Git-Workshop

Fachschaft Informatik, HS Karlsuhe

Felix Bürkle

Basierend auf Material von Julius Plenz

`https://github.com/Feh/git-workshop`

Bevor wir beginnen . . .

- Wer verwendet Linux? – Windows? – Mac?
- Wer arbeitet gelegentlich auf der Shell?
- Wer hat momentan noch *kein* Git installiert?
- Wer kennt oder hat schon mal eines der folgenden Systeme benutzt?
  - CVS/RCS
  - SVN
  - Mercurial, Darcs, Perforce, Bazaar

Wer kennt Git?

Wer hat schonmal ...

- `git` eingegeben
- Ein Git-Repository selbst erstellt?
- ... oder geklont?
- Einen Commit gemacht?
- Per Git mit anderen Leuten zusammengearbeitet?

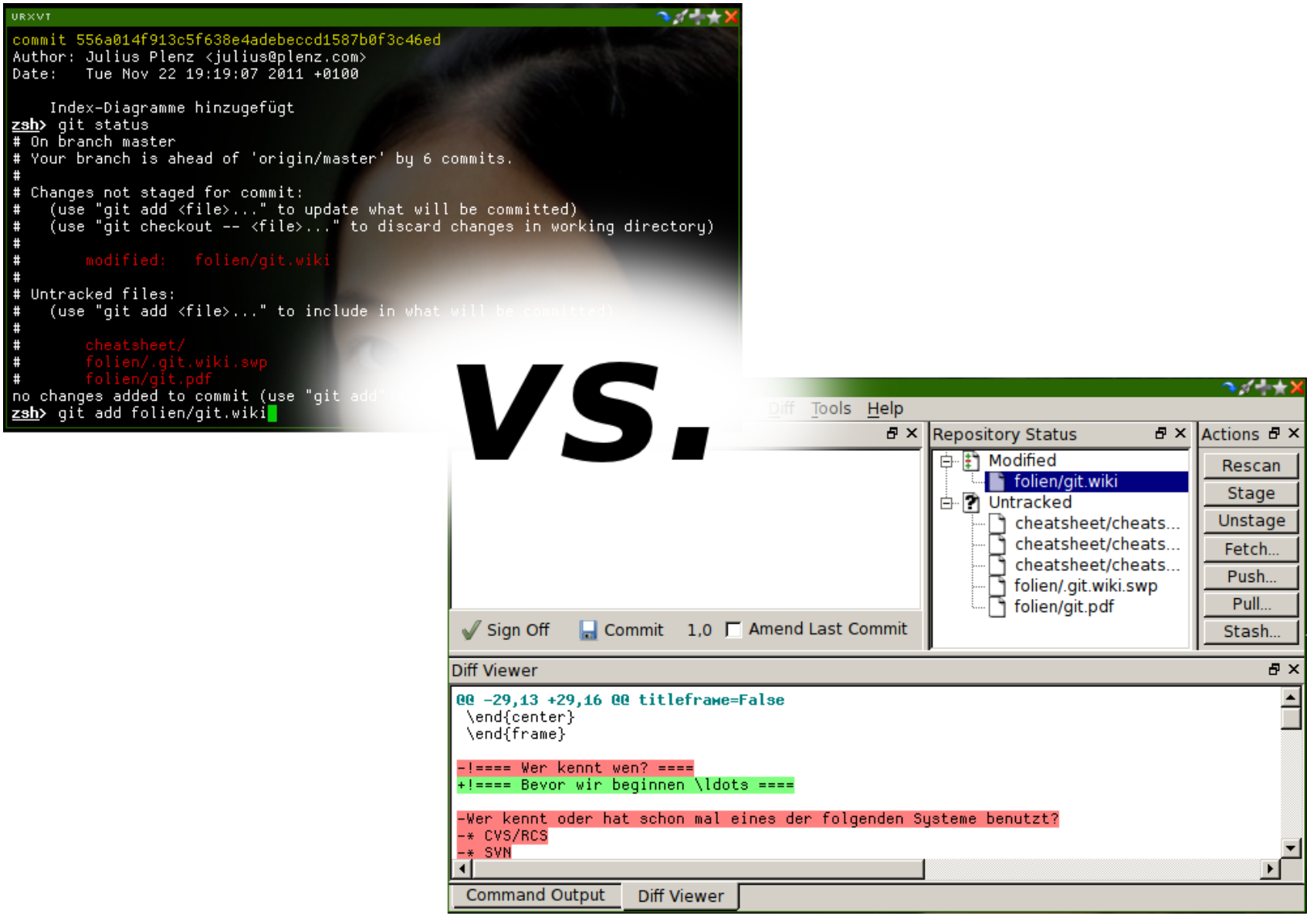
Der Plan

- Grundlegende Arbeitsschritte in Git
- Das Objektmodell – eine theoretische Grundlage
- Parallele Entwicklung: Branches und Merges
- Entwicklung koordinieren: Ein Branching-Modell
- Die Geschichte umschreiben: Rebase
- Verteiltes Git: Commits hoch- und runterladen
- Verschiedene Workflows

Motivation: Warum Versionskontrolle?

- **Sicherheit:** Versionskontrolle schützt vor Verlusten
- **Dokumentation:** Wer hat wann was gemacht?
- **Fokussierung:** Entwicklung logisch gliedern
- **Kollaboration:** Mit anderen Leuten an den gleichen Dateien arbeiten
- **Partizipation:** Jeder kann mitmachen (GitHub etc.)

Interface


URXVT
commit 556a014f913c5f638e4adebeccd1587b0f3c46ed
Author: Julius Plenz <julius@plenz.com>
Date: Tue Nov 22 19:19:07 2011 +0100
Index-Diagramme hinzugefügt
zsh> git status
On branch master
Your branch is ahead of 'origin/master' by 6 commits.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: folien/git.wiki
Untracked files:
(use "git add <file>..." to include in what will be committed)
cheatsheet/
folien/.git.wiki.swp
folien/git.pdf
no changes added to commit (use "git add
zsh> git add folien/git.wiki
vs.
Diff Tools Help
Repository Status
Modified
folien/git.wiki
Untracked
cheatsheet/cheats...
cheatsheet/cheats...
cheatsheet/cheats...
folien/.git.wiki.swp
folien/git.pdf
Actions
Rescan
Stage
Unstage
Fetch...
Push...
Pull...
Stash...
Sign Off
Commit
1,0
Amend Last Commit
Diff Viewer
@@ -29,13 +29,16 @@ titleframe=False
\end{center}
\end{frame}
-!==== Wer kennt wen? ====
+!==== Bevor wir beginnen \ldots ====
-Wer kennt oder hat schon mal eines der folgenden Systeme benutzt?
-* CVS/RCS
-* SVN
Command Output
Diff Viewer

Wer bin ich? – Name und E-Mail einstellen

- Für alle Projekte (wird in `~/.gitconfig` gespeichert)
  - `git config --global user.name "Max Mustermann"`
  - `git config --global user.email max@mustermann.de`
- ... oder alternativ nur für das aktuelle Projekt:
  - `git config user.email maintainer@cool-project.org`
- Außerdem, für die, die wollen: Farbe!
  - `git config --global color.ui auto`

Ein Projekt importieren oder erstellen

- Ein neues Projekt erstellt man wie folgt:
  - `mkdir` *`projekt`*
  - `cd` *`projekt`*
  - `git init`
- Um ein bestehendes Projekt zu importieren, »klont« man es mit seiner gesamten Versionsgeschichte:
  - `git clone git@github.com:fsi-hska/git-workshop.git`

Begriffsbildung

- **Index/Staging Area**: Bereich zwischen dem Arbeitsverzeichnis und dem Repository, in die Änderungen für den nächsten **Commit** gesammelt werden
- **Commit**: Eine Änderung an einer oder mehrerer Dateien, versehen mit Metadaten wie Autor, Datum und Beschreibung
- **Repository**: Eine Datenbank für Commits, dort wird die Versionsgeschichte aufgezeichnet
- **Referenz**: Jeder **Commit** wird durch eine eindeutige SHA1-Summe identifiziert. Eine Referenz »zeigt« auf einen bestimmten Commit
- **Branch**: Ein »Zweig«, eine Abzweigung im Entwicklungszyklus, z. B. um ein neues Feature einzuführen.

Ein typischer Arbeitsablauf

- Eine *datei* verändern, und die Änderungen in das Repository »einchecken«:

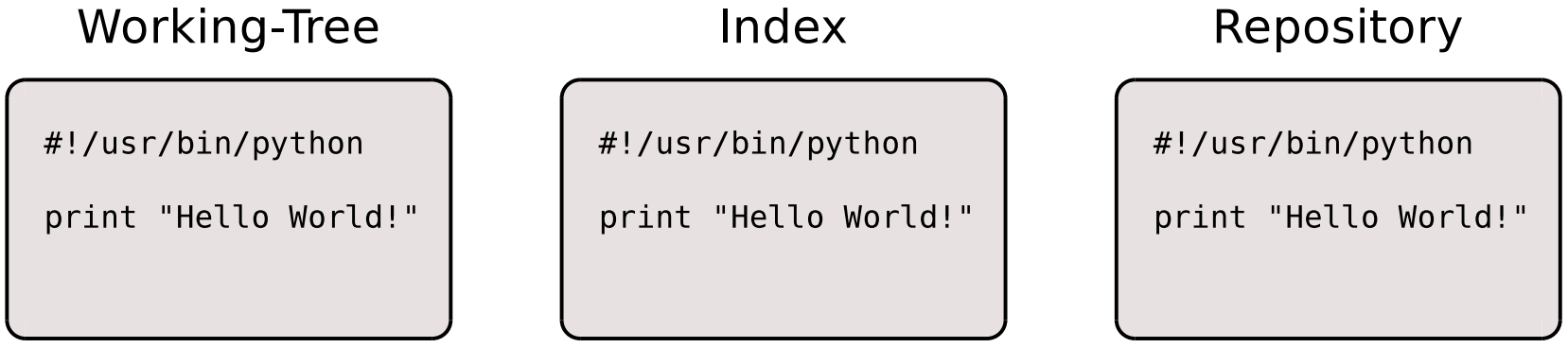
1. `$EDITOR` *datei*
2. `git status`
3. `git add` *datei*
4. `git commit -m 'datei angepasst'`
5. `git show`

Index / Staging Area

- Im *Index* bzw. der *Staging-Area* werden Veränderungen für den nächsten Commit vorgemerkt
- So kann der Inhalt von einem Commit schrittweise aus einzelnen Veränderungen zusammengestellt werden

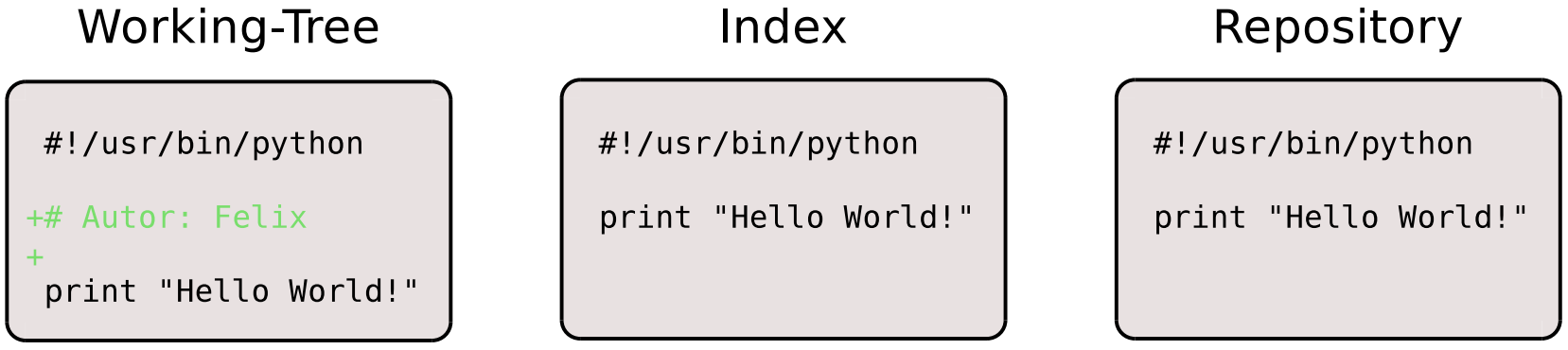
Ausgangsstellung

- Alle auf dem gleichen Stand



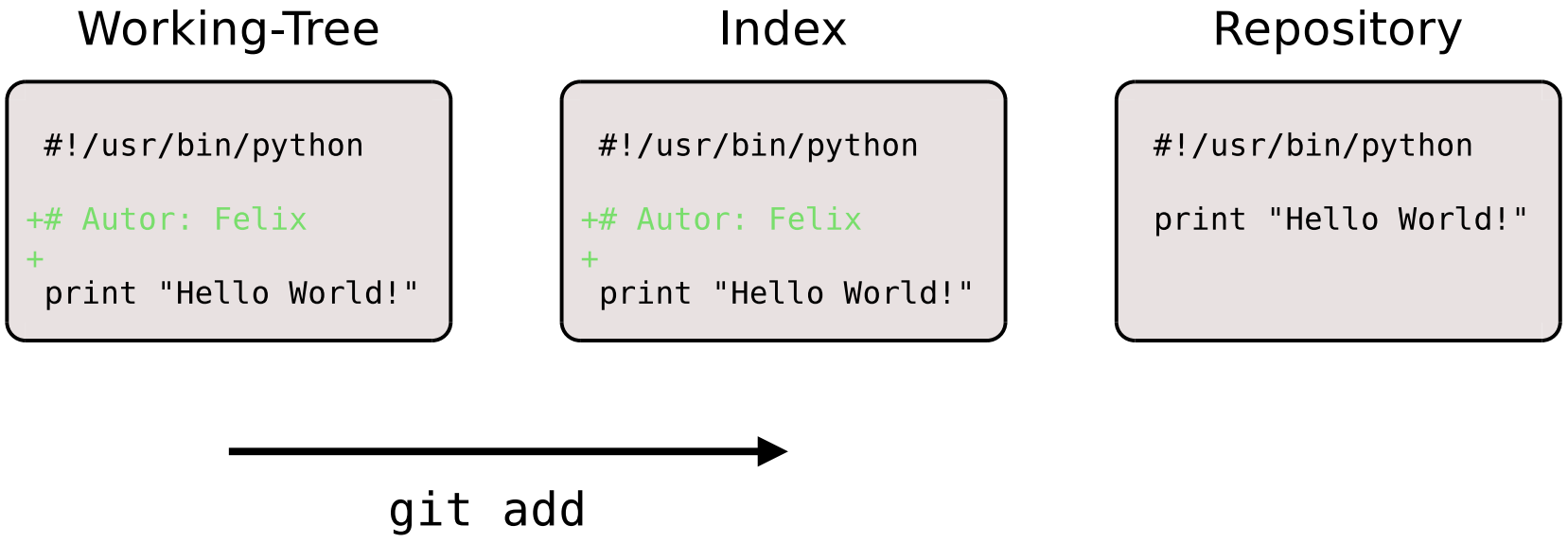
Veränderungen machen

- Veränderungen werden im Working-Tree gemacht



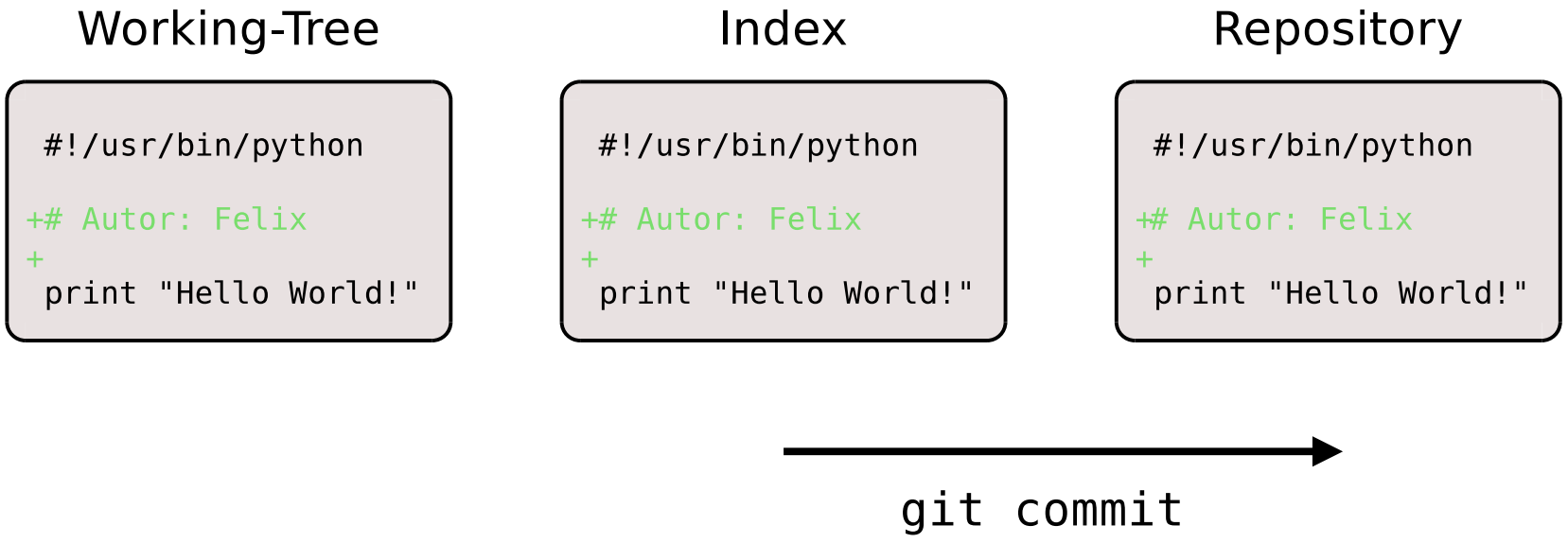
Dem Index hinzufügen – `git add`

- 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: Felix

print "Hello World!"
```

Index

```
#!/usr/bin/python

# Autor: Felix

print "Hello World!"
```

Repository

```
#!/usr/bin/python

# Autor: Felix

print "Hello World!"
```

Referenzen und ignorierte Dateien

Relative Referenzen:

- `HEAD`: Der letzte Commit (wird per `git show` angezeigt)
- `HEAD^`: Der vorletzte Commit
- `HEAD~`*N*: Der *N*.-letzte Commit

Informationen über das Repository erhalten

- Den jüngsten Commit im vollen Umfang anschauen:
  - `git show`
- Die gesamte Versionsgeschichte, die zum aktuellen Zustand führt, anzeigen:
  - `git log`
- Was hat sich verändert?
  - `git diff`
- Das Repository visualisieren:
  - `gitk`
  - `gitg`
- ... oder textbasiert:
  - `tig`

Änderungen rückgängig machen

Einen neuen Commit erstellen, der eine alte Änderung rückgängig macht:

- `git revert` *`commit`*

Den Index zurücksetzen:

- `git reset HEAD`

Den Zustand von vor zwei Commits wiederherstellen:

- `git checkout HEAD~2`

Die letzten zwei Commits **unwiederbringlich** löschen:

- `git reset --hard HEAD~2`

Branches: Abzweigungen

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

Was genau sind Branches? – Nichts anderes als Referenzen auf den jeweils obersten Commit einer Versionsgeschichte.

Branches ...

- erstellen: `git branch` *name*
- auschecken: `git checkout` *name*
- erstellen und direkt auschecken: `git checkout -b` *name*
- auflisten: `git branch -v`
- löschen: `git branch -d` *name*

Idealisierter Workflow: Ein Branch pro neuem Feature oder Bugfix.

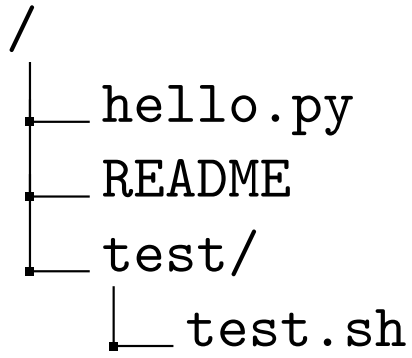
Beispiel: Zwei Branches

Zwei Branches erstellen, und auf jedem einen Commit machen.
Dann das Resultat in `gitk` anschauen.

- `git branch eins`
- `git checkout eins`
- Commit machen
- `git checkout master`
- `git checkout -b zwei`
- Commit machen
- `gitk --all`

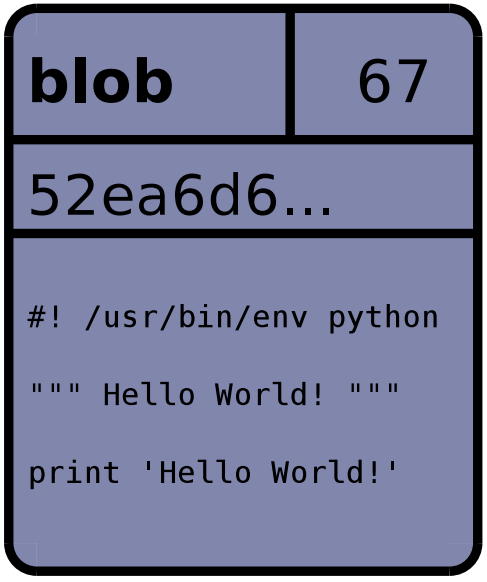
Beispielprojekt: Was wollen wir speichern

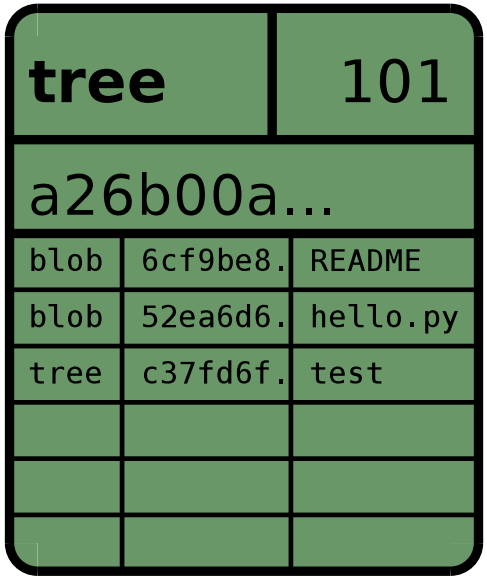
Angenommen, wir wollen folgendes Verzeichnis speichern:



Objektmodell

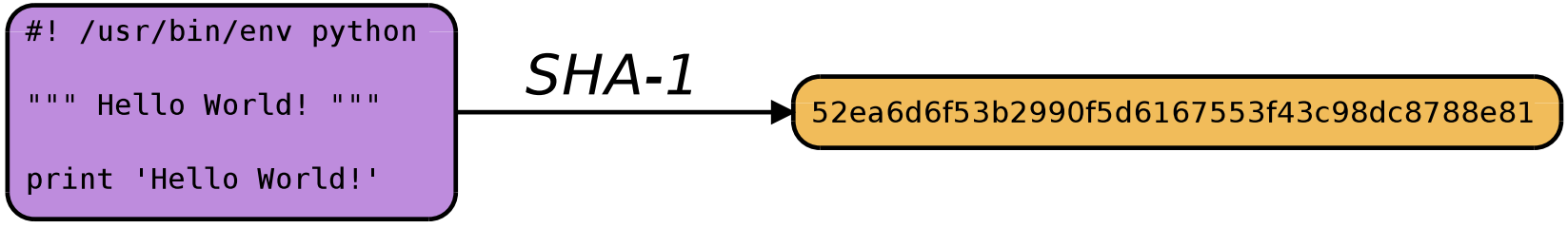
- *Blob*: Enthält den Inhalt einer Datei
- *Tree*: Eine Sammlung von Tree- und Blob-Objekten
- *Commit*: Besteht aus einer Referenz auf einen Tree mit zusätzlichen Informationen
  - *Author* und *Commiter*
  - *Parents*
  - *Commit-Message*





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



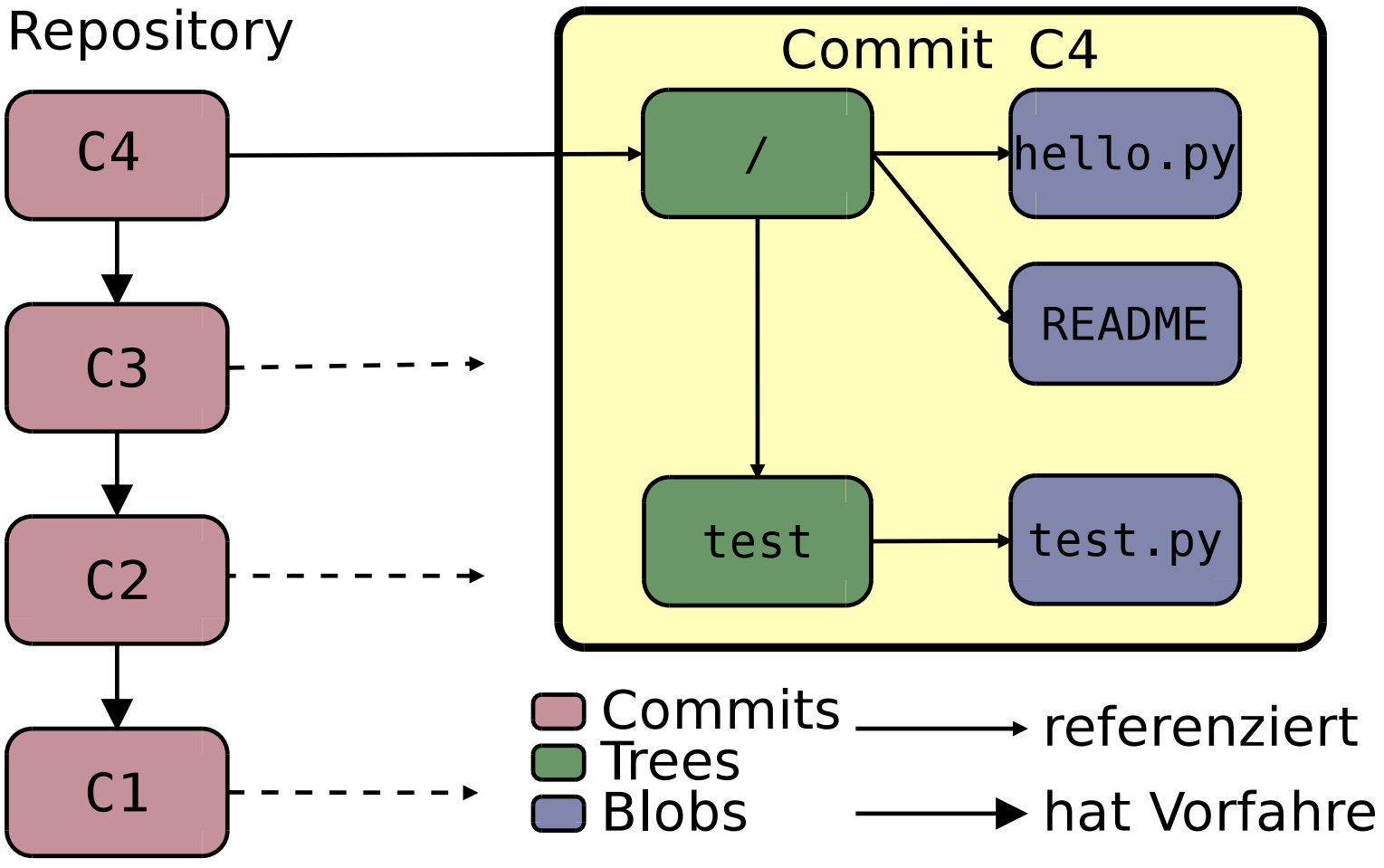
Objektverwaltung

- 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 (SHA1 IDs) auf die in dem Verzeichnis enthaltenen Dateien

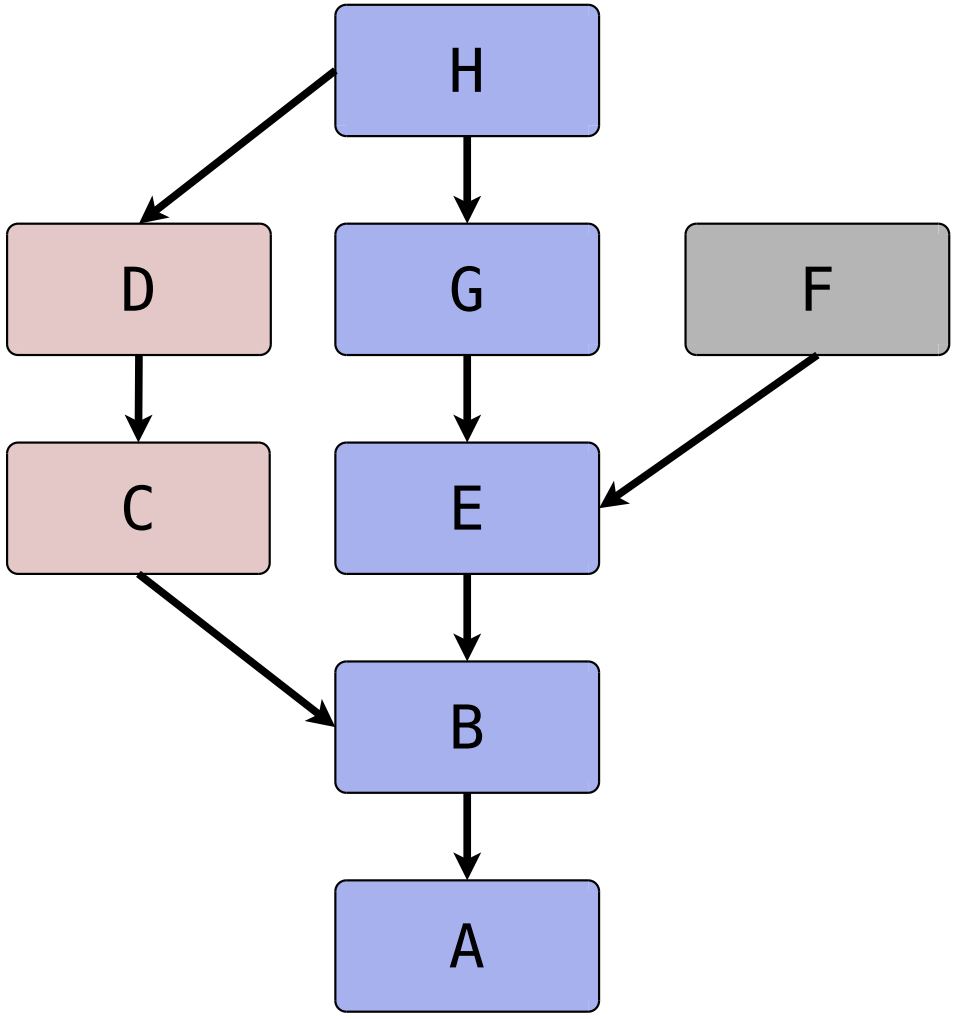
Zusammenfassung

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



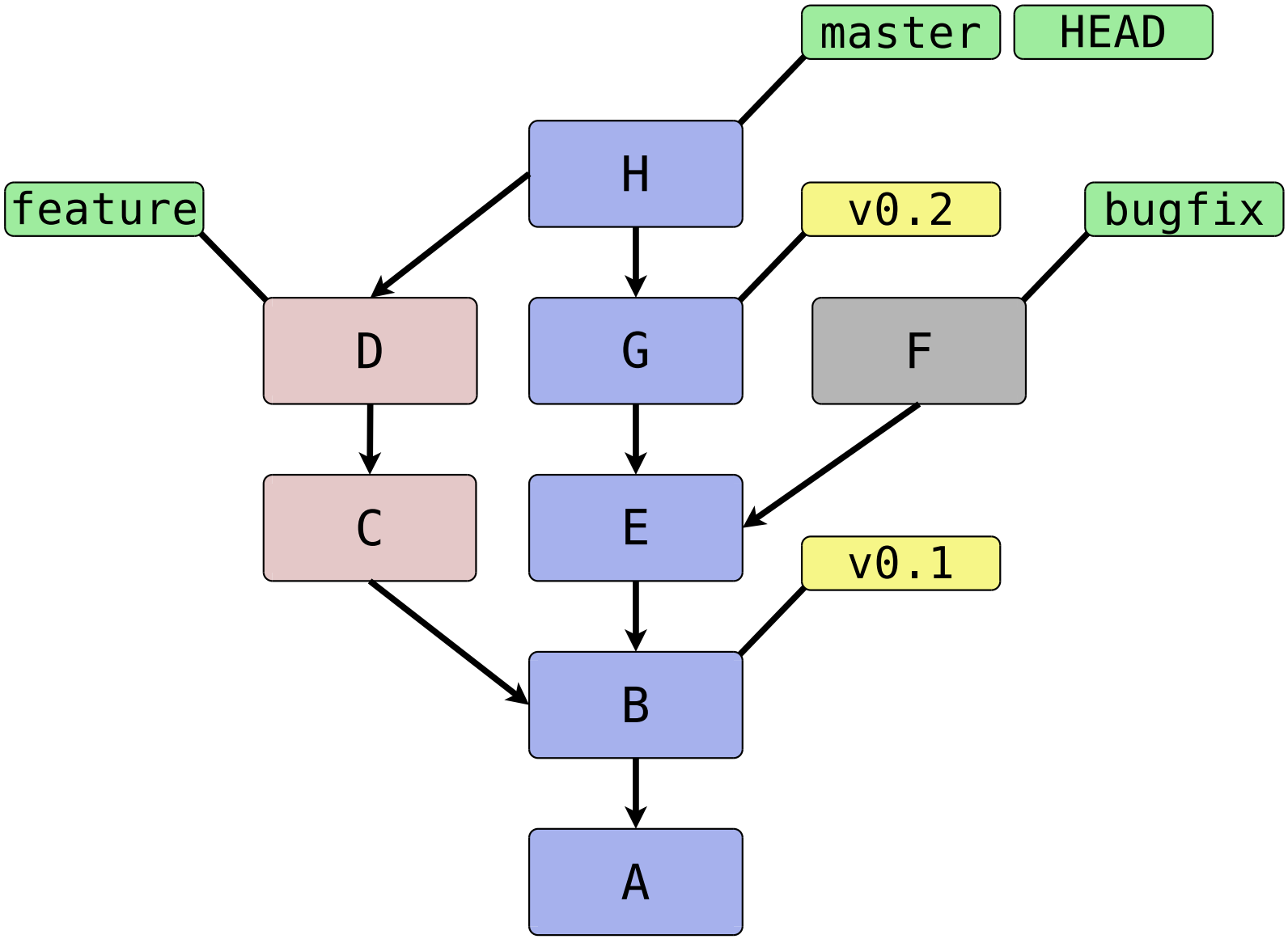
Commit Graph

Ein Repository ist ein *Gerichteter Azyklischer Graph*
Engl.: Directed Acyclic Graph (DAG)



Branches und Tags

Branches und Tags sind Zeiger auf Knoten in dem Graphen.



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

Merging: Branches Zusammenfügen

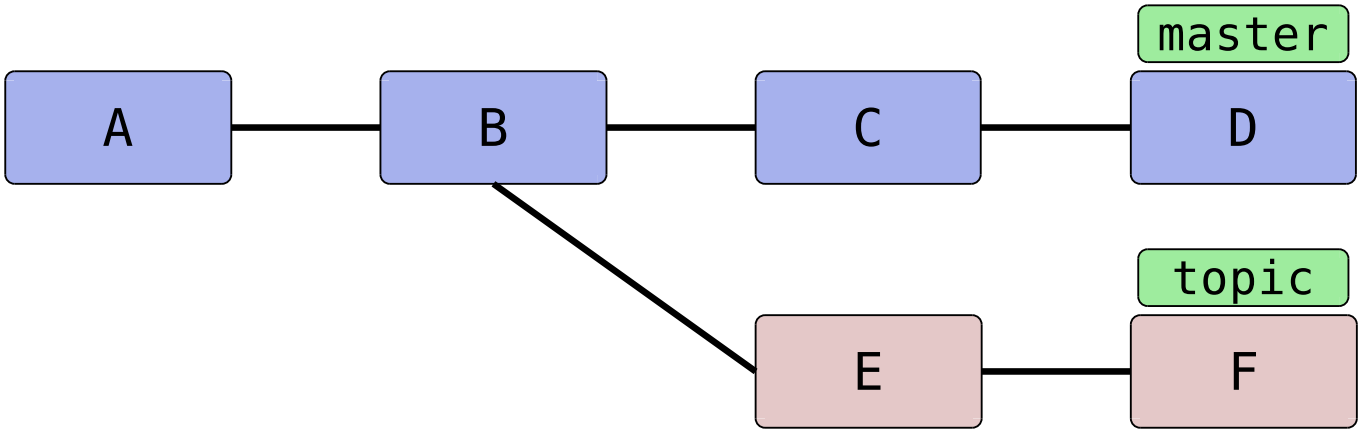
Simple Merge:

- `git merge` *`neues-feature`*

Fast-Forward Merge:

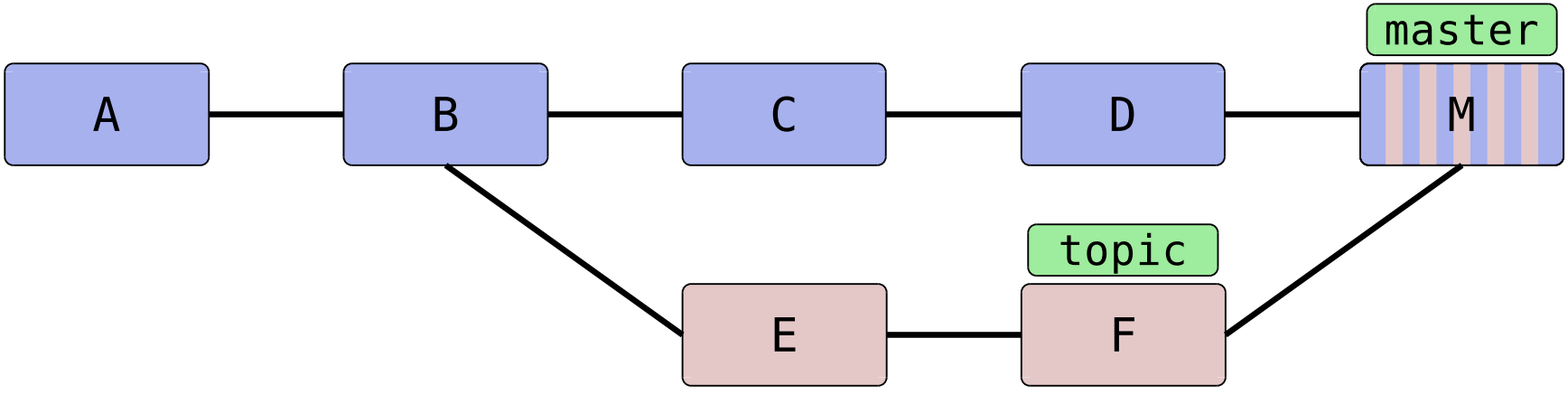
- Wird *`topic`* in *`master`* gemerget und *`topic`* basiert auf *`master`*, dann wird kein Merge-Commit erstellt, sondern nur der Zeiger »weitergerückt« bzw. »vorgespult«.

Vor dem Merge



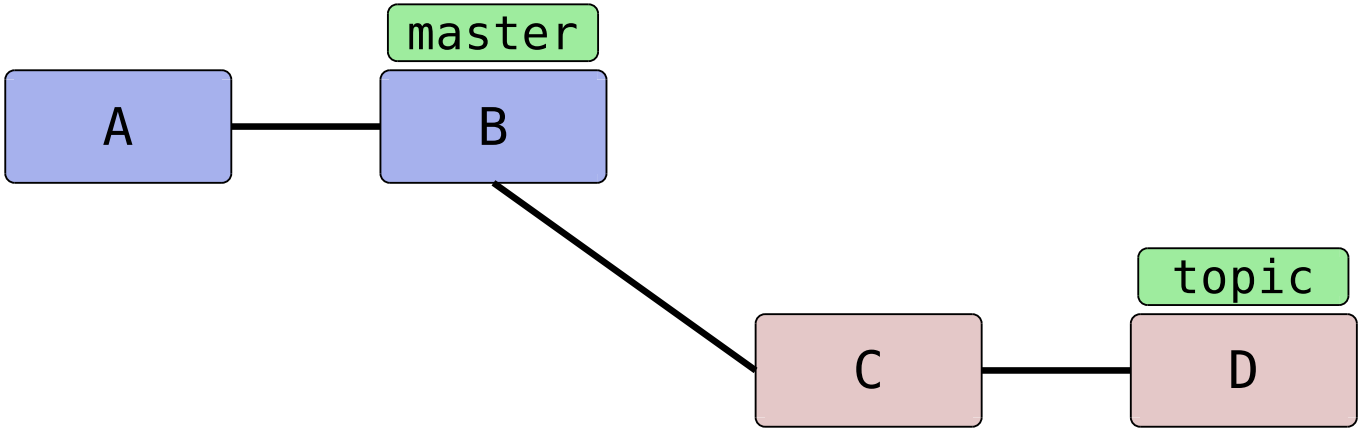
- `topic` ist fertig und soll in `master` integriert werden

Nach dem Merge



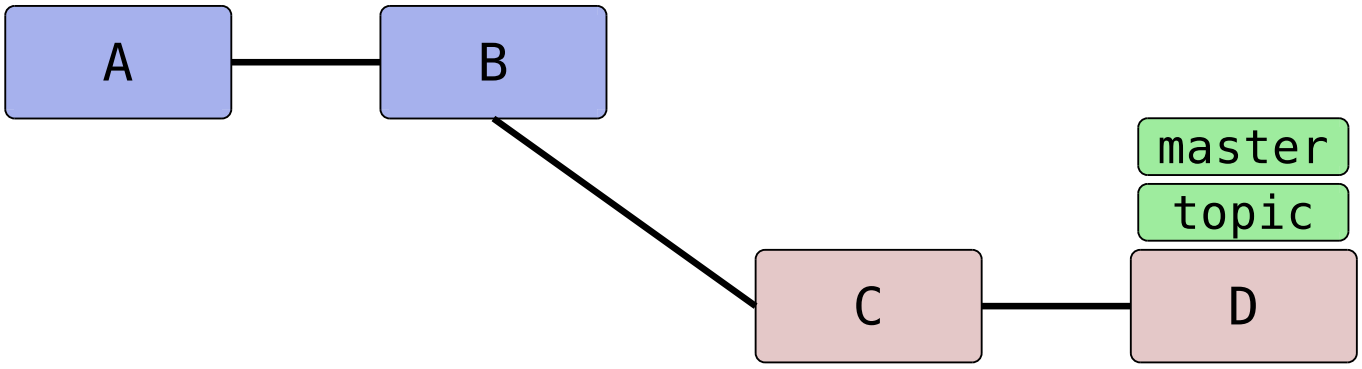
- Im `master` ausführen: `git merge topic`

Vor dem Fast-Forward



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

Nach dem Fast-Forward



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

Hilfe, Konflikte!

Bei einem `merge` kann es zu Konflikten kommen. Wie geht man damit um?

- `$EDITOR` *`konfliktdateien`*
- `git add` *`konfliktdateien`*
- `git commit -m "`*`Merge-Konflikt behoben`*`"`

Das Unterfangen abbrechen:

- `git reset --hard HEAD`

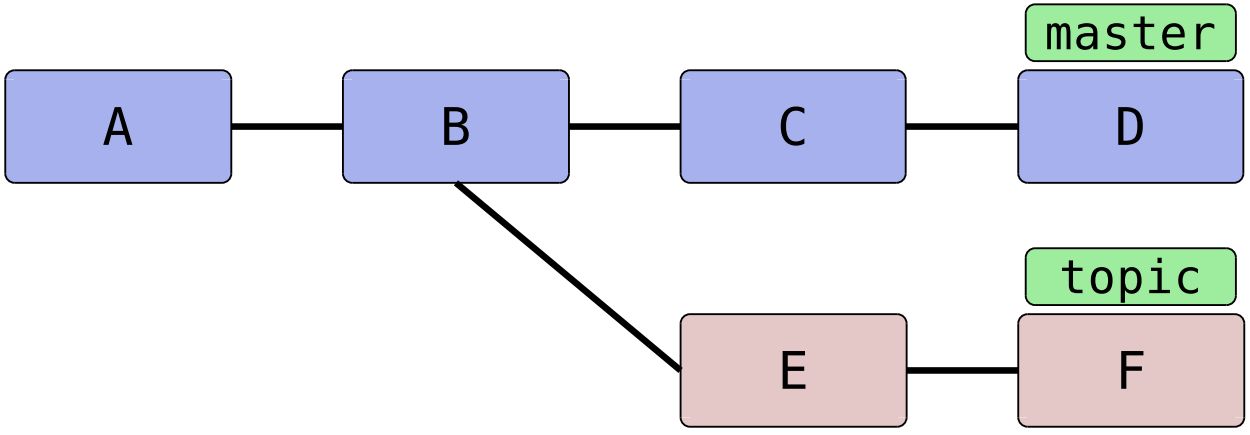
Einen Commit ändern

- Commit ändern = Neuen Commit erstellen, alten wegschmeißen
- Den letzten Commit (`HEAD`) ändern:

1. `$EDITOR` *`datei`*
2. `git add` *`datei`*
3. `git commit --amend`

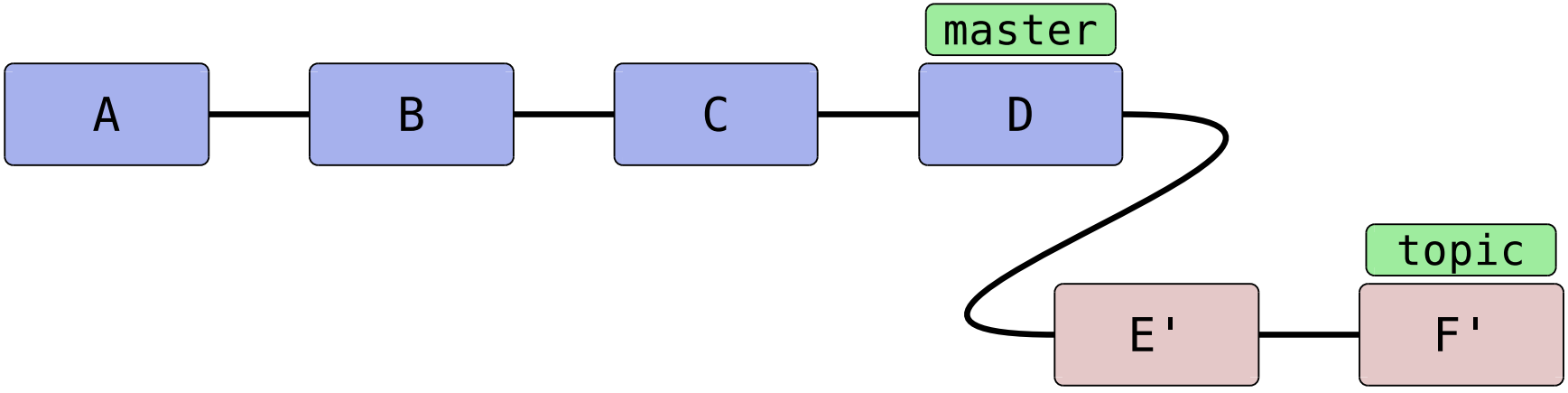
- Tiefer liegende Commits (`HEAD~1` etc.) können so nicht geändert werden!

Vor dem Rebase



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

Nach dem Rebase



- `git rebase master` *topic*

Rebase: Auf eine neue Basis bauen

- **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
```

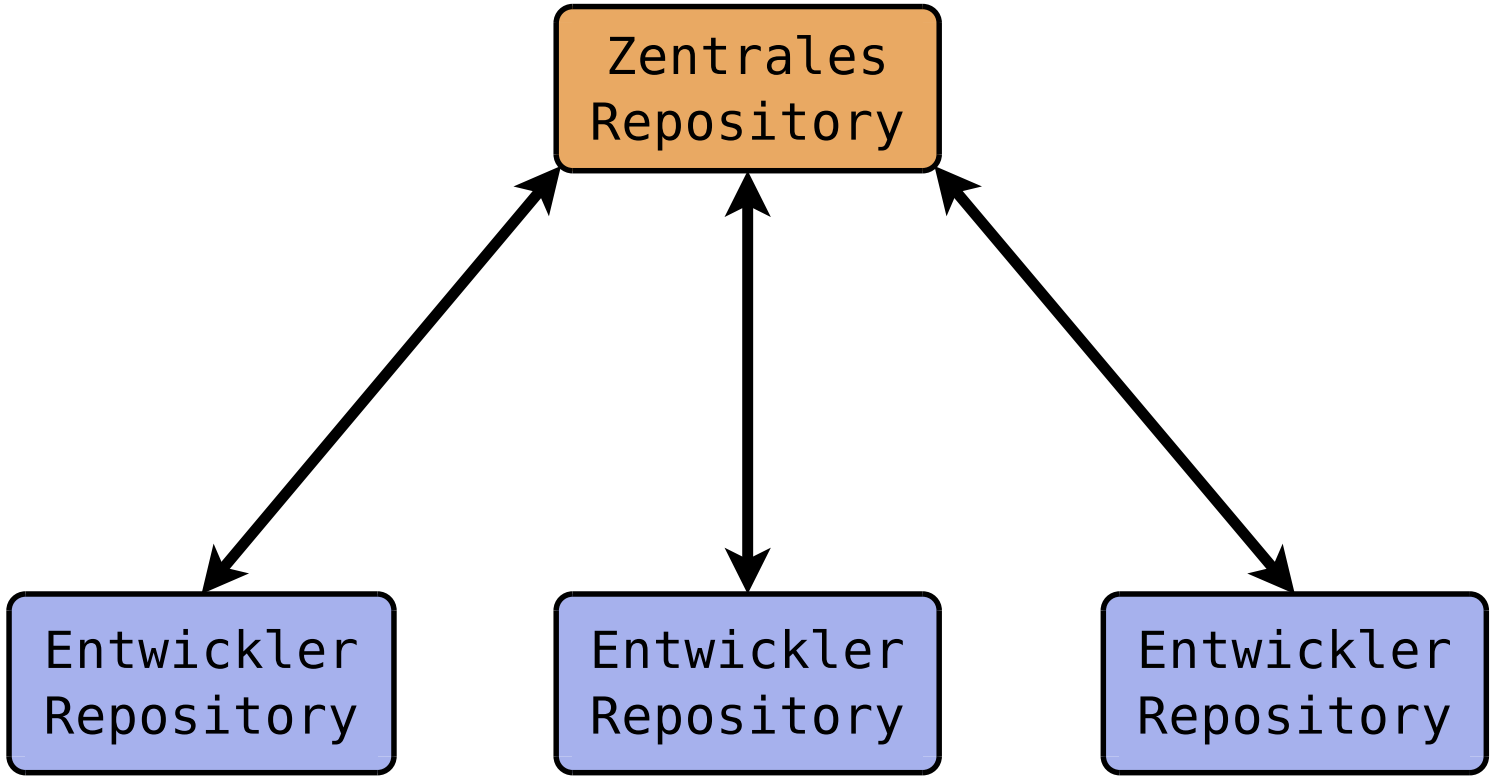
Rebasing: eine Warnung

- **Wichtig:** Man darf ***niemals*** Commits aus einem bereits veröffentlichten Branch – auf dem also womöglich Andere ihre Arbeit basieren – durch `git rebase` verändern!
- **Daher: Nur Unveröffentlichtes gegen Veröffentlichtes rebasen:**
  - `git rebase origin/master`
  - `git rebase v1.1.23`

Hinaus in die weite Welt!

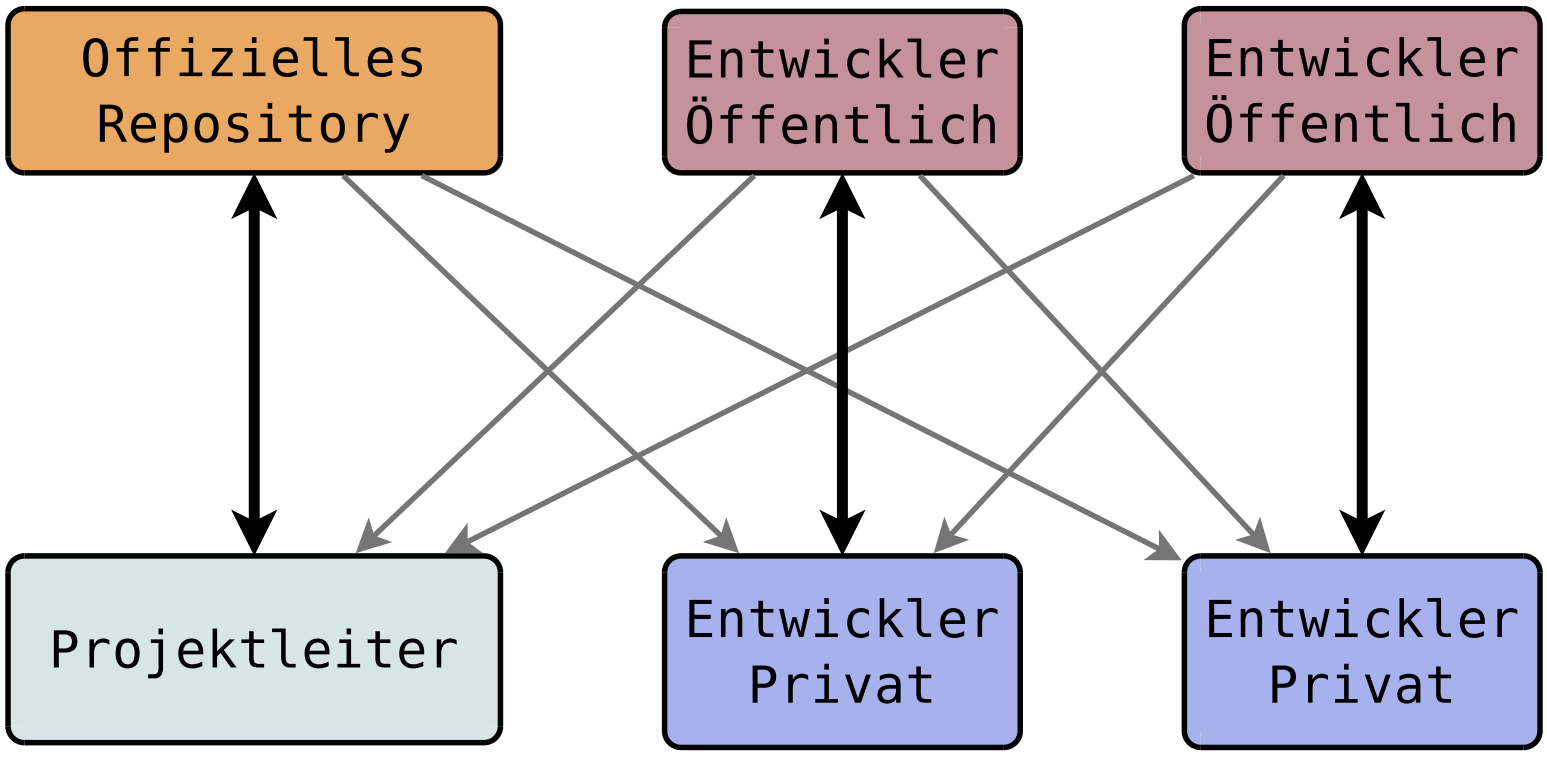
- Wir wollen unsere Arbeit mit der anderer Entwickler austauschen!
- Durch die verteilte Architektur von `git` braucht es keinen *zentralen* Server zu geben.
- Das Entwicklerteam muss sich auf einen *Workflow* einigen:
  - Shared Repository
  - Maintainer/Blessed Repository
  - Patch-Queue per E-Mail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

Zentralisiert



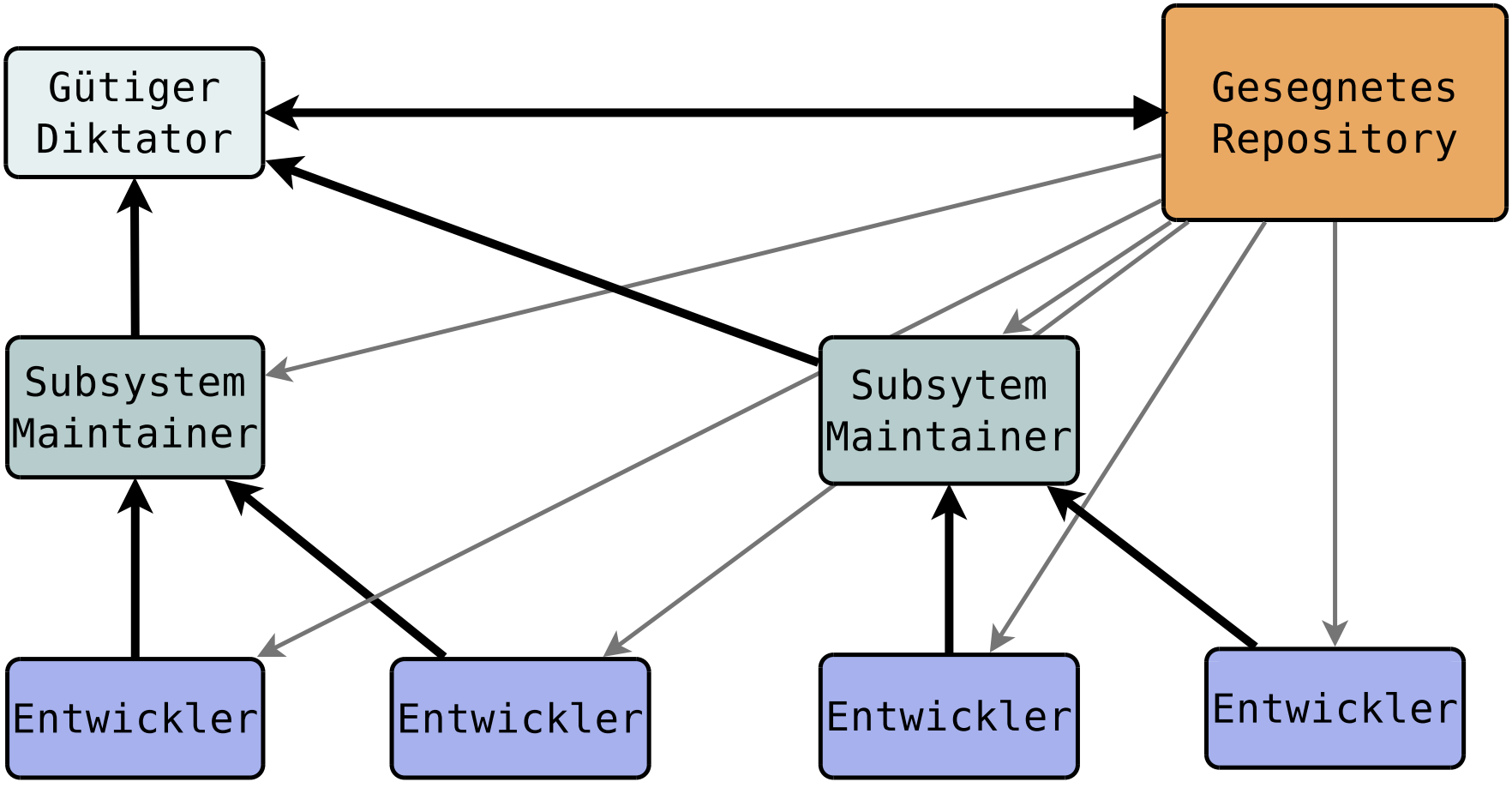
- Ein einziges zentrales Repository
- Alle Entwickler haben Schreibzugriff

Öffentliche Entwickler-Repositories



- Ein öffentliches Repository pro Entwickler
- Der Projektleiter integriert Verbesserungen

Patch-Queue per Email



- Stark vom Kernel und Git selbst verwendet

Remote Repositories / Remote Branches

Remote Repositories verwalten:

- `git remote -v`
- `git remote add` *`name url`*
- `git remote rm` *`name`*
- `git remote update`
  - Fragt bei allen Remote Repositories an, ob es neue Commits gibt. (Eigene Commits werden durch dieses Kommando **nicht** veröffentlicht!)

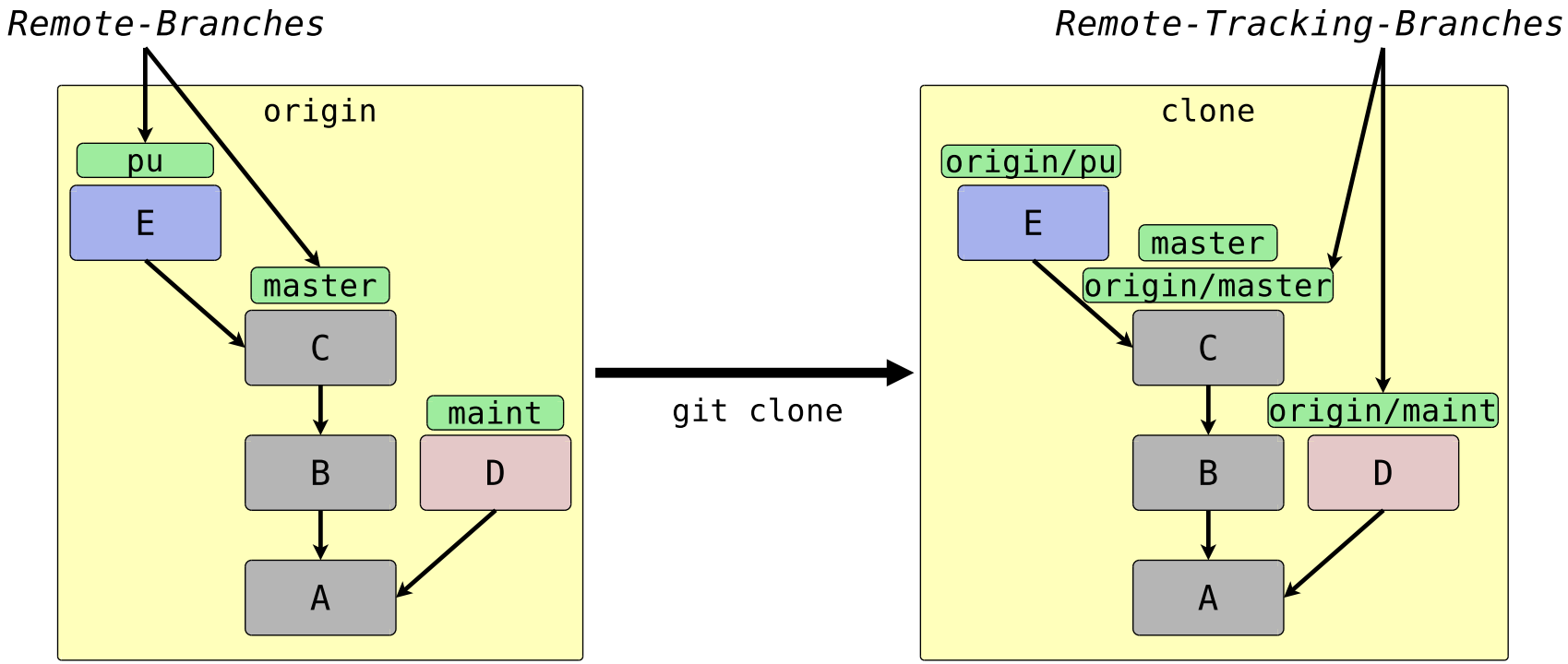
Details der Repositories ändern (z. B. Vertipper):

- `$EDITOR .git/config`

Remote Branches auflisten:

- `git branch -r`

Remote Branches vs. Remote Tracking Branches



Fremden Code holen, eigenen versenden

Aus einem anderen Repository neuen Code »ziehen«:

- `git pull` *`remote branch`*
  - `git pull blessed master`

Was hinter den Kulissen passiert:

1. `git fetch` *`remote branch`*
2. `git merge` *`remote/branch`*

Eigene Commits »pushen« oder per E-Mail senden:

- `git push` *`remote branch`*
- `git format-patch` *`seit-wann`*

Konventionen

- Wiederholter Einsatz von `git pull` erzeugt viele unnötige Merges
- Konvention:
  - Nicht im `master` entwickeln
  - `git remote update`, `master` *immer* Fast-Forwarden
  - Eigene Branches per `merge` in `master` integrieren

FF-Merge erzwingen

```
git merge --ff-only origin/master
git config --global alias.fm 'merge --ff-only'
```

GitHub – „Social Coding“

- GitHub stellt Git-Repositories zur Verfügung
  - Kostenlos und viel genutzt
  - Web-basiertes Interface
  - Aktionen „Fork“, „Follow“ und „Watch“
- Account erstellen:
  - → `http://www.github.com`
  - Authentifizierung per SSH-Schlüssel (ggf. erstellen)
- Ein eigenes Repository hochladen:
  - Repository auf GitHub erstellen
  - `git remote add` *`github`*
    `ssh://git@github.com:`*`user/projekt`*`.git`
  - `git push` *`github`* `master`

Kür: Was noch fehlt

- Rebase
- `git stash`
- Remote Branches löschen
- Git-Aliase
- Tags
- Reflog

Danke!

Vielen Dank für eure Teilnahme!

Fragen und Feedback gerne persönlich oder per Mail:

mail@nicidienase.de

Bonus-Folien

Rebase Interaktiv

- Das ist *Advanced Git Magic* – und will geübt sein!
- Rebase-Prozess anhalten, Commits »mittendrin« ändern, weiterlaufen lassen

Interaktives Rebase

```
git rebase -i master topic
```

- Anwendungsfälle *nur* lokal und für die *eigenen* Commits
  - Patch-Serie neu strukturieren
  - Typos aus den eigenen Commits entfernen
  - Offensichtliche Fehler glattbügeln

Rebase Interaktiv: Beispiele

Zwei Commits zusammenfassen

`git rebase -i HEAD~`n

$\rightarrow$ `pick` des zweiten Commits durch `fixup` ersetzen

- Einen Commit verschieben: Die Zeilen vertauschen
- Einen Commit editieren: mit `edit` markieren
- Einen Commit aufteilen: Siehe Cheatsheet

Whitespace und EOL

- Was ist kaputter Whitespace?
- `git diff --check` (z. B. → Hook)

- *Zeilenende:* Windows (CRLF) vs. UNIX (LF)
- `core.eol` bestimmt, was zu tun ist: `lf`, `crlf` oder `native`
- Git-Attribut `text` für Dateien, die automatisch konvertiert werden sollen
  - `echo '*.c text' > .gitattributes`
- `core.safecrlf`: Konvertierung verbieten, wenn ein Mix aus CRLF und LF vorhanden ist
- Mehr Infos: `gitattributes(5)`