Versionsverwaltung mit git: Warum und wie.

Erstsemester-Einführung Informatik, 22.10.2020

FSFW

https://fsfw-dresden.de/

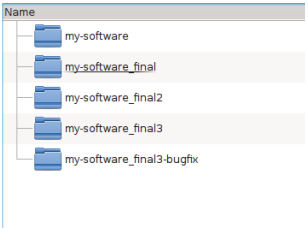
- uni-stick
- mitmachen
- ringvorlesung
- git-ws

- Hochschulgruppe an der TU, Studierende und andere Leute
- Bisherige Projekte:
  - Linux-Install-Party, Linux-Presentation-Day
  - Sprechstunde zu LaTeX u.a.
  - „Uni-Stick“ mit freier Software
  - Ringvorlesung: Freie Software und Freies Wissen als Beruf

Gliederung

Warum Versionsverwaltung?

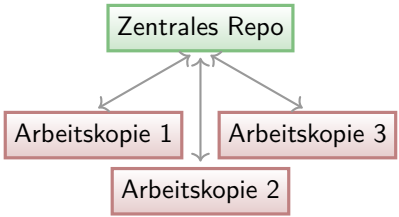
- Projekte bestehen aus schrittweisen Änderungen
- Bedürfnis, zu vorherigem Zustand zurückkehren zu können („Savegame“)
- Naiver Ansatz:



- Probleme:
  - Speicherplatz
  - Fehlende Übersicht
  - Skaliert nicht (Teamwork)

Warum Git? (1)

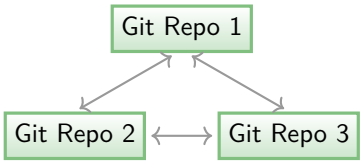
- Lösung 1: zentrale Versionsverwaltung
  - CVS (1986), SVN (2000)
  - Idee: Zentrales Repositorium und Arbeitskopien



- Probleme:
  - Abhängig von Server-Erreichbarkeit
  - Performanz

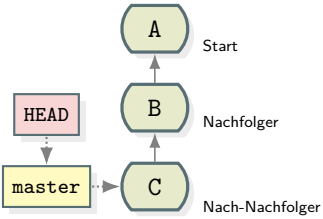
Warum Git? (2)

- Lösung 2: **de**zentrale Versionsverwaltung
  - mercurial (2005), bazaar (2005), **git** (2005)
  - Idee: Jeder hat ein vollwertiges Repositorium



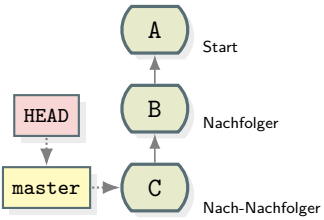
- Vorteile:
  - Alle Operationen lokal → schnell, unabhängig
  - Einfaches „branching“ und „merging“
  - ...

Einführung in git – Was ist ein Repo?



- Graph von Versionen einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

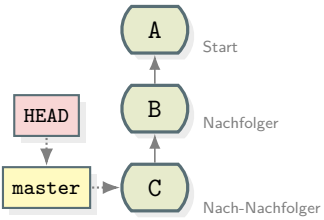
Einführung in git – Was ist ein Repo?



- Graph von Versionen einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

  Beispiel: b52c95e791e1dac76b7f70292e366de7caa76178
- *HEAD*: Knoten im Graphen; momentaner Bezugspunkt für Operationen
- *refs*: referenzieren Knoten im Graphen (Beispiele: `HEAD`, `HEAD^3`, `master`, `my_branch`, `b52c95e` (abgekürzte Commit-ID))

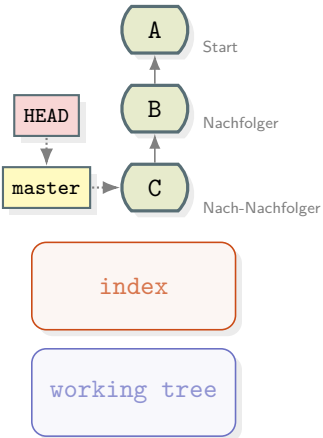
Einführung in git – Was ist ein Repo?



- Graph von Versionen einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

  Beispiel: b52c95e791e1dac76b7f70292e366de7caa76178
- *HEAD*: Knoten im Graphen; momentaner Bezugspunkt für Operationen
- *refs*: referenzieren Knoten im Graphen (Beispiele: `HEAD`, `HEAD^3`, `master`, `my_branch`, `b52c95e` (abgekürzte Commit-ID))

Einführung in git – Was ist ein Repo?



- Graph von Versionen einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

  Beispiel: b52c95e791e1dac76b7f70292e366de7caa76178
- *HEAD*: Knoten im Graphen; momentaner Bezugspunkt für Operationen
- *refs*: referenzieren Knoten im Graphen (Beispiele: `HEAD`, `HEAD^3`, `master`, `my_branch`, `b52c95e` (abgekürzte Commit-ID))

Einführung in git – Verwendung

- Wir empfehlen: git Bedienung via Kommandozeile
- Syntax: `git <command> [<args>]`
- Beispiele:
  - `git init`
  - `git add myscript.py`
  - `git commit -m "add basic functionality"`
  - `git push`

Einführung in git – Verwendung

- Wir empfehlen: git Bedienung via Kommandozeile
- Syntax: `git <command> [<args>]`
- Beispiele:
  - `git init`
  - `git add myscript.py`
  - `git commit -m "add basic functionality"`
  - `git push`
  - `git status`
  - `git log`
  - `git branch develop`
  - `git checkout master`
  - `git merge develop`
  - `git blame myscript.py`
  - `git diff`
  - `git difftool`

Einführung in git – Verwendung

- Wir empfehlen: git Bedienung via Kommandozeile
- Syntax: `git <command> [<args>]`
- Beispiele:
  - `git init`
  - `git add myscript.py`
  - `git commit -m "add basic functionality"`
  - `git push`
  - `git status`
  - `git log`
  - `git branch develop`
  - `git checkout master`
  - `git merge develop`
  - `git blame myscript.py`
  - `git diff`
  - `git difftool`
  - `git clone`
  - `git help <command>`
  - `git rebase`
  - `git config`
  - `gitk`

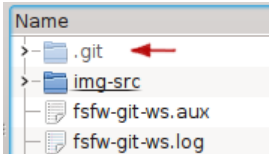
Praxis 1: Erste Schritte

- Installation: `apt install git` oder https://git-scm.com/download/win
- Konfiguration anpassen
  - `git config --global user.email "foo@bar.de"`
  - `git config --global user.name "Your Name"`
  - ...
- Eigenes Repo `foo` erstellen
  - `mkdir foo`
  - `cd foo`
  - `git init`
- Oder bestehendes Repo klonen
  - `git clone <url>`

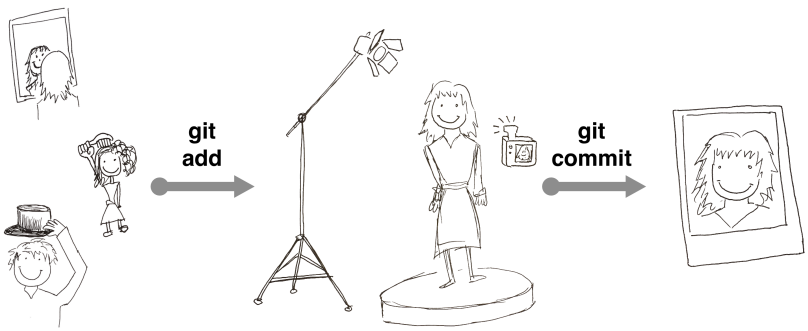
Praxis 1: Erste Schritte

- Installation: `apt install git` oder `https://git-scm.com/download/win`
- Konfiguration anpassen
  - `git config --global user.email "foo@bar.de"`
  - `git config --global user.name "Your Name"`
  - ...
- Eigenes Repo `foo` erstellen
  - `mkdir foo`
  - `cd foo`
  - `git init`
- Oder bestehendes Repo klonen
  - `git clone <url>`
- Hintergrund: Wo speichert git die relevanten Informationen?

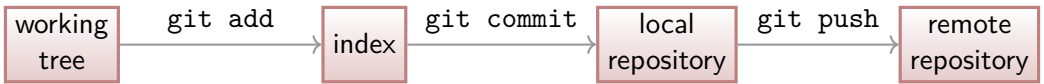
→ Versteckes Verzeichnis `.git`



Theorie: typischer Ablauf / „staging area“ (1)



Theorie: typischer Ablauf / „staging area“ (2)



Wozu zweiphasiger Commit-Prozess?

- Ermöglicht präzise, hoch aufgelöste Commits
  - Änderungen mancher Dateien (`git add dir1/*.html`)
  - Nur bestimmte Änderungen einer Datei (`git add -p`)
  - Alle Änderungen übernehmen und comitten (`git commit -a`)

⇒ nachvollziehbare, aussagekräftige Commit-History

Praxis: minimales Repo

- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`

Praxis: minimales Repo

- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`
- Änderungen durchführen, anzeigen und committen
  - `sed -i -- "s/Welt/Leute/g" README.md`
  - `git diff`

Praxis: minimales Repo

- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`
- Änderungen durchführen, anzeigen und committen
  - `sed -i -- "s/Welt/Leute/g" README.md`
  - `git diff`

```
14:58 $ git diff
diff --git a/README.md b/README.md
index ee7ae9a..31d5401 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 Hallo
-Welt
+Leute
```

Praxis: minimales Repo

- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`
- Änderungen durchführen, anzeigen und committen
  - `sed -i -- "s/Welt/Leute/g" README.md`
  - `git diff`
  - `git commit -am "change Hello-message"`

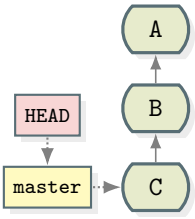
```
14:58 $ git diff
diff --git a/README.md b/README.md
index ee7ae9a..31d5401 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 Hallo
-Welt
+Leute
```

Praxis: minimales Repo

- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`
- Änderungen durchführen, anzeigen und committen
  - `sed -i -- "s/Welt/Leute/g" README.md`
  - `git diff`
  - `git commit -am "change Hello-message"`
- Sich Überblick verschaffen
  - `git status`
  - `git log`
  - `gitk`

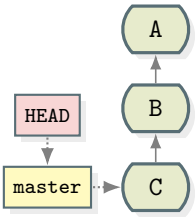
```
14:58 $ git diff
diff --git a/README.md b/README.md
index ee7ae9a..31d5401 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 Hallo
-Welt
+Leute
```

Theorie: Branches



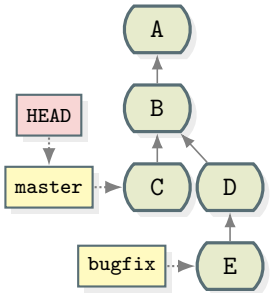
- Unkompliziertes paralleles Arbeiten an verschieden Versionen

Theorie: Branches



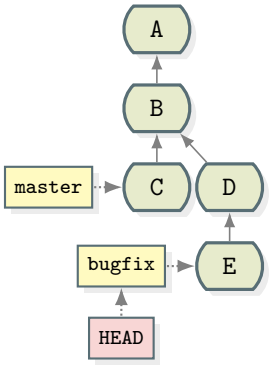
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`

Theorie: Branches



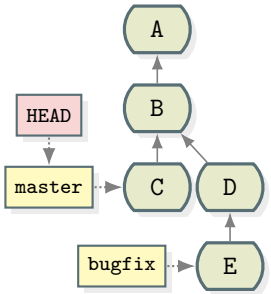
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich

Theorie: Branches



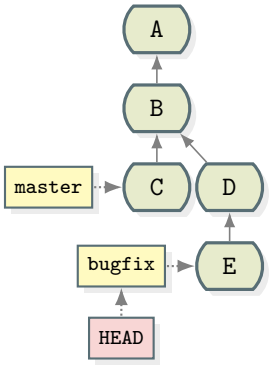
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout bugfix`

Theorie: Branches



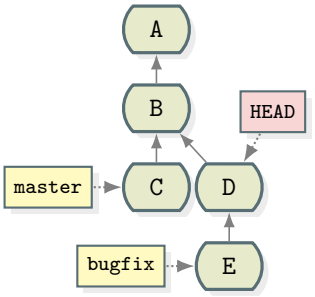
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout master`

Theorie: Branches



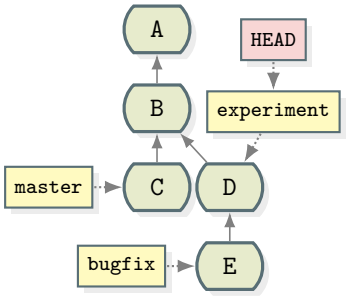
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout bugfix`

Theorie: Branches



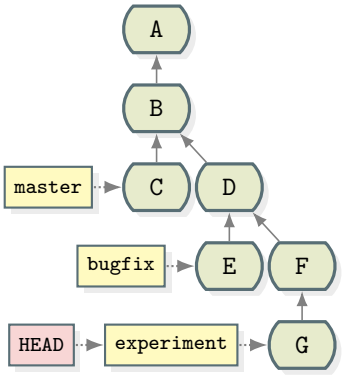
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`

Theorie: Branches



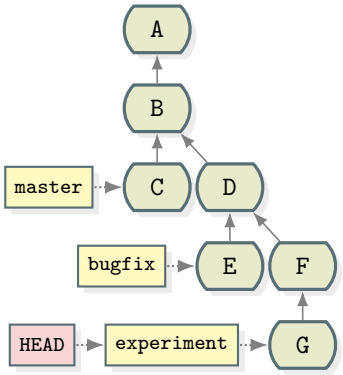
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`
- neuer Branch auf `HEAD` erstellen:
  `git checkout -b experiment`

Theorie: Branches



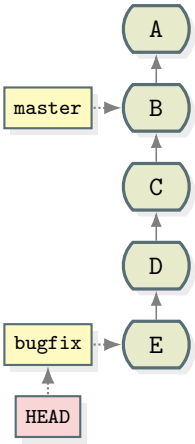
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`
- neuer Branch auf `HEAD` erstellen:
  `git checkout -b experiment`

Theorie: Zusammenfassung Branches



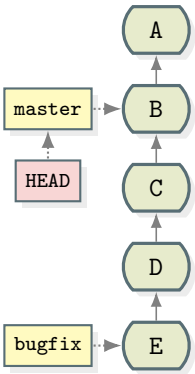
Branches sind *lokale* Lesezeichen auf Knoten im Revisionsgraphen. Beim Anlegen eines neuen Commits folgt der aktive Branch dem neuen `HEAD`.

Theorie: Mergen – Zusammenführen von Zweigen



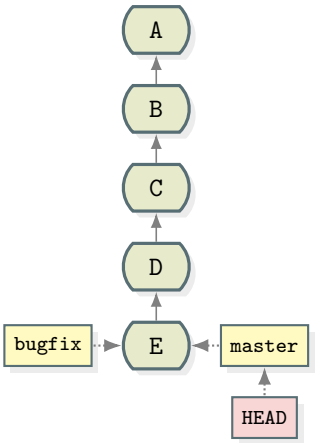
- Fall 1: Fast-Forward

Theorie: Mergen – Zusammenführen von Zweigen



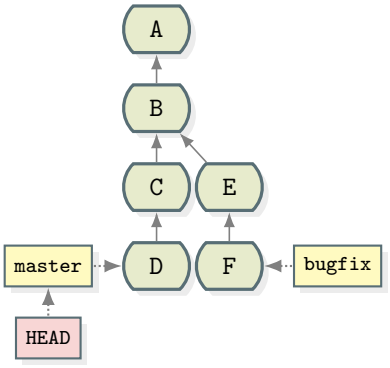
- Fall 1: Fast-Forward
  - `git checkout master`

Theorie: Mergen – Zusammenführen von Zweigen



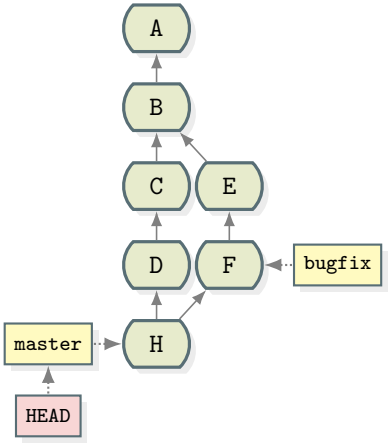
- Fall 1: Fast-Forward
  - `git checkout master`
  - `git merge bugfix`

Theorie: Mergen – Zusammenführen von Zweigen



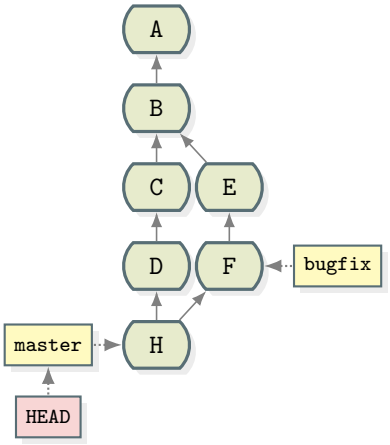
- Fall 1: Fast-Forward
  - `git checkout master`
  - `git merge bugfix`
- Fall 2: Parallele Zweige
  - `git checkout master`

Theorie: Mergen – Zusammenführen von Zweigen



- Fall 1: Fast-Forward
  - `git checkout master`
  - `git merge bugfix`
- Fall 2: Parallele Zweige
  - `git checkout master`
  - `git merge bugfix`
  - ⇒ Erzeugung eines „Merge-Commits“

Theorie: Mergen – Zusammenführen von Zweigen



- Fall 1: Fast-Forward
  - `git checkout master`
  - `git merge bugfix`
- Fall 2: Parallele Zweige
  - `git checkout master`
  - `git merge bugfix`
  - ⇒ Erzeugung eines „Merge-Commits“
  - Automatische Konfliktlösung ziemlich gut
  - Gelegentlich manueller Eingriff notwendig

Theorie: Mergen – Konflikte auflösen

- Konflikte beim Mergen: beide Versionen werden in der Datei markiert eingefügt

```
Gleiche Zeilen 1,
<<<<<<< HEAD
in unserem Zweig geänderte Zeilen,
=======
im anderen Zweig geänderte Zeile,
>>>>>>> other-branch
Gleiche Zeilen 2
```

- manuell editieren um den Konflikt aufzuheben (z. B. beide Zeilen behalten, die Änderungen in beiden Zeilen zusammenführen, eine Version behalten), die Marker entfernen
- `git add <conflicting-file>`
- `git commit`

Praxis

- Bereitgestelltes nicht-triviales Repo:

  `https://github.com/fsfw-dresden/git-ws-lyrik.git`

- Basis für verschiedene Aufgaben (Anregungen zum spielen)
  1. Repo klonen `git clone <url>`
  2. Überblick verschaffen: `gitk --all`
     a) Wie viele Commits, Committer gibt es?
     b) Wie viele Branches?
  3. Änderungen vornehmen
     a) zum Branch *weimar* wechseln
     b) in Datei `gedichte/prometheus.md` 'YYY' durch 'ich' ersetzen,
     c) committen
     d) analog im Branch *london* `sonnets/text1.md` 'XXX' durch 'thee' ersetzen

Praxis (2)

4. commit-History einzelner Dateien anzeigen
   a) `git blame AUTHORS.md`
   b) `git blame sonnets/text1.md`
5. Änderungen anzeigen
   a) ... seit dem vorletzten Commit: `git diff HEAD~`
   → beliebige Änderungen vornehmen
   b) ... seit dem letzten Commit: `git diff`
6. Branch *london* in *master* mergen
   a) master auschecken: `git checkout master`
   b) merge durchführen: `git merge london`
   c) Ergebnis anschauen: `gitk --all`

Praxis (3)

7. Irrelevante oder geheime Dateien ignorieren
   a) `git status` → Hilfsdateien oder geheime Dateien bemerken (`out.log` oder `id_rsa`)
      Häufig gibt es lokale Dateien, deren Änderungen nicht von git verfolgt werden sollen → Datei `.gitignore` hilft
   b) ignore-Datei ergänzen: `echo "out.log" >> .gitignore`
   c) Staus-Änderung zur Kenntnis nehmen: `git status`
   d) Committen: `git commit -am "ignore log file"`
   e) Zur Kenntnis nehmen: `git status` → „working directory clean“

Hinweis: Inhalt der Datei `.gitignore` (nachprüfen):

```
# This file specifies which files should not be tracked by git
out.log
```

- Bedeutung der Zeilen
  7.1 Erklärender Kommentar (stand vorher schon drin)
  7.2 Ignoriere Dateien mit dem Namen `out.log`

Praxis (3)

8. Branch *rom* in *master* mergen
   a) merge durchführen → Konflikt zur Kenntnis nehmen
   b) Überblick verschaffen: `git status` `gitk --all`
   c) Manuell Konflikt in `AUTHORS.md` beheben
   d) Merge abschließen durch commiten der Änderungen:

```
git commit --add -m "merge branch rom after manual conflict resolution"
```

Weitere Ideen:

- Eigenen Branch anlegen mit bestimmten Eltern-Knoten
  - Commit-ID herausfinden: `git log` (ersten 4 Zeichen reichen)
  - `git checkout <id>`
  - `git checkout -b mybranch`
- Rebase aller Branches, damit das repo linear wird
  - Hintergrundwissen:
    - `git help rebase`
    - https://onlywei.github.io/...
    - dort `git rebase master` eintippen, Animation anschauen und Text lesen

Schlussbemerkungen (1)

- github $\neq$ git
  - git: Freies Tool zur Versionsverwaltung
  - github: Kommerzieller Webservice *basierend auf* git

Schlussbemerkungen (1)

- github $\neq$ git
  - git: Freies Tool zur Versionsverwaltung
  - github: Kommerzieller Webservice *basierend auf* git
- git nicht gut für (große) Binärdateien
  - Merges werden ungemütlich
  - Grund: Delta-Kompression basiert auf zeilenweisen Diffs

  $\rightarrow$ `.git`-Verzeichnis wird ggf. sehr groß

Schlussbemerkungen (1)

- github $\neq$ git
  - git: Freies Tool zur Versionsverwaltung
  - github: Kommerzieller Webservice *basierend auf* git
- git nicht gut für (große) Binärdateien
  - Merges werden ungemütlich
  - Grund: Delta-Kompression basiert auf zeilenweisen Diffs

  $\rightarrow$ `.git`-Verzeichnis wird ggf. sehr groß
- Nicht behandelte wichtige Konzepte/Kommandos
  - `git fetch`, `git pull`, `git push`, `git rebase`, ...
  - Siehe Cheat-Sheet

Schlussbemerkungen (1)

- github $\neq$ git
  - git: Freies Tool zur Versionsverwaltung
  - github: Kommerzieller Webservice *basierend auf* git
- git nicht gut für (große) Binärdateien
  - Merges werden ungemütlich
  - Grund: Delta-Kompression basiert auf zeilenweisen Diffs

  $\rightarrow$ `.git`-Verzeichnis wird ggf. sehr groß
- Nicht behandelte wichtige Konzepte/Kommandos
  - `git fetch`, `git pull`, `git push`, `git rebase`, ...
  - Siehe Cheat-Sheet
- Weitere Tipps:
  - Doku kennen
  - Status-Infos im Bash-Prompt

  - Aliase in `.gitconfig` (z.B.: `git co` $\rightarrow$ `git checkout`)
  - Globale gitignore-Datei anlegen
  - Bewährtes Branching-Modell anwenden

Schlussbemerkungen (2): Blick übern Tellerrand

Bits und Bäume

- Digitalisierung und Nachhaltigkeit verbinden
- Ressourcen, Freie Software, Privatsphäre, Demokratie, ...
- `https://dresden.bits-und-baeume.org/`

TU–Umweltinitiative

- Umweltringvorlesungen
- `https://tuuwi.de/`

Students4Future

- → `https://fffdd.de/`

Schlussbemerkungen (3)

- Fragen?

- Unterstützung: (im Rahmen unserer Möglichkeiten)
  - https://fsfw-dresden.de/sprechstunde
  - https://fsfw-dresden.de/git-ws
  - https://ifsr.de/

  - kontakt@fsfw-dresden.de
  - fsr@ifsr.de

Quellen und Links (Auswahl)

- https://git-scm.com/documentation
- https://git-scm.com/documentation/external-links
- https://stackoverflow.com/questions/tagged/git
- ...