Git-Workshop

Brandenburger Linux-Info-Tag

Valentin Hänel, Julius Plenz

6. November 2010

`http://github.com/Feh/git-workshop/raw/pdf/folien/git.pdf`

Wer kennt wen?

Wer kennt oder hat schon mal eines der folgenden Systeme benutzt?

- CVS/RCS
- SVN
- Mercurial, Darcs, Perforce, Bazaar
- Git

Wer kennt Git?

Wer hat schonmal ...

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

Ablaufplan

- Wir Führen die wichtigsten Git-Kommandos ein und üben damit Arbeitsschritte.

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://git.plenz.com/`*`git-tips`*

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

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

Dateien ignorieren:

- Globbing-Muster in `.gitignore` schreiben. Z. B.:
  - `*.aux`
  - `*.bak`
  - `*.swp`
  - `.gitignore` (die Datei selbst)

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`
- ... 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 Version von *datei* anschauen, wie sie vor zwei Commits war:

- `git show HEAD~2:`*`datei`*

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.

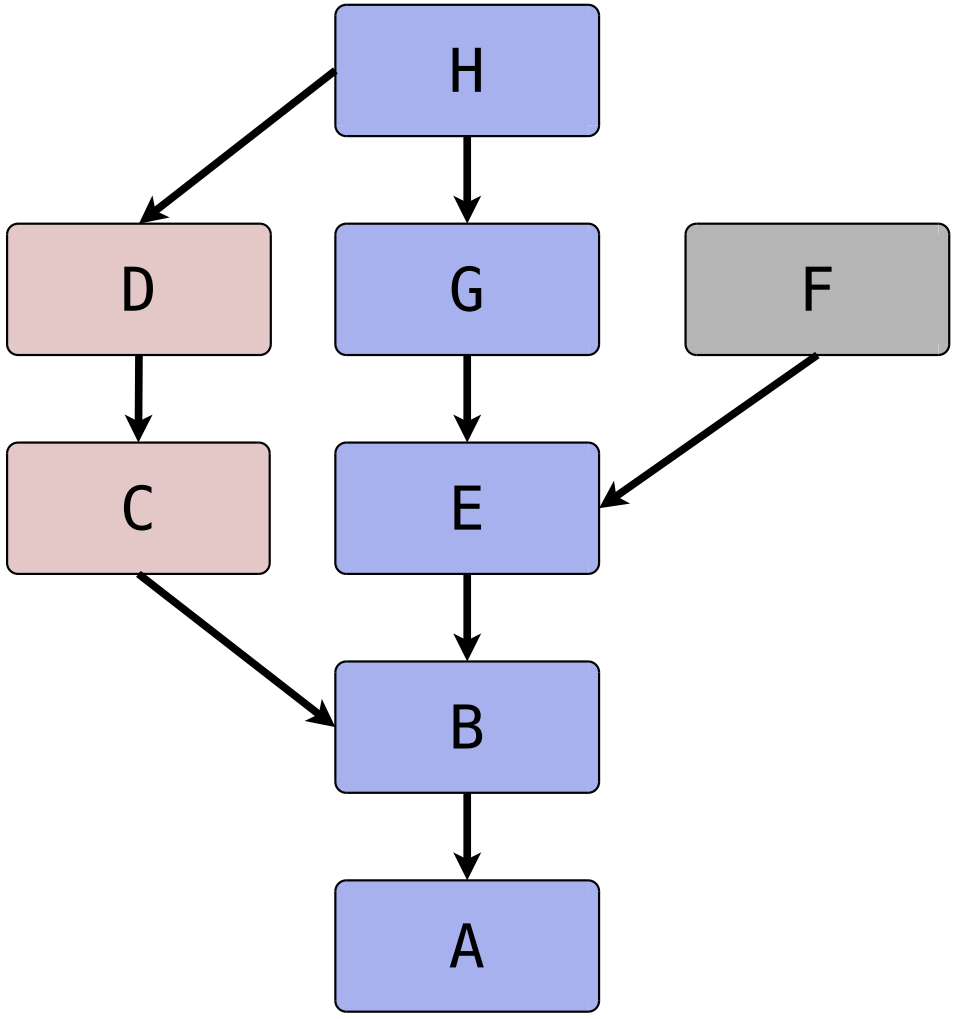
Branches ...

- 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.

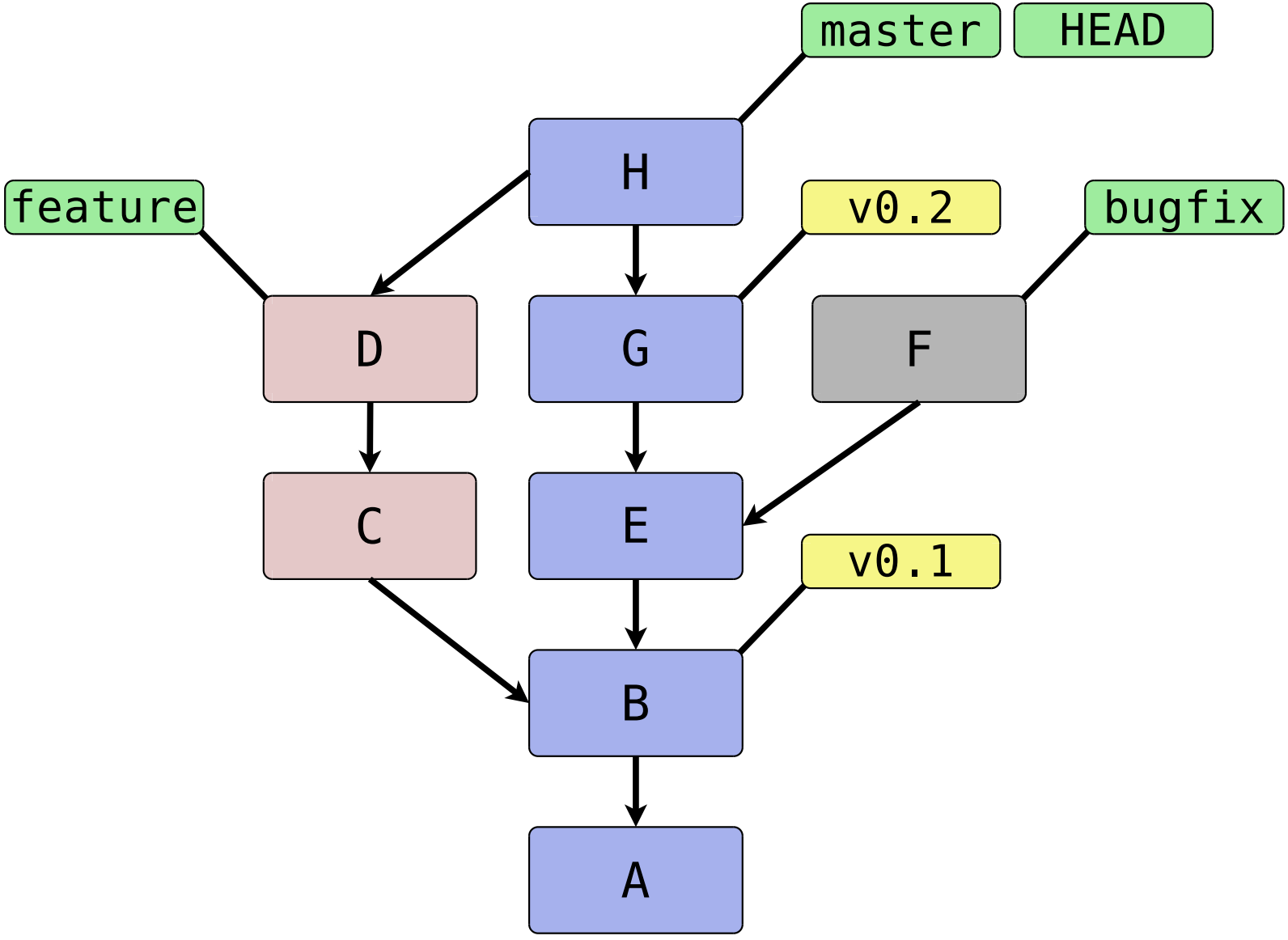
Commit Graph

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.



Beispiel: Zwei Branches

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

- `git branch apfel`
- `git checkout apfel`
- Commit machen
- `git checkout master`
- `git checkout -b birne`
- Commit machen
- `gitk --all`

Merging: Branches Zusammenfügen

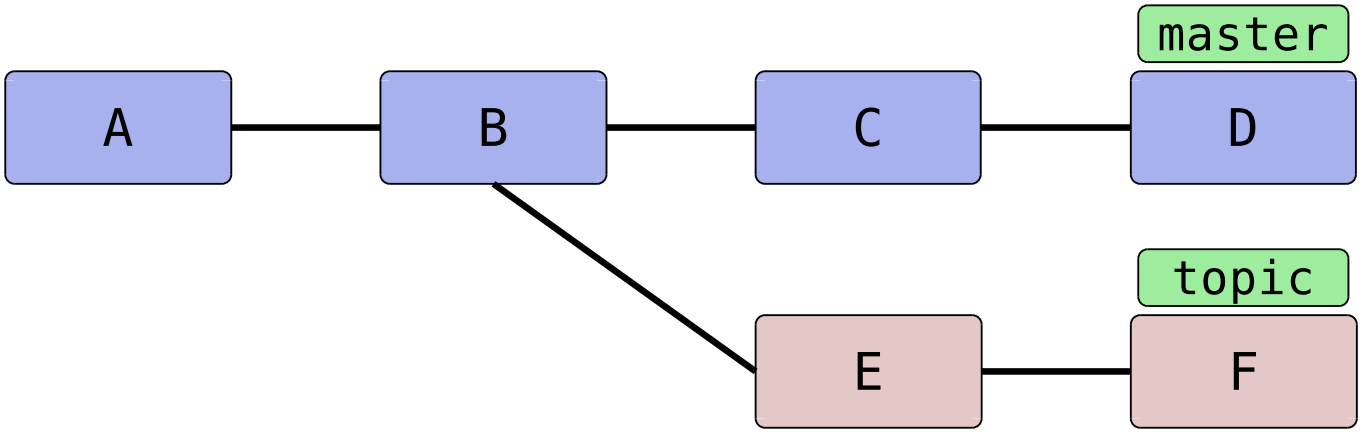
Simple Merge:

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

Fast-Forward Merge:

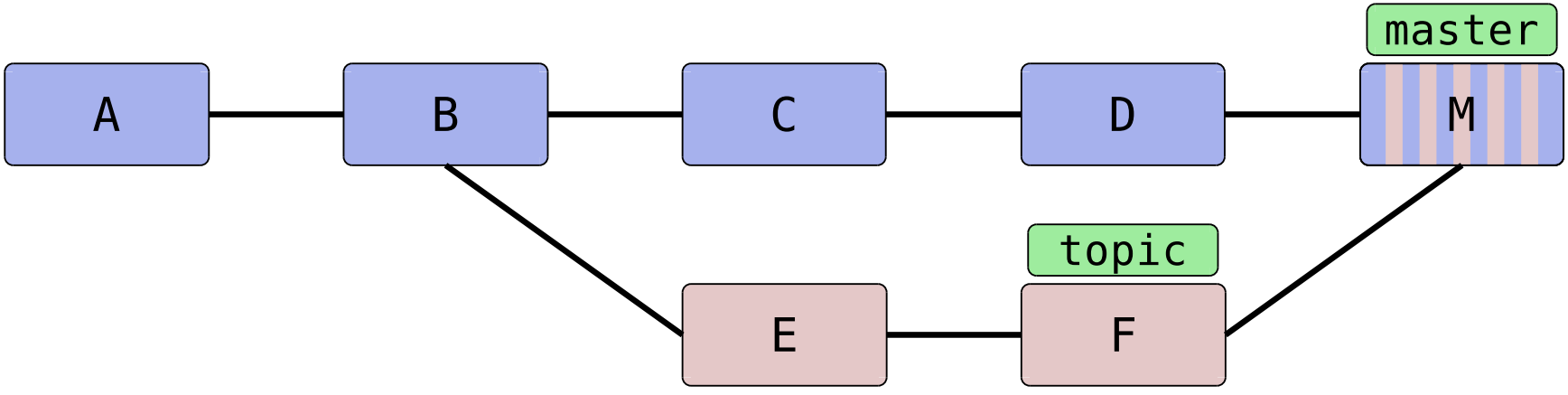
- Wird *`topic`* in *`master`* gemerget und *`topic`* basiert auf *`master`*, dann wird kein Merge-Commit erstellt, sondern nur der Zeiger »weitergerückt« bzw. »vorgespult«.

Vor dem Merge



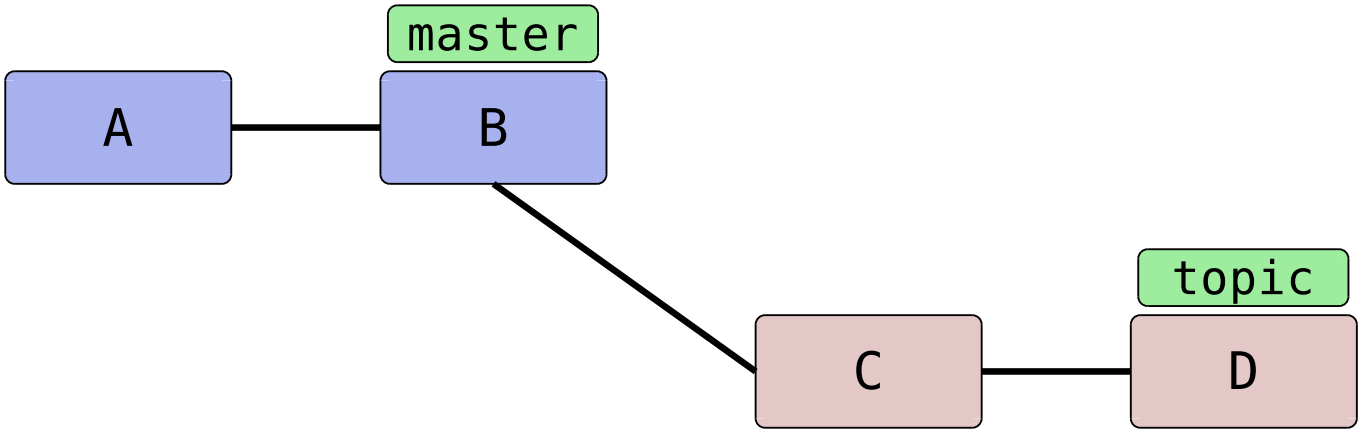
- `topic` ist fertig und soll in `master` integriert werden

Nach dem Merge



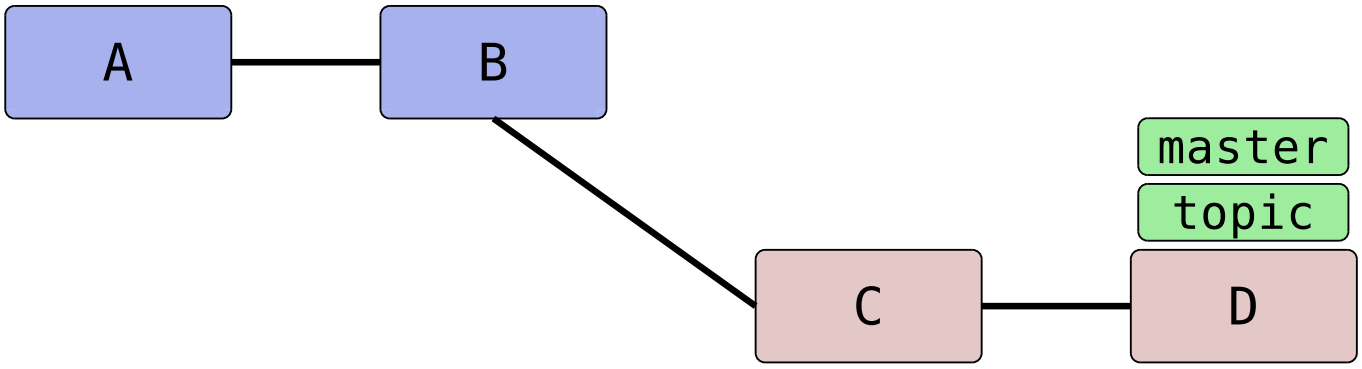
- Im `master` ausführen: `git merge topic`

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?

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

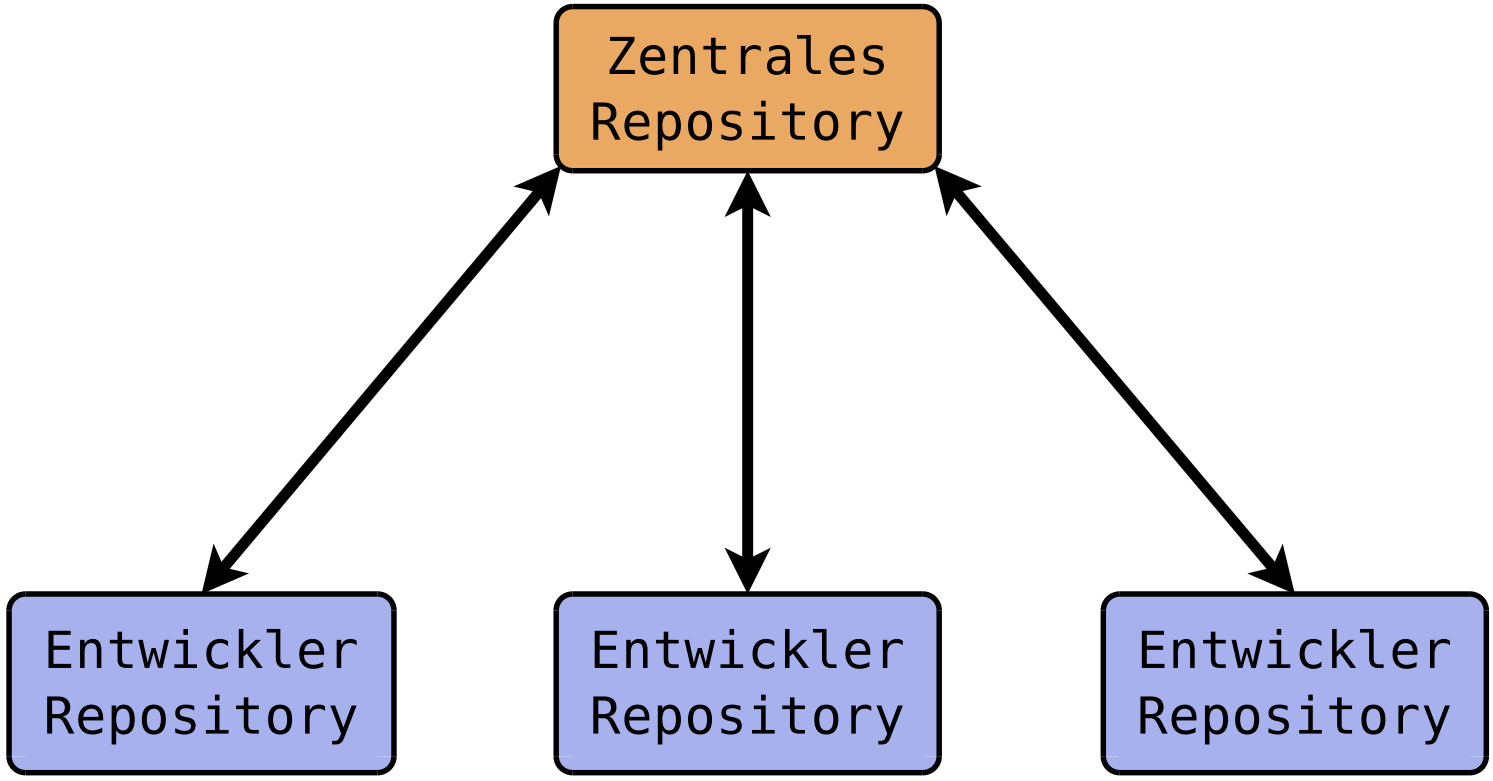
Das Unterfangen abbrechen:

- `git reset HEAD`

Hinaus in die weite Welt!

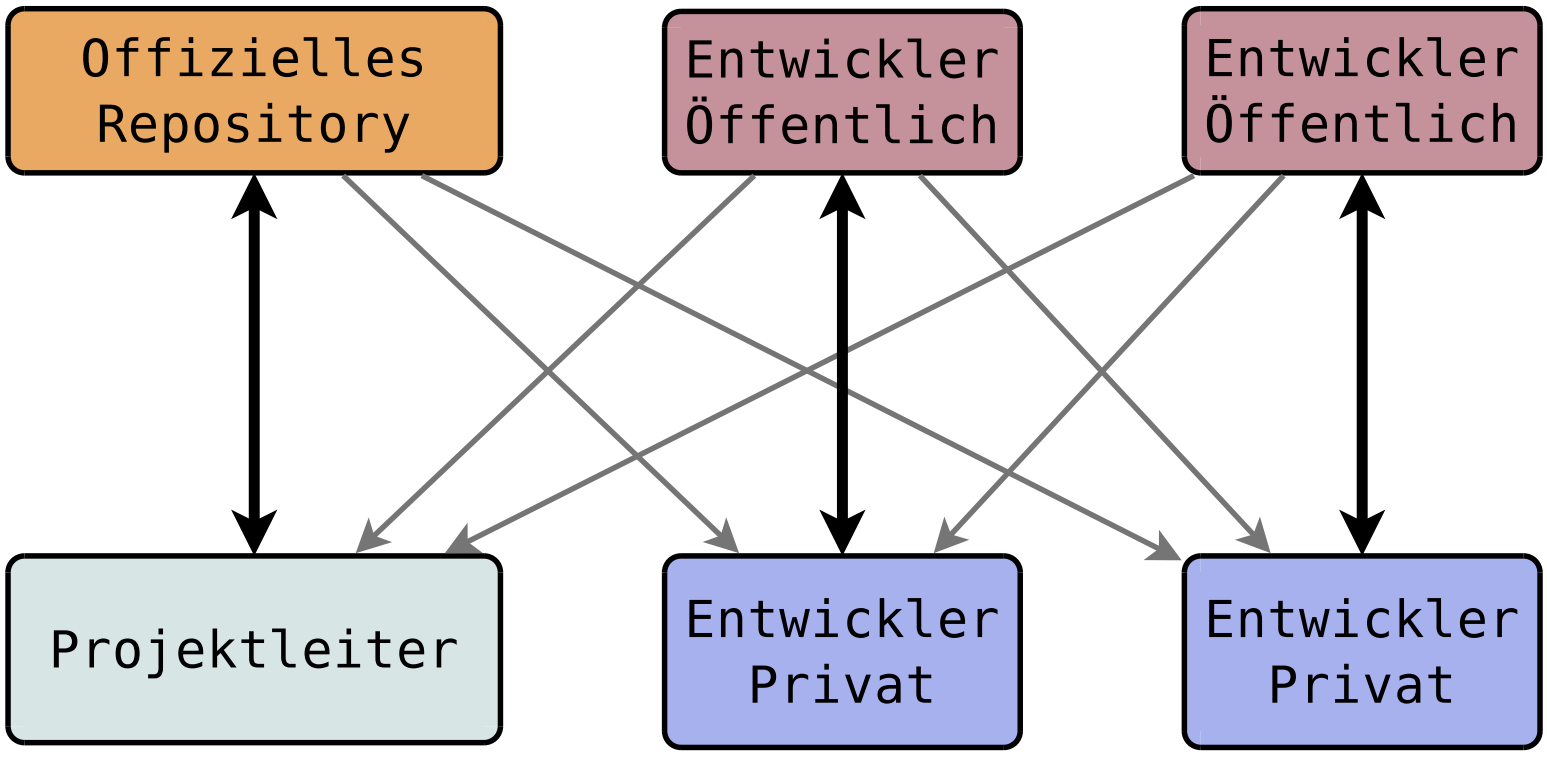
- Wir wollen unsere Arbeit mit der anderer Entwickler austauschen!
- Durch die verteilte Architektur von `git` braucht es keinen *zentralen* Server zu geben.
- Das Entwicklerteam muss sich auf einen *Workflow* einigen:
  - Shared Repository
  - Maintainer/Blessed Repository
  - Patch-Queue per E-Mail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

Zentralisiert



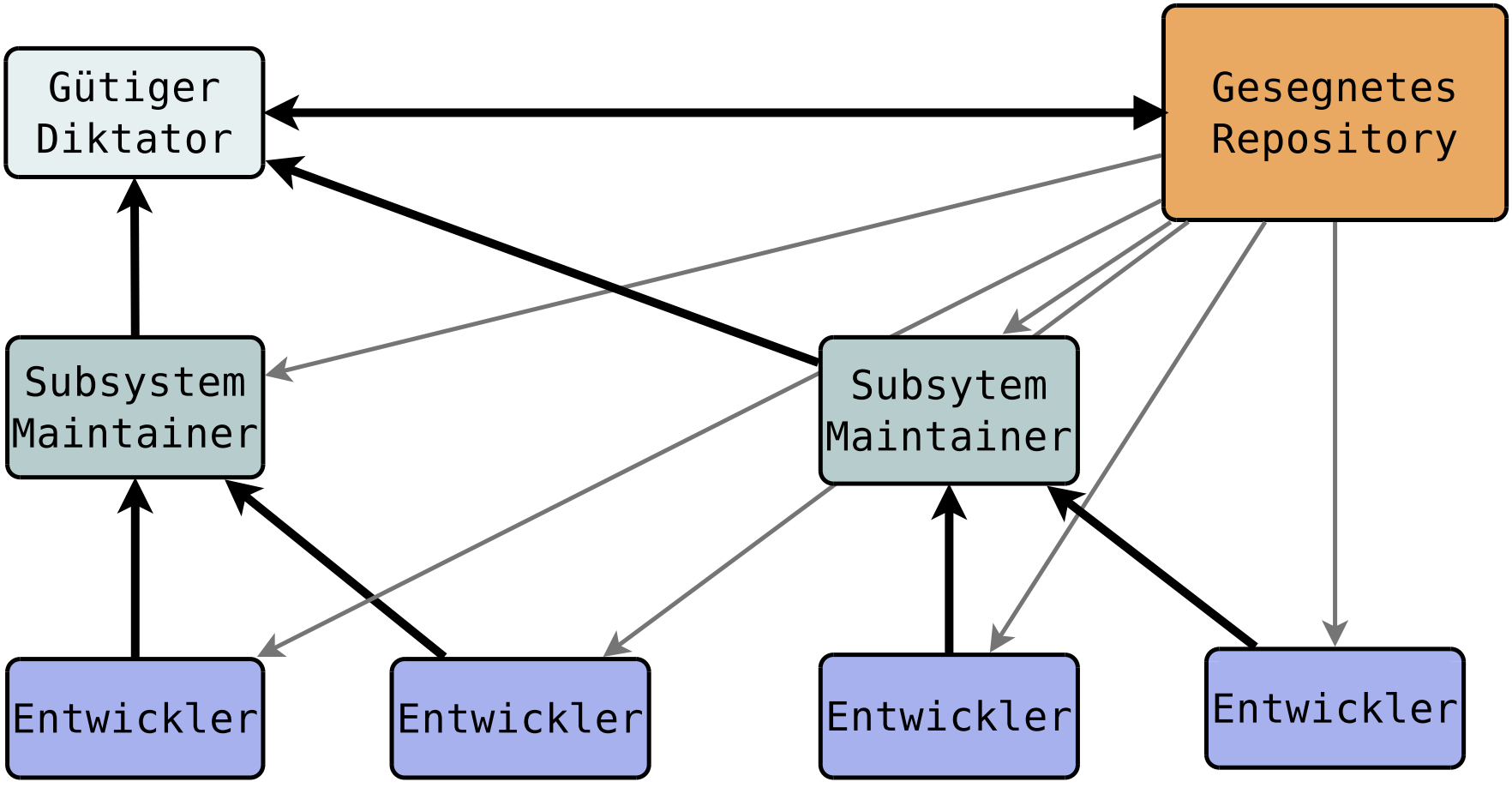
- Ein einziges zentrales Repository
- Alle Entwickler haben Schreibzugriff

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):

- `vim .git/config`

Remote Branches auflisten:

- `git branch -r`

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`*

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 per Mail:

Valentin Haenel
valentin.haenel@gmx.de

Julius Plenz
blit-2010@plenz.com

Rebasing

- **Rebase**: Einen Branch auf eine »neue Basis« stellen:
  - `git rebase master` *`topic`*
- Interaktiv Commits neu ordnen, bearbeiten, zusammenfassen oder aufteilen:
  - `git rebase -i HEAD~5`
- **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`