Git-Schulung

–Firma–

Valentin Hänel, Julius Plenz

–Datum–

Ablaufplan: Tag 1

- **Tag 1, vormittags:** Einführung in Git
  - Ein Repository erstellen
  - Die wichtigsten Git-Kommandos
  - Der Index
  - Git-Interna
- **Tag 1, nachmittags:** Mit Branches arbeiten
  - Branches erstellen und wieder zusammenführen
  - Änderungen rückgängig machen
  - Merge-Konflikte lösen

Ablaufplan: Tag 2

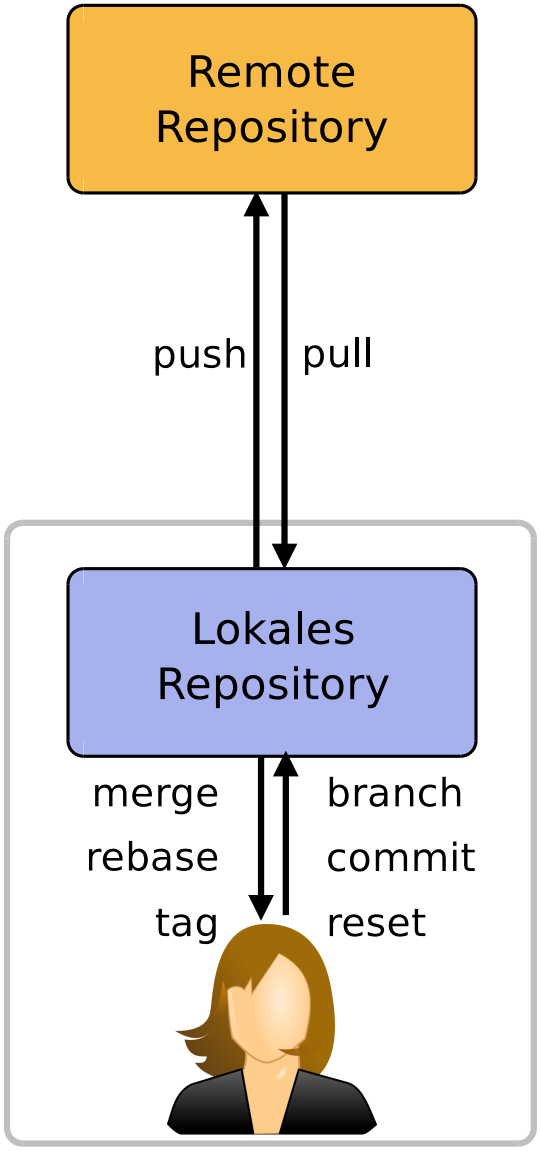
- **Tag 2, vormittags:** Kollaboration
  - Rebase
  - Parallele Entwicklung mit Git
  - Workflows
  - Praktischer Teil: Zusammen ein Projekt erstellen
- **Tag 2, nachmittags:** Erweitertes Git
  - Hilfreiche Git-Kommandos
  - Fehlersuche
  - Automatisierung
  - Evtl. Auffangbecken für noch nicht geklärte Fragen

Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

Autonomie des eigenen Repositories



- *Remote* und lokales Repository sind gleichberechtigt
- Austausch zwischen Repositories via Push/Pull
  - *Push*: Eigene Änderungen hochladen
  - *Pull*: Änderungen herunterladen
- Alle anderen Aktionen passieren zunächst *nur* lokal

Vor- und Nachteile verteilter Versionskontrollsysteme

Vorteile:

- Jeder Entwickler besitzt eine *komplette* Kopie der Versionsgeschichte
  - Kommandos laufen sehr schnell
  - Offline-Arbeit möglich
  - Impliziter Schutz vor Manipulation
- Es gibt keinen »single point of failure«
  - Serverausfall, Hack, wütender Entwickler, ...

Vor- und Nachteile verteilter Versionskontrollsysteme

Vorteile:

- Kein Streit um Commit-Rechte
- Delegation von Aufgaben ist leichter
- Beliebige Workflows

Vor- und Nachteile verteilter Versionskontrollsysteme

Nachteile:

- Viel Freiheit: Policies müssen geschaffen werden
- Komplexeres Setup als zentralisierte Systeme

Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

Begriffsbildung

- **Commit**: Eine Änderung an einer oder mehrerer Dateien, versehen mit Metadaten wie Autor, Datum und Beschreibung
- **Commit-ID**: Jeder **Commit** wird durch eine eindeutige SHA1-Summe identifiziert, seine **ID**
- **Repository**: »Behältnis« für gespeicherte **Commits**
- **Working-Tree**: Arbeitsverzeichnis, die Dateien die man sieht
- **Branch**: Ein »Zweig«, eine Abzweigung im Entwicklungszyklus, z. B. um ein neues Feature einzuführen.
- **Referenz**: Eine Referenz »zeigt« auf einen bestimmten Commit, z. B. ein **Branch**

- **Index/Staging-Area**: Bereich zwischen dem **Working-Tree** und dem **Repository**, in dem Änderungen für den nächsten **Commit** gesammelt werden

Git konfigurieren

- Mit `git config` wird die Konfiguration abgefragt und angepasst
- Grundsätzlich nur für das aktuelle Projekt
  - Wird in `.git/config` gespeichert
- Mit dem Zusatz `--global` für den aktuellen Benutzer
  - Wird in der Datei `~/.gitconfig` gespeichert

Wer bin ich? – Name und E-Mail einstellen

- Bevor wir Git einsetzen, müssen wir uns vorstellen
- Information wird beim Erstellen von Commits verwendet
- Default-Einstellung sind `$USER` und `hostname`

Allgemein für den Benutzer

```
git config --global user.name "Max Mustermann"
git config --global user.email max@mustermann.com
```

... alternativ nur für das aktuelle Projekt

```
git config user.email maintainer@cool-project.org
```

Achtung! *Kein* = aber "" verwenden!

```
git config --global user.name = Max Mustermann
```

Übung: Git konfigurieren

1. Setzen Sie mit `git config` Benutzernamen und E-Mail-Adresse
2. Fragen Sie die Einstellungen auf der Kommandozeile ab
3. Schauen Sie in `~/.gitconfig` nach, welche Auswirkungen Ihre Kommandos hatten

Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

Ein Projekt importieren oder erstellen

Ein neues Projekt erstellen

```
git init projekt
```

Um ein bestehendes Projekt zu importieren, »klont« man es mit seiner gesamten Versionsgeschichte

```
git clone git://gitschulung.de/projekt
```

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`

Übung: Repository aufsetzen

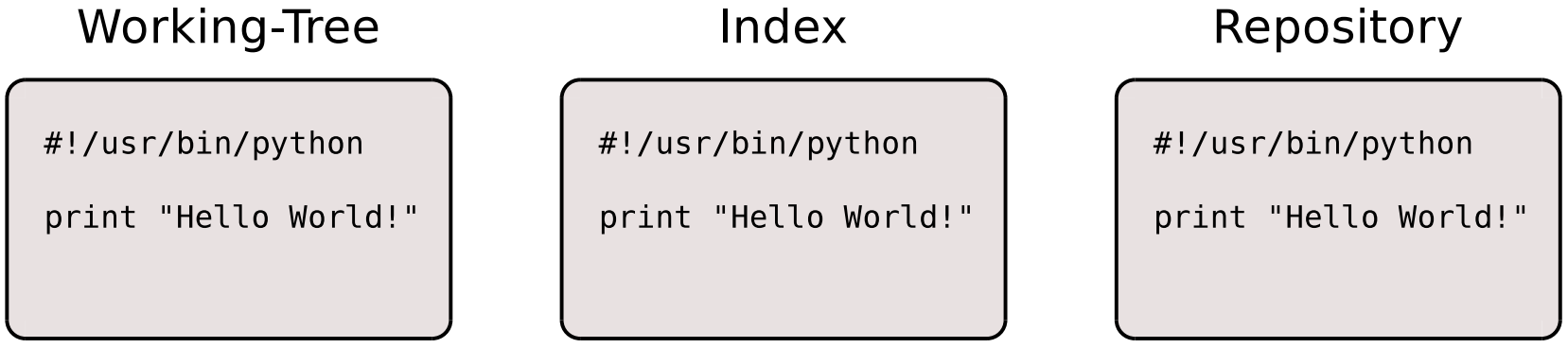
1. Verwenden Sie `git init`, um ein leeres Repository aufzusetzten
2. Erstellen Sie zwei Dateien inklusive Inhalt, und betrachten Sie die Ausgabe von `git status`
3. Fügen Sie per `git add` diese Dateien dem Index hinzu
4. Benutzen Sie `git commit`, um Ihren ersten Commit zu erstellen
5. Betrachten Sie erneut die Ausgabe von `git status`

Index / Staging Area

- Im *Index/Staging-Area* werden Veränderungen für den nächsten Commit vorgemerkt
- So kann der Inhalt von einem Commit schrittweise aus einzelnen Veränderungen zusammengestellt werden
- Nach einem Commit enthält der Index genau die Versionen der Dateien wie in dem Commit

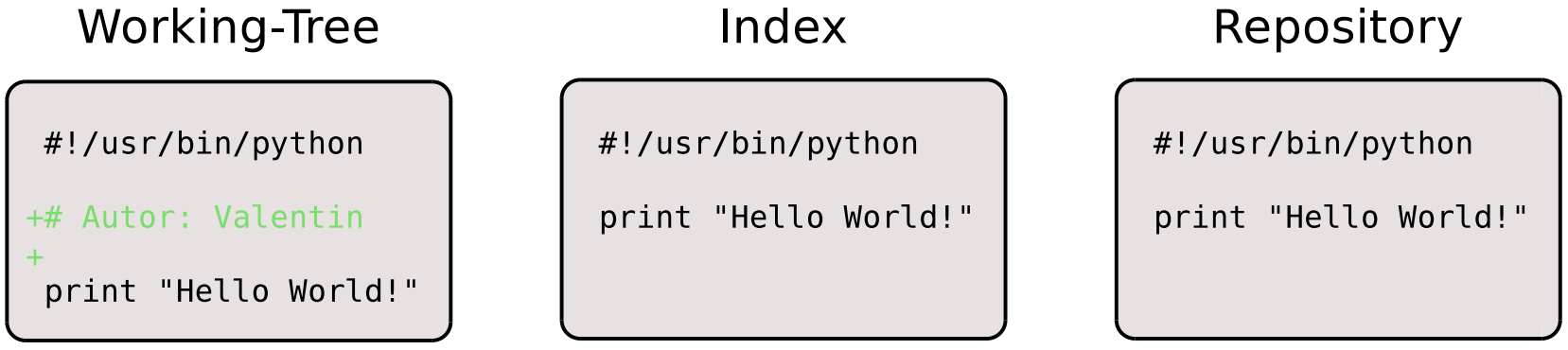
Ausgangsstellung

- Alle auf dem gleichen Stand



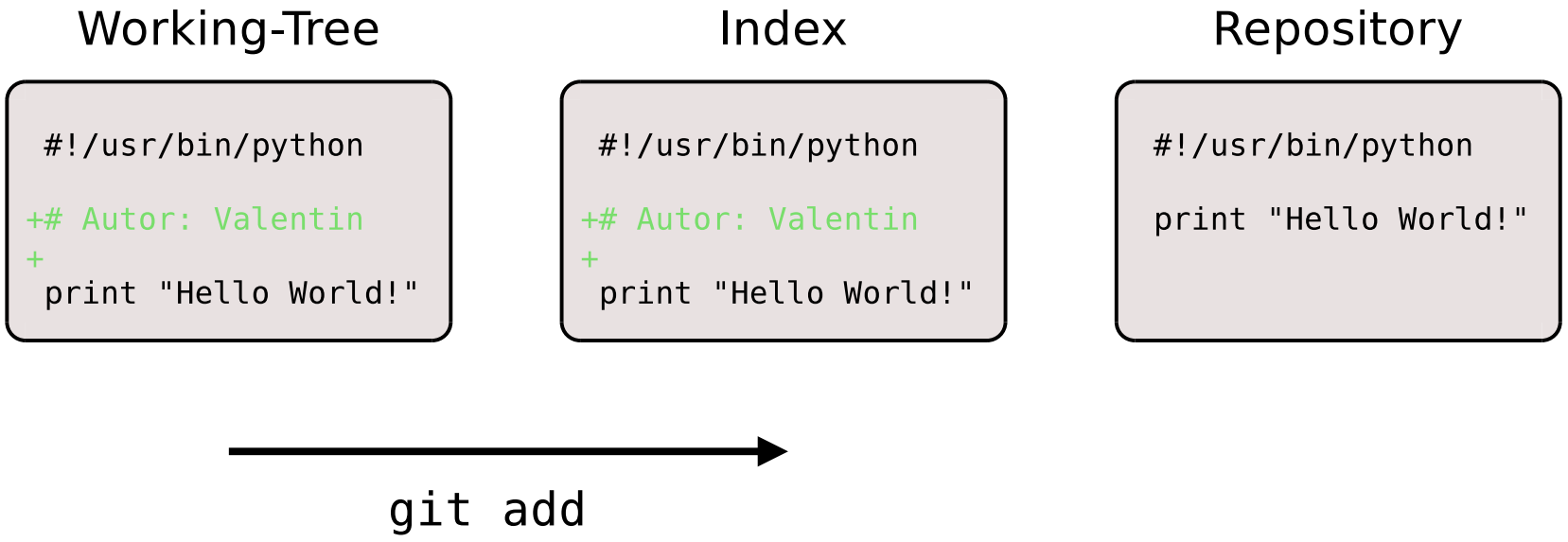
Veränderungen machen

- Veränderungen werden im Working-Tree gemacht



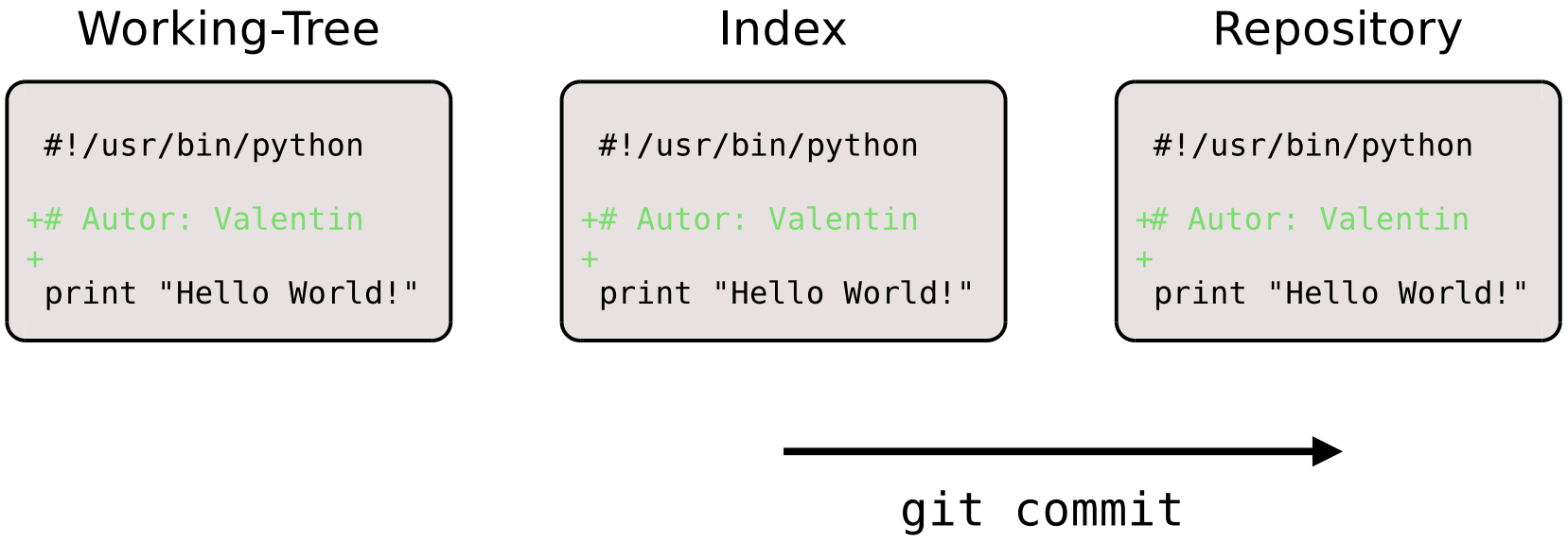
Dem Index hinzufügen – `git add`

- 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: Valentin

print "Hello World!"
```

Index

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

Repository

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

HEAD

HEAD (mehr oder weniger)

Der neuste Commit in der Versionsgeschichte wird als `HEAD` bezeichnet.

`git status` – Wie ist der Zustand?

Status abfragen

```
git status
```

- Welche Dateien wurden modifiziert?
- Welche Veränderungen sind schon im Index?
- Gibt es Git nicht bekannte Dateien? (*untracked files*)

Dateien dem Index hinzufügen

Alle Veränderungen in einer Datei hinzufügen

```
git add datei
```

Interaktives Hinzufügen

```
git add -p
```

Interaktives Hinzufügen nur für eine Datei

```
git add -p datei
```

Index: Unterschiede und Zurücksetzen

Unterschiede zwischen Working-Tree und Index

```
git diff
```

Unterschiede zwischen Index und `HEAD`

```
git diff --staged
git diff --cached
```

Index zurücksetzen

```
git reset
```

Übung: Machen Sie sich mit dem Index vertraut

1. Erstellen Sie eine weitere Datei, und betrachten Sie anschließend die Ausgabe von `git status`
2. Verändern Sie beide Dateien, am besten sowohl am Anfang als auch am Ende der jeweiligen Datei
3. Sehen Sie sich die Ausgabe von `git diff` an
4. Probieren Sie `git add -p` aus!
5. Sehen Sie sich die Ausgabe von `git diff --staged` an
6. Bringen Sie den Index mit `git reset` in seine Ausgangsstellung zurück
7. Betrachten Sie nun die Ausgabe von `git status`

`git commit` – Commits erzeugen

- Das meistgebrauchte Kommando!
- Veränderungen im Index werden zu einem *Commit* zusammengefasst

Commit erzeugen

```
git commit
```

Commit-Nachricht direkt angeben

```
git commit -m "message"
```

Alle Veränderungen im Working-Tree

```
git commit -a
```

Fortgeschrittene Nutzung

Den jüngsten Commit verbessern

```
git commit --amend
```

Leeren Commit erzeugen

```
git commit --allow-empty
```

Autor anpassen

```
git commit --author="Maxine Mustermann \
    maxine@mustermann.de"
```

Mit der Zeile `Signed-off-by:`

```
git commit -s
```

Übung: Commits erstellen

1. Erstellen Sie mit den Dateien ein paar Commits
2. Machen Sie sich hierbei mit `git commit -m 'message'` und `git commit -a` vertraut
3. Versuchen Sie, einen Commit im Nachhinein zu modifizieren, benutzen Sie hierfür `git commit --amend`
4. Erstellen Sie einen leeren Commit mit `git commit --allow-empty`

Commit-Message

- Die erste Zeile der Commit-Message sollte maximal 50 Zeichen lang sein
- Kurz, prägnant formulieren – aber trotzdem informativ!
- Beschreiben, *warum* etwas geändert wurde!
  - *Was* geändert wurde, sieht man an der `diff`-Ausgabe

Beispiel

```
commit 95ad6d2de1f762f20edb52d139d3cc19529a581a
Author: Matthieu Moy <Matthieu.Moy@imag.fr>
Date:   Fri Sep 24 18:43:59 2010 +0200

update comment and documentation for :/foo syntax

The documentation in revisions.txt did not match the
implementation, and the comment in sha1_name.c was
incomplete.
```

Die Versionsgeschichte anzeigen

- `git log` listet alle Commits auf, den neusten zuerst

Versionsgeschichte seit Anfang

```
git log
```

Versionsgeschichte mit Patches

```
git log -p
```

Nur den aktuellen Commit

```
git log -1
```

Als Einzeiler

```
git log --oneline
```

Übung: Schauen Sie sich die Geschichte an

1. Betrachten Sie, was Sie bisher gemacht haben mit `git log`
2. Machen Sie einen neuen Commit, und vergleichen Sie die Ausgabe von `git log`
3. Betrachten Sie nur die n neusten Commits mit `git log -`n
4. Passen Sie die Ausgabe von `git log` an, verwenden Sie:
   - `git log --oneline`
   - `git log --stat`
   - `git log -p`

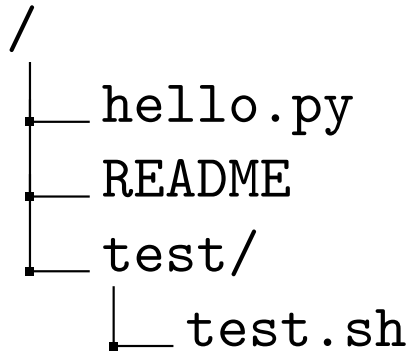
Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

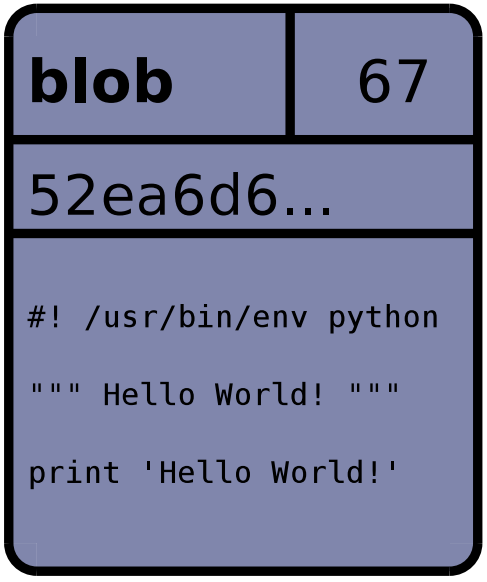
Was wollen wir speichern?

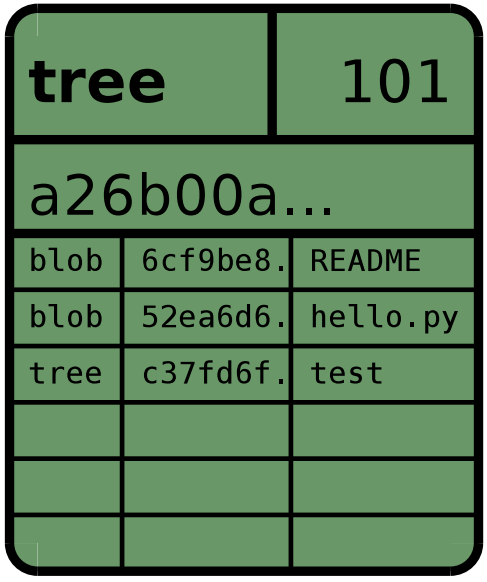
Angenommen, wir wollen folgendes Verzeichnis speichern:



Objektmodell

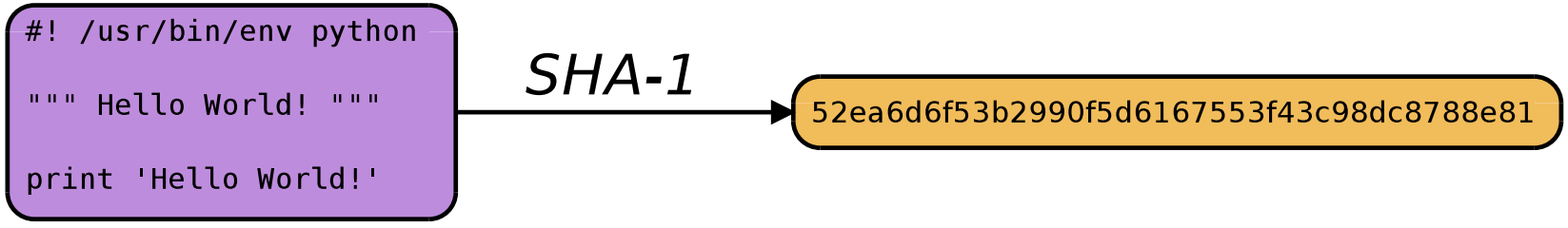
- *Blob*: Enthält den Inhalt einer Datei
- *Tree*: Eine Sammlung von Tree- und Blob-Objekten
- *Commit*: Besteht aus einer Referenz auf einen Tree mit zusätzlichen Informationen
  - *Author* und *Commiter*
  - *Parents*
  - *Commit-Message*





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 (≈ 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



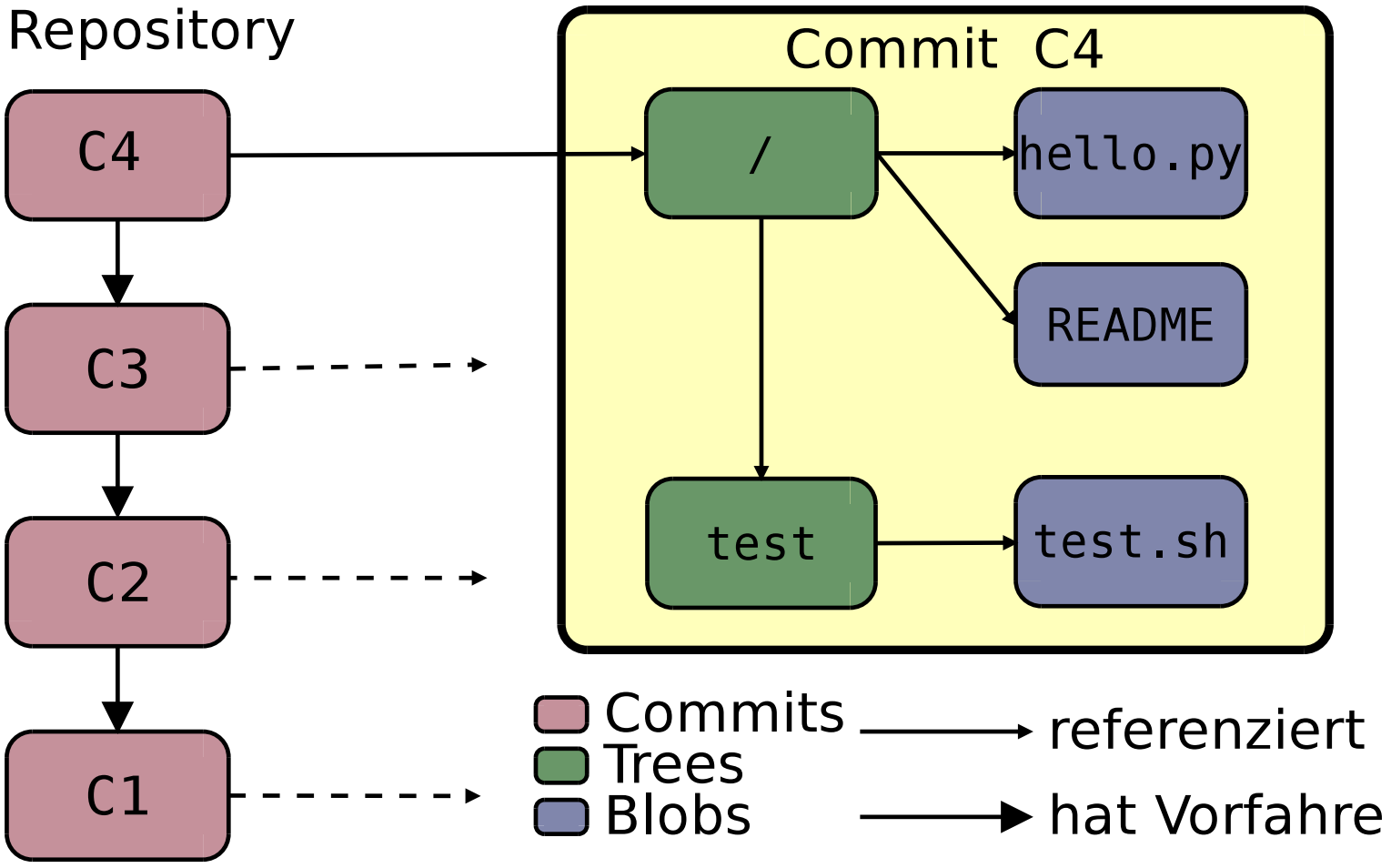
Objektverwaltung

- 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 (SHA-1-IDs) auf die in dem Verzeichnis enthaltenen Dateien

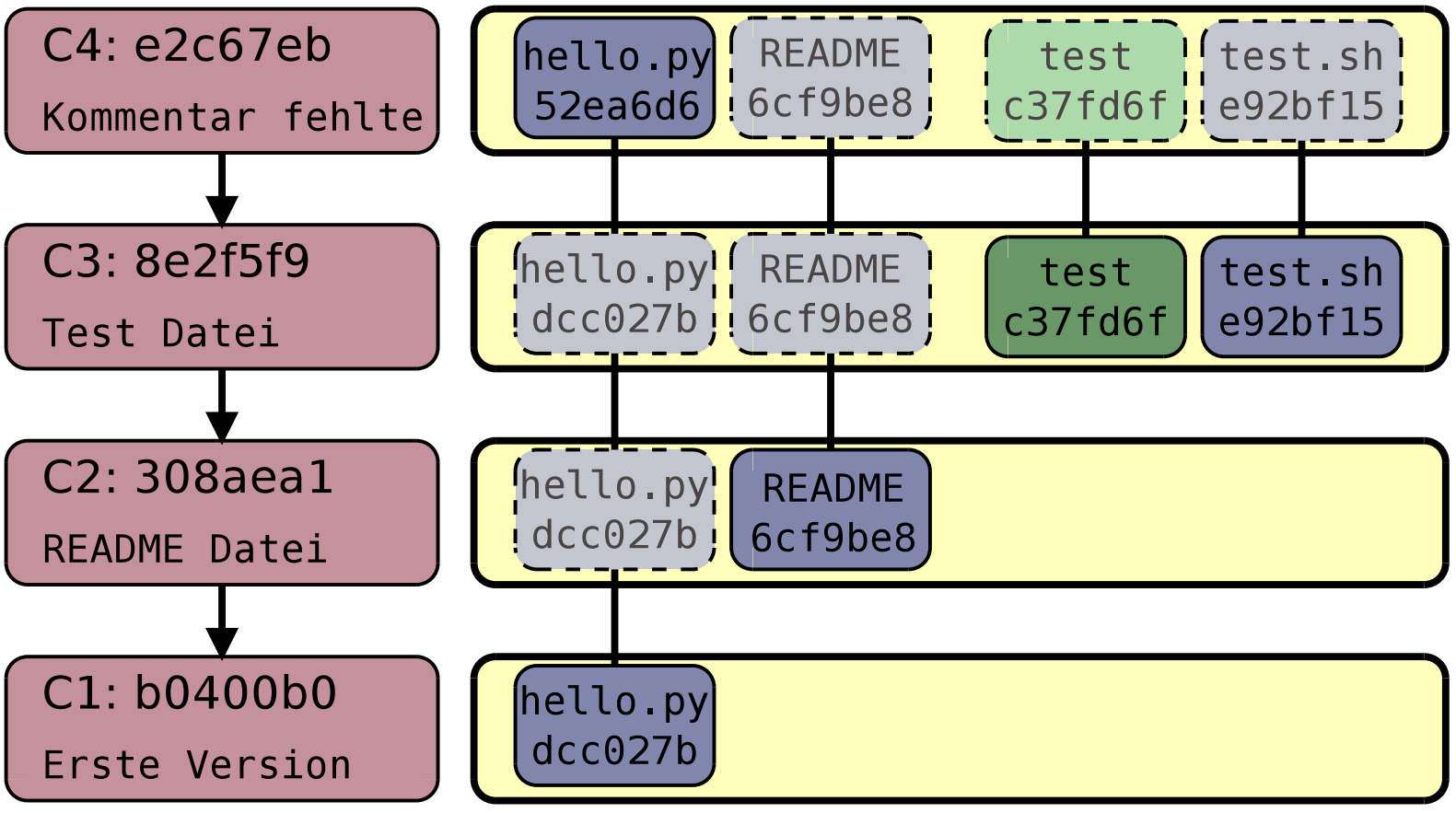
Zusammenfassung

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



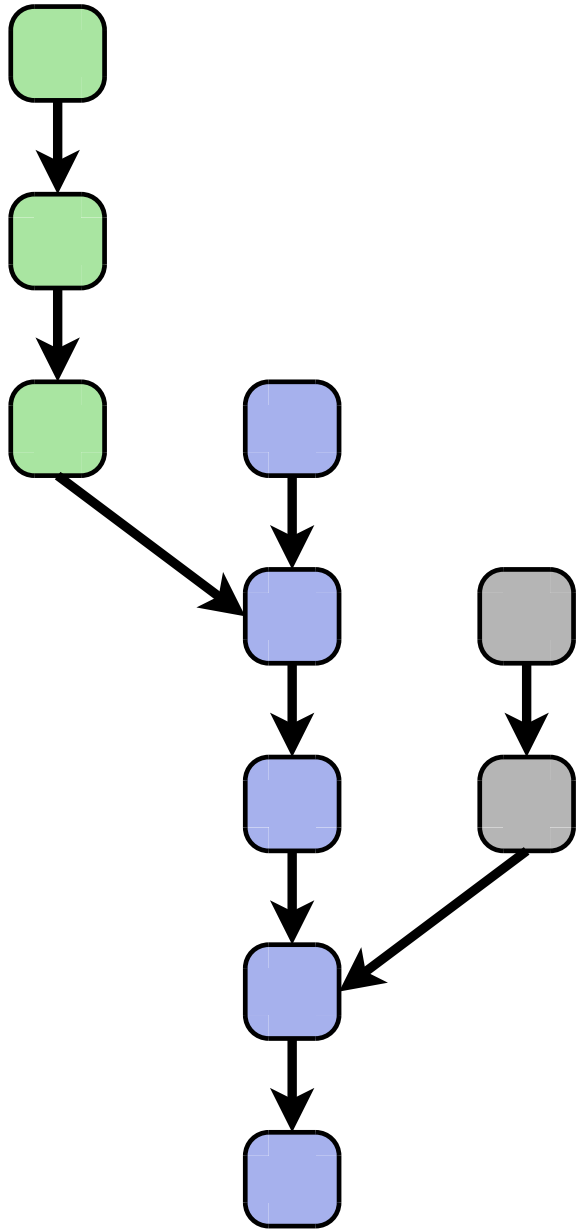
Commit = Dateibaum

Ein Commit hält den Zustand *jeder* Datei zum gegebenen Zeitpunkt fest. (Auch den der nicht geänderten.)



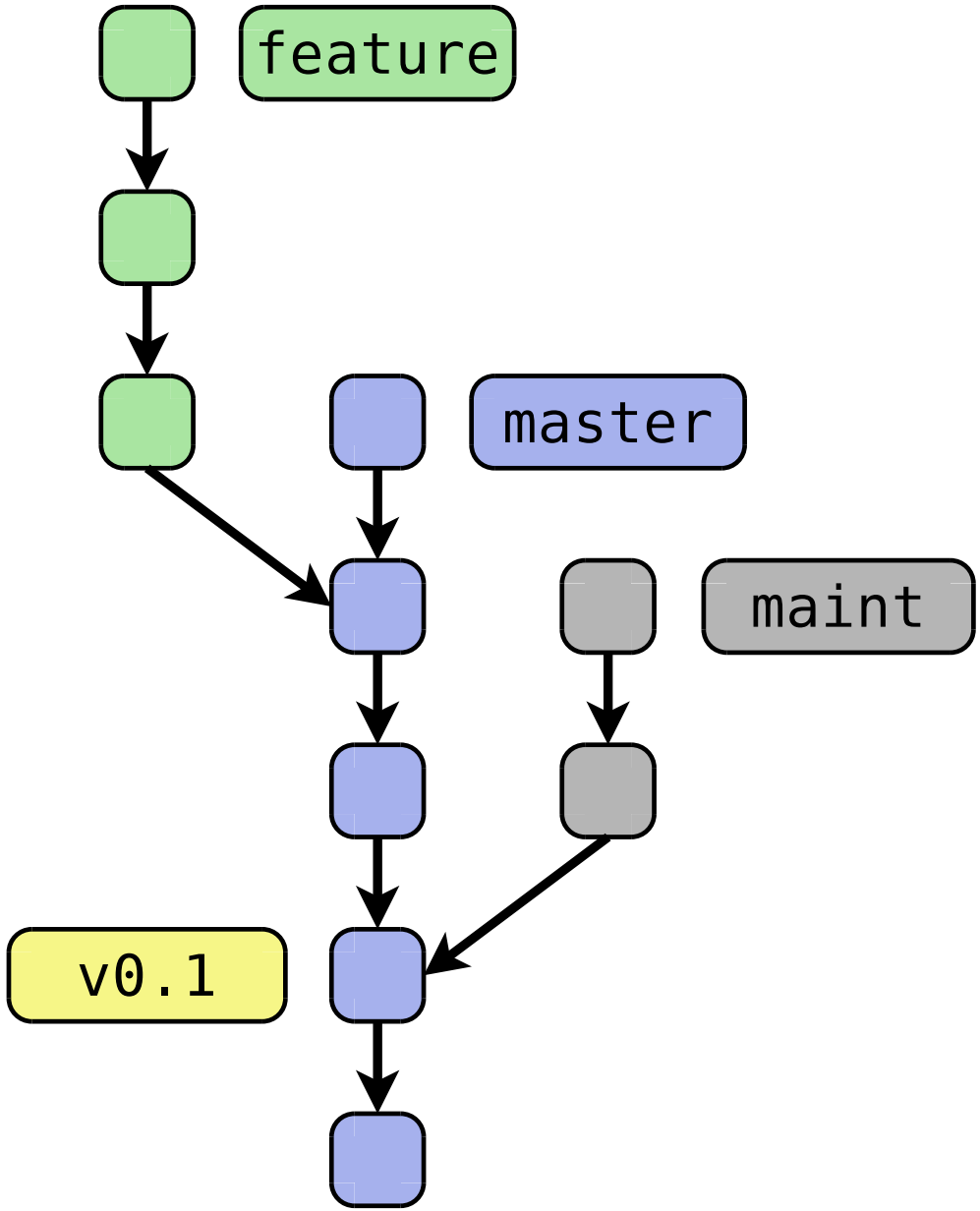
Commit Graph

Ein Repository ist ein *Gerichteter Azyklischer Graph* (DAG)



Branches und Tags

Branches und Tags sind Zeiger



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

Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

Dateien entfernen

- Dateien und Verzeichnisse wenn möglich immer mit Git-Befehlen entfernen
- Dateien können nur entfernt werden, wenn sie keine aktuellen Veränderungen enthalten

Entfernen

```
git rm datei
```

Rekursiv entfernen

```
git rm -r verzeichnis
```

Nur aus dem Index löschen

Aus dem Index entfernen, aber Working-Tree nicht verändern

```
git rm --cached datei
```

- Wenn die Datei schon regulär per `rm` gelöscht wurde
- Wenn die Working-Tree-Kopie erhalten bleiben soll

Dateien verschieben und umbenennen

Dateien innerhalb eines Git-Repositories immer mit Git-Befehlen verschieben/umbenennen.

Verschieben

```
git mv datei verzeichnis
```

Umbenennen

```
git mv datei-alt datei-neu
```

Erzwingen, auch wenn das Ziel bereits existiert

```
git mv -f datei-alt datei-neu
```

So tun als ob: »dry run«

```
git mv -n
```

Übung: Umbenennen und löschen

1. Verschieben Sie eine Datei mit `git mv` und schauen Sie die Veränderung per `git status` an
2. Verschieben Sie die Datei mit `mv`, und führen Sie anschließend ein `git add` mit dem alten und neuen Dateinamen aus
3. Fügen Sie *dem Index* eine neue Datei hinzu, und löschen Sie sie dann wieder (aber nur aus dem Index!)

Übersicht

Session 1: Einführung

- Grundlegendes zu verteilter Versionskontrolle
- Starten
- Git Basics
- Git Interna
- Repository verwalten
- Einführung Branches

Branching einfach gemacht

- Branches funktionieren in Git schnell und intuitiv
  - Revolutioniert die Arbeitsweise (Workflow)
- Ein Branch ist *keine* komplette Kopie des Projektes
  - »Branching is cheap«
- Vorstellung eher: Ein »Label«, das man an einen Commit heftet

Branches auflisten

```
git branch [-v]
```

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

Branches erstellen

- Keine Dateien werden kopiert
  - Erstellung dauert wenige Millisekunden

Einen Branch erstellen

```
git branch name
```

Ausgangscommit des Branches explizit angeben

```
git branch name start
```

Branches wechseln

- Um auf einem Branch zu arbeiten, wird er »ausgecheckt«

Branch auschecken

```
git checkout branch
```

- Für SVN-Umsteiger
  - Das aktuelle Verzeichnis wird *nicht* gewechselt
  - Stattdessen: Inhalt des Branches → Working-Tree

Branch erstellen und auschecken

```
git checkout -b name
```

Branches manipulieren

Branch umbenennen

`git branch -m` *alt neu*

`git branch -M` *alt neu* (forciert)

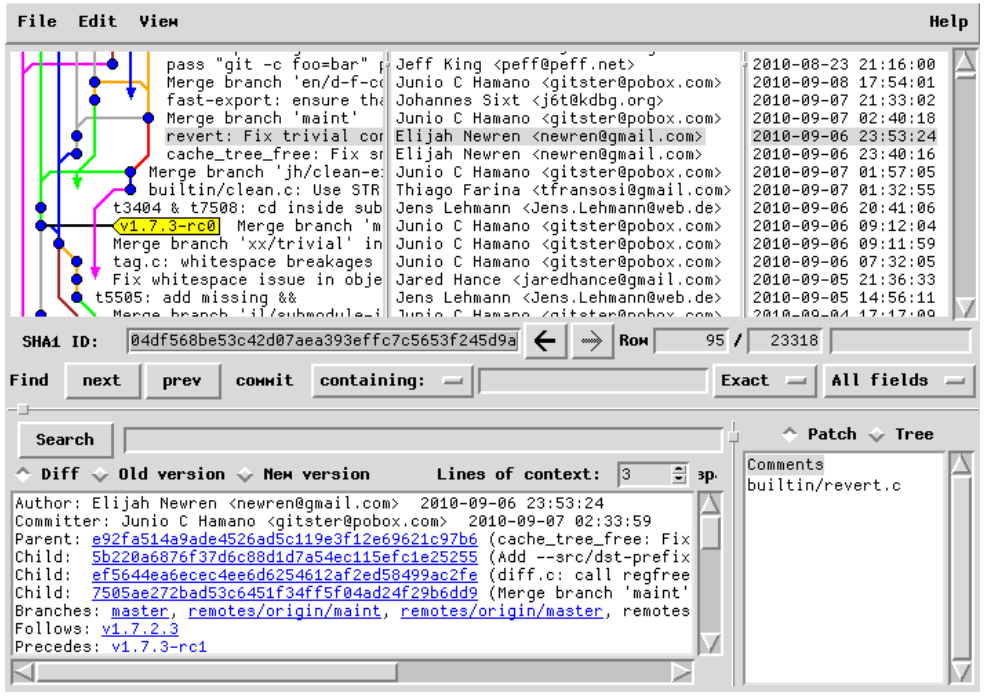
Branch löschen

`git branch -d` *name*

`git branch -D` *name* (forciert)

- Die forcierte Version (`-M` bzw. `-D`) ist dann notwendig, wenn Commits oder Branches überschrieben werden

gitk: Das Repository untersuchen – grafisch



`gitk --all`

Tags

- Tags markieren wichtige Commits in der Entwicklungsgeschichte von einem Projekt, z. B. Releases.

Lightweight Tags

Nur eine Referenz auf einen Commit

Annotated Tags

Enthalten zusätzliche Informationen (Autor, Datum) eine Message, und können digital signiert werden. (Diese Tags sind der bevorzugte Weg)

Tag-Befehle

Anzeigen aller Tags

```
git tag
```

Lightweight Tag erzeugen

```
git tag v1.0
```

Annotated Tag erzeugen

```
git tag -a v1.0 -m "tag message"
```

Tag löschen

```
git tag -d v1.0
```

Übung: Branches und Tags erstellen und löschen

1. Erstellen Sie zwei Branches, die vom gleichen Commit ausgehen
2. Erstellen Sie Commits auf beiden Branches
3. Schauen Sie sich das Ergebnis mit folgendem Kommando an: `git log --oneline --graph --decorate --all`
4. Rufen Sie `gitk --all` auf
5. Listen Sie alle vorhandenen Branches mit `git branch -av` auf
6. Erstellen Sie ein *Annotated Tag* `v1.0`

Git Stash

- `git stash` lagert aktuelle Veränderungen aus
- Zustand von Index und Working-Tree werden gespeichert
- Git erzeugt keinen Commit

Stashes anlegen

Veränderungen auslagern

```
git stash
```

Optional mit Beschreibung

```
git stash save "message"
```

- Sonst wird folgende Nachricht gesetzt:
  - `WIP on branch: sha1 commit-message`

Stashes verwalten

- Es können beliebig viele *Stashes* angelegt werden
- Stashes haben die folgenden Referenzen:
  - `stash@{0}`
  - `stash@{1}`
  - `stash@{2}`
  - `...`

Stashes anzeigen

```
git stash list
```

Inhalt vom jüngsten Stash ansehen

```
git stash show
```

Stash mit Patch anzeigen

```
git stash show -p stash@{3}
```

Stashes verwenden

- Ein Stash kann jederzeit verwendet werden
- Erzeugt das jedoch Konflikte, siehe → Konfliktlösung bei Merges

Den jüngsten Stash verwenden

```
git stash apply
```

Jüngsten Stash löschen

```
git stash drop
```

Verwenden und auch gleich löschen (bevorzugte Möglichkeit)

```
git stash pop
```

Alle Stashes löschen

```
git stash clear
```