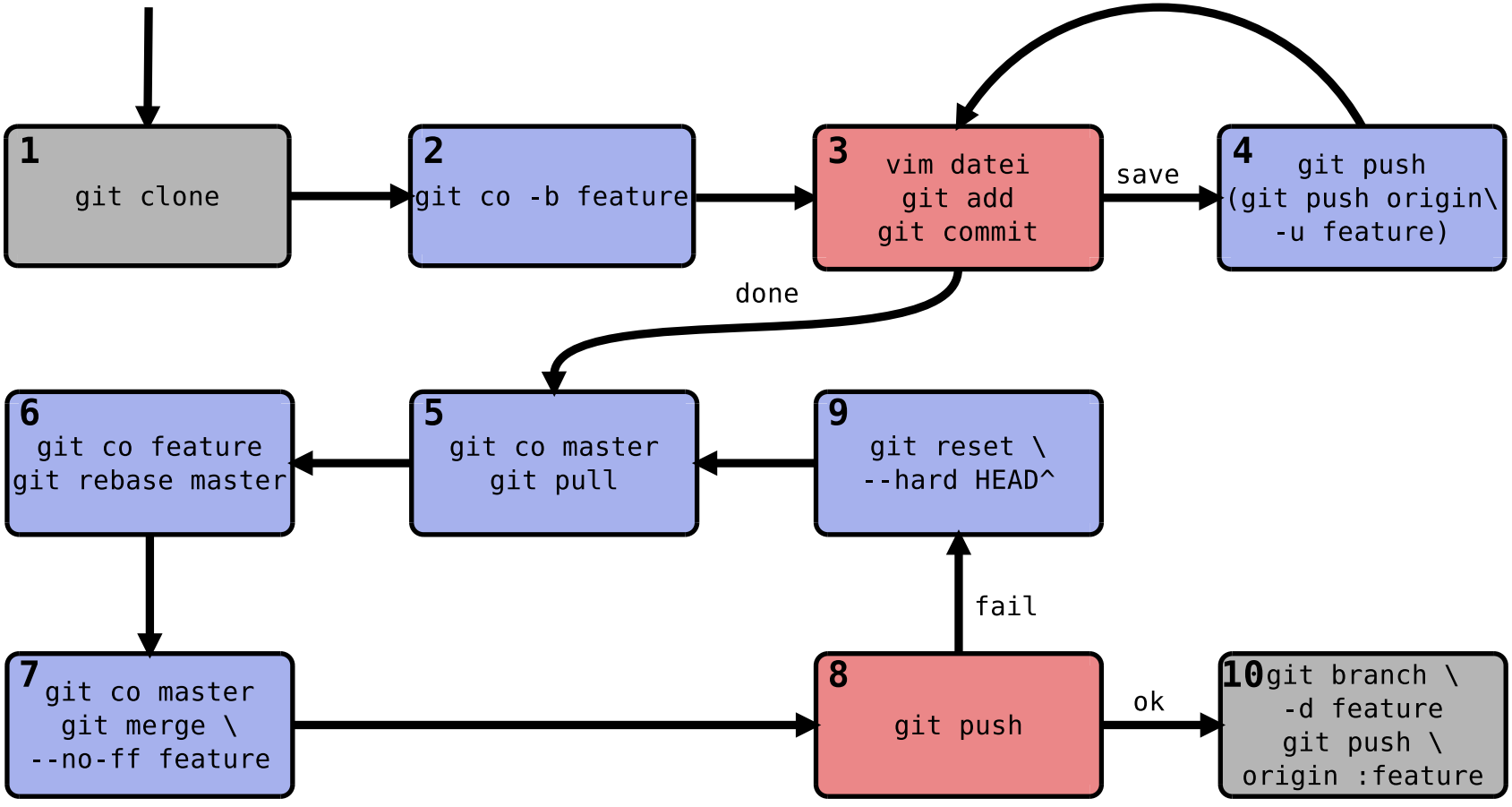
- Vorteile
  - Leicht für Anfänger
  - Nur weniger Kommandos
- Nachteile
  - Es entstehen Merge-Commits
  - »Aber wir arbeiten doch alle auf `master`?!«
  - Rebase ist eine Option (für Anfänger?)

Push'n'pull Workflow Resultat mit `--rebase`



- Vorteile
  - Keine Merge-Commits
- Nachteile
  - »Sinnloses« Linearisieren
  - Feature-Commits in zufälliger Reihenfolge

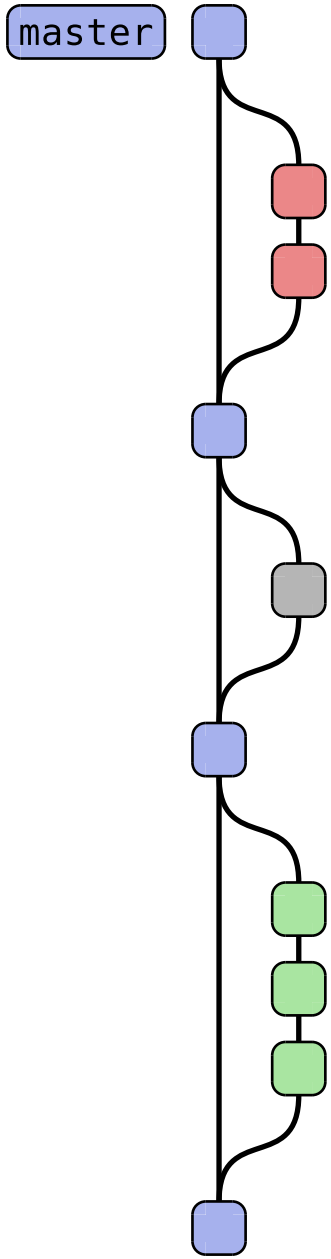
Rebase 'n' Force-Merge Workflow



Beschreibung der Schritte

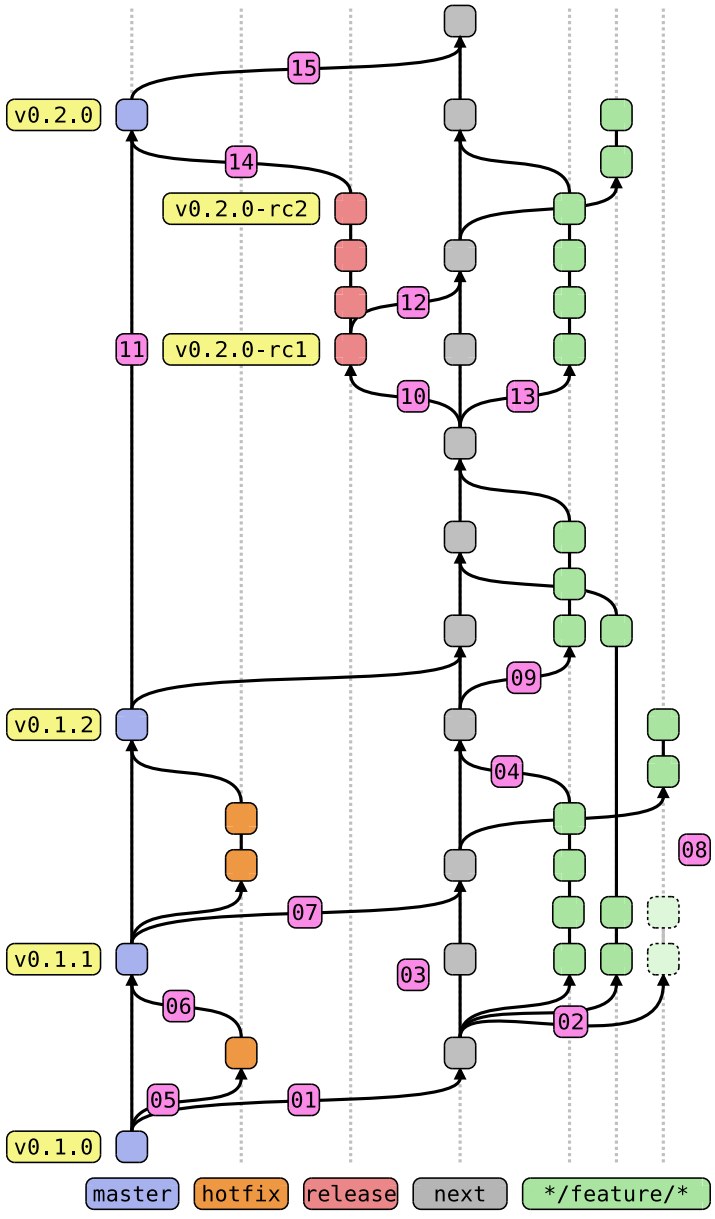
1. **Repsoitory clonen:**
   ```
   git clone <url>
   ```
2. **Feature-Branch anlegen:**
   ```
   git checkout -b feature
   ```
3. **Arbeiten:**
   ```
   vim file
   git add
   git commit
   ```
4. **Getaene Arbeit hochladen:**
   ```
   #beim ersten Mal
   git push origin -u feature
   git push
   ```
5. **Lokalen `master` aktualisieren:**
   ```
   git checkout master
   git pull
   ```
6. **Rebase `feature` auf `master`:**
   ```
   git checkout feature
   git rebase master
   ```
7. **Forcierter Merge von `feature`:**
   ```
   git checkout master
   git merge --no-ff feature
   ```
8. **`master` Hochladen:**
   ```
   git push
   ```
9. **Fehlschlag: Merge rückgängig:**
   ```
   git reset --hard HEAD^
   ```
10. **Erfolg: `feature` löschen, lokal und remote:**
    ```
    git branch -d feature
    git push origin :feature
    ```

Rebase 'n' Force-Merge Resultat



- Vorteile
  - Saubere history
  - Feature-Branches
  - Merges sinnvoll
- Nachteile
  - Verständnis von Git gebraucht
  - Meherer Kommandos nötig

Branch-Modell



- Angelehnt an gitflow

- Branches
  - `master` → Stabil
  - `next` → Vorbereitung
  - `feature` → Feature-Entwicklung
  - `release` → Letzte Releasearbeiten
  - `hotfix` → Hotfixes

Beschreibung

1. Der Next-Branch wird von dem letzten stabilen Release abgezweigt.
2. Die Entwicklung beginnt, Feature-Branches werden von dem Next-Branch abgezweigt.
3. Triviale Commits könnten auf dem Next-Branch eingespielt werden.
4. Feature-Branches werden nach Fertigstellung im Next-Branch gesammelt und ggf. getagged.
5. Hotfixes werden im Hotfix-Branch eingespielt und ggf. getagged.

Beschreibung (cont)

6. Hotfixes werden in den Master-Branch gemergt und auf jeden Fall getagged.
7. Der Master-Branch wird nach einem Hotfix-Release auch wieder in den Next-Branch gemergt.
8. Feature-Branches werden ggf. per Rebase aktualisiert.
9. Neue Feature-Branches können jederzeit wieder vom Next-Branch abgezweigt werden.
10. Sind alle Feature-Branches für das Release im Next-Branch angekommen wird der Release-Branch abgezweigt. Dort findet dann die Release-Vorbereitung Statt.

Beschreibung (cont)

11. Release-Candidates können von hier aus getagged und deployed werden.
12. Kritische Bug-Fixes die für Feature-Branches wichtig sind können jederzeit wieder in den Next-Branch gemergt werden.
13. Während der Release-Vorbereitung können trotzdem weiter neue Feature-Branches für das nächste Release abgezweigt werden.
14. Ist der Release fertig, wird der Release-Branch in den Master-Branch gemergt, getagged und in die Produktion deployed.
15. Der Master-Branch wird zuletzt noch in den Next-Branch gemergt um alle Änderungen aus dem Release-Branch dort verfügbar zu machen und damit das Tag “v0.2.0“ von dem Next-Branch aus erreichbar ist.

Iterationen

1. Push & Pull
2. Fetch & Rebase
3. Eigene Branches hochladen
4. Eigene Branches lokal, selbständiges Mergen

Iteration 1: Push & Pull

1. Commits direkt auf `master` machen
2. Fertig: `git push`
3. Schlug fehl?
   - 3.1 `git pull`
   - 3.2 `git push`

Iteration 2: Fetch & Rebase

1. Commits direkt auf `master` machen
2. Fertig: `git push`
3. Schlug fehl?
   - 3.1 `git pull --rebase`
   - 3.2 `git push`

Iteration 3: Eigene Branches hochladen

1. Eigenen Branch erstellen, z. B. `jp/feature`
2. Dort Commits machen, periodisch hochladen, Bescheid sagen
3. Integration Manager kümmert sich um `master`
4. Eigenen Branch löschen
5. Remote-Pruning (`git remote prune origin`)

Iteration 4: Eigene Branches lokal, selbständiges Mergen

1. Eigenen Branch erstellen, z. B. `jp/feature`
2. Dort Commits machen, *nicht* hochladen
3. Gerne mit interaktivem Rebase aufräumen
4. Wenn fertig:
   - 4.1 `master` auschecken, Pullen
   - 4.2 `git merge` *`feature-branch`*
   - 4.3 `git push`

Referenz: Workflow-Diagramm