



Versionsverwaltung mit git: Warum und wie.

Erstsemester-Einführung Informatik, 01.10.2018

<Eigenwerbung>

Wer sind wir?



- ▶ Hochschulgruppe an der TU (gegründet 2014, ca. 8 P.)
- ▶ Studierende (TU, HTW) und andere Leute
- ▶ Hochschulen als Zielgruppe (Multiplikationswirkung) und Arbeitsfeld (Räume, Strukturen)

Wer sind wir?



- ▶ Hochschulgruppe an der TU (gegründet 2014, ca. 8 P.)
- ▶ Studierende (TU, HTW) und andere Leute
- ▶ Hochschulen als Zielgruppe (Multiplikationswirkung) und Arbeitsfeld (Räume, Strukturen)

- ▶ Bisherige Projekte
 - ▶ Linux-Install-Party, Linux-Presentation-Day
 - ▶ Verschlüsselungsgewinnspiel
 - ▶ Monatliche Sprechstunde zu \LaTeX u.a.
 - ▶ Formulierung eines Programmpapiers
 - ▶ „Uni-Stick“: 130 × 8 GB mit freier Software

Warum machen wir das? Aus Überzeugung!



- ▶ *Überzeugung 1*: freie und quelloffene Software ist (oft) besser (technische + nicht technische Argumente)

Warum machen wir das? Aus Überzeugung!



- ▶ *Überzeugung 1*: freie und quelloffene Software ist (oft) besser (technische + nicht technische Argumente)
- ▶ *Überzeugung 2*: öffentlich finanzierte wissenschaftliche Inhalte (AutorInnen, GutachterInnen) sollten nicht von öffentlich finanzierten Bibliotheken für horrenden Summen von Zeitschriften-Verlagen gekauft werden müssen

Projekt Uni-Stick (3. Auflage)



- ▶ **4000** Flyer in Ersti-Tüten: **Gutscheine** für 8 GB Stick mit freier Software fürs Studium, ≈ 800 € vom StuRa für 130 Stk.
- ▶ Live-Linux / freie Windows-Programme

Projekt Uni-Stick (3. Auflage)



- ▶ **4000** Flyer in Ersti-Tüten: **Gutscheine** für 8 GB Stick mit freier Software fürs Studium, ≈ 800 € vom StuRa für 130 Stk.
- ▶ Live-Linux / freie Windows-Programme
- ▶ Vorbereitung hat Arbeit und Spaß gemacht



Projekt Uni-Stick (3. Auflage)



- ▶ **4000** Flyer in Ersti-Tüten: **Gutscheine** für 8 GB Stick mit freier Software fürs Studium, ≈ 800 € vom StuRa für 130 Stk.
- ▶ Live-Linux / freie Windows-Programme
- ▶ Vorbereitung hat Arbeit und Spaß gemacht
- ▶ Ist gut angekommen (ca. 250 TN)



Ringvorlesung



Im WS 2018/19 organisiert die FSFW erstmals eine Ringvorlesung
Thema:

Freie Software und Freies Wissen als Beruf

→ Ist interessant und bringt bis zu 3 CP

Zeit: Dienstags ab 16.10.2018, 17-18:30 Uhr

Ort: HTW Dresden, Raum Z 254

Infos: fsfw-dresden.de/ringvorlesung

Zukunftsideen



- ▶ Fortführung „Uni-Stick“
- ▶ Studierende zum Nutzen/Verbessern freier Software animieren
 - ▶ Mehr Blog-Beiträge
 - ▶ Kurse (\LaTeX / Python / **Git** / Inkscape / ...)
 - ▶ Infrastruktur-Stipendium
 - ▶ OpenSource-Wettbewerb/Preis
 - ▶ ...
- ▶ Aufmerksamkeit erzeugen / Lobby-Arbeit („Landesverträge“)
- ▶ Vernetzung mit anderen Städten

Weitere Informationen



<https://fsfw-dresden.de/>

uni-stick

blog

newsletter

mitmachen

fork

ringvorlesung

git-ws



</Eigenwerbung>

Gliederung



Warum Versionsverwaltung?

Warum Git?

Git Einführung (mit Praxis)

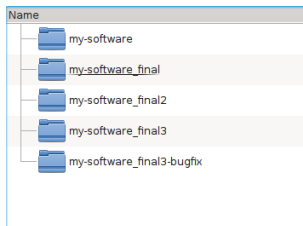
Schlussbemerkungen

Warum Versionsverwaltung?



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

- ▶ Naiver Ansatz:



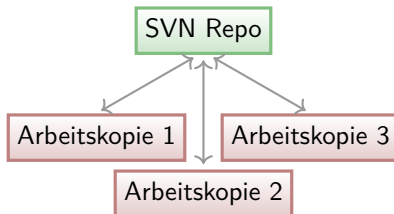
- ▶ Probleme:

- ▶ Speicherplatz
- ▶ Fehlende Übersicht
- ▶ Skaliert nicht (Teamwork)

Warum Git? (1)



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

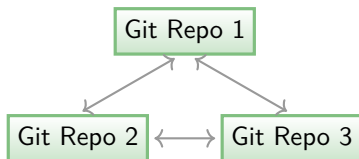


- ▶ Probleme:
 - ▶ Abhängig von Server-Erreichbarkeit
 - ▶ Performanz

Warum Git? (2)



- ▶ Lösung 2: **dezentrale** Versionsverwaltung
 - ▶ mercurial (2005), bazaar (2005) **git** (2005)
 - ▶ Idee: Jeder hat ein vollwertiges Repository

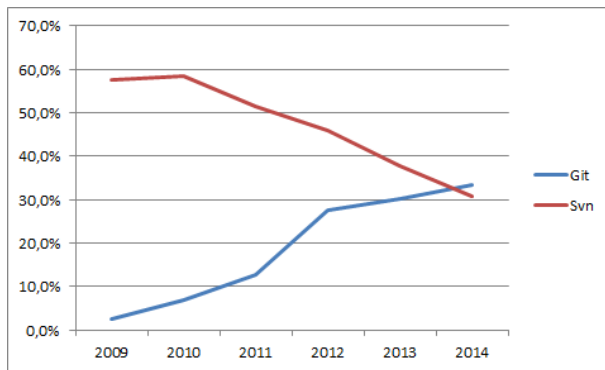


- ▶ Vorteile:
 - ▶ Alle Operationen lokal → schnell, unabhängig
 - ▶ Einfaches „branching“ und „merging“
 - ▶ ...

Warum Git? (3)

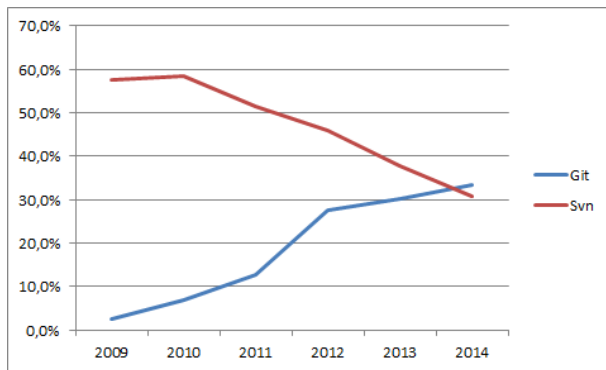


Warum Git? (3)



Gefühlter Grad der Verbreitung: Git vs SVN

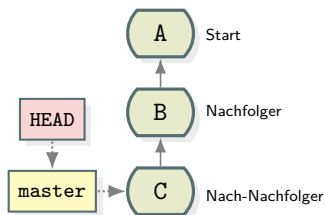
Warum Git? (3)



Gefühlter Grad der Verbreitung: Git vs SVN

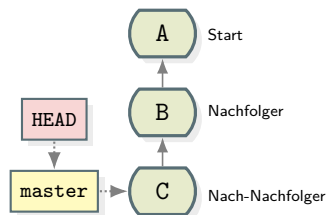
Seit 2016: Git $\hat{=}$ defacto Standard

Einführung in git – Was ist ein Repo?



- ▶ gerichteter, azyklischer Graph von Versionen (*revisions*) 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?

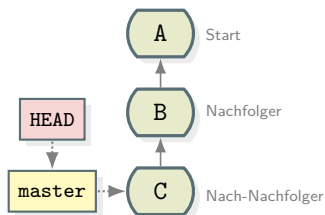


- ▶ gerichteter, azyklischer Graph von Versionen (*revisions*) einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- ▶ Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

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?

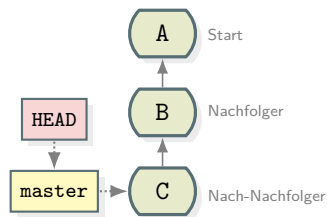


- ▶ gerichteter, azyklischer Graph von Versionen (*revisions*) einer Ordnerstruktur und deren Inhalt mit Metadaten (*Commit-ID*, Autor, Beschreibungstext)
- ▶ Commit-ID abgeleitet aus dem Inhalt und dem Graphen (kryptographische Hash-Funktion)

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?



index

working tree

- ▶ gerichteter, azyklischer Graph von Versionen (*revisions*) 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

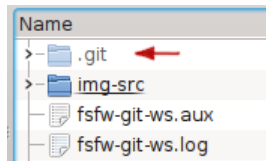


- ▶ Konfiguration anpassen
 - ▶ `git config --global user.email "foo@bar.de"`
 - ▶ `git config --global user.name "Your Name"`
 - ▶ ...
- ▶ Eigenes Repo foo erstellen
 - ▶ `mkdir foo`
 - ▶ `cd foo`
 - ▶ `git init`
- ▶ Alternativ: Bestehendes Repo klonen
 - ▶ `git clone <url>`

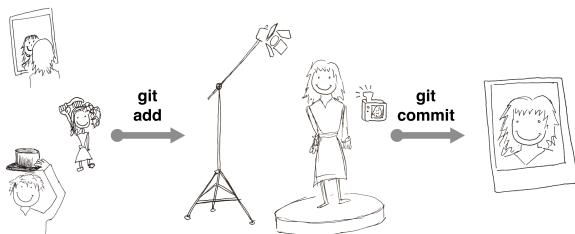
Praxis 1: Erste Schritte



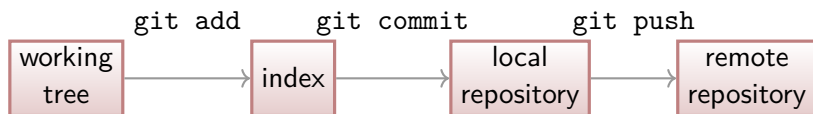
- ▶ Konfiguration anpassen
 - ▶ `git config --global user.email "foo@bar.de"`
 - ▶ `git config --global user.name "Your Name"`
 - ▶ ...
- ▶ Eigenes Repo foo erstellen
 - ▶ `mkdir foo`
 - ▶ `cd foo`
 - ▶ `git init`
- ▶ Alternativ: Bestehendes Repo klonen
 - ▶ `git clone <url>`
- ▶ Hintergrund: Wo speichert git die relevanten Informationen?
→ Verstecktes 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
```

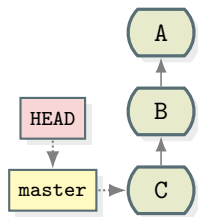
Praxis: minimales Repo



- ▶ 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`

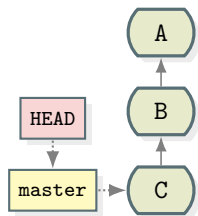
```
14:58 $ git diff
diff --git a/README.md b/README.md
index ee7ae9a..31d5401 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 Hallo
-welt
+Leute
```

Theorie: Branches



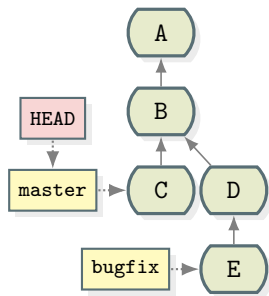
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen

Theorie: Branches



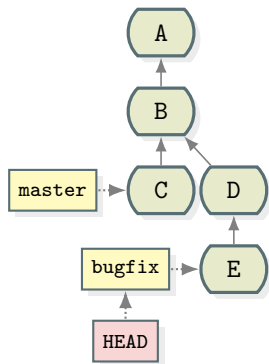
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD

Theorie: Branches



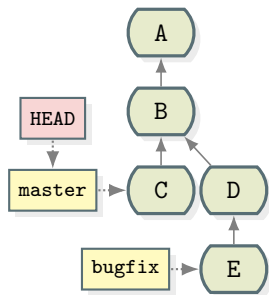
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich

Theorie: Branches



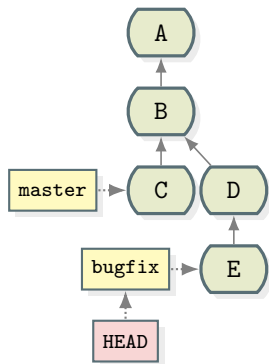
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout bugfix`

Theorie: Branches



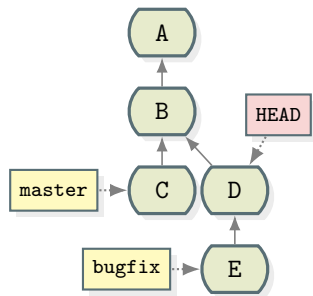
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout master`

Theorie: Branches



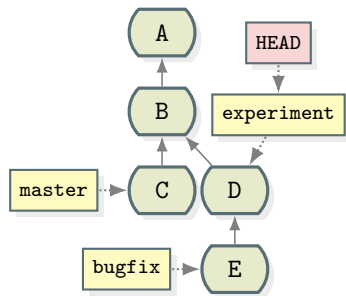
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout bugfix`

Theorie: Branches



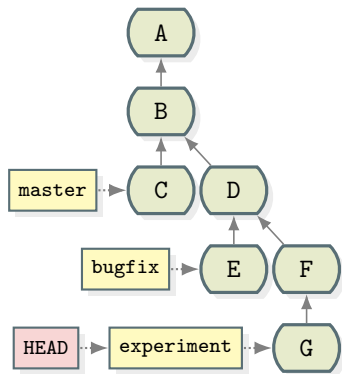
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout <ref>`

Theorie: Branches



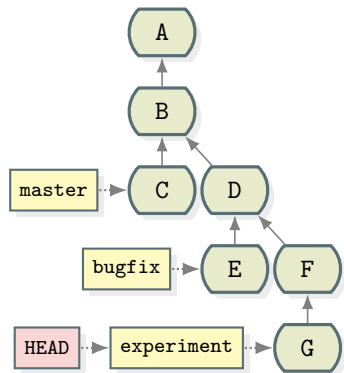
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout <ref>`
- ▶ neuer Branch auf HEAD erstellen:
`git checkout -b experiment`

Theorie: Branches



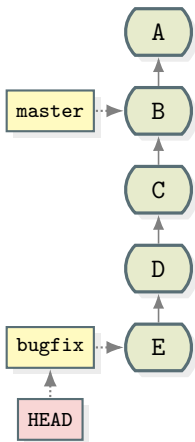
- ▶ Unkompliziertes paralleles Arbeiten an verschiedenen Versionen
- ▶ Der aktive Branch folgt HEAD
- ▶ beliebig viele Branches möglich
- ▶ Branch/Revision wechseln:
`git checkout <ref>`
- ▶ neuer Branch auf HEAD erstellen:
`git checkout -b experiment`

Theorie: Zusammenfassung Branches



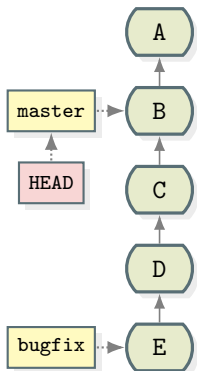
Branches sind *lokale* Lesezeichen auf Knoten im Revisionsgraphen. Beim Anlegen eines neuen Commits folgt der aktive Branch dem neuen HEAD.

Theorie: Mergen – Zusammenführen von Zweigen



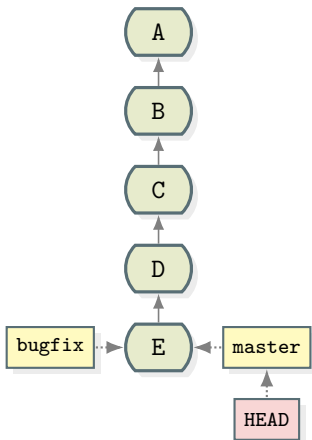
- ▶ Fall 1: Fast-Forward

Theorie: Mergen – Zusammenführen von Zweigen



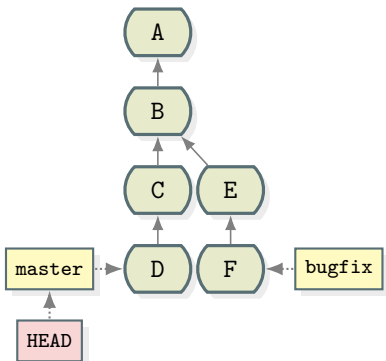
- ▶ Fall 1: Fast-Forward
 - ▶ `git checkout master`

Theorie: Mergen – Zusammenführen von Zweigen



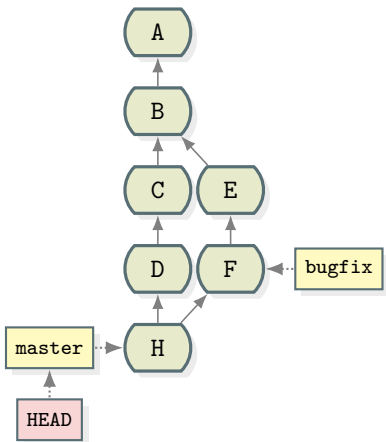
- ▶ Fall 1: Fast-Forward
 - ▶ `git checkout master`
 - ▶ `git merge bugfix`

Theorie: Mergen – Zusammenführen von Zweigen



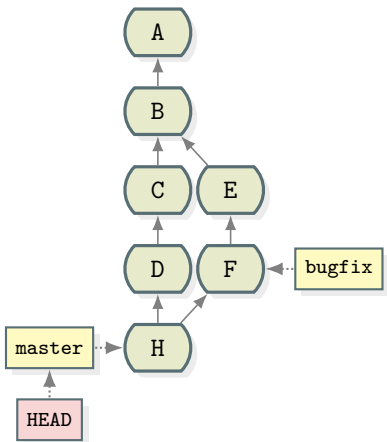
- ▶ Fall 1: Fast-Forward
 - ▶ `git checkout master`
 - ▶ `git merge bugfix`
- ▶ Fall 2: Parallele Zweige
 - ▶ `git checkout master`

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-trivials 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
4. commit-History einzelner Dateien anzeigen
 - a) `git blame AUTHORS.md`
 - b) `git blame sonnets/text1.md`

Praxis (2)



5. Änderungen anzeigen

a) ... seit dem vorletzten Commit: `git diff HEAD~`

→ beliebige Änderungen vornehmen

b) ... seit dem letzten Commit: `git diff`

c) Grafische diff-Anzeige: `diff` durch `difftool` ersetzen
vorher:

```
git config --global diff.tool meld (im PC-Pool schon ✓)
```

```
oder: git config --global diff.tool kdiff3
```

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 Dateien ignorieren

- a) sicherstellen, dass `git status` „working directory clean“ liefert
- b) Skript ausführen: `python3 scripts/find-shortest-poem.py`
- c) `git status` → Hilfsdateien bemerken (`__pycache__`)

Häufig treten lokale Dateien auf, deren Änderungen nicht von git verfolgt werden sollen → Datei `.gitignore` hilft

- d) ignore-Datei ergänzen: `printf "__pycache__" >> .gitignore`
- e) Staus-Änderung zur Kenntnis nehmen: `git status`
- f) Committen: `git commit -am "ignore python-bytecode-dir"`
- g) 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
__pycache__
```

► Bedeutung der Zeilen

7.1 Erklärender Kommentar (stand vorher schon drin)

7.2 Ignoriere alle Dateien in Unterverzeichnissen namens `__pycache__`

Praxis (3)



8. Branch *rom* in *master* mergen

- merge durchführen → Konflikt zur Kenntnis nehmen
- Überblick verschaffen: `git status gitk --all`
- Manuell Konflikt in `AUTHORS.md` beheben
- 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, so dass repo linear wird
 - ▶ Hintergrundwissen:
 - ▶ `git help rebase`
 - ▶ [https://onlywei.github.io/...](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 (Binärdateien verwenden z. B. oft Offsets)
 - ▶ Grund: Delta-Kompression basiert auf zeilenweisen Diffs
- .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 (Binärdateien verwenden z. B. oft Offsets)
 - ▶ Grund: Delta-Kompression basiert auf zeilenweisen Diffs
- .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 (Binärdateien verwenden z. B. oft Offsets)
 - ▶ Grund: Delta-Kompression basiert auf zeilenweisen Diffs

→ `.git`-Verzeichnis wird ggf. sehr groß
- ▶ Nicht behandelte wichtige Konzepte/Kommandos
 - ▶ `git fetch`, `git pull`, `git push`, `git rebase`, ...
 - ▶ Siehe Cheat-Sheet
- ▶ Weitere Tipps:
 - ▶ Status-Infos im Bash-Prompt
 - ▶ Aliase in `.gitconfig` (z.B.: `git co` → `git checkout`)
 - ▶ Globale `gitignore`-Datei anlegen
 - ▶ Bewährtes Branching-Modell anwenden

```
~/git-workshop [feature/ck-folien ↑·5|+ 2..3]
11:02 $ █
```

Schlussbemerkungen (2): Blick über den Tellerrand



- ▶ Umundu-Festival (01. - 06. Okt.)
 - ▶ Festival für nachhaltige Entwicklung
 - ▶ Fokus Thema 2018: „Utopien“
 - ▶ Filme, Vorträge, Workshops, ...
 - ▶ <https://umundu.de/>



Schlussbemerkungen (2): Blick über den Tellerrand



- ▶ Umundu-Festival (01. - 06. Okt.)
 - ▶ Festival für nachhaltige Entwicklung
 - ▶ Fokus Thema 2018: „Utopien“
 - ▶ Filme, Vorträge, Workshops, ...
 - ▶ <https://umundu.de/>

- ▶ Kritische Einführungstage (08. - 18. Okt.)
<https://www.kreta-dresden.org/>
 - ▶ Fahrrad-Stadt-Rundgang
 - ▶ Kritische Wissenschaft
 - ▶ Klimawandel und Gesundheit (Kohle)
 - ▶ ...



Schlussbemerkungen (3)



- ▶ Fragen?

- ▶ **Unterstützung:** (im Rahmen unserer Möglichkeiten)
 - ▶ <https://fsfw-dresden.de/sprechstunde>
 - ▶ <https://fsfw-dresden.de/git-ws>

 - ▶ kontakt@fsfw-dresden.de

Quellen und Links (Auswahl)



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