Git-Schulung

–Firma–

Valentin Hänel, Julius Plenz

–Datum–

Übersicht

Session 2: Versionsgeschichte, Branching und Merging

- Aliase
- Veränderungen einsehen
- Rückgängig machen
- Branches zusammenführen

Git-Aliase

- Aliase verkürzen die Eingabe von Arbeitsschritten
- Sinnvoll für häufig benötigte Kommandos

Beispiele für sinnvolle Aliase

```
git config --global alias.co checkout
git config --global alias.cm commit -m
git config --global alias.st status
git config --global alias.lol \
    'log --oneline --graph --decorate --all'
git config --global alias.k '!  gitk --all'
```

Aliase in der Shell

- In der Shell lassen sich natürlich auch sinnvolle Aliase einrichten

Aliase für z. B. `bash`

```
alias giu='git remote update'
alias ga='git add'
alias gcm='git commit -m'
alias gis='git status'
alias gil='git log'
```

Übersicht

Versionsgeschichte inspizieren

- `git log` listet alle Commits auf, den neusten zuerst
- Durch Angabe von Argumenten lässt sich die Menge der Commits reduzieren
- `git show` zeigt einen *einzelnen* Commit

Die letzten drei Commits inkl. Zeilen-Bilanz anzeigen

```
git log --stat -3
```

Nur merge commits anzeigen

```
git log --merges
```

Den aktuellen Commit im `master` anzeigen

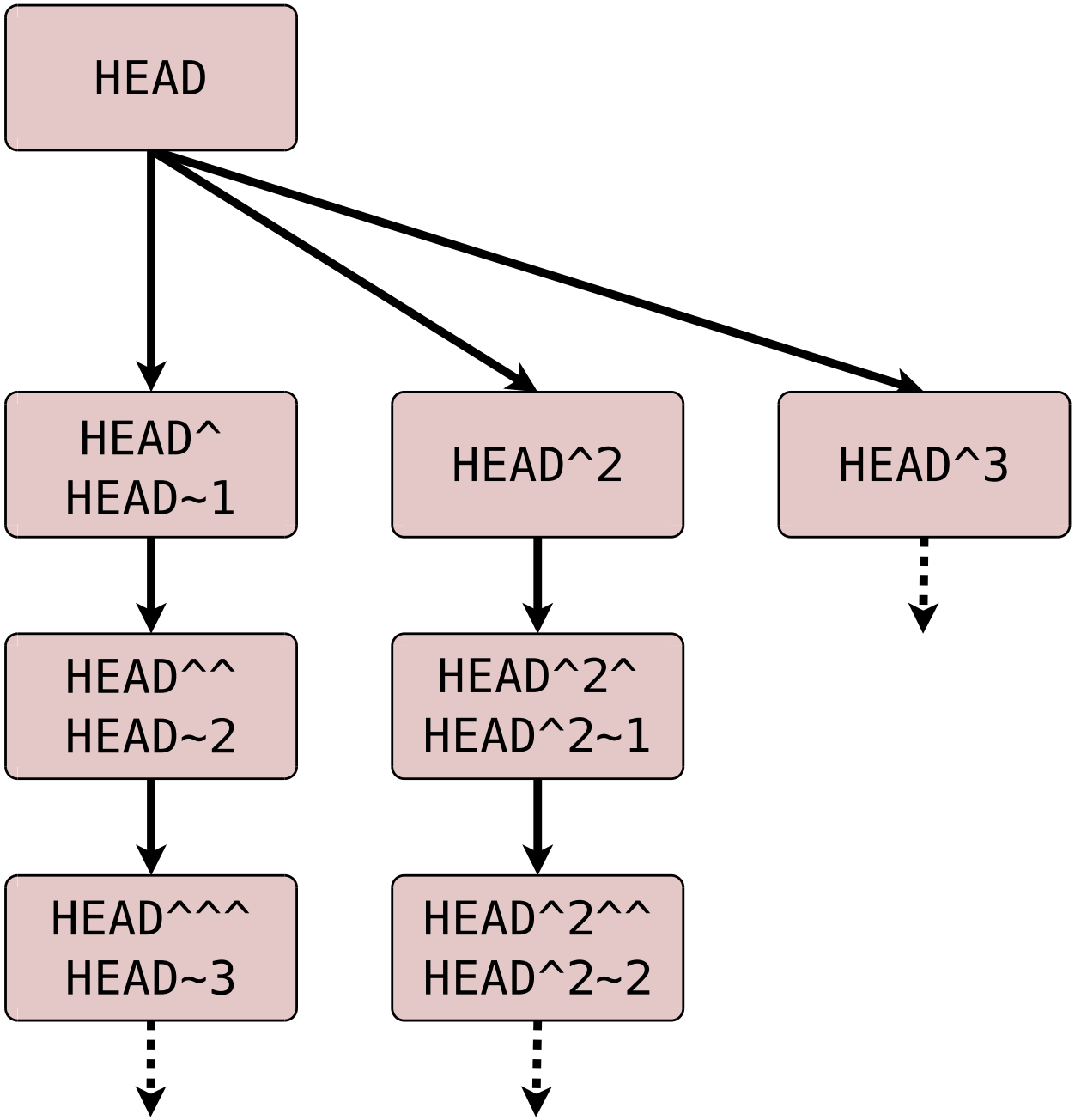
```
git show master
```

Revisionen auflisten

- `HEAD`: aktuell ausgecheckter Commit
  - Voreinstellung für viele Kommandos, die einen Commit als Argument erwarten
- `HEAD^`: Vorgänger von `HEAD`
  - analog: `master^`, `f160742^`, ...
- `HEAD~`*n*: *n*-ter Vorgänger von `HEAD`
  - analog: `master~`*n*, `f160742~`*n*, ...
  - `HEAD^` entspricht also `HEAD~1`

- `git log` traversiert die Geschichte bis zum Root-Commit
- `git show` zeigt nur den Commit an, nicht jedoch seine Vorgänger

Relative Referenzen



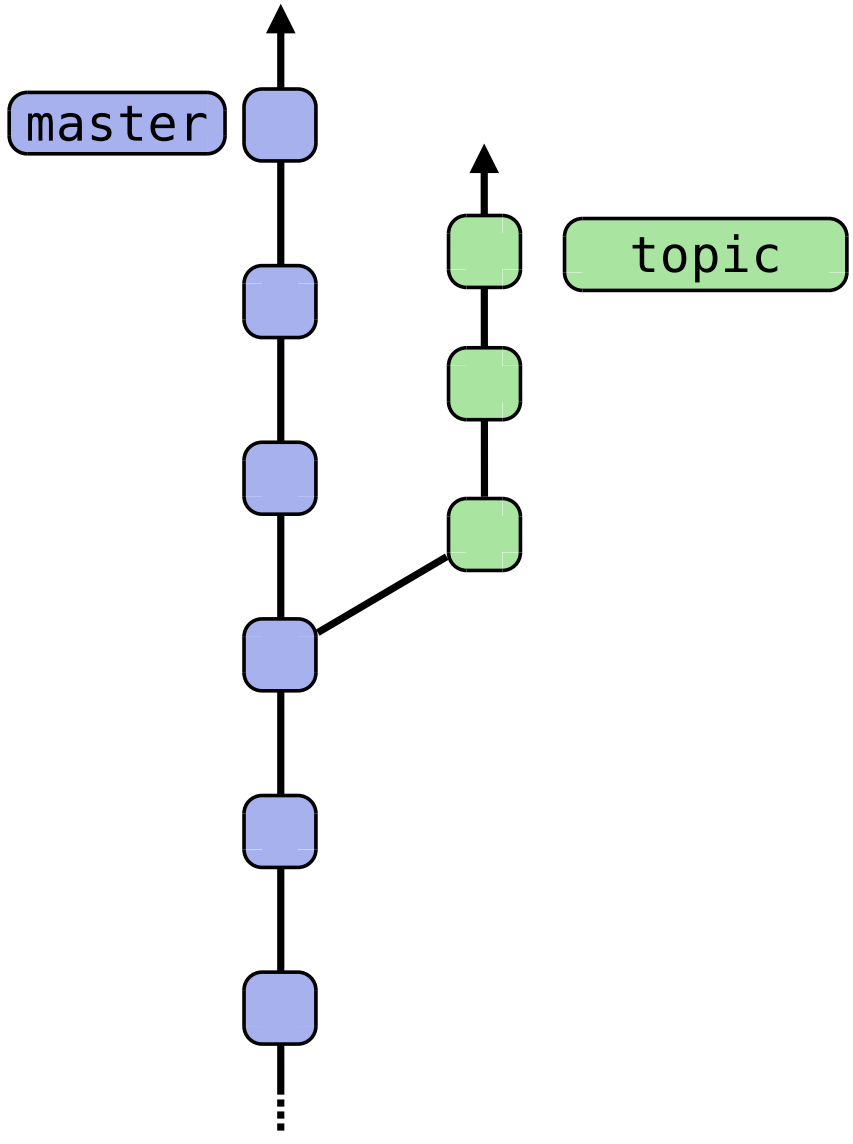
Revisionen auflisten (cont.)

- Negation: `^` oder `--not`
- `git log ^master` *`feature`*
  - Alle Commits, die in *`feature`* sind, aber nicht im `master`
- Kurzform: `git log master..`*`feature`*
  - Welche Commits aus *`feature`* sind noch nicht im `master`?
  - Fehlt einer der beiden Namen, nimmt Git `HEAD` an

In welchen Commits wurde die Datei `stack.c` modifiziert?

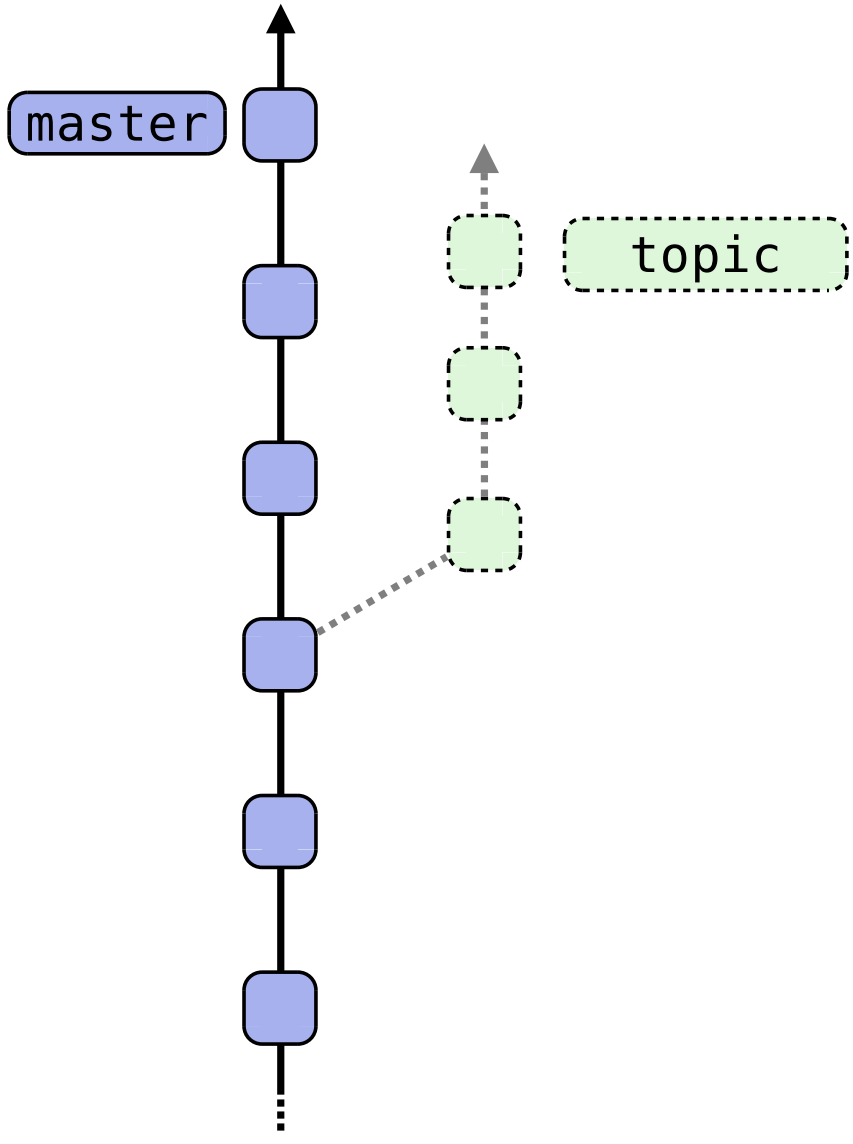
```
git log ..feature -- stack.c
```

»Differenzen« von Branches



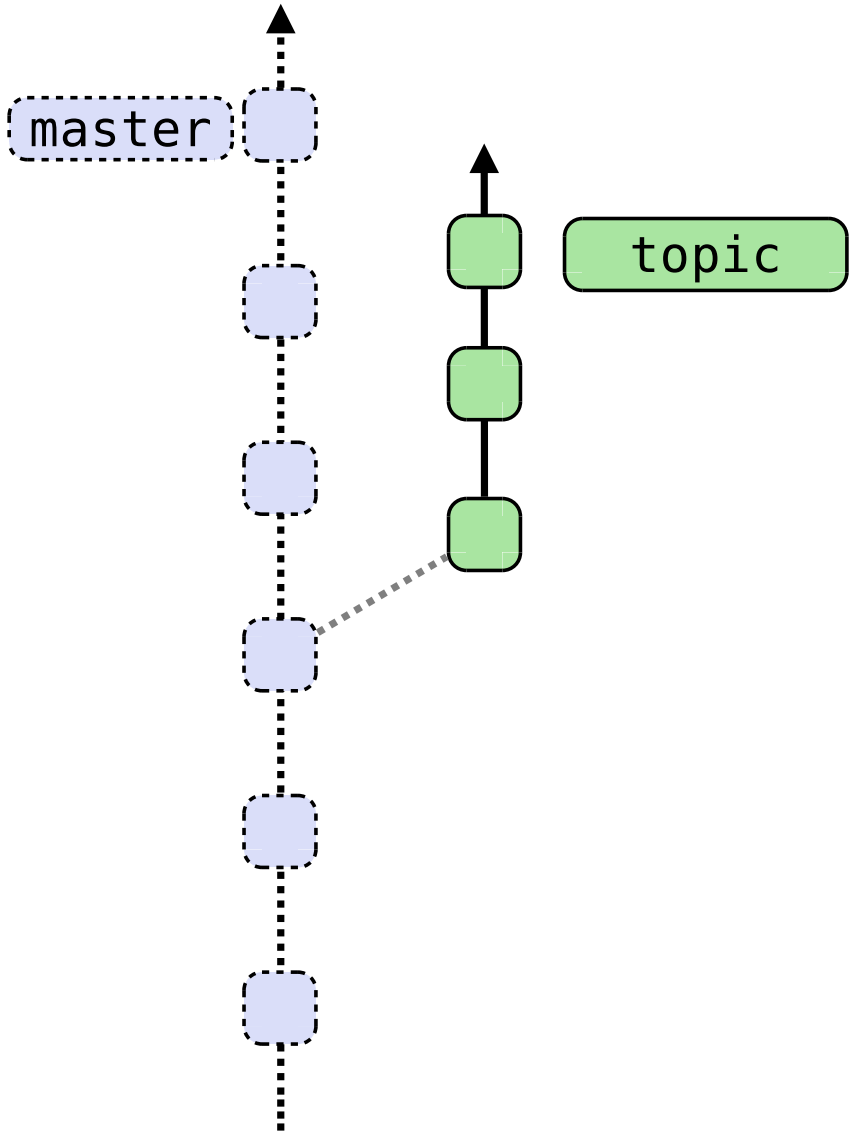

```
git log --all
```

»Differenzen« von Branches (2)



`git log master`

»Differenzen« von Branches (3)




```
git log master..topic
```

Den Wald vor lauter Bäumen...

Baumstruktur als ASCII-Diagramm anzeigen

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

- Statt `--all` kann auch eine Liste von Commits angegeben werden
- `--date-order` ordnet die Commits strikt nach Datum
- Ausgabe ist äquivalent zu der von `gitk`

Übersicht

Checkout

Version 1.0 auschecken

```
git checkout v1.0
```

Wie sah der `master` vor einer Woche aus?

```
git checkout 'master@{one week ago}'
```

- »Detached `HEAD`«: `HEAD` zeigt direkt auf einen Commit und nicht auf einen Branch
  - Änderungen können verloren gehen, wenn der `HEAD` wieder verändert wird

Checkout (cont.)

Datei aus dem Repository wiederherstellen

```
git checkout -- datei
```

`master` auschecken

```
git checkout -f master
```

- Beide Kommandos sind möglicherweise destruktiv
- Getätigte Änderungen werden überschrieben!

Neuen Branch erstellen

Branch von `master` erstellen und direkt auschecken

```
git checkout -b neues-feature master
```

Übung: Checkout

1. Verwenden Sie `git checkout`, um den Working-Tree in einen Zustand von heute Morgen zu versetzten
2. Beobachten Sie hierbei die Ausgabe von Git bzgl. des »Detached `HEAD`«
3. Schauen Sie sich nun die Ausgabe von `git branch` an
4. Verändern Sie eine Datei im Working-Tree. Stellen Sie danach mit `git checkout` den alten Zustand wieder her
5. Verändern Sie mehrere Dateien im Working-Tree, und stellen Sie danach mit einem einzigen Aufruf von `git checkout` den Zustand aller Dateien wieder her

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!"
```

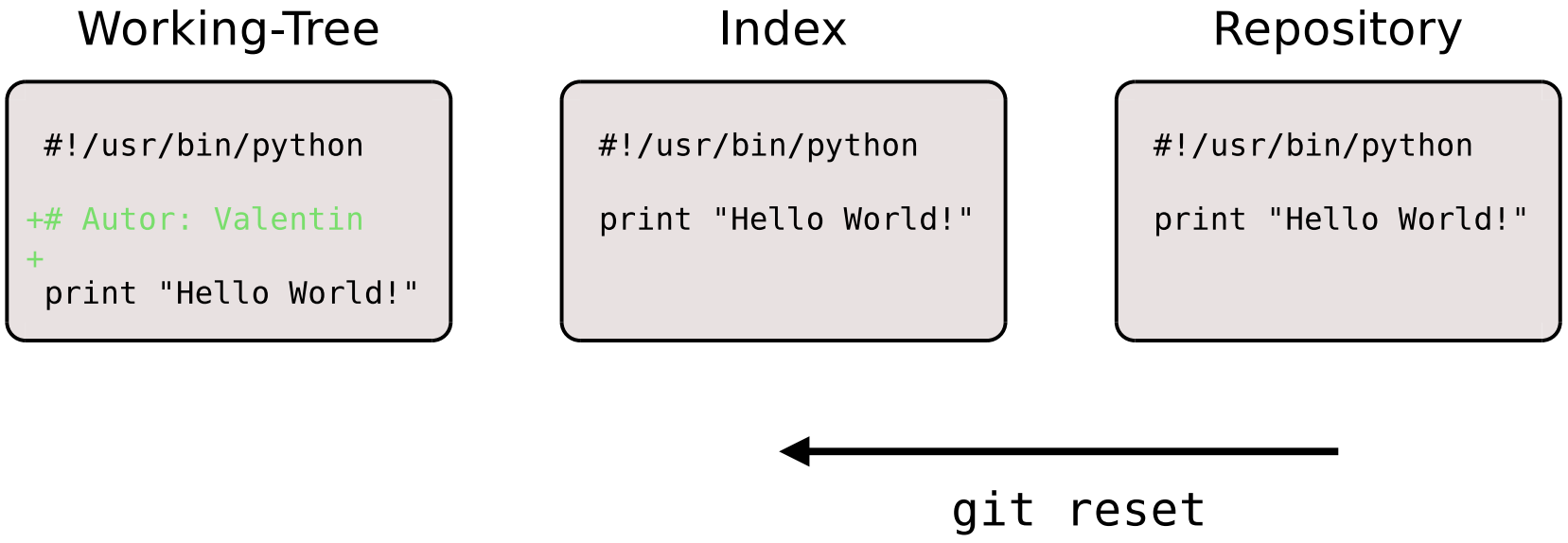
Repository

```
#!/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!"
```

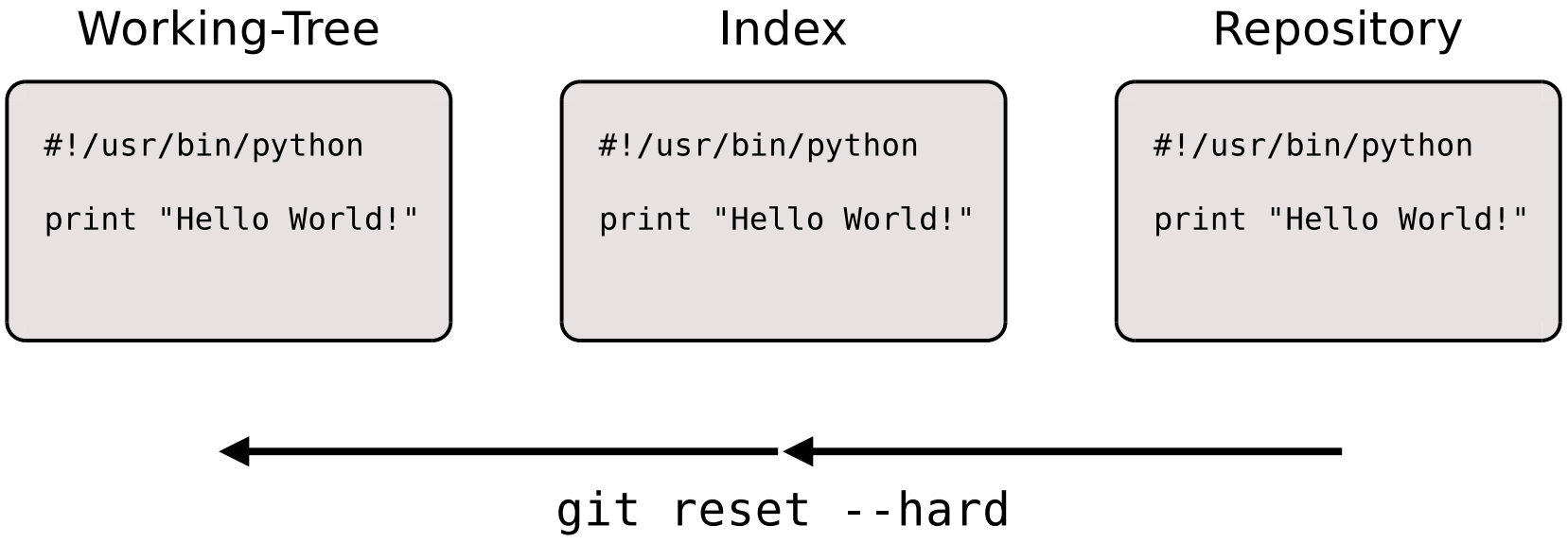
Repository

```
#!/usr/bin/python

print "Hello World!"
```

Index zurücksetzten – `git reset --hard`

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



Einen Commit rückgängig machen: `revert`

- `revert` macht einen Commit rückgängig
  - Das Gegenteil des Commits wird als neuer Commit erstellt
  - Ähnlich wie `patch -r`

Gegenteil des jünsten Commits

```
git revert HEAD
```

Übung: Reset und Revert

1. Machen Sie einen Commit. Setzen Sie den aktuellen `HEAD` mit `git reset HEAD^` zurück. Was passiert mit den Veränderungen?
2. Wiederholen Sie den Commit. Setzen Sie diesmal den `HEAD` mit `git reset --soft HEAD^` zurück. Was ist diesmal anders?
3. Wiederholen Sie den Commit ein drittes mal. Wiederholen Sie auch das Zurücksetzen, aber nun mit der Option `--hard`
4. Wiederholen Sie den Commit ein viertes mal. Machen Sie diesmal jedoch den Commit mit `git revert` »rückgängig«
5. Verändern Sie mehrere Zeilen am Anfang und Ende einer Datei. Fügen Sie die Datei dem Index hinzu, und entfernen Sie einzelne Zeilen aus dem Index (`reset -p`)

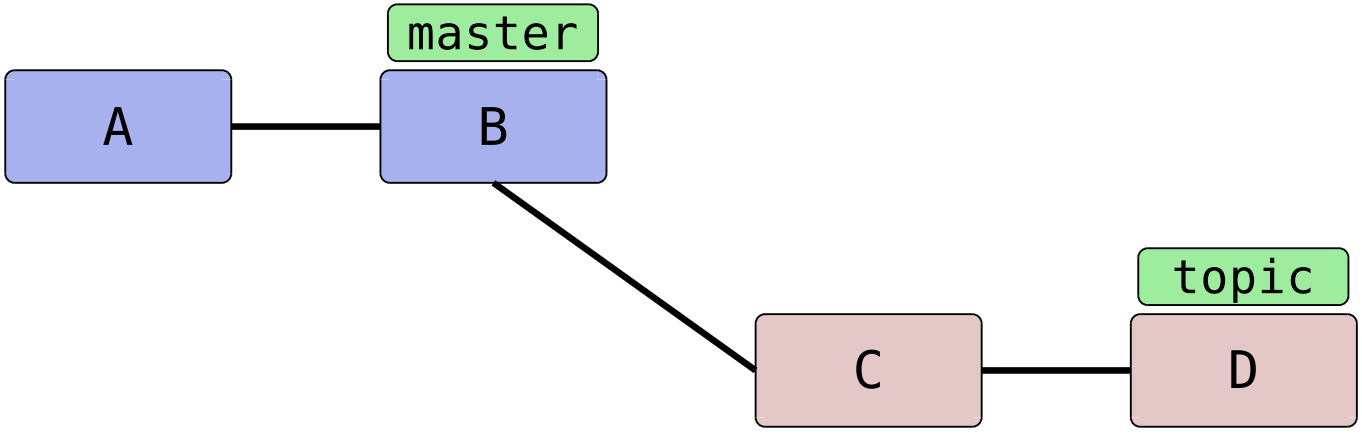
Übersicht

Merge

`git merge` fügt zwei oder mehr Branches zusammen

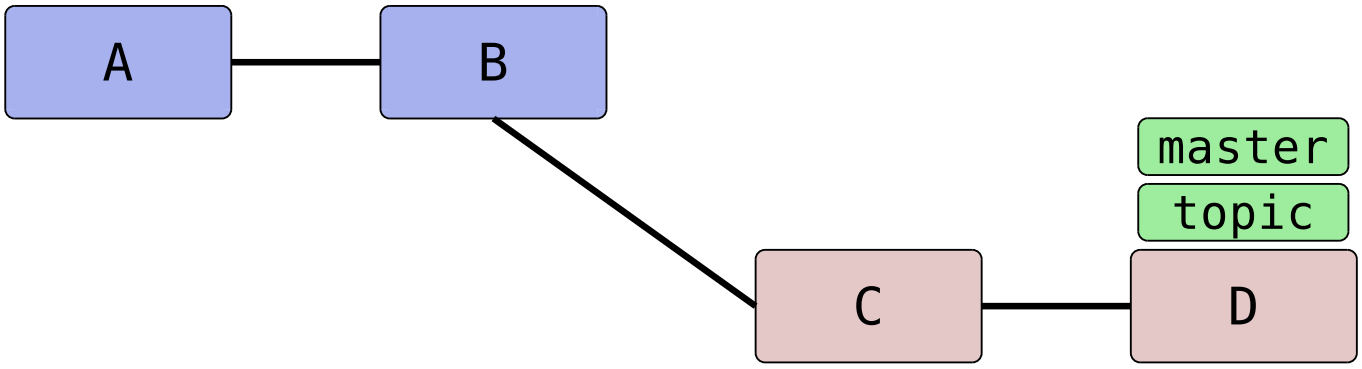
- *Fast-Forward*: Es existieren keine Abzweigungen, und der Branch wird einfach »weitergerückt«
- Sonst wird ein *Merge-Commit* erstellt, der beide Branches als »Parents« referenziert
  - Treten Konflikte auf, werden diese in dem Merge-Commit behoben
  - Andernfalls ist der Merge-Commit »leer«
- Die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

Vor dem Fast-Forward



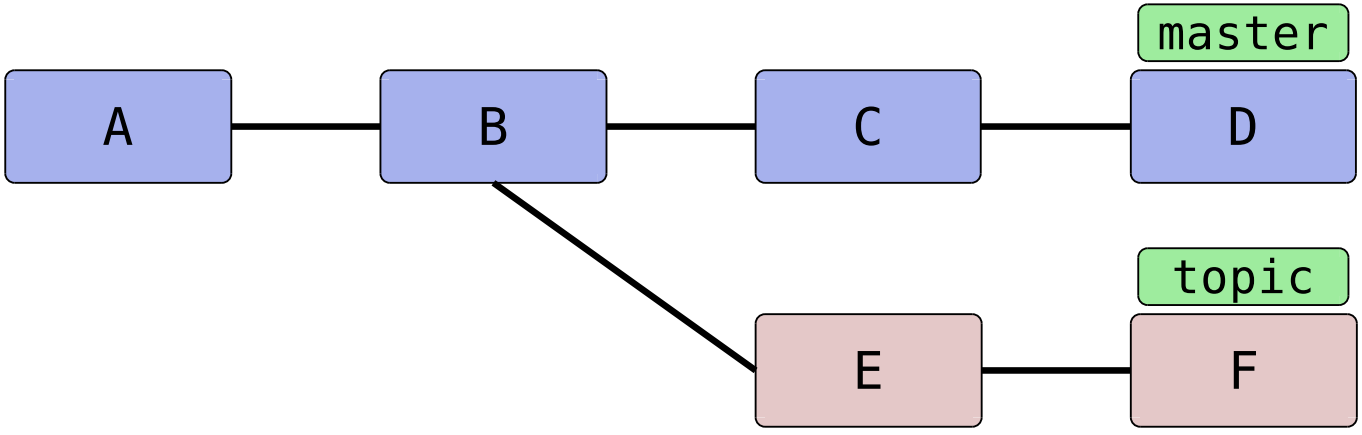
- In `master` hat sich nichts getan, `topic` ist fertig

Nach dem Fast-Forward



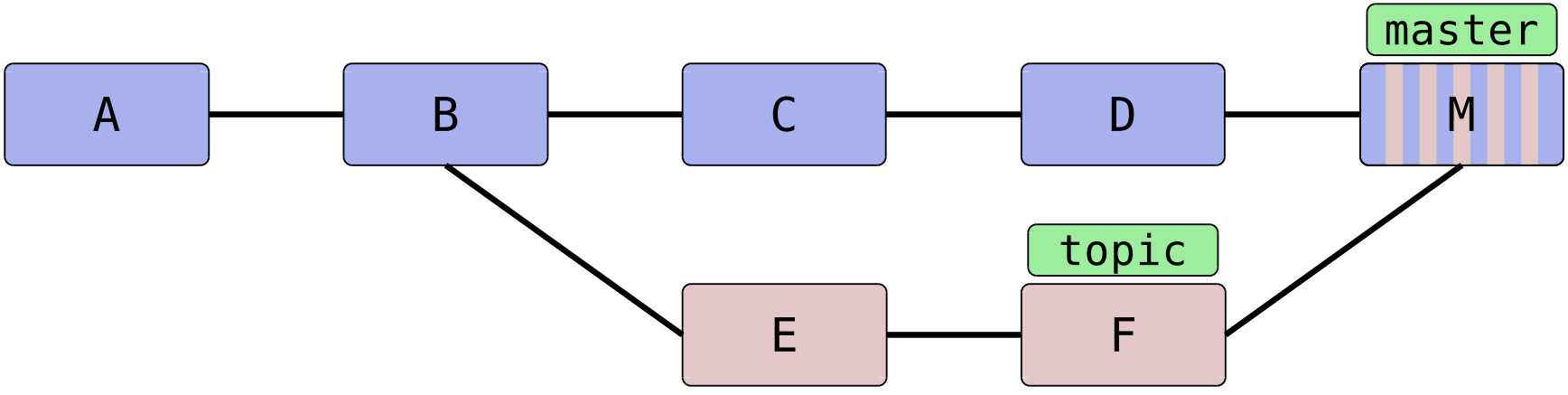
- `master` wird »weitergerückt«, bzw. »vorgespult«

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
```

Übung: Branches zusammenführen

1. Führen Sie zwei Branches zusammen
2. Integrieren Sie *mehrere* Branches gleichzeitig in den `master` (*octopus*)

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
```

Übung: Konflikte lösen

1. Erstellen Sie auf zwei verschiedenen Branches konfliktierende Änderungen, um den Konflikt dann nach einem `merge` zu lösen
2. Definieren Sie ein Mergetool und lösen Sie damit einen Konflikt
3. Lösen Sie einen Merge-Konflikt mit der Strategie-Option *ours* oder *theirs*

Throw-Away-Integration

- Idee: Einen Merge vorher *testen*
  - Funktioniert das Feature? Gibt es Konflikte?
- `git branch throw-away-feature1`
- `git merge feature1`
- Funktioniert alles?
  - Nein: Fehler suchen, verantwortlichen Entwickler kontaktieren
  - Ja: `git branch -D throw-away-feature1` und den gleichen Merge in den Entwicklungs-Branch machen

rerere: reuse recorded resolution

- Es tritt ein Merge-Konflikt auf
  - Trat genau dieser Konflikt schon einmal auf, so wird die damals gefundene Lösung als »Voreinstellung« bereitgestellt
  - Sonst muss der Konflikt manuell behoben werden, die Lösung wird automatisch abgespeichert

- In Kombination mit Throw-Away-Integration sehr praktisch für Topic-Branches, die lange Zeit entwickelt werden

rerere-Funktionalität global aktivieren

```
git config --global rerere.enabled true
```

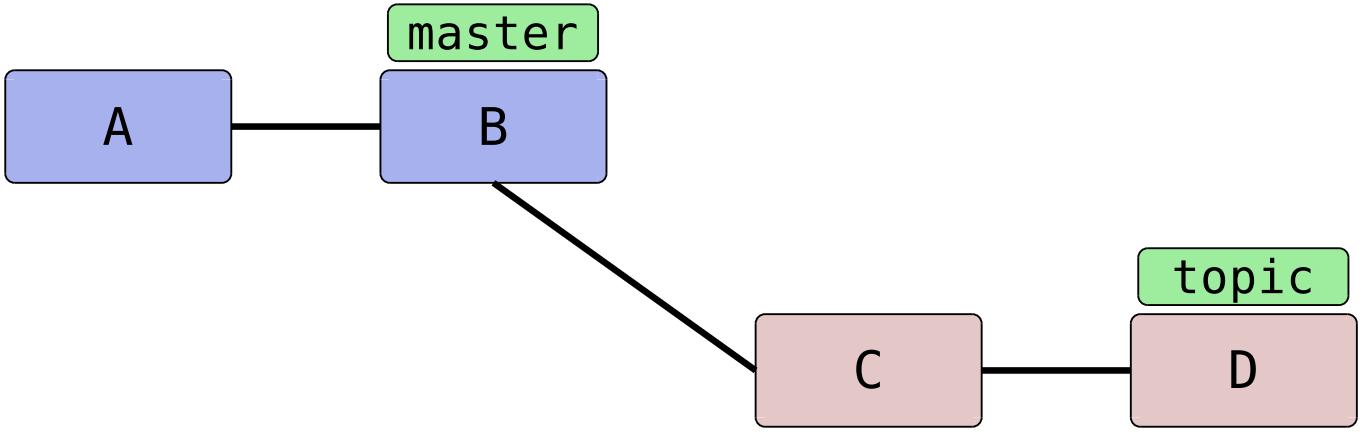
Merge-Commit Forcieren

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

Das ist sehr sinnvoll:

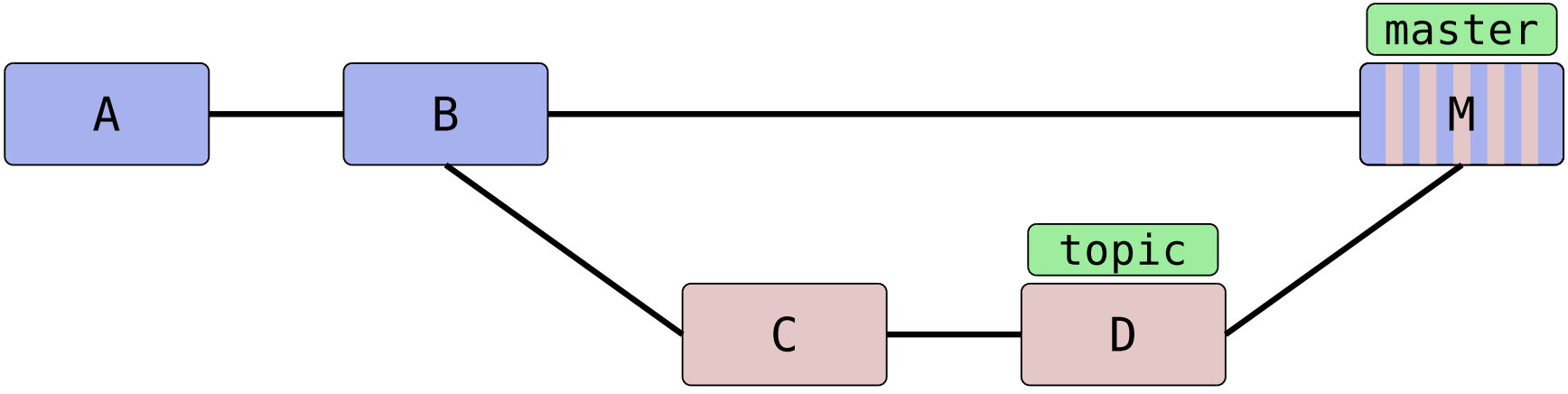
- Die Integration eines Feature-Branches deutlich machen
- In der Ansicht von `git log` wird die Geschichte immer linear dargestellt
- In der Baumansicht sieht man die Abzweigung und Zusammenführung

Vor dem Force-Merge



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

Nach dem Force-Merge



- Ein Merge-Commit wurde forceirt

Cherry-Pick

- «Die Rosinen finden»: Einzelne Commits aus anderen Branches übernehmen
  - Nicht der ganze Branch soll integriert werden (`merge`), sondern vorerst nur Teile, z. B. Bugfixes
- `rebase` erkennt solche Commits und überspringt sie
  - Wird ein Commit aus `feature` in `master` übernommen (Cherry-Pick), dann ignoriert `git rebase master feature` diesen Commit – er erscheint danach nicht mehr in `feature`

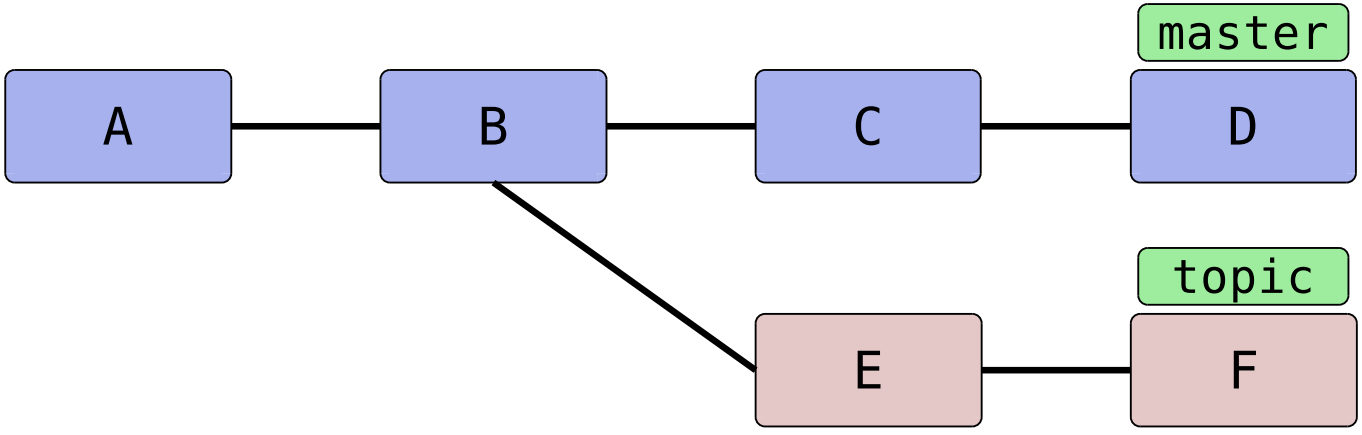
Commits aus *upstream* anzeigen

```
git cherry -v --abbrev upstream
```

Commit übernehmen

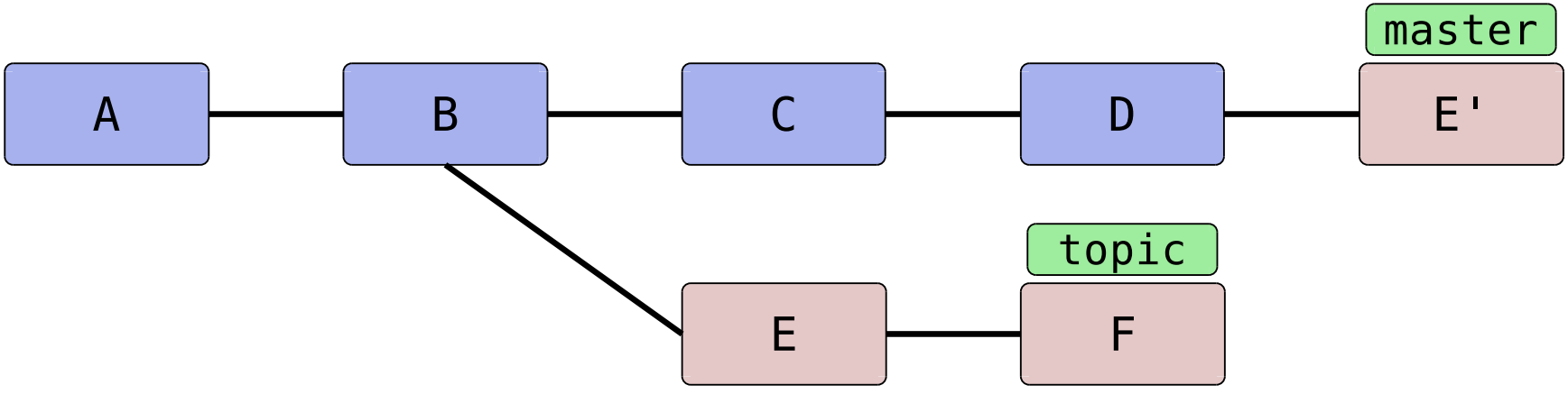
```
git cherry-pick c1de5ac
```

Ausgangslage



- Commit E soll in `master` übernommen werden

Nach dem Cherry-Pick



- `git cherry-pick` *E*

Übung: Rerere und Cherry-Pick

1. Erstellen Sie einen Throw-Away-Integration-Branch und probieren Sie, mehrere Branches gleichzeitig zu integrieren
2. Schalten Sie `rerere.enabled` auf `true`. Lösen Sie dann einen Konflikt, um den entstandenen Merge-Commit anschließend mit `reset --hard` wieder zu löschen. Führen Sie dann den Merge noch einmal aus. (Rerere sollte Ihre vorige Konfliktlösung gespeichert haben.)
3. Übernehmen Sie einzelne Commits aus einem Branch nach `master` (`cherry-pick`)

Branch-Modell

- Dieses Modell wird in vielen OSS-Projekten verwendet
- Es gibt vier Haupt-Branches:
  - `maint`: Maintenance (Security und Bugfixes)
  - `master`: Vorbereitung des neuen Releases
  - `next`: Neue Features werden auf Stabilität getestet (Beta)
  - `pu`: *proposed updates*, unfertige, experimentelle Features
- `next` und `pu` werden möglicherweise verändert (neu aufgebaut, etc.)
- Bugfixes werden von `maint` in `master` übernommen
- Topic-Branches werden bei Vollendung in `next` übernommen
- Wird `next` stabil, wandern die Änderungen in den `master`

Branch-Modell – Visualisiert