Versionsverwaltung mit git: Warum und wie.

Amine Othmane

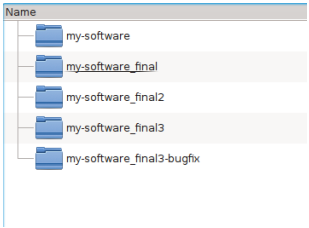
Modellierung und Simulation technischer Systeme (SMS)
Universität des Saarlandes
Saarbrücken, Deutschland

Folieninhalt hauptsächlich aus
https://github.com/fsfw-dresden/git-ws

Warum Versionsverwaltung?

Warum Versionsverwaltung?

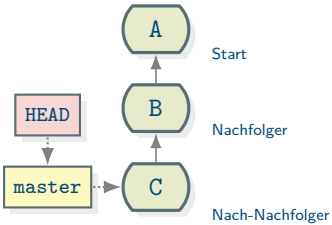
- Projekte bestehen aus schrittweisen Änderungen
- Bedürfnis, zu vorherigem Zustand zurückkehren zu können
  - ("'Savegame"')
- Naiver Ansatz:



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

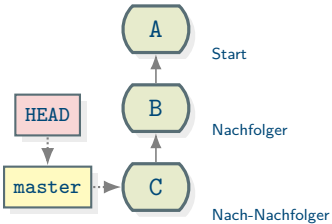
Git Einführung (mit Praxis)

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)

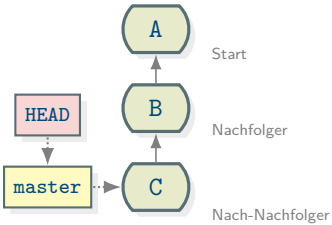
Einführung in git – Was ist ein Repo?



- 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))

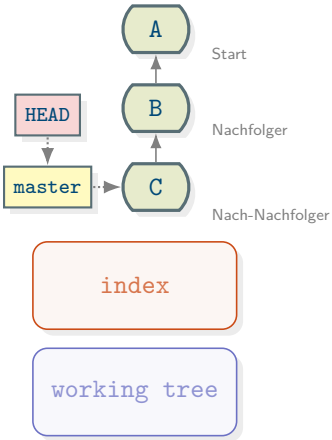
Einführung in git – Was ist ein Repo?



- 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?



- 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

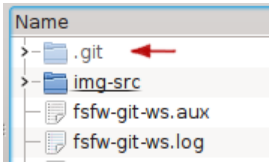
- Konfiguration anpassen
  - `git config --global user.email "foo@bar.de"`
  - `git config --global user.name "Your Name"`
  - ...
- Eigenes Repo `foo` erstellen
  - `mkdir foo`
  - `cd foo`
  - `git init`
- Alternativ: Bestehenedes Repo klonen
  - `git clone <url>`

Praxis 1: Erste Schritte

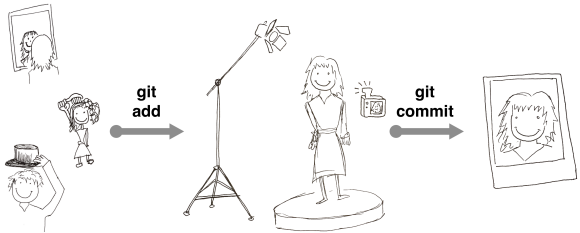
- Konfiguration anpassen
  - `git config --global user.email "foo@bar.de"`
  - `git config --global user.name "Your Name"`
  - ...
- Eigenes Repo `foo` erstellen
  - `mkdir foo`
  - `cd foo`
  - `git init`
- Alternativ: Bestehenedes Repo klonen
  - `git clone <url>`

- Hintergrund: Wo speichert git die relevanten Informationen?

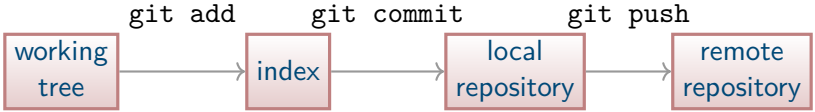
→ Verstecktes Verzeichnis `.git`



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`)

$\Rightarrow$ nachvollziehbare, aussagekräftige Commit-History

Praxis: minimales Repo

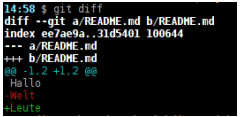
- Inhalt erzeugen
  - `printf "Hallo\nWelt\n" > README.md`
  - `git status`
  - `git add README.md` Tipp: Auto-Vervollständigung mit TAB
  - `git status`
  - `git commit -m "New content of README"`
  - `git status`

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`

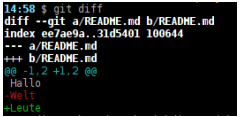
Praxis: minimales Repo

- 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`



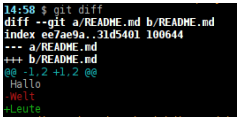
Praxis: minimales Repo

- 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"`

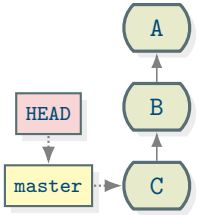


Praxis: minimales Repo

- 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`

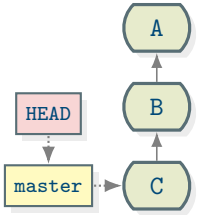


Theorie: Branches



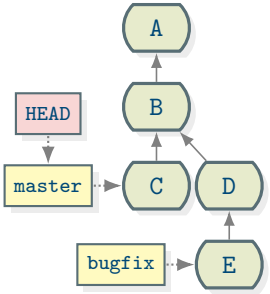
- Unkompliziertes paralleles Arbeiten an verschieden Versionen

Theorie: Branches



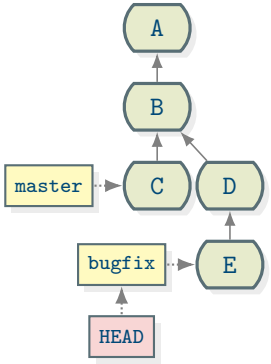
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`

Theorie: Branches



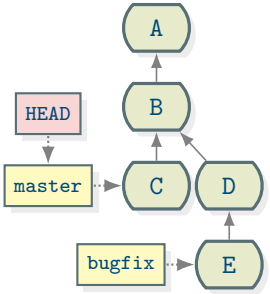
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich

Theorie: Branches



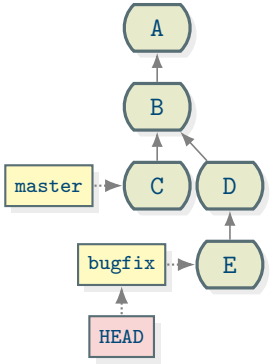
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout bugfix`

Theorie: Branches



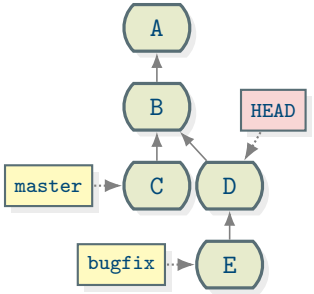
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout master`

Theorie: Branches



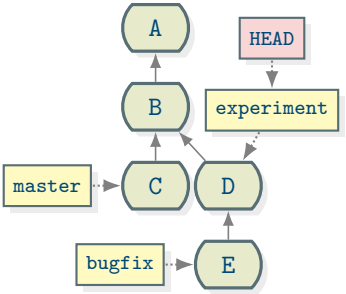
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout bugfix`

Theorie: Branches



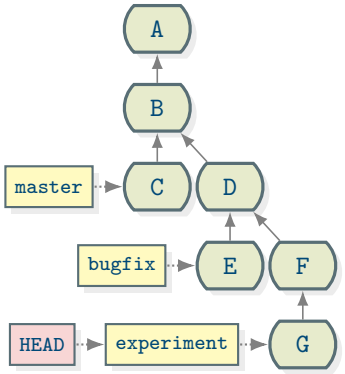
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`

Theorie: Branches



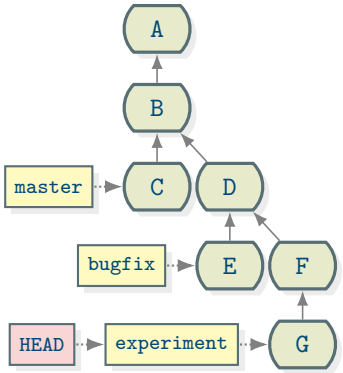
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`
- neuer Branch auf `HEAD` erstellen:
  `git checkout -b experiment`

Theorie: Branches



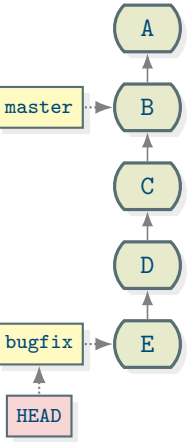
- Unkompliziertes paralleles Arbeiten an verschieden Versionen
- Der aktive Branch folgt `HEAD`
- beliebig viele Branches möglich
- Branch/Revision wechseln:
  `git checkout <ref>`
- neuer Branch auf `HEAD` erstellen:
  `git checkout -b experiment`

Theorie: Zusammenfassung Branches



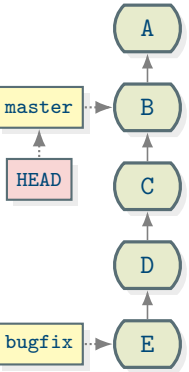
Branches sind *lokale* Lesezeichen auf Knoten im Revisionsgraphen. Beim Anlegen eines neuen Commits folgt der aktive Branch dem neuen `HEAD`.

Theorie: Mergen – Zusammenführen von Zweigen



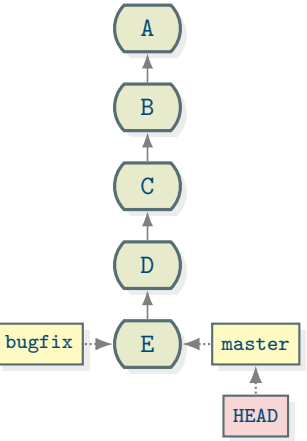
- Fall 1: Fast-Forward

Theorie: Mergen – Zusammenführen von Zweigen



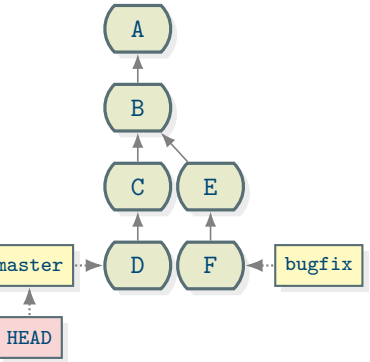
- Fall 1: Fast-Forward
  - `git checkout master`

Theorie: Mergen – Zusammenführen von Zweigen



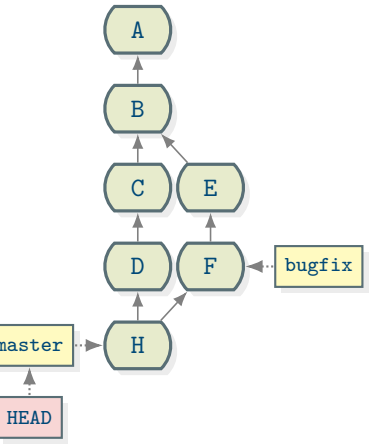
- Fall 1: Fast-Forward
  - `git checkout master`
  - `git merge bugfix`

Theorie: Mergen – Zusammenführen von Zweigen



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

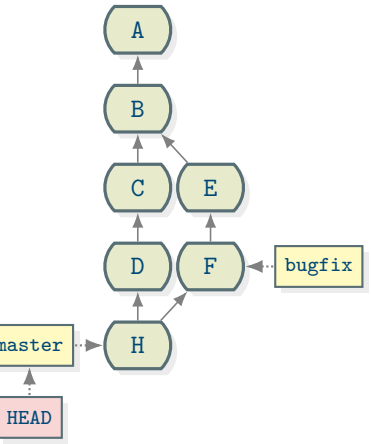
Theorie: Mergen – Zusammenführen von Zweigen



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

  $\Rightarrow$ 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`

Schlussbemerkungen

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`)

Quellen und Links (Auswahl)

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