# Git-Schulung

## –Firma–

Valentin Hänel, Julius Plenz

–Datum–

# Übersicht

# Vorläufiges Programm:

- Begriffsbildung
- Staging-Area/Index
- Objektmodell und gerichtete azyklishe Graphen
- Der Push'n'Pull Workflow
- Branches, Tags, Merges und Rebase
- Remote-Repositories

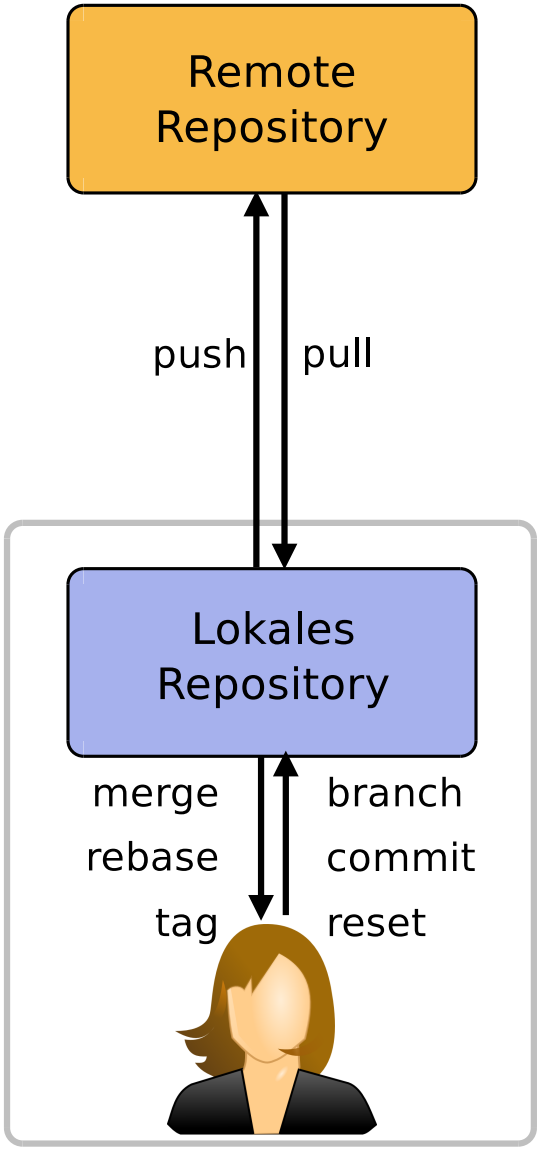
Folien: siehe ****(?)

# Ziele:

- Git Basiswissen erarbeiten
- Durch Verständnis der Interna, Git besser verstehen
- Einen einfachen Workflow lernen

# Übersicht

# Autonomie des eigenen Repositories




- *Remote* und lokales Repository sind gleichberechtigt
- Austausch zwischen Repositories via Push/Pull
  - *Push*: Eigene Änderungen hochladen
  - *Pull*: Änderungen herunterladen
- Alle anderen Aktionen passieren zunächst *nur* lokal

# Vor- und Nachteile verteilter Versionskontrollsysteme

**Vorteile:**

- Jeder Entwickler besitzt eine *komplette* Kopie der Versionsgeschichte
  - Kommandos laufen sehr schnell
  - Offline-Arbeit möglich
  - Impliziter Schutz vor Manipulation
- Es gibt keinen »single point of failure«
  - Serverausfall, Hack, wütender Entwickler, ...

# Vor- und Nachteile verteilter Versionskontrollsysteme

**Vorteile:**

- Kein Streit um Commit-Rechte
- Delegation von Aufgaben ist leichter
- Beliebige Workflows

# Vor- und Nachteile verteilter Versionskontrollsysteme

**Nachteile:**

- Viel Freiheit: Policies müssen geschaffen werden
- Komplexeres Setup als zentralisierte Systeme

# Begriffsbildung

- **Commit**: Eine Änderung an einer oder mehrerer Dateien, versehen mit Metadaten wie Autor, Datum und Beschreibung
- **Commit-ID**: Jeder **Commit** wird durch eine eindeutige SHA1-Summe identifiziert, seine **ID**
- **Repository**: »Behältnis« für gespeicherte **Commits**
- **Working-Tree**: Arbeitsverzeichnis, die Dateien die man sieht
- **Branch**: Ein »Zweig«, eine Abzweigung im Entwicklungszyklus, z. B. um ein neues Feature einzuführen.
- **Referenz**: Eine Referenz »zeigt« auf einen bestimmten Commit, z. B. ein **Branch**

- **Index/Staging-Area**: Bereich zwischen dem **Working-Tree** und dem **Repository**, in dem Änderungen für den nächsten **Commit** gesammelt werden

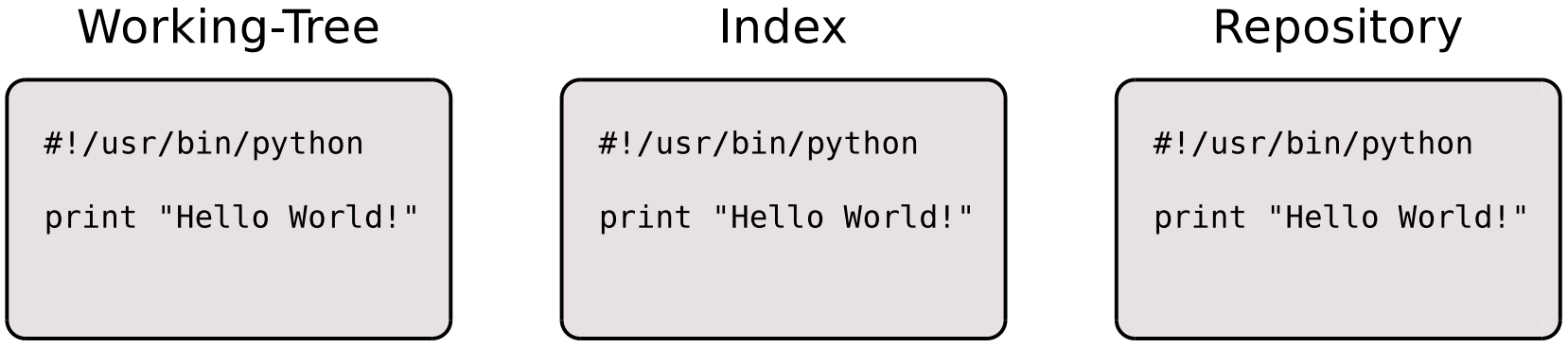
# Übersicht

# Index / Staging Area

- Im *Index/Staging-Area* werden Veränderungen für den nächsten Commit vorgemerkt
- So kann der Inhalt von einem Commit schrittweise aus einzelnen Veränderungen zusammengestellt werden
- Nach einem Commit enthält der Index genau die Versionen der Dateien wie in dem Commit

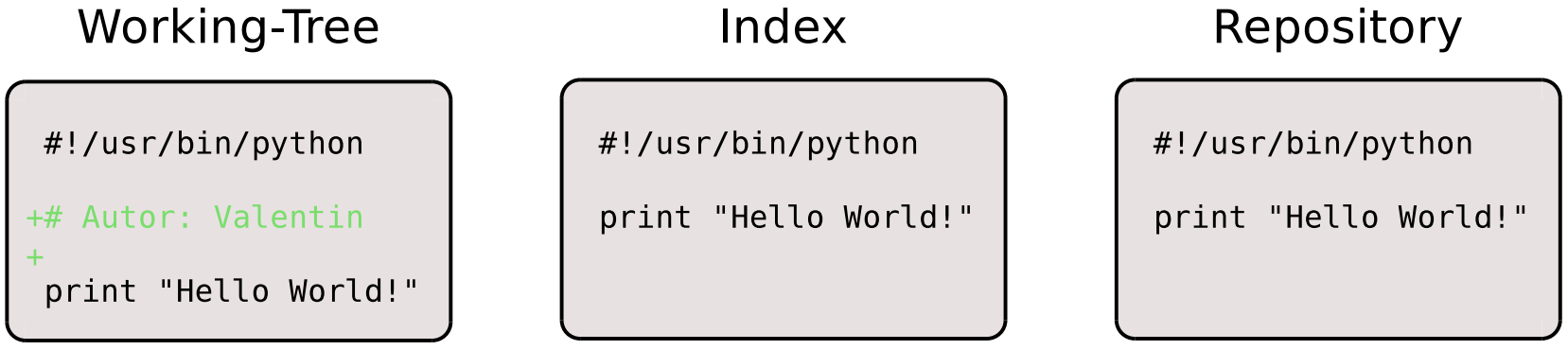
# Ausgangsstellung

- Alle auf dem gleichen Stand



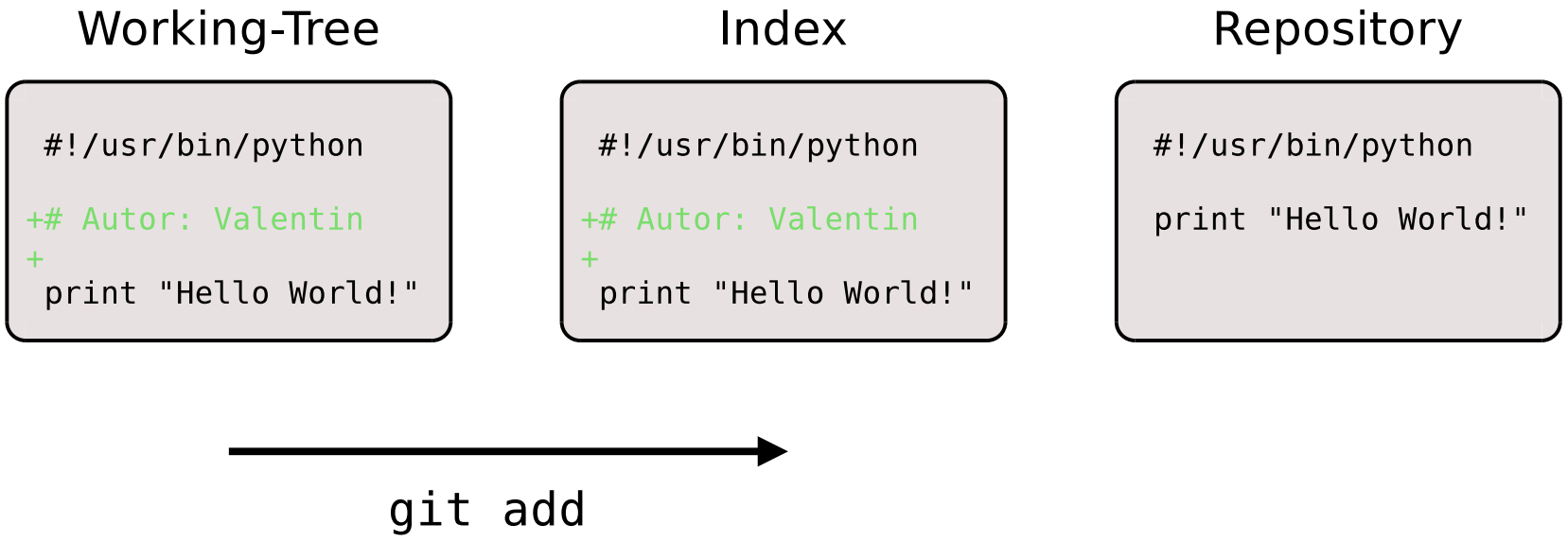
# Veränderungen machen

- Veränderungen werden im Working-Tree gemacht



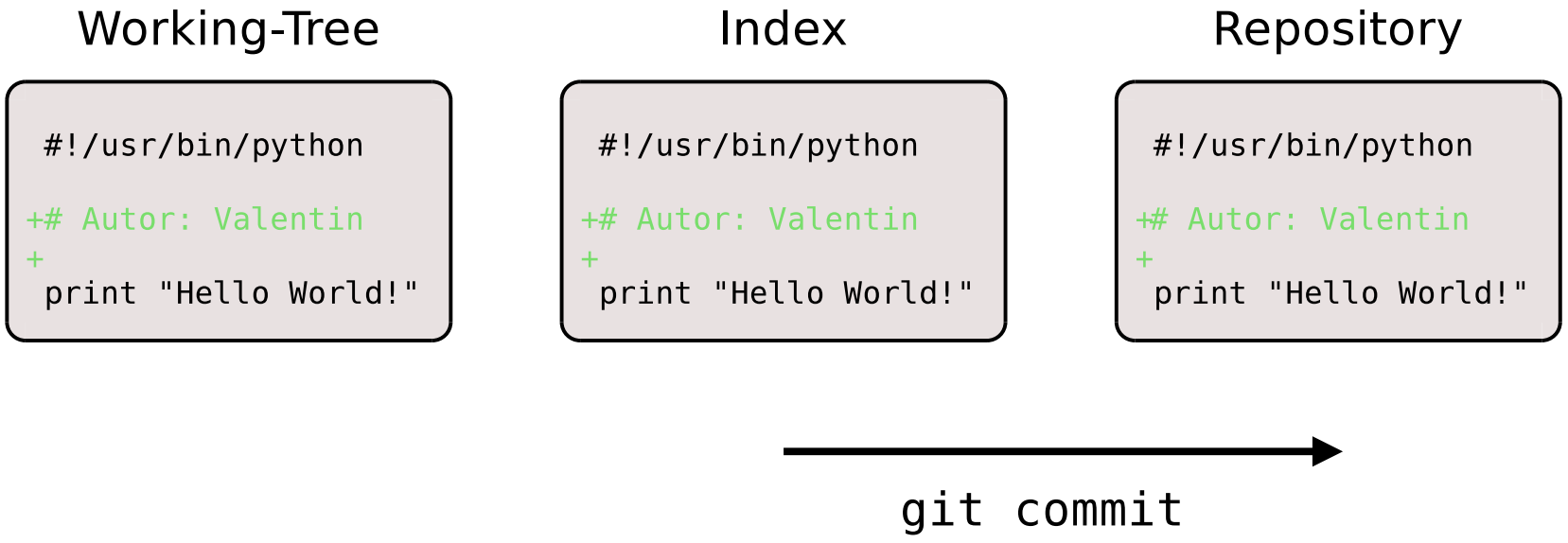
# Dem Index hinzufügen – `git add`

- Die Veränderungen im Working-Tree → Index



# Einen Commit erzeugen – `git commit`

- Alle Veränderungen im Index → Commit



# Resultat

- Alle wieder auf dem gleichen Stand

Working-Tree

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

Index

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

Repository

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

# HEAD

## HEAD (mehr oder weniger)

Der neuste Commit in der Versionsgeschichte wird als `HEAD` bezeichnet.

# `git status` – Wie ist der Zustand?

## Status abfragen

```
git status
```

- Welche Dateien wurden modifiziert?
- Welche Veränderungen sind schon im Index?
- Gibt es Git nicht bekannte Dateien? (*untracked files*)

# Dateien dem Index hinzufügen

## Alle Veränderungen in einer Datei hinzufügen

```
git add datei
```

## Interaktives Hinzufügen

```
git add -p
```

## Interaktives Hinzufügen nur für eine Datei

```
git add -p datei
```

# Index: Unterschiede und Zurücksetzen

## Unterschiede zwischen Working-Tree und Index

```
git diff
```

## Unterschiede zwischen Index und `HEAD`

```
git diff --staged
git diff --cached
```

## Index zurücksetzen

```
git reset
```

# Index und Working-Tree manipulieren: `reset`

- `git reset` erlaubt die Manipulation ...
  - vom aktuellen `HEAD` (bzw. Branch)
  - des Indexes (Staging Area)
  - der Working-Tree

- *soft reset*
  - Ändert nur den `HEAD`
- *mixed reset* (default)
  - Ändert den `HEAD` und Index
  - Working-Tree bleibt unverändert
    - Eventuelle Änderungen werden nicht verworfen
- *hard reset*
  - Forciert den `HEAD`, den Index und die Working-Tree auf den selben Stand
  - Achtung: Hierdurch können Commits und Änderungen verloren gehen!

# `git reset`: Manipulationen am Index

## Den Index leeren

```
git reset
```

## Änderungen an *datei* aus dem Index löschen

```
git reset -- datei
```

## Änderungen zeilenweise aus dem Index entfernen

```
git reset -p
```

- Implizit beziehen sich alle Kommandos auf `HEAD`
- Diese Kommandos sind nicht-destruktiv
- Änderungen werden aus dem Index gelöscht (bleiben aber im Working-Tree erhalten
  - Gegenteil zu `git add`

# Ausgangszustand

- Veränderungen wurden im Working-Tree und Index gemacht

Working-Tree

```
 #!/usr/bin/python

+# Autor: Valentin
+
 print "Hello World!"
```

Index

```
 #!/usr/bin/python

+# Autor: Valentin
+
 print "Hello World!"
```

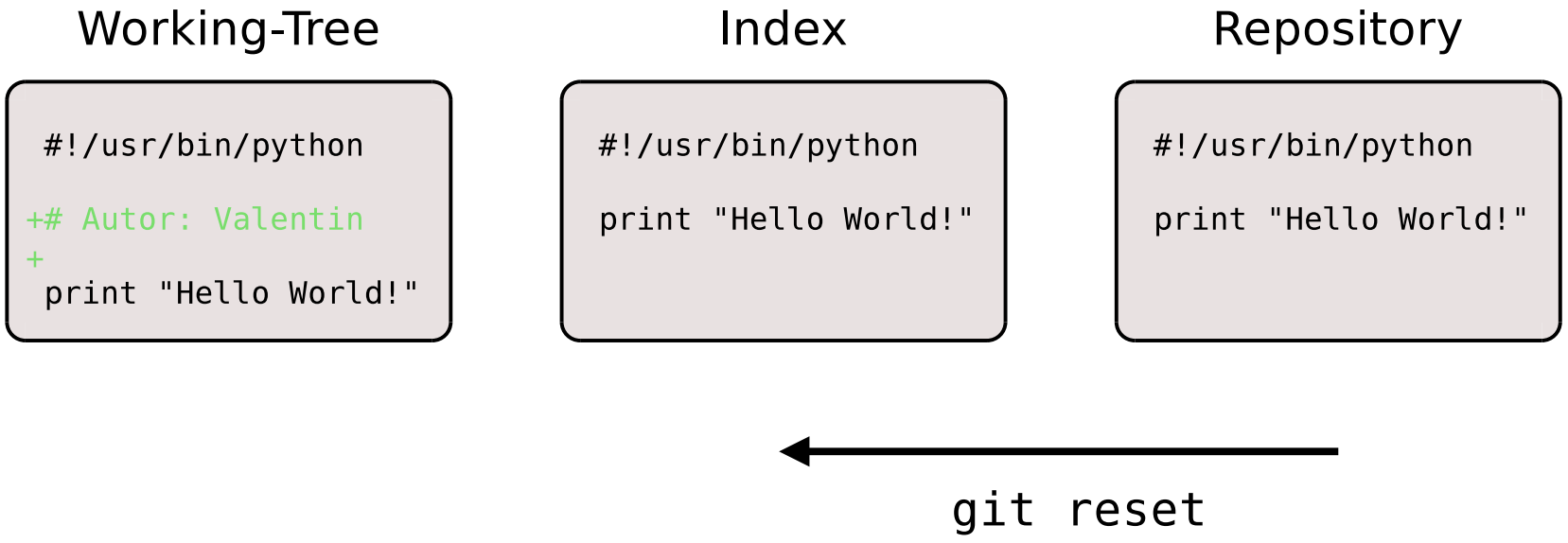
Repository

```
#!/usr/bin/python

print "Hello World!"
```

# Index zurücksetzten – `git reset`

- Veränderungen aus `HEAD` auf den Index abbilden



# `reset`: Destruktive Rezepte

## Änderungen im Index und im Working-Tree löschen

```
git reset --hard
```

## Zuletzt gemachten Commit und alle Änderungen verwerfen

```
git reset --hard HEAD^
```

- Ein harter Reset synchronisiert `HEAD`, Index und Working-Tree
- `git status` wird »nothing to commit« zurückgeben
- Änderungen am Working-Tree werden weggeworfen (und können nicht wiederhergestellt werden)

# Ausgangszustand

- Veränderungen wurden im Working-Tree und Index gemacht

Working-Tree

```
 #!/usr/bin/python

+# Autor: Valentin
+
 print "Hello World!"
```

Index

```
 #!/usr/bin/python

+# Autor: Valentin
+
 print "Hello World!"
```

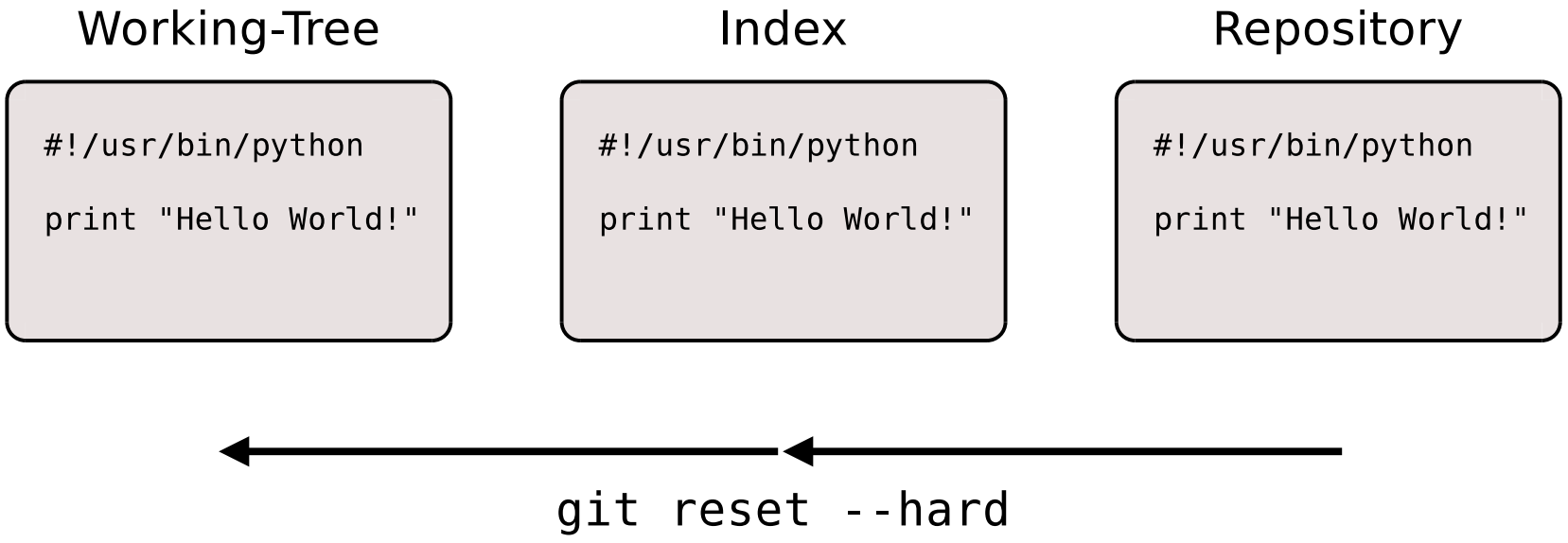
Repository

```
#!/usr/bin/python

print "Hello World!"
```

# Index zurücksetzten – `git reset --hard`

- Veränderungen aus `HEAD` auf den Index und WT abbilden



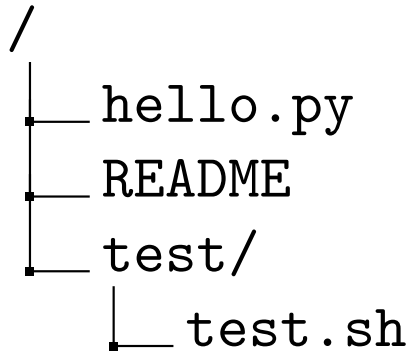
# Übersicht



Jetzt geht es an die Interna

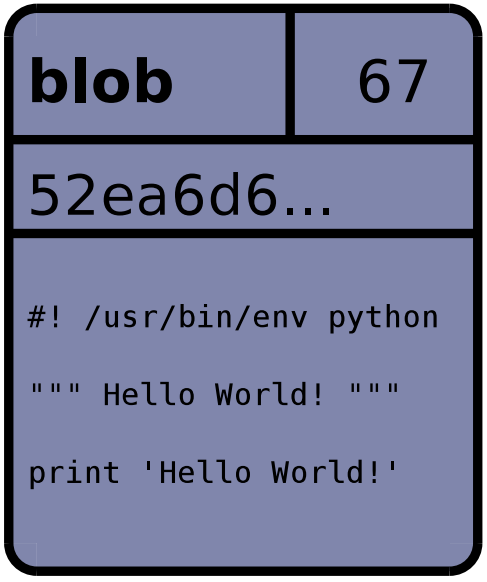
# Was wollen wir speichern?

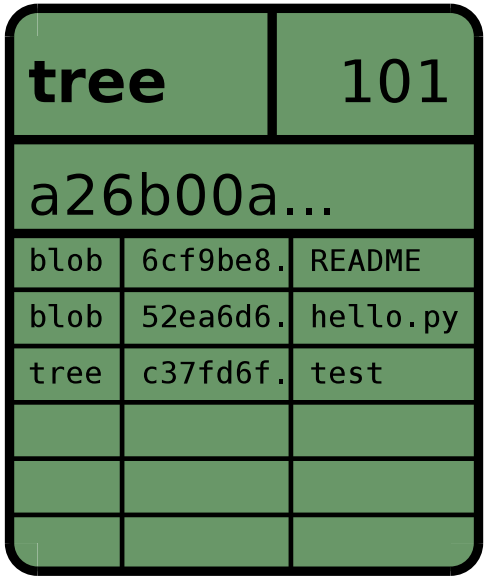
Angenommen, wir wollen folgendes Verzeichnis speichern:



# Objektmodell

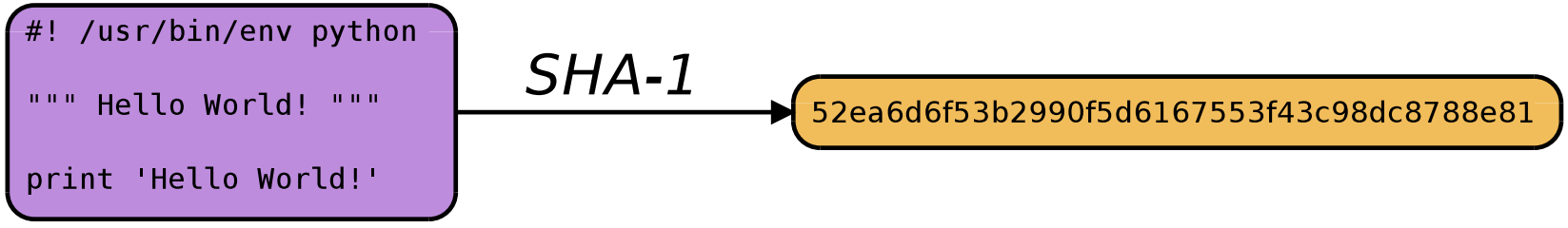
- *Blob*: Enthält den Inhalt einer Datei
- *Tree*: Eine Sammlung von Tree- und Blob-Objekten
- *Commit*: Besteht aus einer Referenz auf einen Tree mit zusätzlichen Informationen
  - *Author* und *Commiter*
  - *Parents*
  - *Commit-Message*




# SHA-1 IDs

- Objekte werden mit *SHA-1 IDs* identifiziert
- Dies ist der *Objekt-Name*
- Wird aus dem Inhalt berechnet
- *SHA-1* ist eine sogenannte Hash-Funktion; sie liefert für eine Bit-Sequenz mit der maximalen Länge von $2^{64} - 1$ Bit ($\approx 2$ Exbibyte) in eine Hexadezimal-Zahl der Länge 40 (d. h. 160 Bits)
- Die resultierende Zahl ist eine von $2^{160} (\approx 1.5 \cdot 10^{49})$ möglichen Zahlen und ziemlich einzigartig



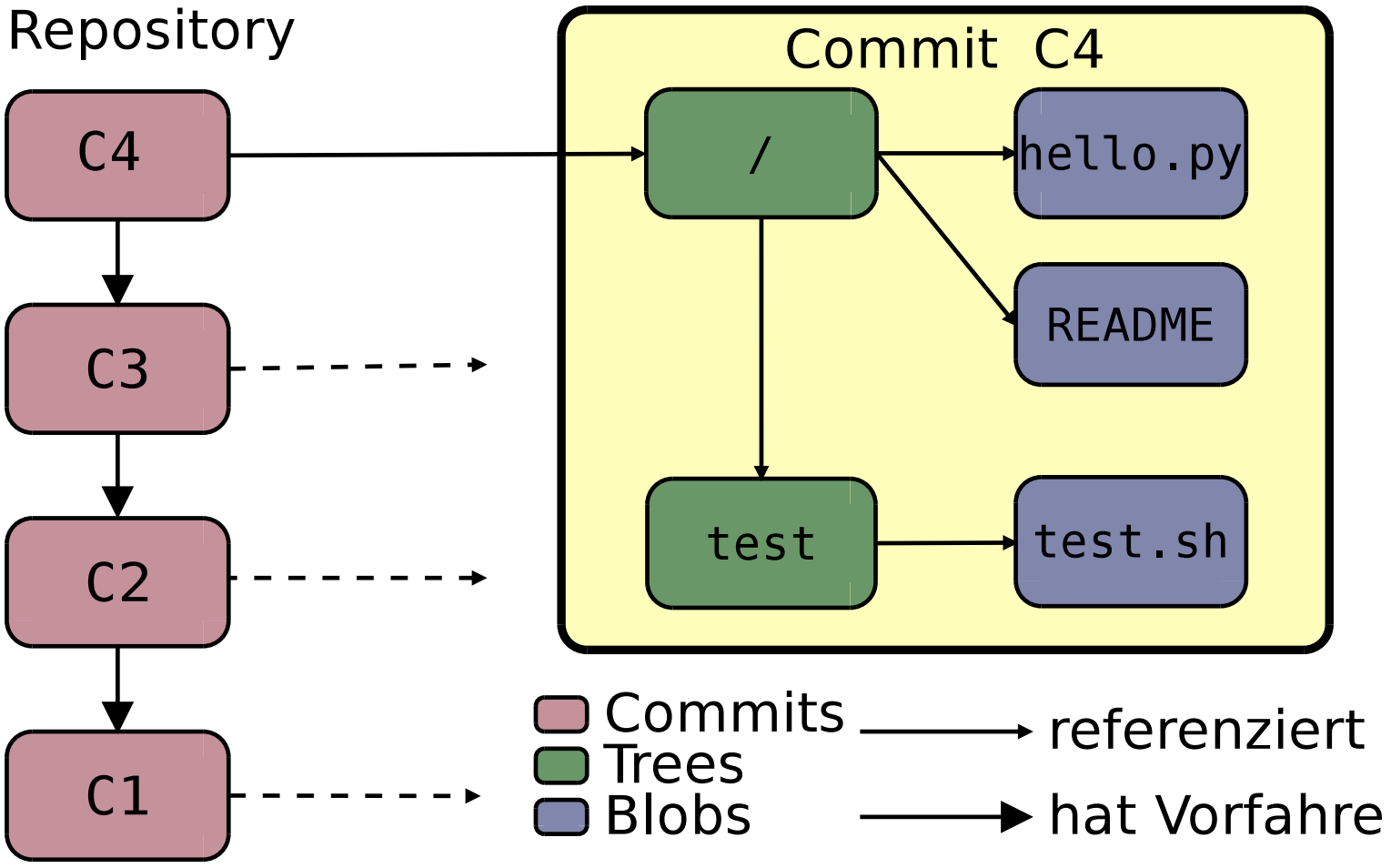
# Objektverwaltung

- Alle Objekte werden von Git in der *Objektdatenbank* (genannt Repository) gespeichert
- Die Objekte sind durch ihre SHA-1-ID eindeutig adressierbar

- Für jede Datei erzeugt Git ein Blob-Objekt
- Für jedes Verzeichnis erzeugt Git ein Tree-Objekt
- Ein Tree-Objekt enthält die Referenzen (SHA-1-IDs) auf die in dem Verzeichnis enthaltenen Dateien

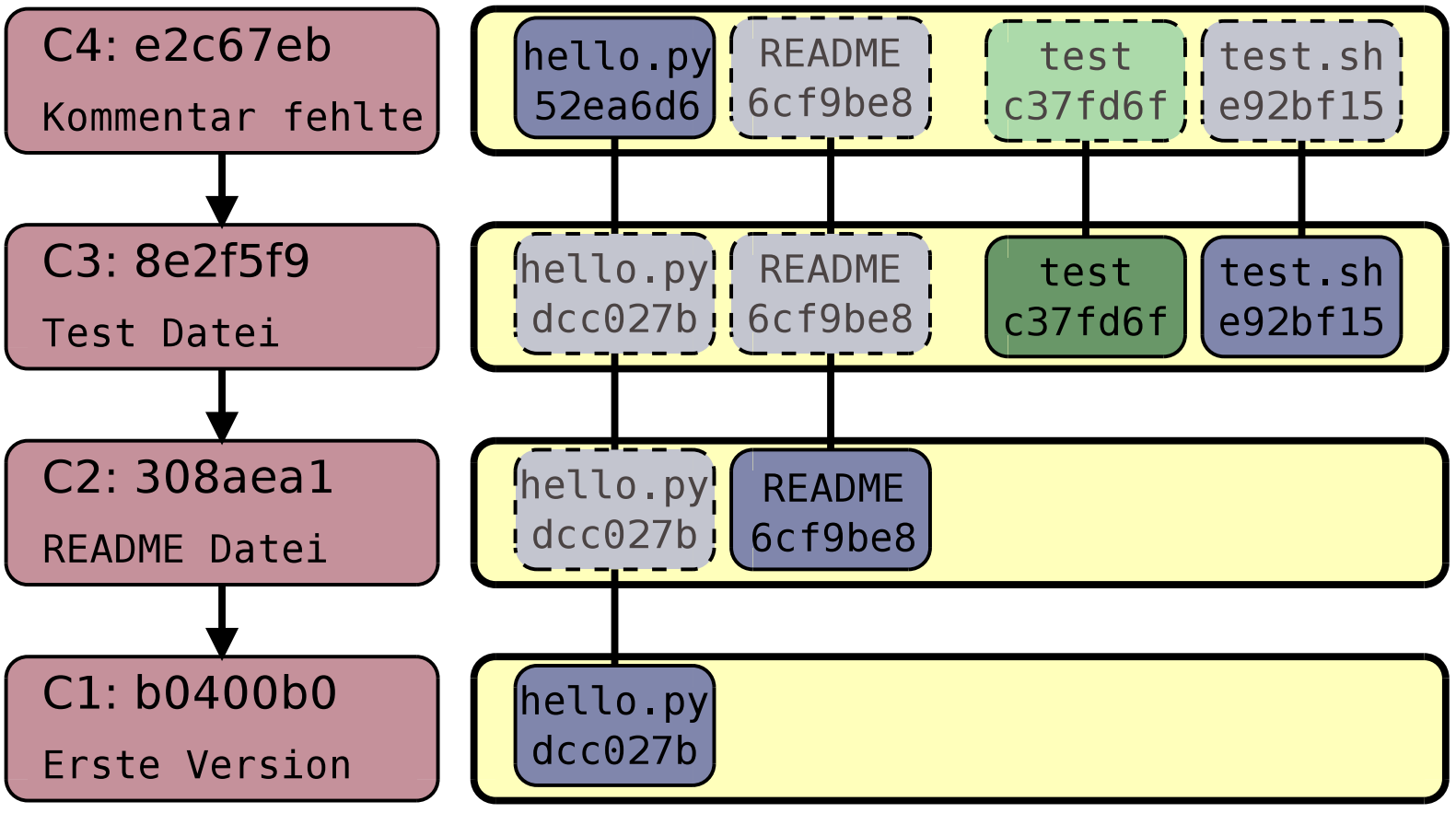
# Zusammenfassung

Ein Git-Repository enthält Commits; diese wiederum referenzieren Trees und Blobs, sowie ihren direkten Vorgänger



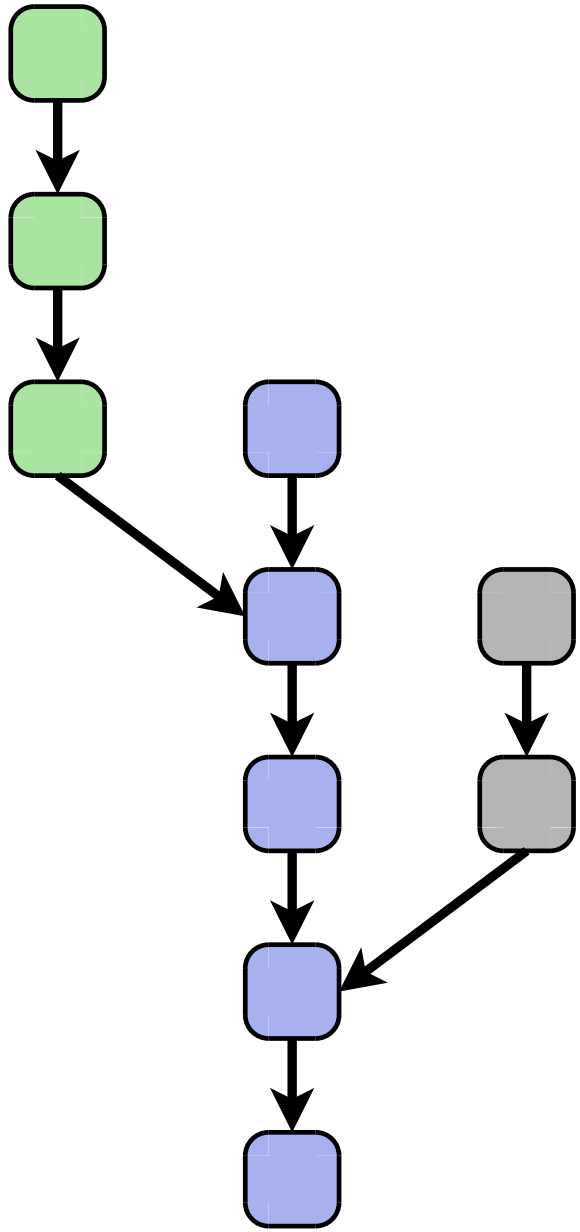
# Commit = Dateibaum

Ein Commit hält den Zustand *jeder* Datei zum gegebenen Zeitpunkt fest. (Auch den der nicht geänderten.)



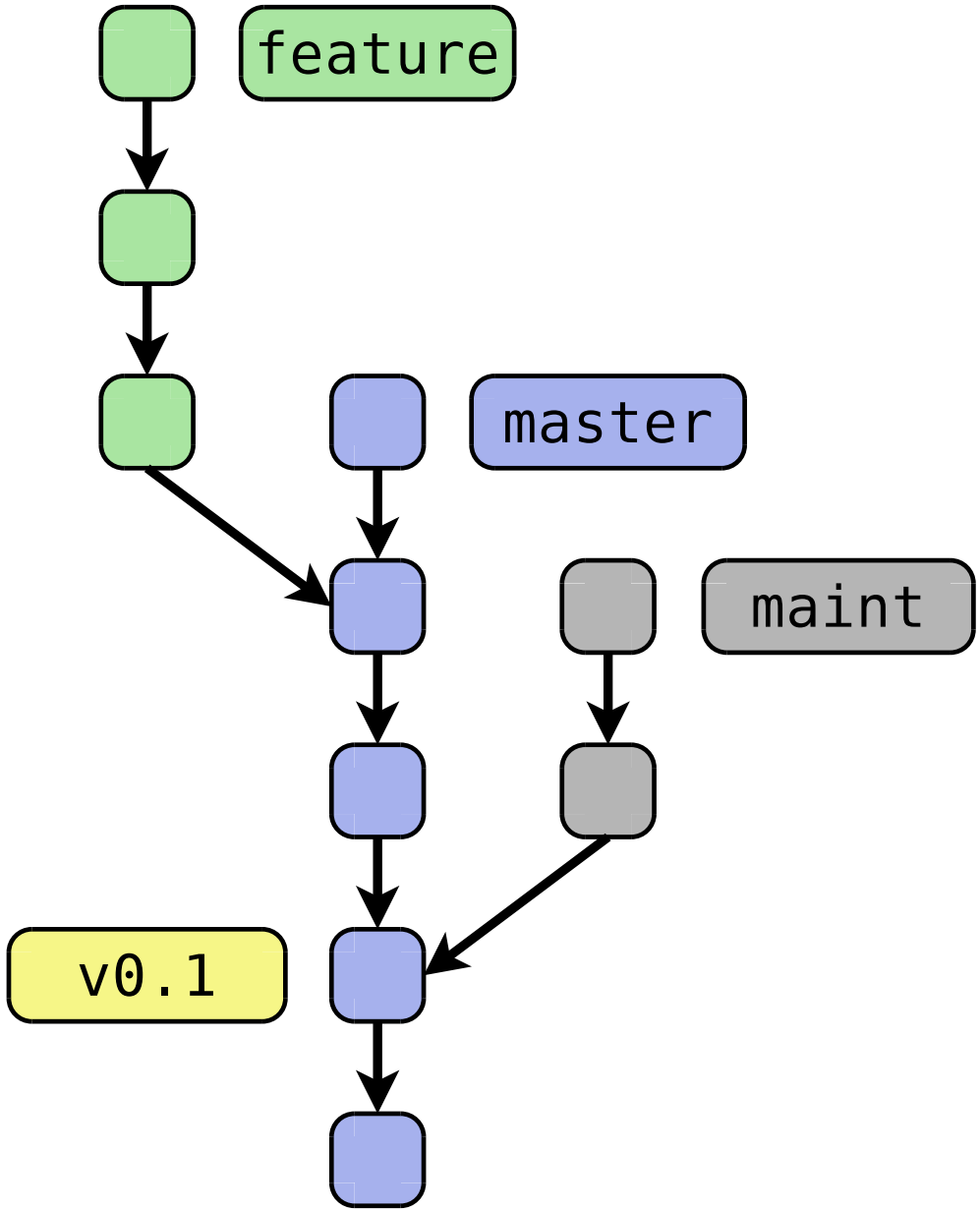
# Commit Graph

Ein Repository ist ein *Gerichteter Azyklischer Graph* (DAG)



# Branches und Tags

Branches und Tags sind Zeiger

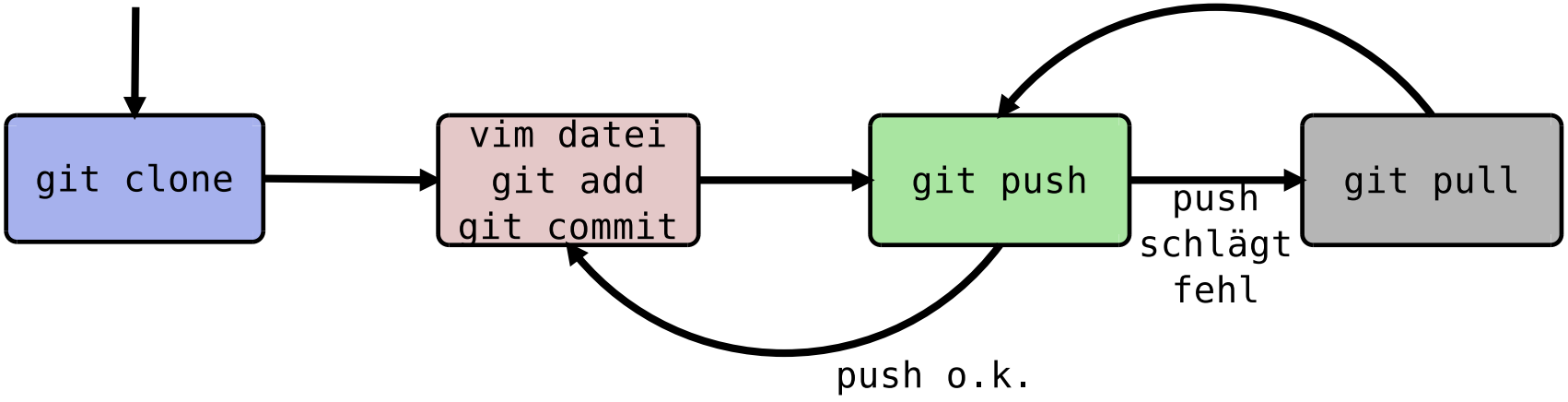


# Graph-Struktur

- Die gerichtete Graph-Struktur entsteht, da in jedem Commit Referenzen auf direkte Vorfahren gespeichert sind
- Integrität kryptographisch gesichert
- Git-Kommandos manipulieren die Graph-Struktur

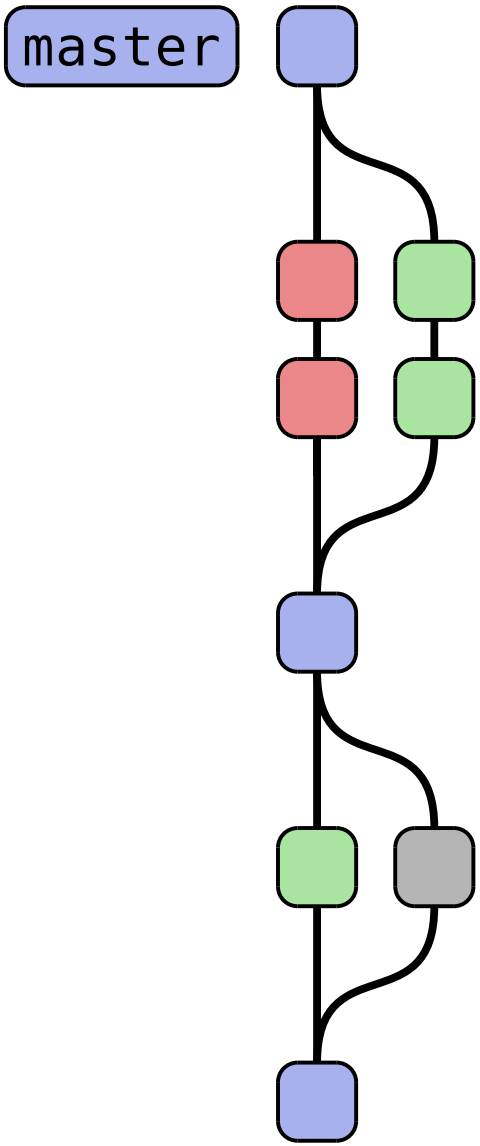
# Übersicht

# Push'n'pull Workflow



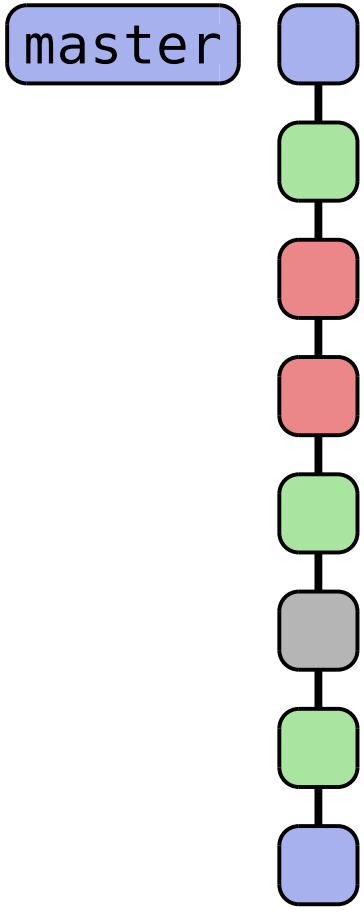

1. Lokale Änderungen
   - `vim` *`datei`*
   - `git add` *`datei`*
   - `git commit -m "`*`msg`*`"`
2. Änderungen veröffentlichen
   - `git push`
   - Wenn `push` fehlschlägt
   - `git pull`, dann `git push`

# 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

# Übersicht

# Branching einfach gemacht

- Branches funktionieren in Git schnell und intuitiv
  - Revolutioniert die Arbeitsweise (Workflow)
- Ein Branch ist *keine* komplette Kopie des Projektes
  - »Branching is cheap«
- Vorstellung eher: Ein »Label«, das man an einen Commit heftet

## Branches auflisten

```
git branch [-v]
```

- Wir arbeiten die ganze Zeit schon im Branch `master`

# Branches erstellen

- Keine Dateien werden kopiert
  - Erstellung dauert wenige Millisekunden

## Einen Branch erstellen

```
git branch name
```

## Ausgangscommit des Branches explizit angeben

```
git branch name start
```

# Branches wechseln

- Um auf einem Branch zu arbeiten, wird er »ausgecheckt«

## Branch auschecken

```
git checkout branch
```

- Für SVN-Umsteiger
  - Das aktuelle Verzeichnis wird *nicht* gewechselt
  - Stattdessen: Inhalt des Branches → Working-Tree

## Branch erstellen und auschecken

```
git checkout -b name
```

# Branches manipulieren

## Branch umbenennen

`git branch -m` *alt neu*

`git branch -M` *alt neu* (forciert)

## Branch löschen

`git branch -d` *name*

`git branch -D` *name* (forciert)

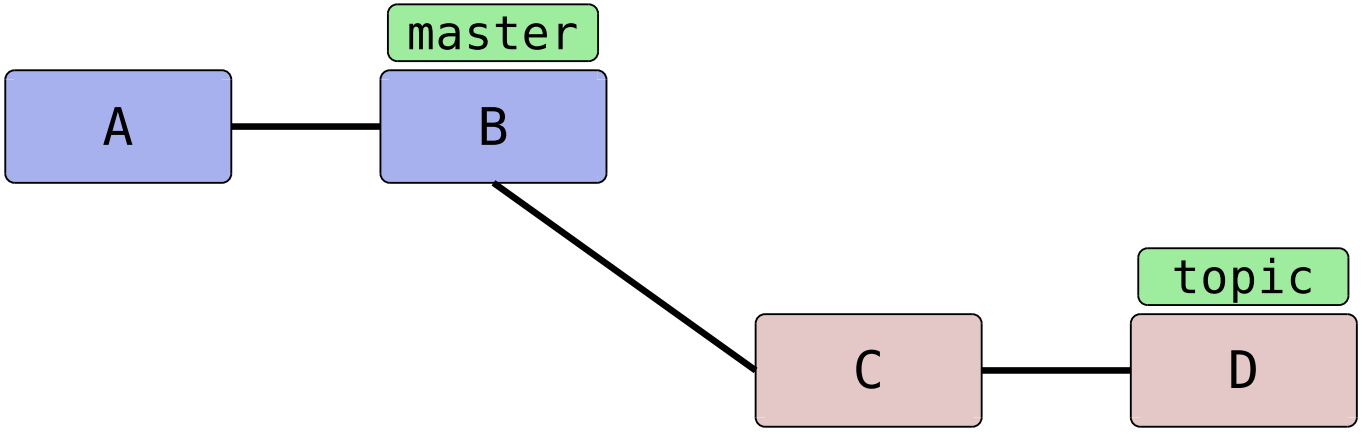
- Die forcierte Version (`-M` bzw. `-D`) ist dann notwendig, wenn Commits oder Branches überschrieben werden

# Merge

`git merge` fügt zwei oder mehr Branches zusammen

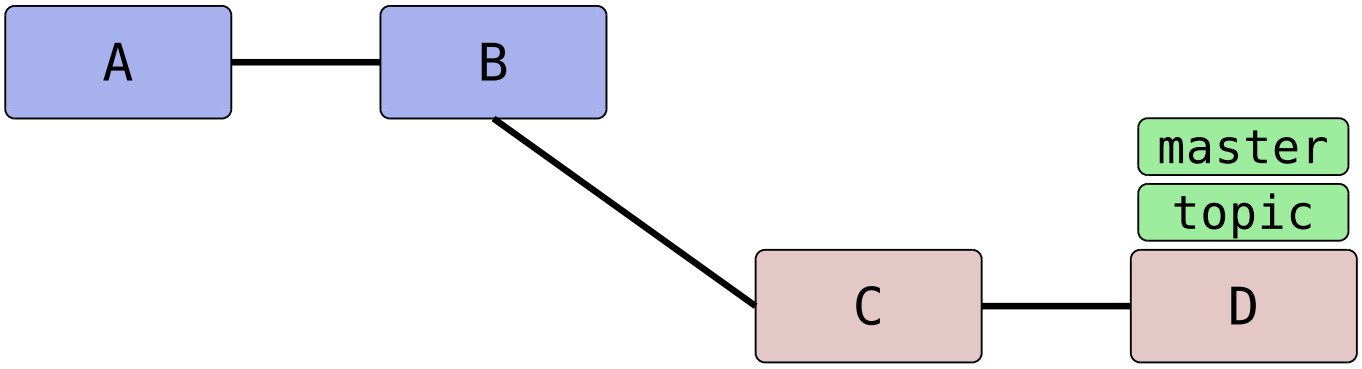
- *Fast-Forward*: Es existieren keine Abzweigungen, und der Branch wird einfach »weitergerückt«
- Sonst wird ein *Merge-Commit* erstellt, der beide Branches als »Parents« referenziert
  - Treten Konflikte auf, werden diese in dem Merge-Commit behoben
  - Andernfalls ist der Merge-Commit »leer«
- Die weitere Entwicklung basiert auf den Commits *beider* Branches
  - Die Branches können also nicht wieder entkoppelt werden

# Vor dem Fast-Forward



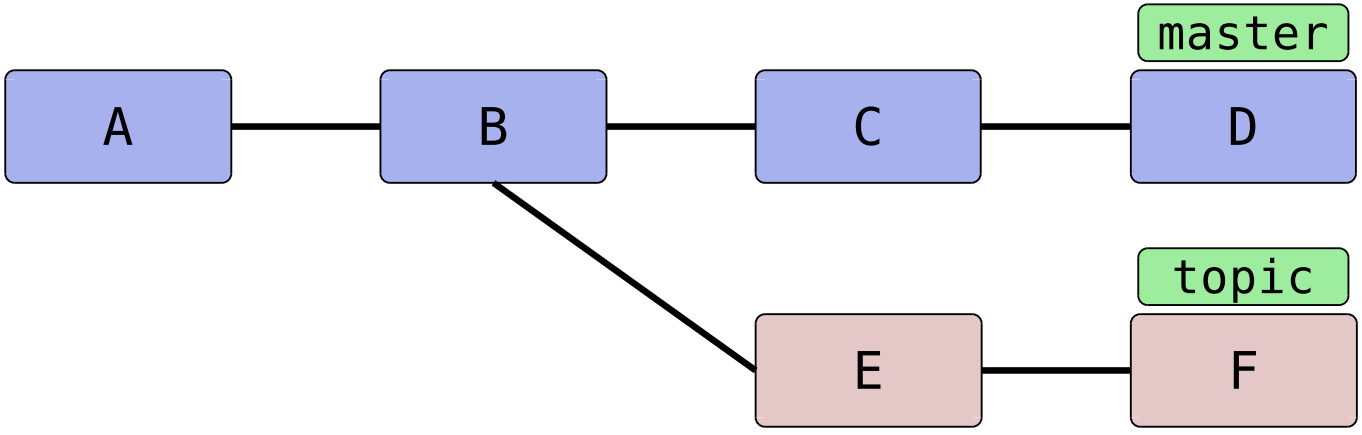

- In `master` hat sich nichts getan, `topic` ist fertig

# Nach dem Fast-Forward



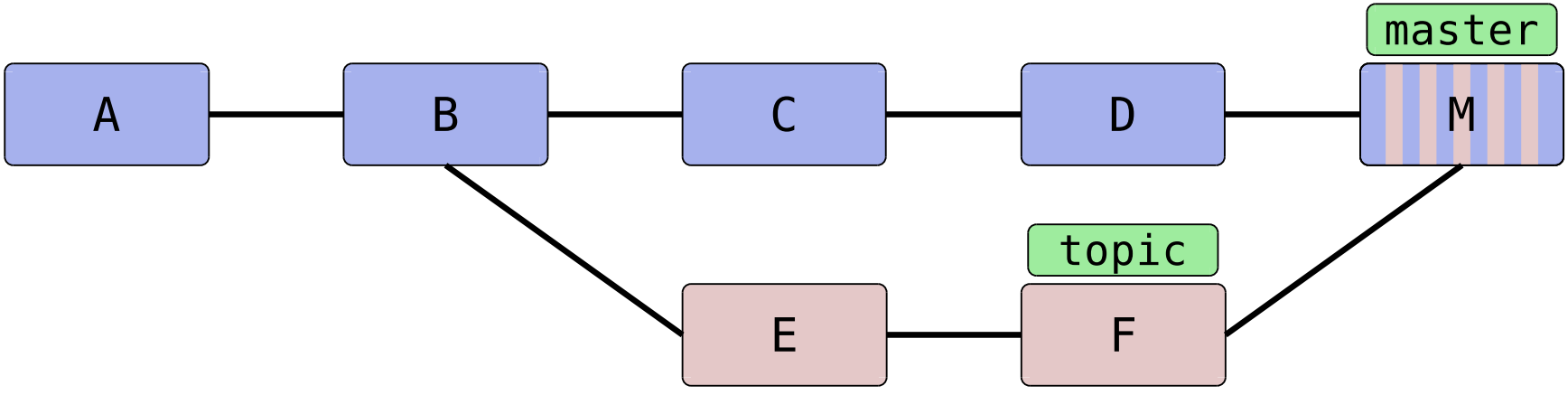

- `master` wird »weitergerückt«, bzw. »vorgespult«

# Vor dem Merge




- `topic` ist fertig und soll in `master` integriert werden

# Nach dem Merge




- Im `master` ausführen: `git merge topic`

# Merge-Commit

- Konfliktlos: Merge-Commit enthält keine Änderungen
  - Allerdings werden zwei oder mehr Branches als »Parent« referenziert
- Beschreibung wird automatisch erstellt
  - Falls andere Beschreibung gewünscht:
    `git merge -m` *`nachricht`* `...`

**Zusammenfassung aller Commits in Merge-Nachricht schreiben**

```
git config --global merge.log true
```

# Merge-Strategien

Git kennt mehrere Strategien, um einen Merge auszuführen. Die beiden wichtigsten sind:

- **recursive**
  - Standard-Strategie: 3-Wege-Merge, platziert im Konfliktfall entsprechende Marker in den betroffenen Dateien
- **octopus**
  - Strategie, um mehrere Branches zusammenzufügen. Bricht bei Konflikten ab.

# Konflikte lösen: Manuell

- `git checkout master`
- `git merge` *`topic`*
  - ... bricht mit einem Konflikt in *`datei`* ab
- `$EDITOR` *`datei`*
  - Suche nach den Markern >>>>>, <<<<< und =====
  - Behebung des Konfliktes
- `git add` *`datei`*
- `git commit`
  - Konflikt und Lösung beschreiben

# Konflikte lösen: Mergetool

## Ein Mergetool konfigurieren

```
git config --global merge.tool vimdiff
```

- `git merge` *`topic`*
  - ... bricht mit einem Konflikt in *`datei`* ab
- `git mergetool`
  - Konflikt lösen
- `git commit`

# Konflikte lösen: automatisch

- Die Merge-Strategie *recursive* kennt zwei Optionen, die einen Merge automatisch lösen können
- Treten konfliktierende Hunks (Bündel geänderter Zeilen) auf, bevorzugt
  - **ours** die Änderungen aus dem aktuellen Branch
  - **theirs** die Änderungen aus dem zu integrierenden Branch

## Den Änderungen aus `master` Vorzug geben

```
git checkout master
git merge -X ours topic
```

## Den Änderungen aus *topic* Vorzug geben

```
git checkout master
git merge -X theirs topic
```

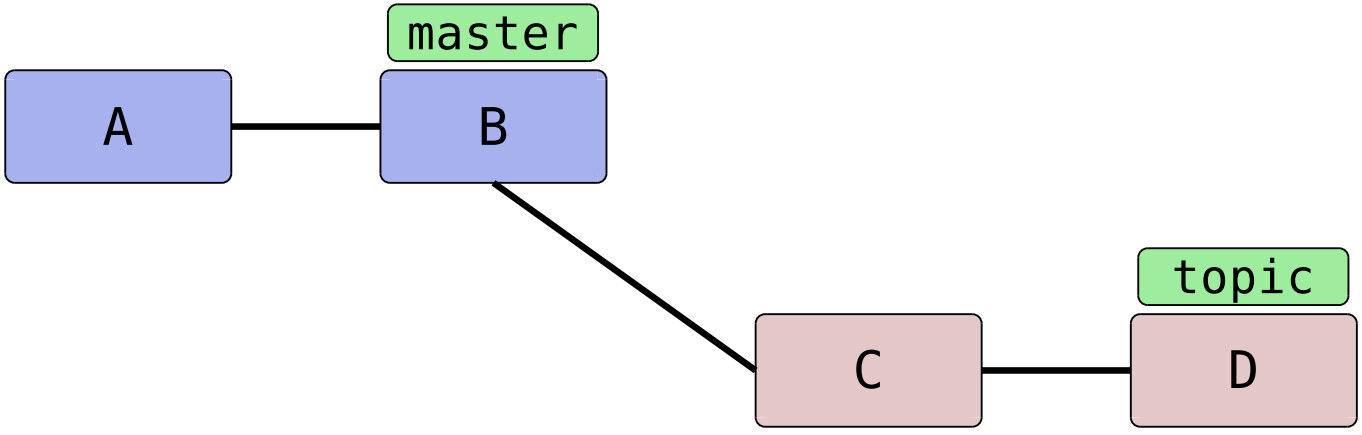
# Merge-Commit Forcieren

Mit `git merge --no-ff` wird in jedem Fall ein Merge-Commit erstellt, auch wenn ein Fast-Forward möglich wäre.

Das ist sehr sinnvoll:

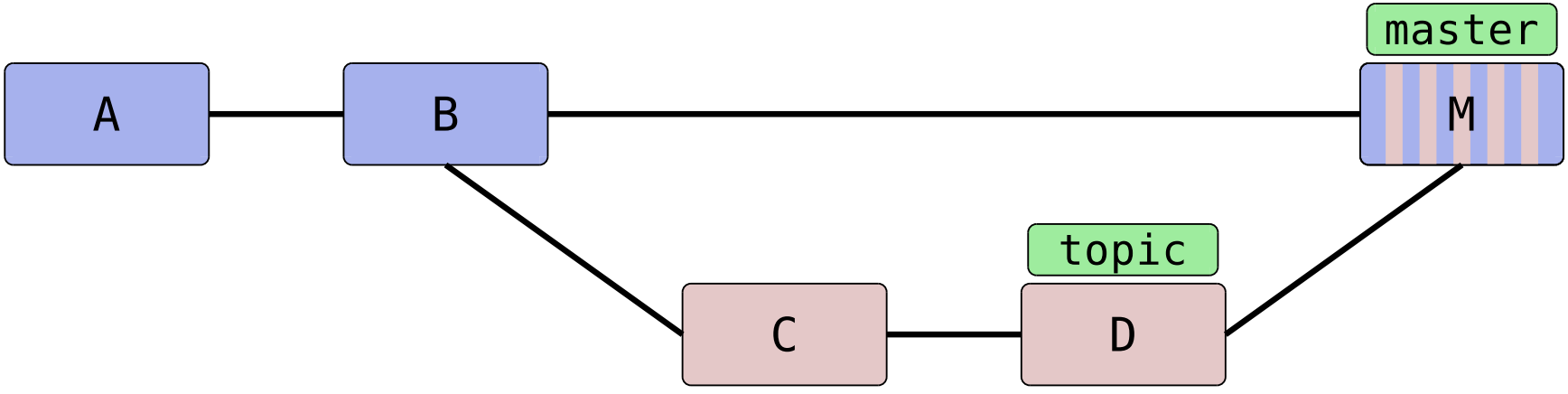
- Die Integration eines Feature-Branches deutlich machen
- In der Ansicht von `git log` wird die Geschichte immer linear dargestellt
- In der Baumansicht sieht man die Abzweigung und Zusammenführung

# Vor dem Force-Merge




- In `master` hat sich nichts getan, `topic` ist fertig

# Nach dem Force-Merge




- Ein Merge-Commit wurde forceirt

# Rebase: Auf eine neue Basis stellen

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

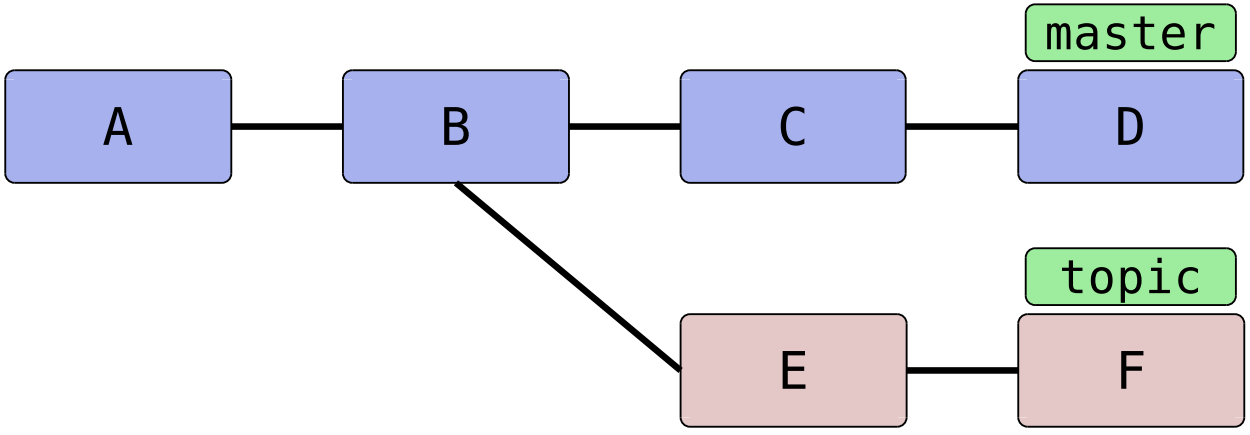
**`master` als neue Basis für *`topic`***

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

**Alternativ**

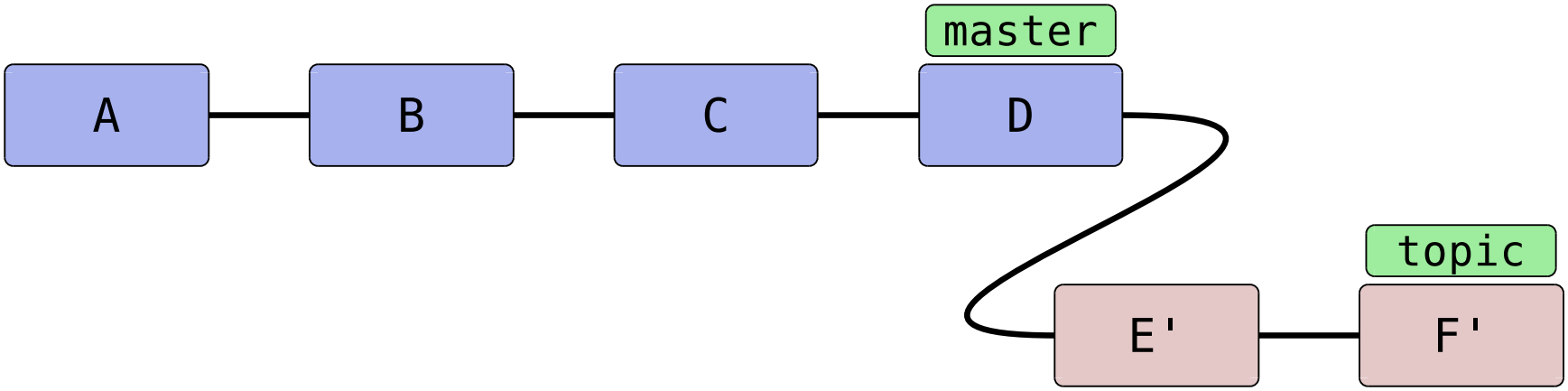
```
git rebase master topic
```

# Vor dem Rebase




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

# Nach dem Rebase



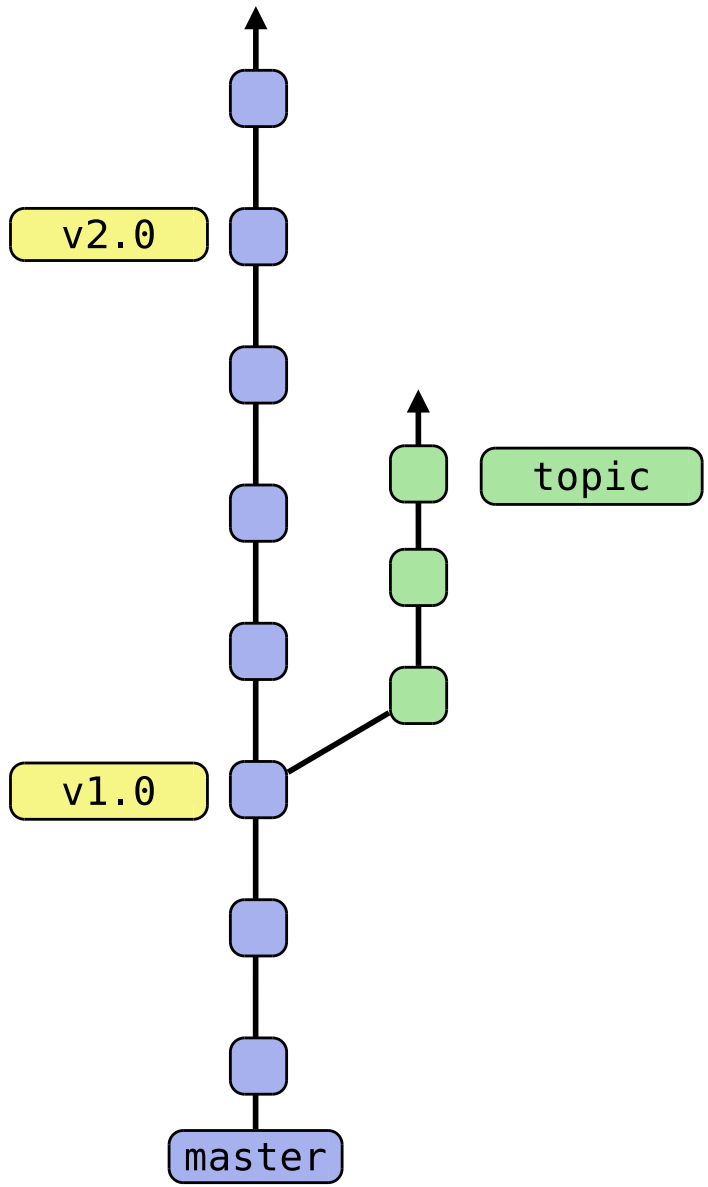

- `git rebase master` *topic*

# Rebase: Wozu?

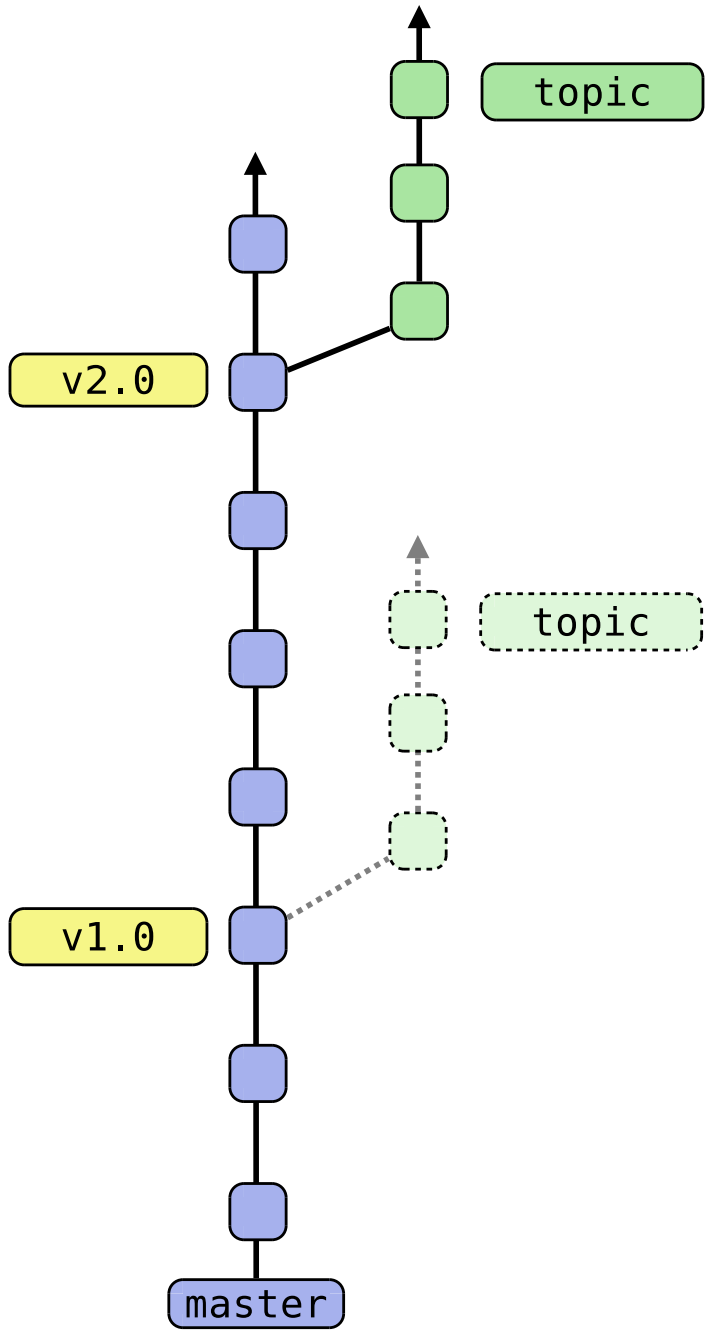
Mit einem Rebase kann man »die Geschichte umschreiben« – aber wozu?

- 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

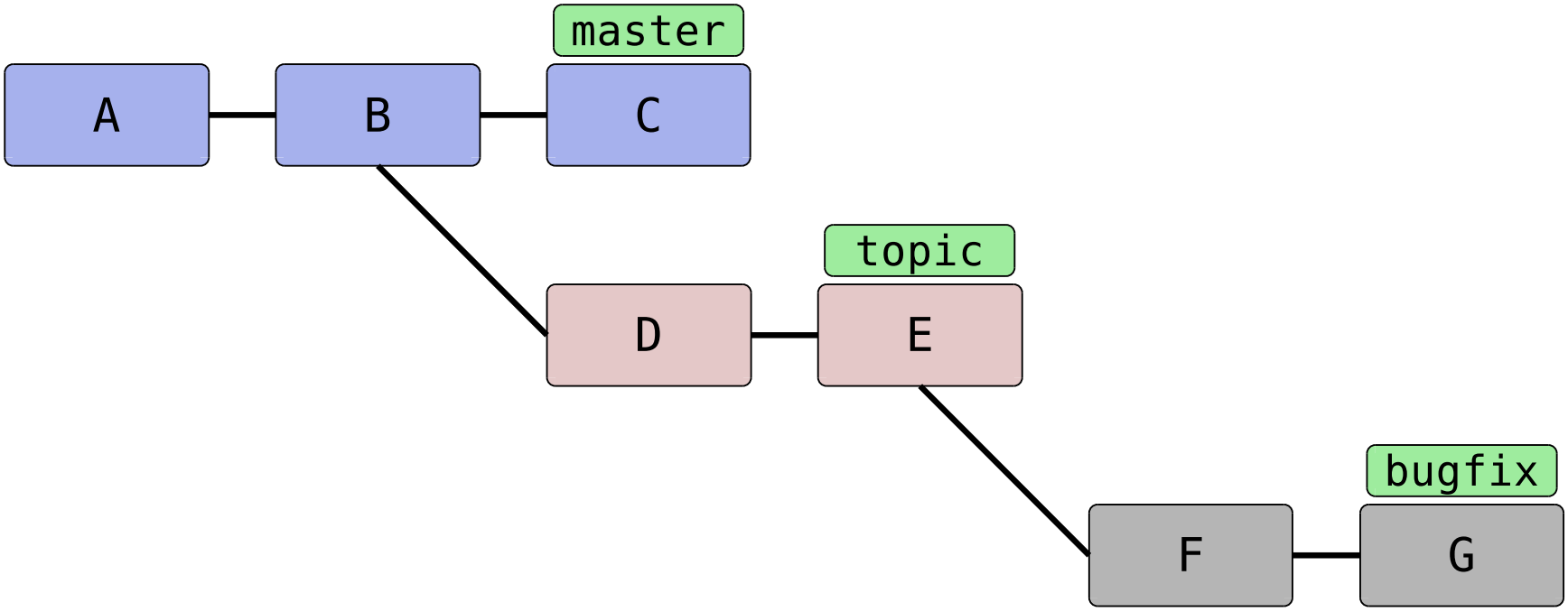
# Rebase Onto: Commits »transplantieren«

- 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`

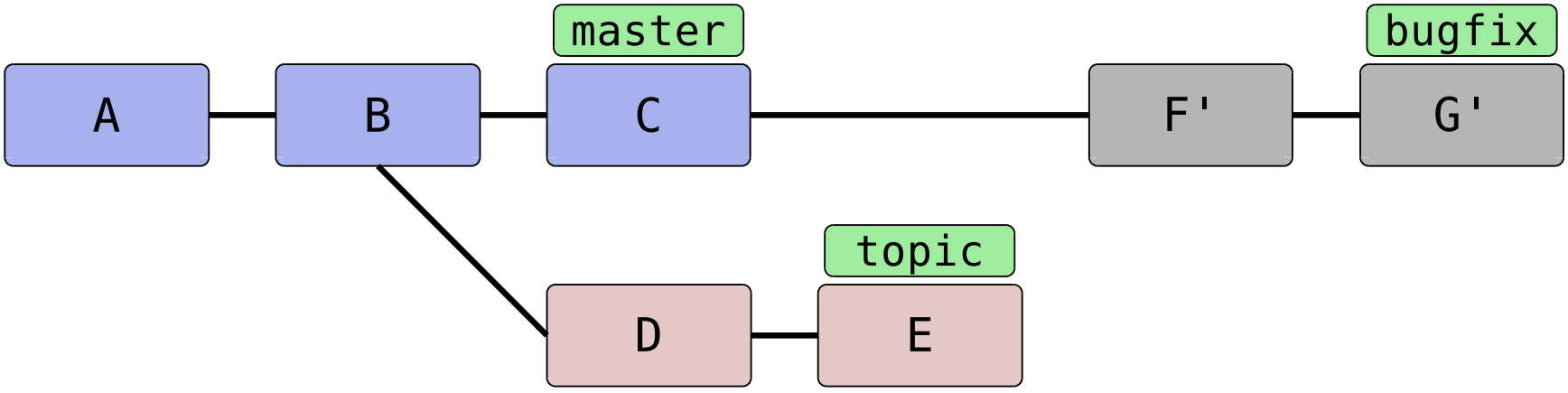
**Teile eines Branches »transplantieren«**

```
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«

# 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
```

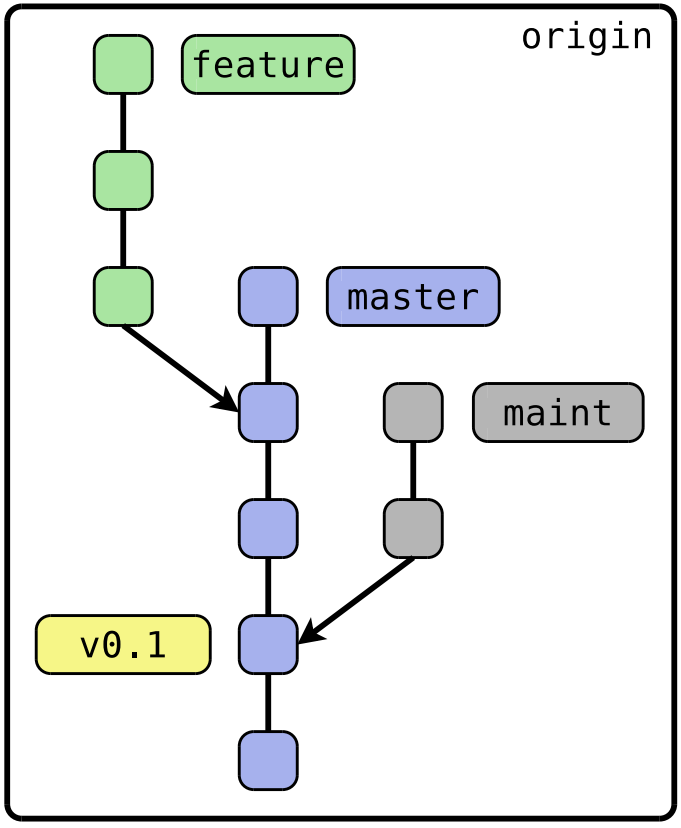
# Remote-Tracking Branches

- 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

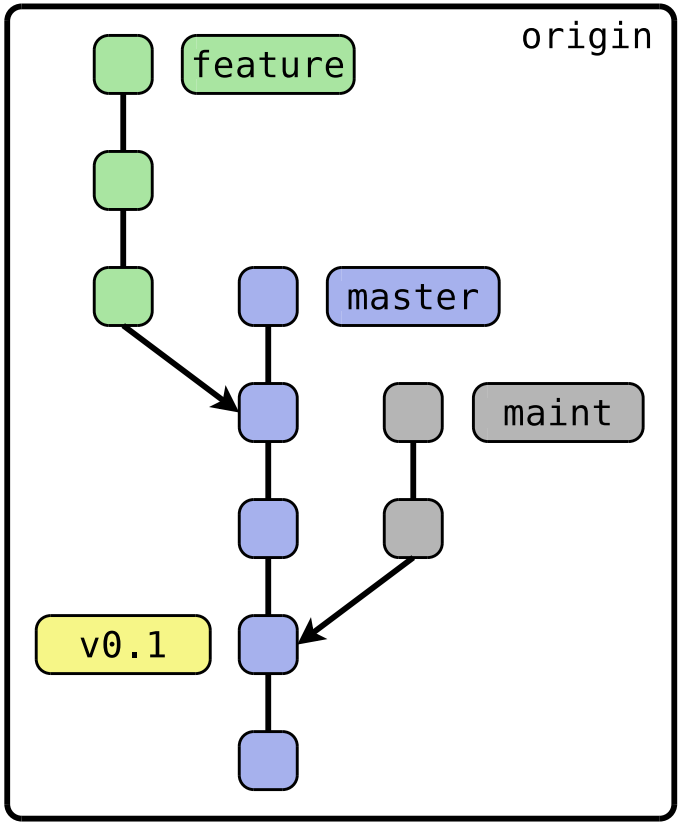
**`origin/master`**

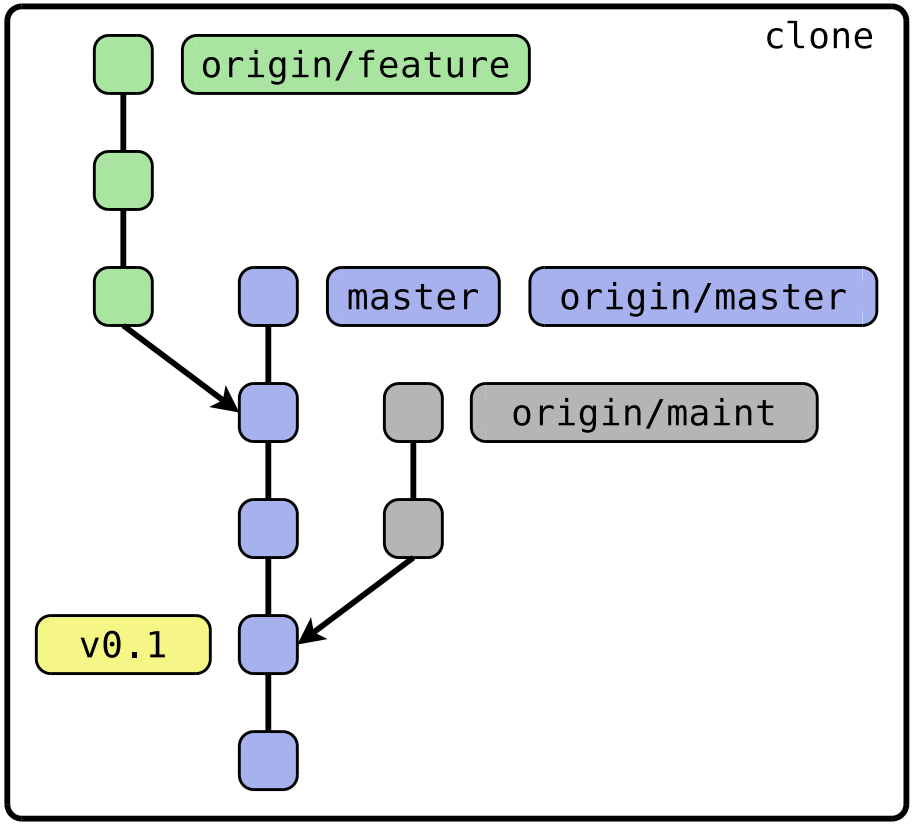
`origin/master` ist der *Remote-Tracking-Branch* des Branches `master` aus dem Remote `origin`

# Vor `git clone`



# Nach `git clone`



# `git push` – Änderungen hochladen

## Änderungen im *branch* nach *remote* hochladen

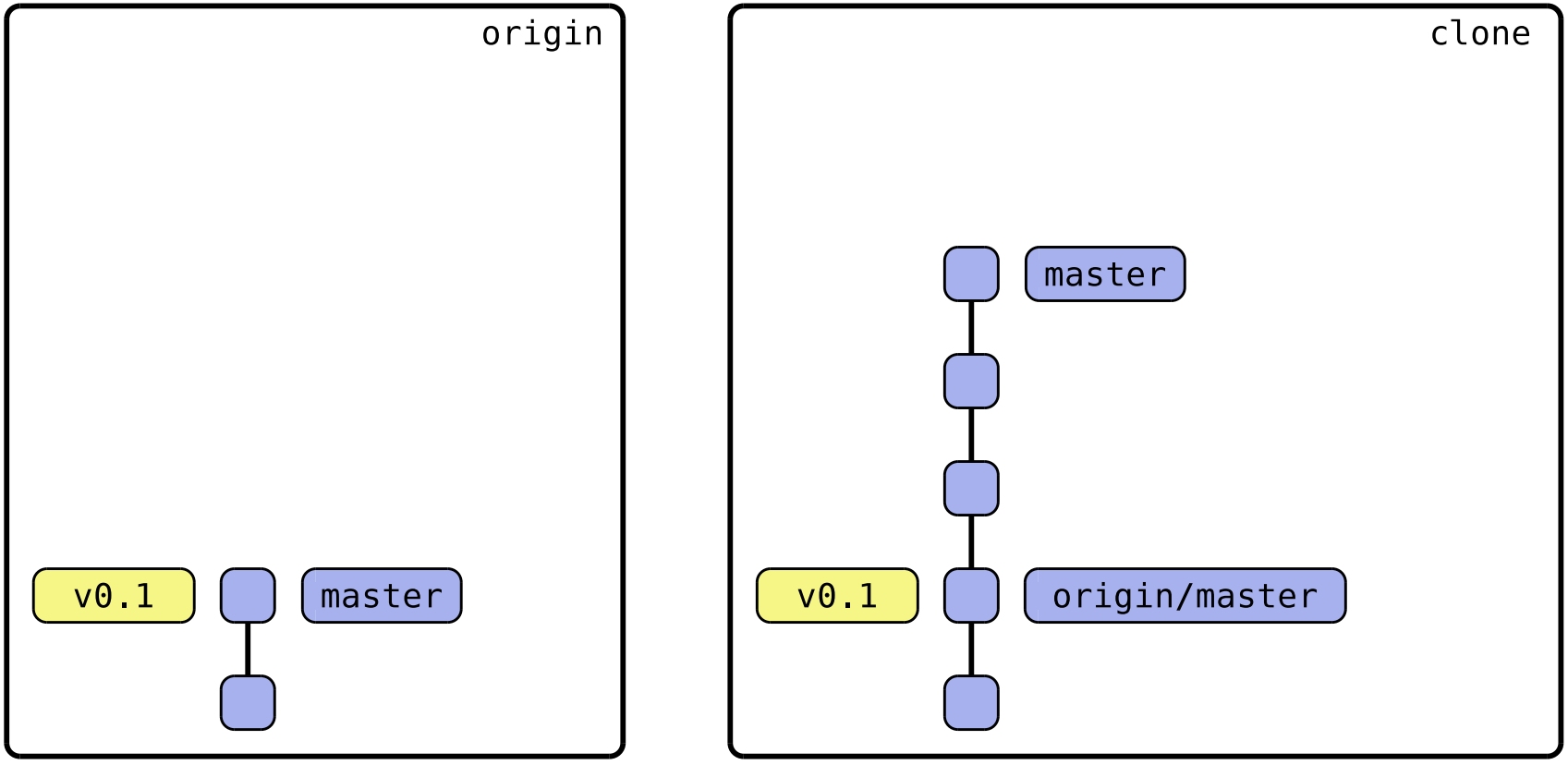
```
git push remote branch
git push origin master
```

## Soll der Branch im Remote anders heißen

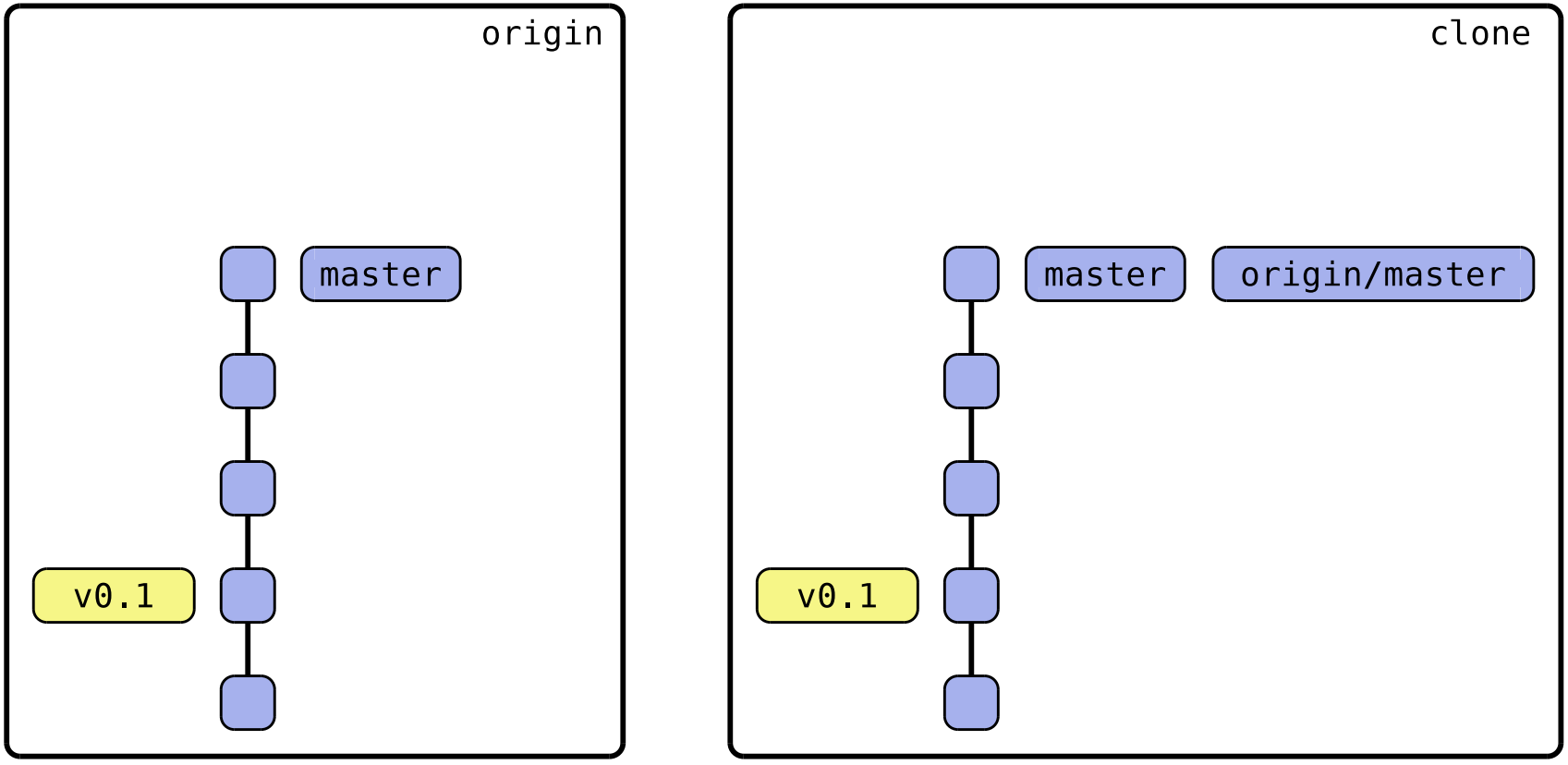
```
git push remote branch-lokal:branch-remote
```

- 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**

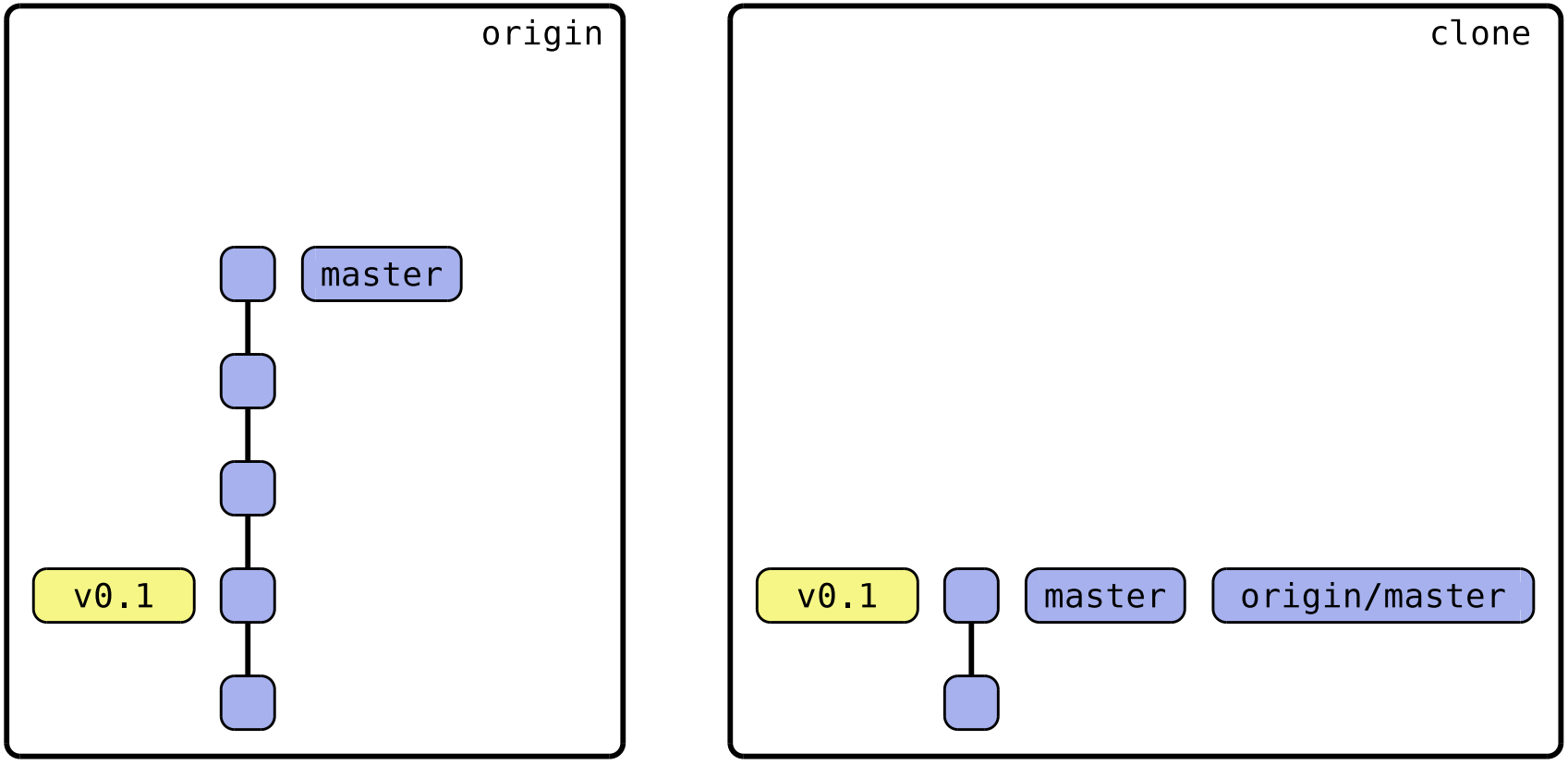
```
git fetch remote
```

**Veränderungen aus allen Remotes herunterladen**

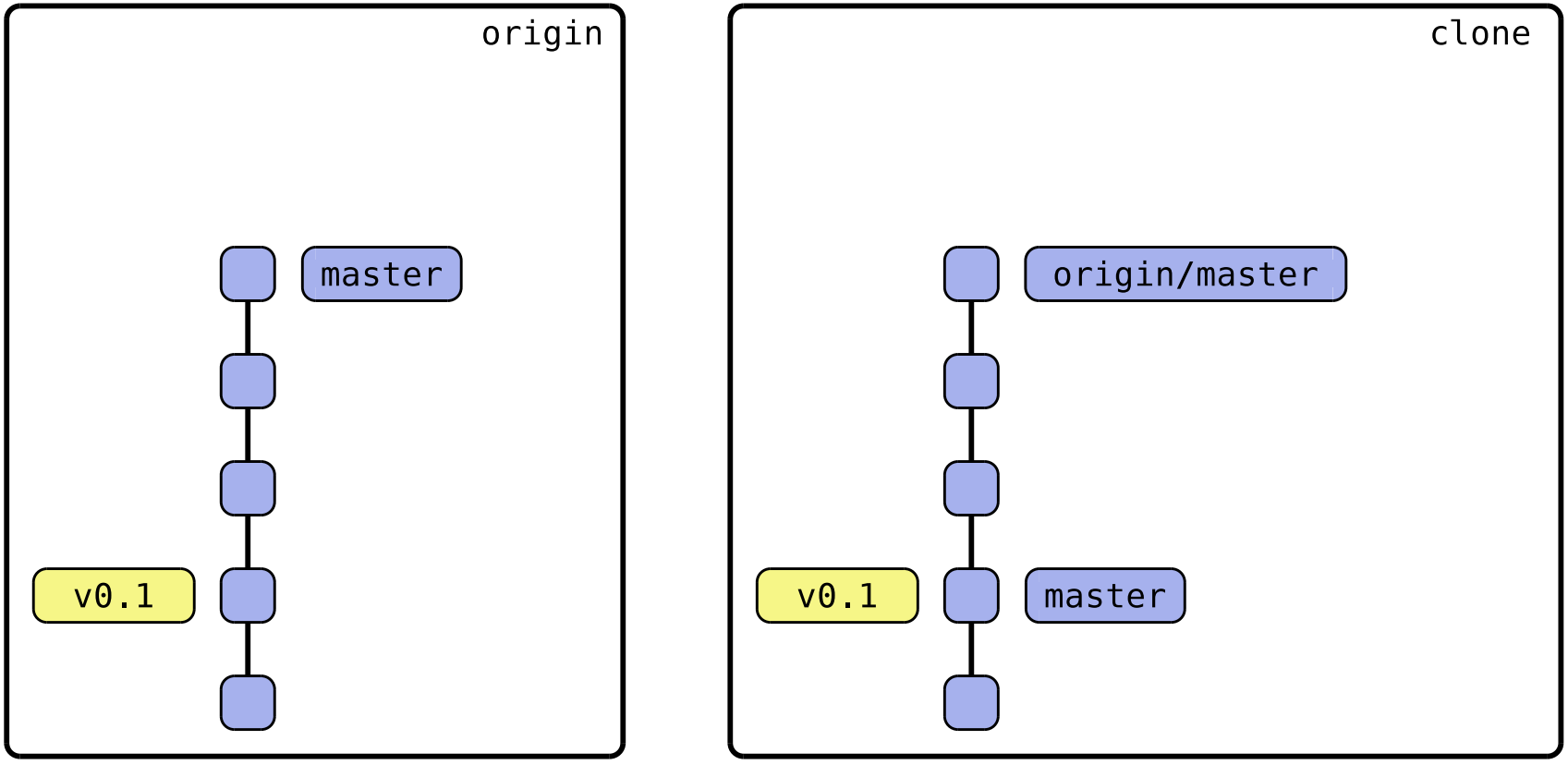
```
git remote update   (oder alternativ)
git fetch --all
```

- 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`

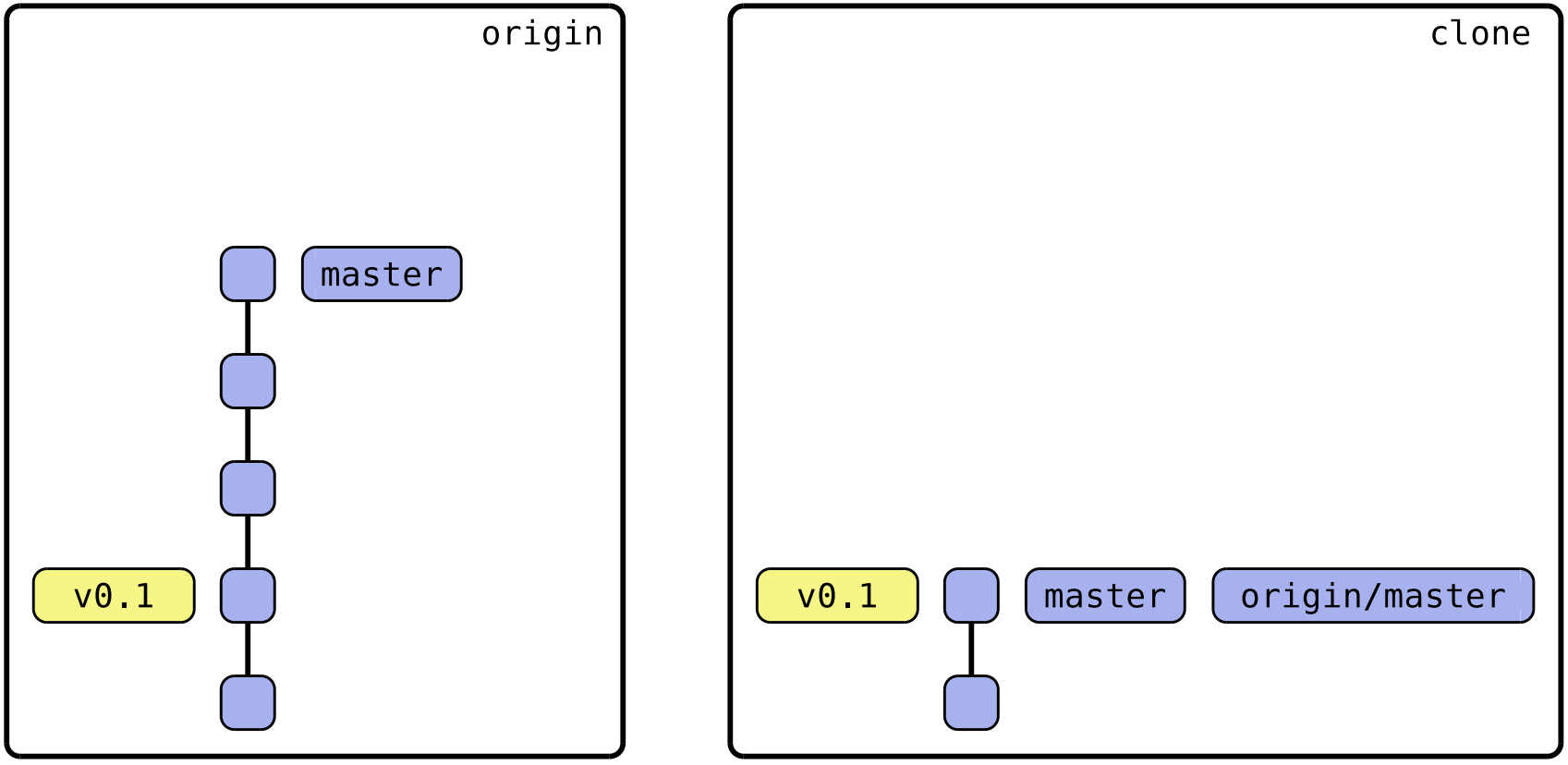
**Fetch und Merge**

```
git pull
```

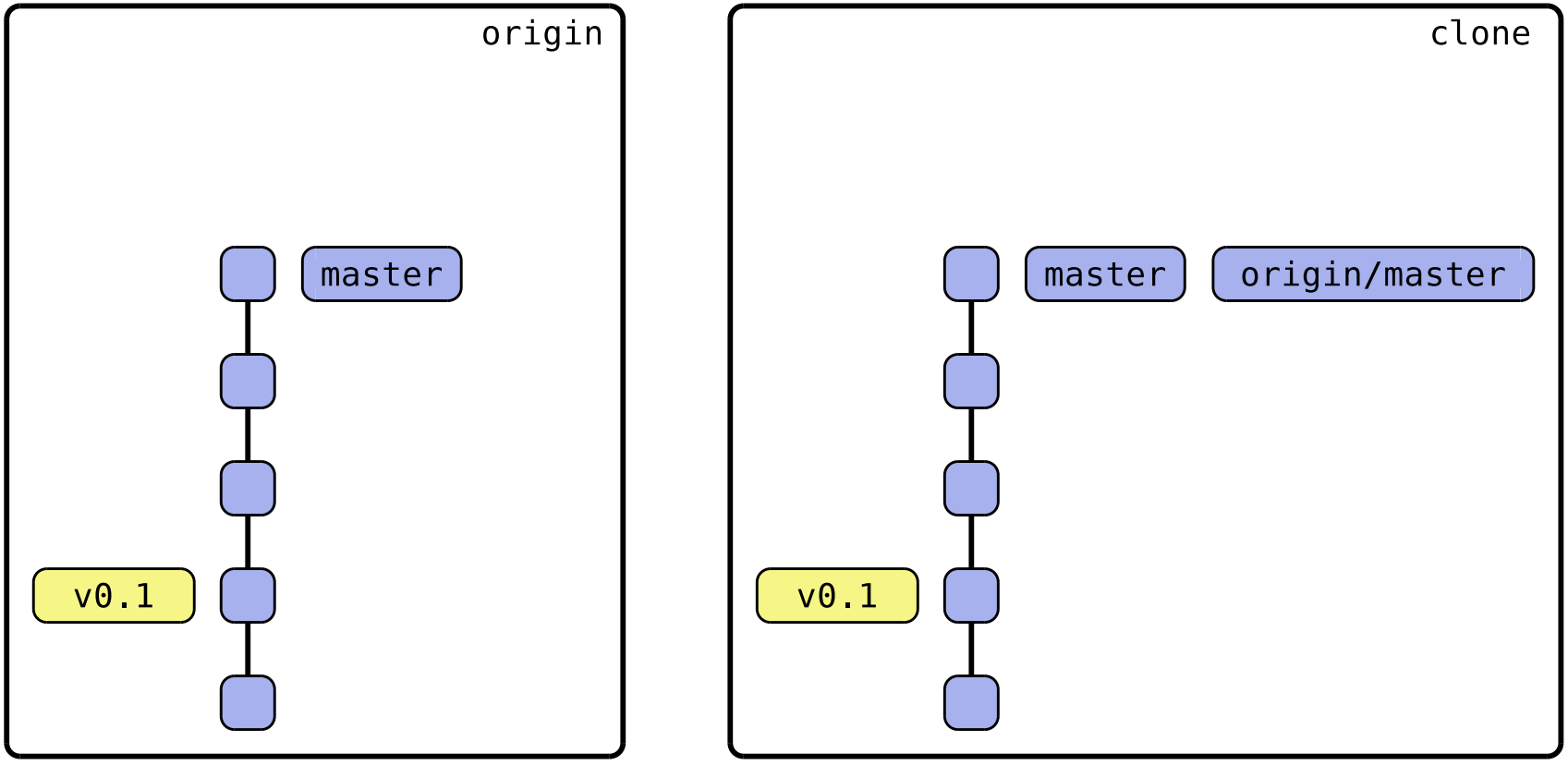
**Fetch und Rebase**

```
git pull --rebase
```

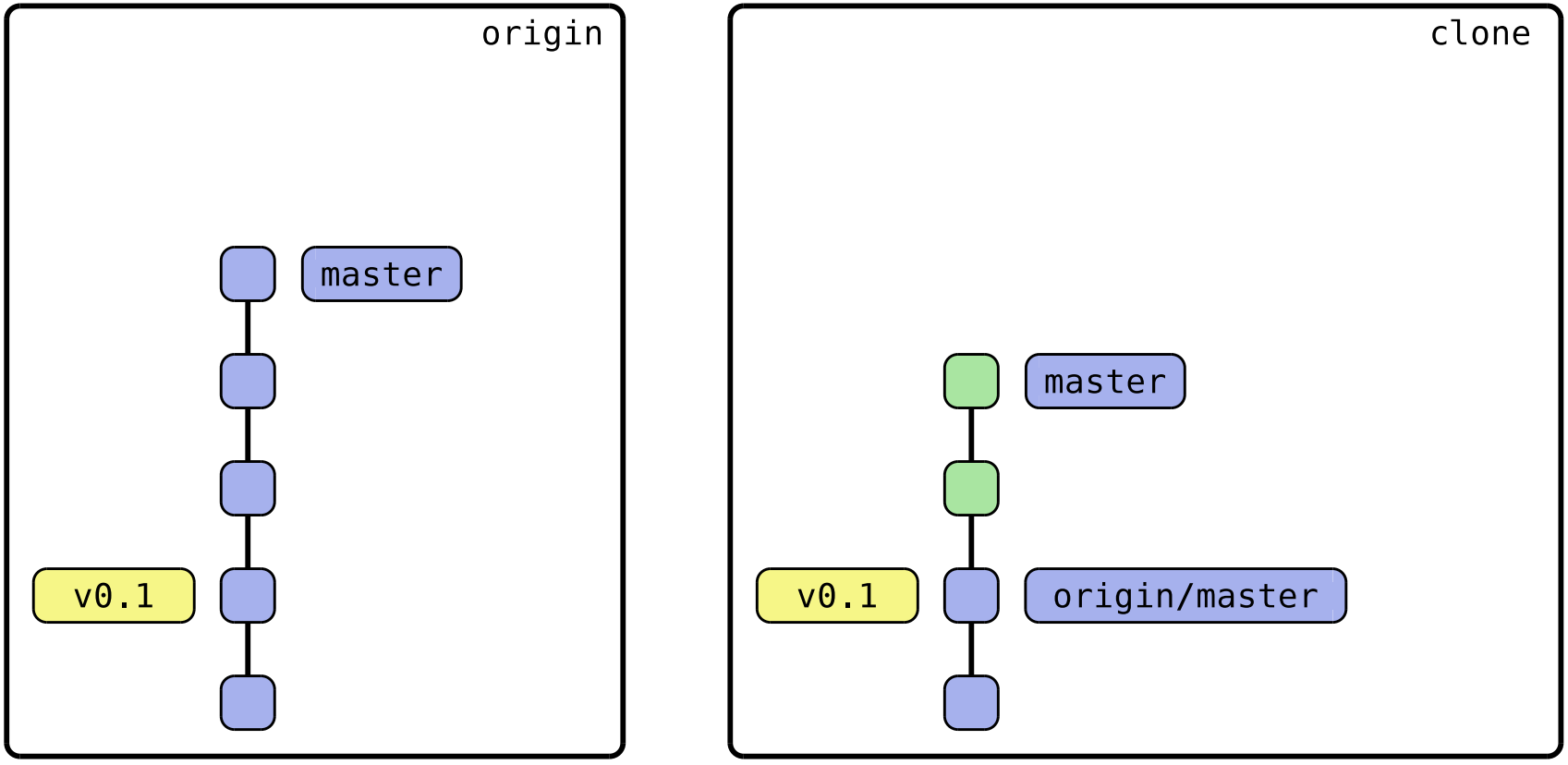
# Vor einem Pull mit Fast-Forward



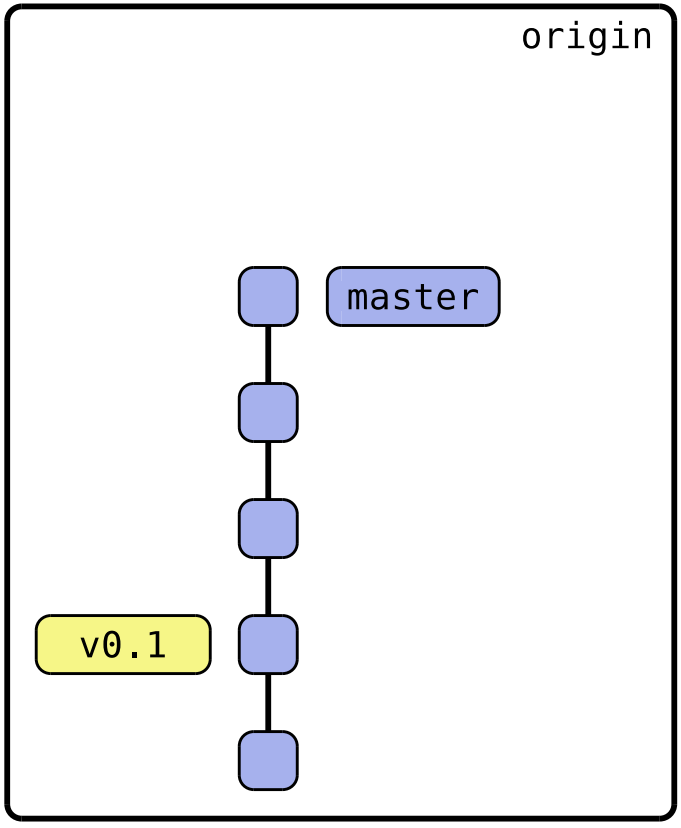
# Nach einem Pull mit Fast-Forward

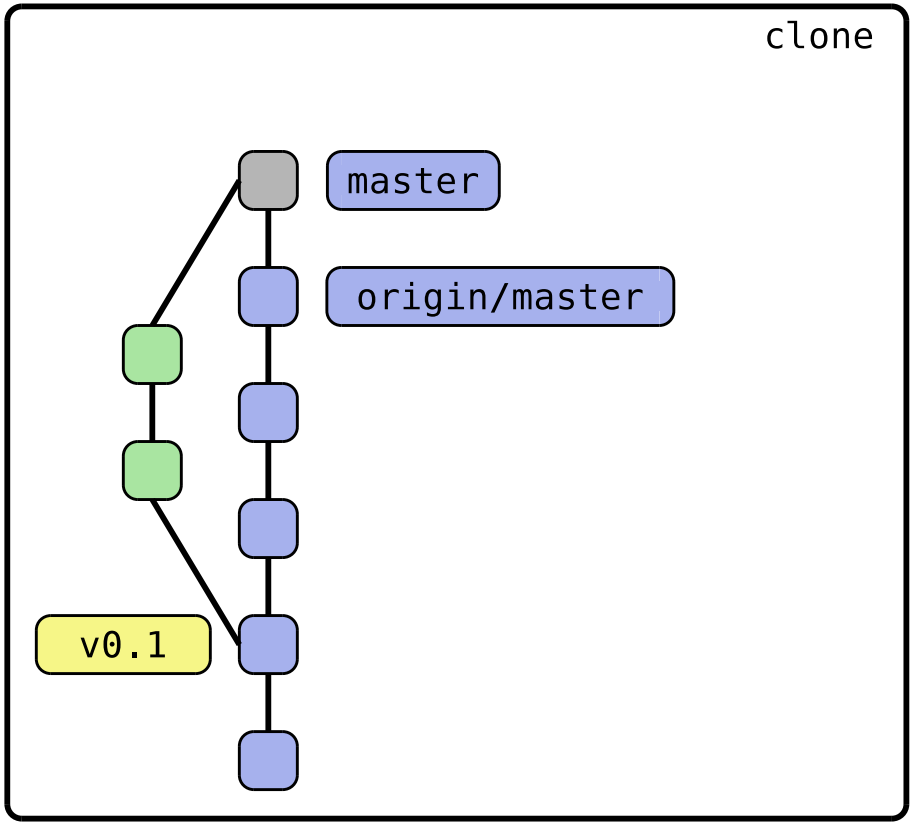


# Vor einem Pull mit Merge

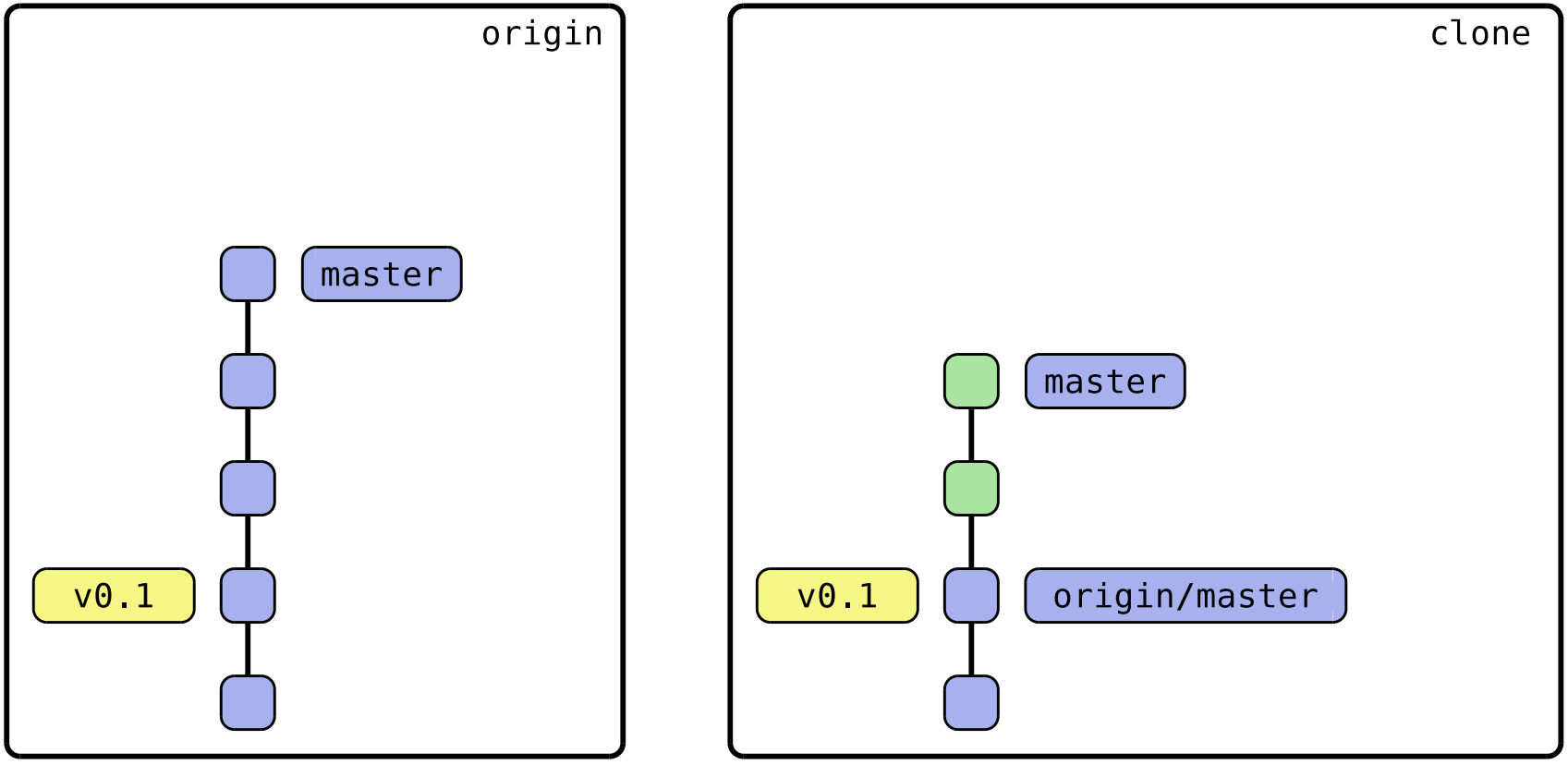


# Nach einem Pull mit Merge

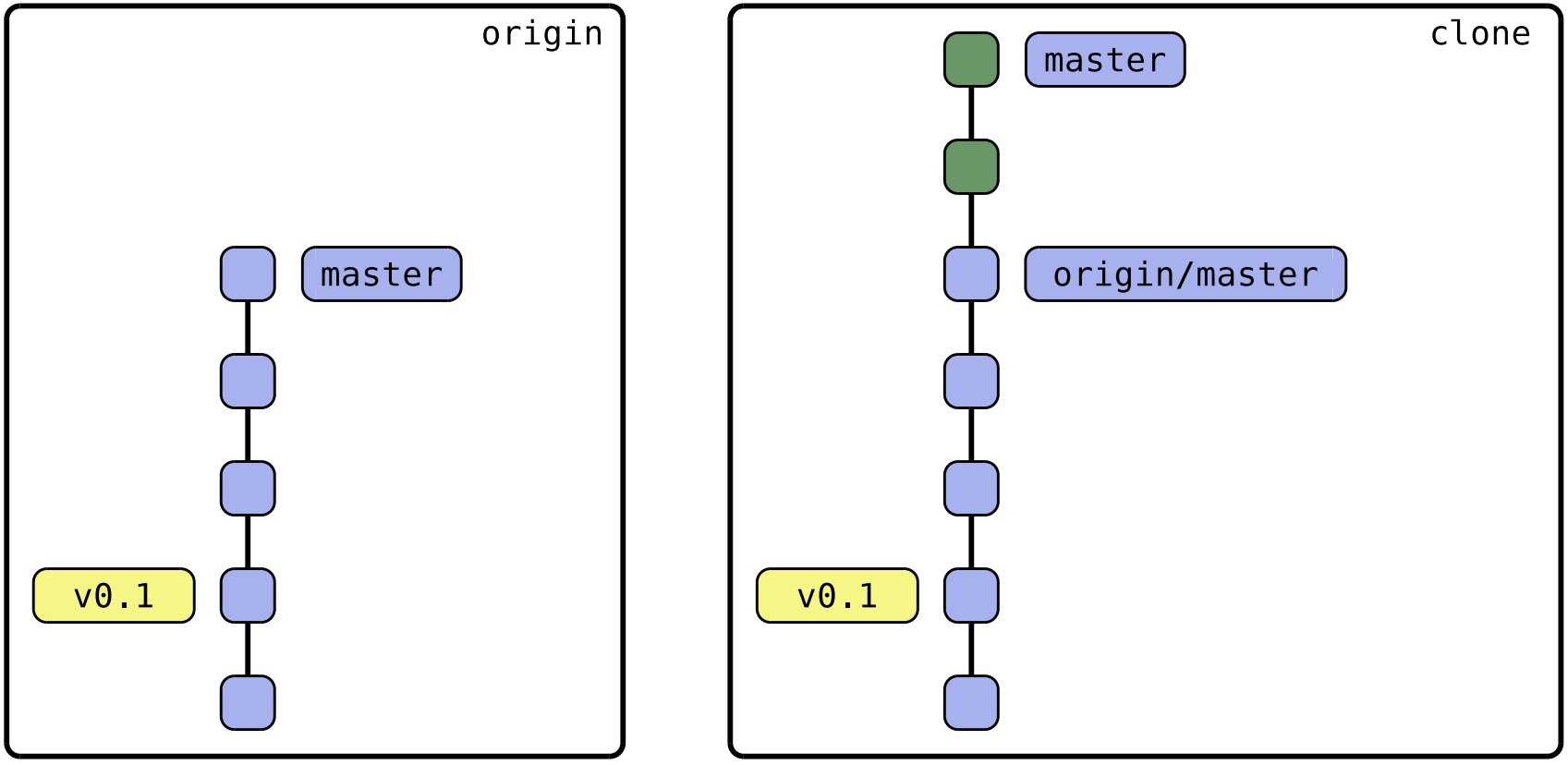


# Vor einem Pull mit Rebase



# Nach einem Pull mit Rebase



# 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
```