Git verstehen und nutzen

Stephan Beyer

11. Chemnitzer Linux-Tage

14. März 2009 (π-Tag)

Teil I

Einleitung

Was ist Git?

. . . ein Versionsverwaltungssystem, oder

- „**S**oftware **C**onfiguration **M**anagement“ (Software-Engineering, Rational ClearCase, BitKeeper)
- „**S**ource **C**ode **M**anagement“ (Darcs)
- „**S**ource **C**ontrol **M**anagement“ (Wikipedia)
- „**R**evision **C**ontrol **S**ystem“ (RCS, GNU arch)
- „**S**ource **C**ode **C**ontrol **S**ystem“ (SCCS)
- „**V**ersion **C**ontrol **S**ystem“ (CVS, Subversion, Mercurial, GNU Bazaar, Monotone)
- „**G**lobal **I**nformation **T**racker“ (Linus Torvalds)

Was ist Git?

Versionsverwaltung – wozu?

Problem

Daten ändern sich im Laufe der Zeit.

Diese Änderungen sollen festgehalten werden.
Hilfreich: *Wer* hat *wann was warum* geändert?

Daten in

- Dateien
- Projekten
- abstrakt: Objekten

bspw.

- Programmcode
- Dokumentation
- Konfigurationsdateien
- Binärdateien

Was ist Git?

Versionsverwaltung – wozu?

Fähigkeiten Versionsverwaltung nach Eric S. Raymond

Verwaltung von Daten mit

- *Umkehrbarkeit* – Zurückholen bekannter, gespeicherter Zustände, z. B. bei Fehlern oder schlechten Ideen
- *Nebenläufigkeit* – Fähigkeit, dass viele Personen auch gleichzeitig denselben Code ändern dürfen, ohne dass Änderungen einer Person einfach verloren gehen
- *Kommentierung* – Änderungen können kommentiert werden mit Intention, Gründen, Umsetzung, Hinweisen, ...

Einschub: zwei grobe Begriffe

Projekt: Gesamtheit aller Daten (jetzt: Verzeichnisse mit Dateien), die verwaltet werden sollen

Repository: dt. *Lager*, *Depot*, *Repositorium*; kurz *Repo*; ein von einer Versionsverwaltung verwaltetes Projekt

Was ist Git?

. . . ein verteiltes oder dezentrales Versionsverwaltungssystem (DVCS):

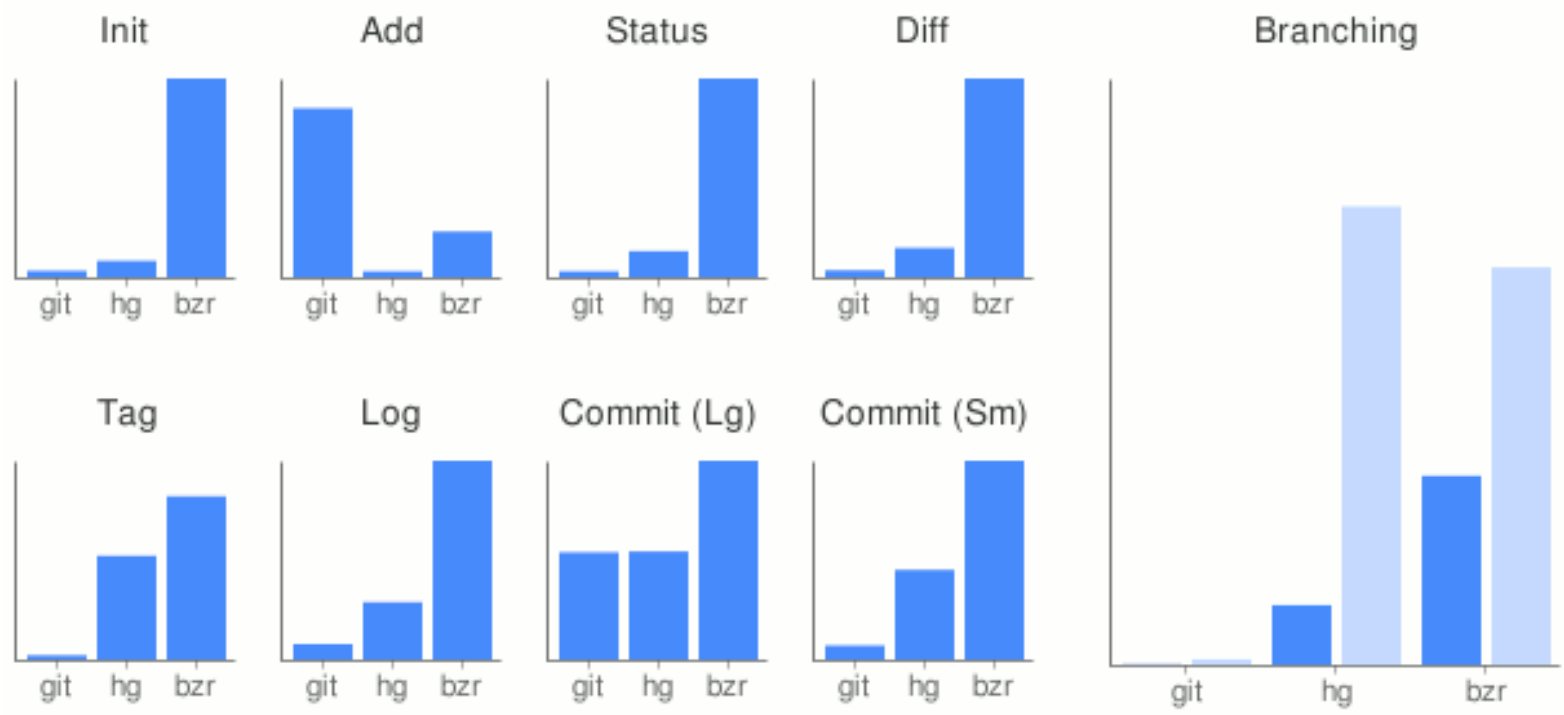
- Jeder Nutzer hat eigenes Repo eines Projektes
- mit der gesamten Geschichte.
- Jedes Repo kann einen Zustand haben, der sich von den anderen unterscheidet.

Neben Git zählen auch Arch, Bazaar, BitKeeper, Darcs, Mercurial, Monotone zu den DVCS.

Warum Git?

- einfach zu bedienen (wenn Konzepte verstanden)
- schnell
- mittlerweile große Nutzerbasis und Community
  - schnelle Entwicklung
  - viel Support (auch Blogosphere)
  - Google: *Results 1 - 20 of about 138,000,000 for git*
  - Sachzwang: ein Projekt nutzt Git
- flexibel und mächtig
- stellt sich nicht in den Weg des Anwenders
- viele nützliche Kleinigkeiten
- Linus schrieb's

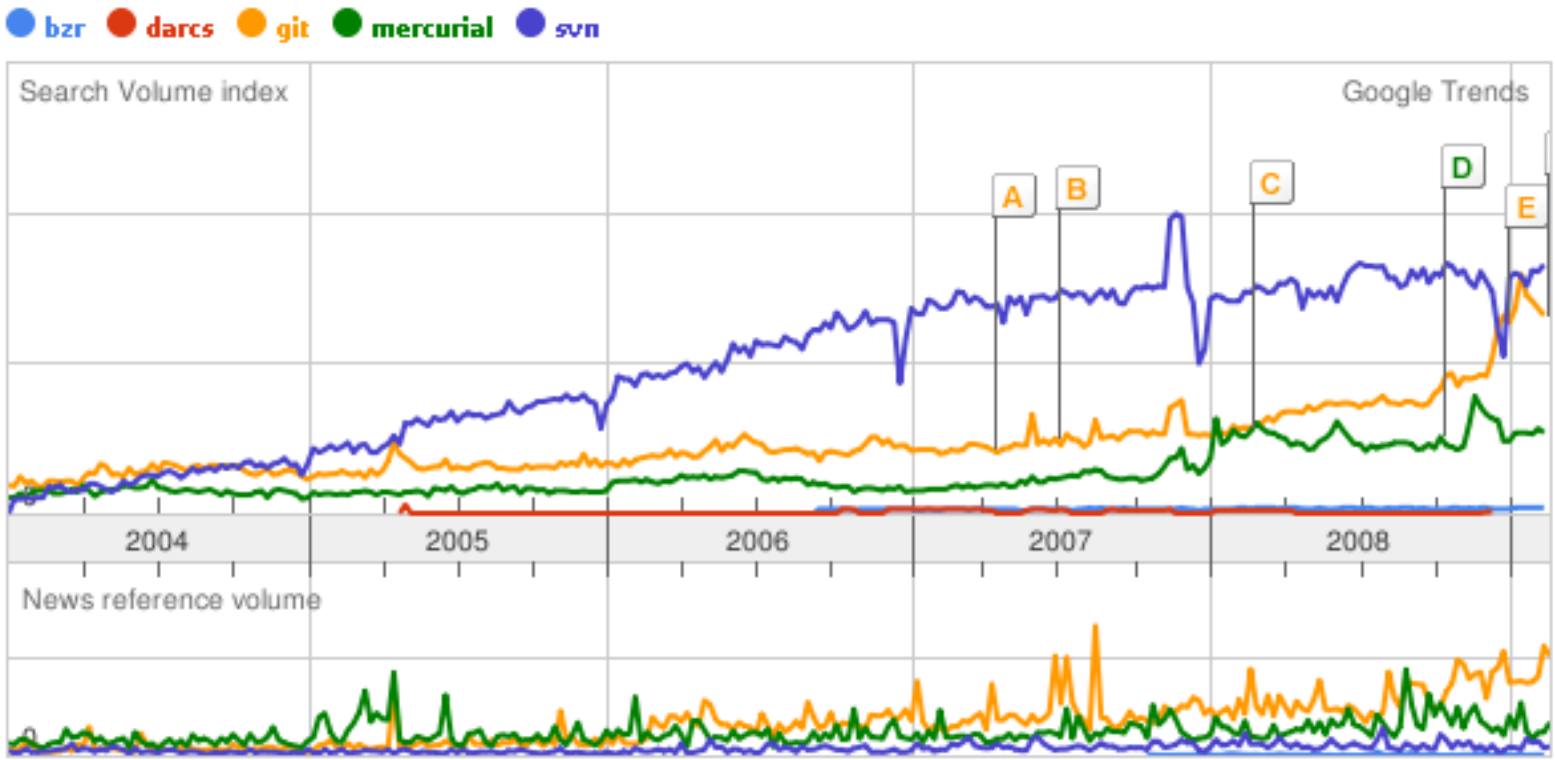
Zeitverhalten



Daten- und Bildquelle: Why Git is Better Than X → *Git is fast*
Plot von Google Charts API

Abbildung: verschiedene DVCS-Operationen auf Django-Repo (roh: ≈19M)

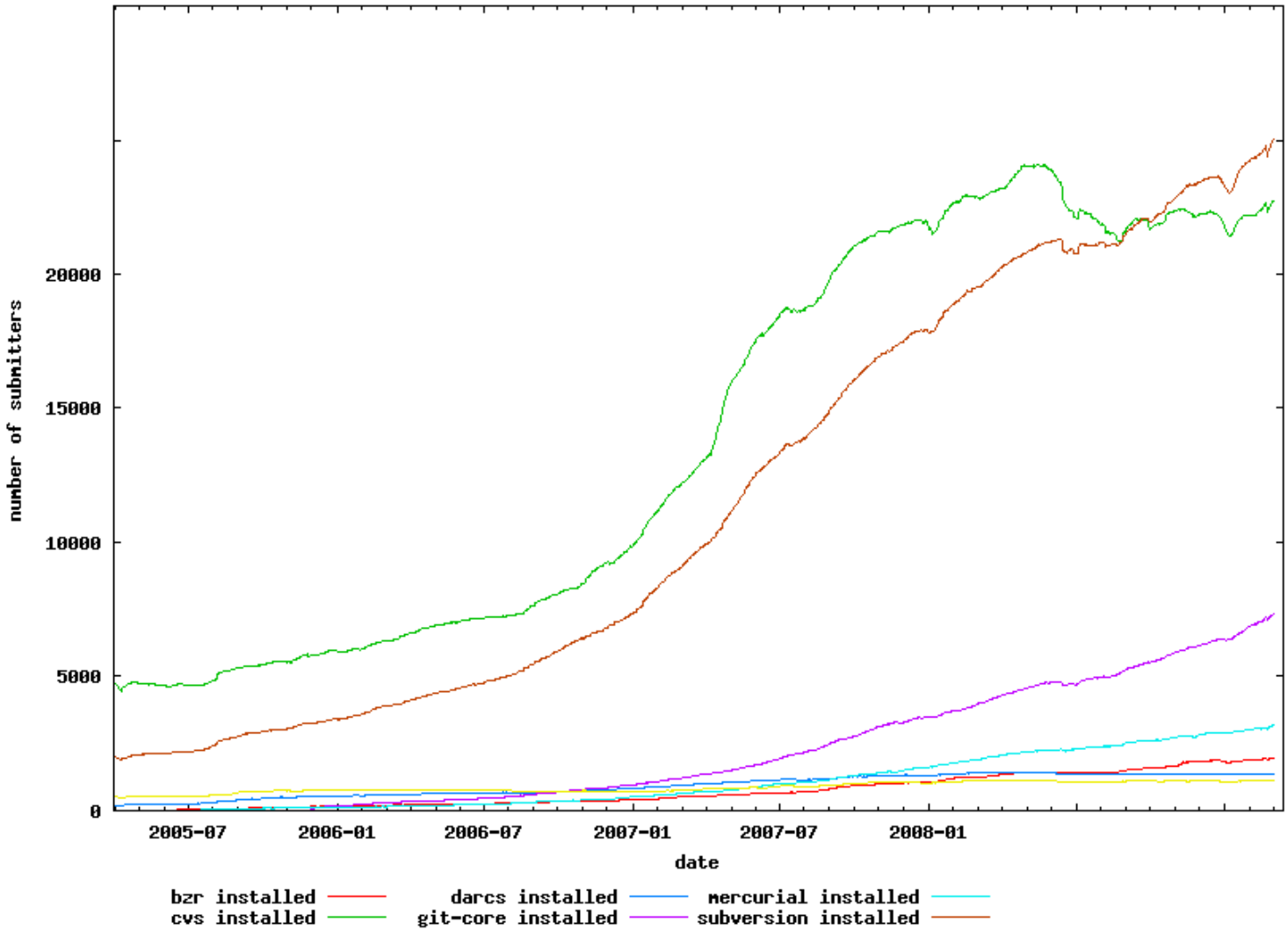
Nutzerentwicklung



Quelle: trends.google.com, 4.3.2009

Abbildung: Google Trends, Suchanfragen

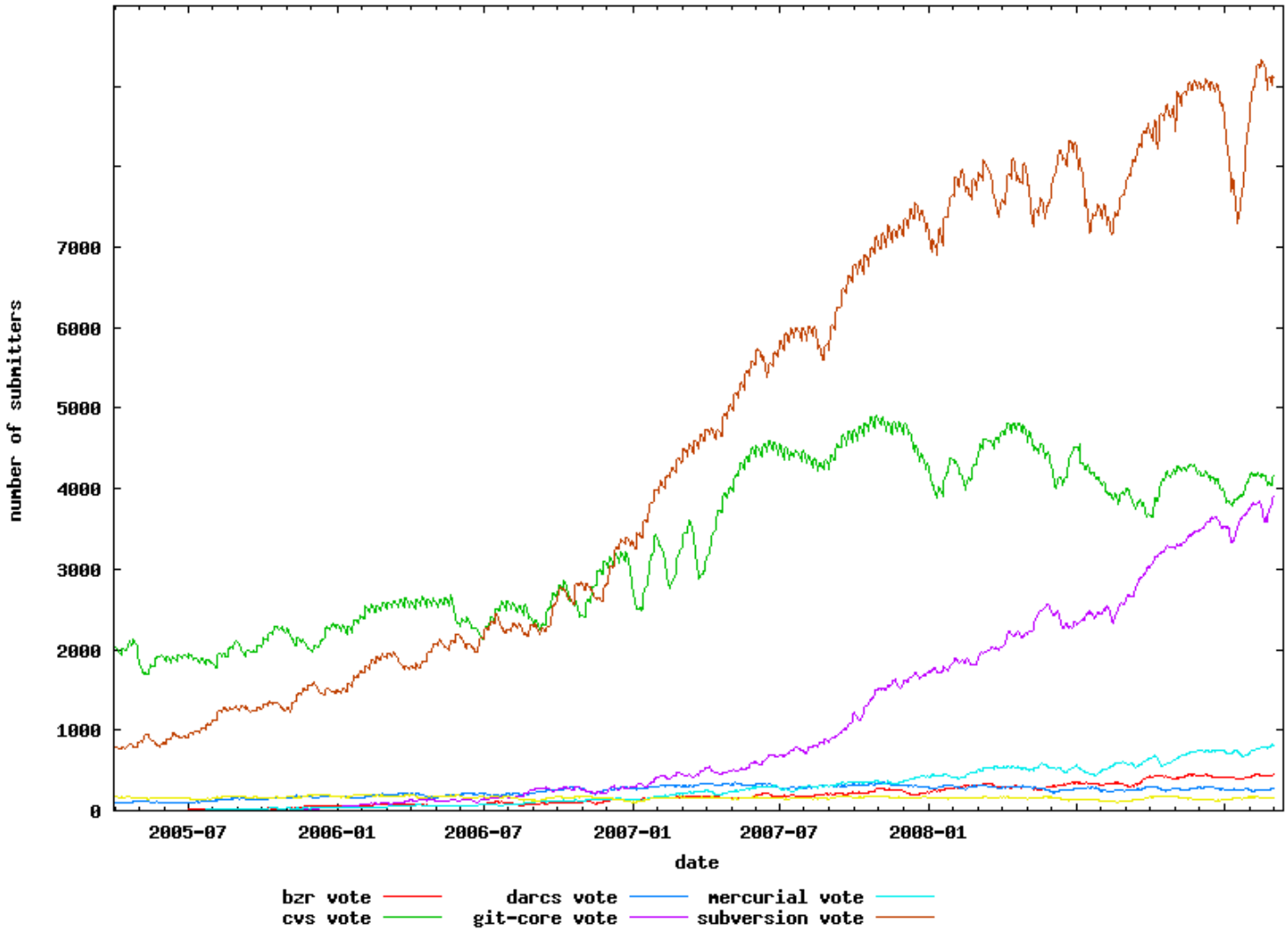
Nutzerentwicklung



Quelle: qa.debian.org popcon statistics (inst), 4.3.2009

Abbildung: Debian Popularity Contest, Installationen von (D)VCS

Nutzerentwicklung



Quelle: qa.debian.org popcon statistics (vote), 4.3.2009

Abbildung: Debian Popularity Contest, reguläre Nutzung von (D)VCS

Wer nutzt Git?

- ALSA
- Android
- awesome
- bucardo
- buildd-tools
- Cairo
- centerim
- Cinelerra-CV
- cobbler
- Compiz Fusion
- dash
- DBus
- DirectFB
- ELinks
- Erlware
- Fedora
- fluxbox
- fontconfig
- fprint
- Frugalware
- gEDA
- GHC
- Git
- GNU autoconf
- GNU automake
- GNU coreutils
- GNU Libtool
- GNU TLS
- grml
- HAL
- LilyPond
- Linux
- Merb
- Mesa3D
- netconf
- OLPC
- OpenVZ
- OpenXPKI
- pacman
- pbuilder
- Perl 5
- Phoronix Test Suite
- Poppler
- Prototype
- Rails
- rox
- Rubinius
- Samba
- Slash
- swfdec
- SWI-Prolog
- SysLinux
- The Mana World
- Thousand Parsec
- uade
- U-Boot
- VLC
- VLE
- Wexus
- WINE
- XMMS2
- X.org
- xwax
- yum
- ZeroInstall
- ...diese Folien

Teil II

Grundlegende Konzepte

Konzepte

Wichtig für Grundverständnis:

- wichtige Objekte: Blobs, Trees, Commits, ...
- Referenzen (Refs): Branches, (leichtgewichtige) Tags, ...
- der Index

Objekte in Git

Git als inhaltsadressiertes Dateisystem

Idee:

- jedes Objekt hat eindeutigen Namen
- Name = SHA-1 des Objekts
  - SHA-1 berechnet sich aus Inhalt des Objekts
  - Länge 160 Bit (20 Bytes)

  $\Rightarrow$ Hexadezimalstring aus 40 Zeichen $\in \{0, \ldots, 9, a, \ldots, f\}$

Konsequenz:

- schneller Zugriff in Objektdatenbank nach Namen
- gleiche Objekte haben gleichen Namen
- Änderungen am Objekt ändern dessen Namen
- schneller Vergleich von Objekten
- implizite Integritätsprüfung
- maximal $256^{20} > 10^{48}$ Objekte

Blob-Objekt

- Daten bzw. Inhalt von Dateien

```
Hello world
```

802992c4220de19a90767f3000a79a31b98d0df7

```
#!/bin/sh
for i in "$@"
do
    convert "$i" -scale 0.5 "small/$i"
done
```

97d9fed61674aa80332e4d6596b875c66099044d

e69de29bb2d1d6434b8b29ae775ad8c2e48c5391

Tree-Objekt

- gibt die Datei- und Verzeichnisstruktur wieder
- Gebilde aus Blobs und Trees
- mit Attributen wie Dateinamen und -rechten, Symlinks usw.
- Dateirechte: eingeschränkt auf Unterscheidung *ausführbar* / *nicht-ausführbar*

```
802992c: 100644 hello
e69de29: 100644 empty-file
```

a6097b449283b490eb4147ffbf3f920ddbbfdf21

```
e69de29: 100644 empty
97d9fed: 100755 small.sh
a6097b4: 040000 small
```

4b15187b78c70255e53857fead9181bbcc22405d

Verzeichnisstruktur:

- `empty`
- `small.sh`
- `small/`
  - `hello`
  - `empty-file`

Commit-Objekt

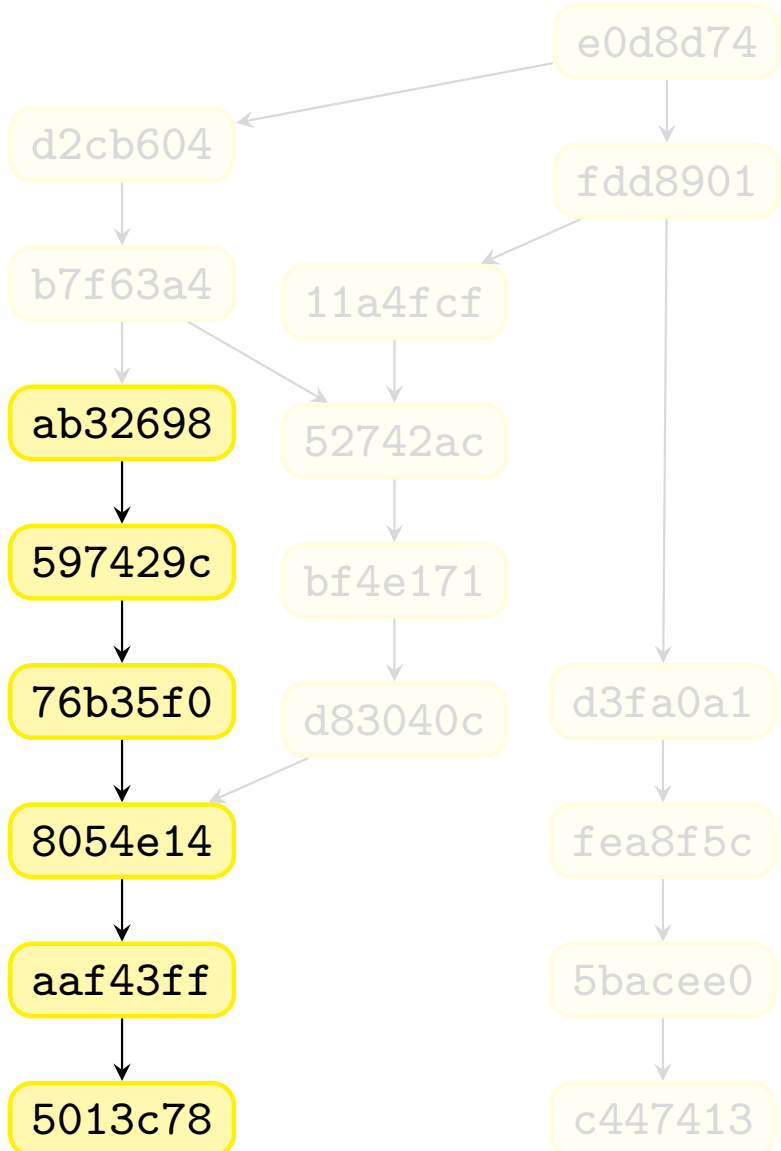
besteht aus

- Committer (Ersteller des Commits)
- Autor
- Zeit
- Parent(s), d. h. Eltern-Commit(s) → Verweis auf Geschichte
- Tree-Objekt
- Zusammenfassung der Änderungen; idealerweise:
  - erste Zeile fasst Änderungen *kurz* zusammen
  - dann leere Zeile
  - ausführliche Zusammenfassung
  - ggf. spezielle Angaben wir `Signed-off-by:` am Ende

d. h. ein Commit beschreibt, wer wann was warum geändert hat.

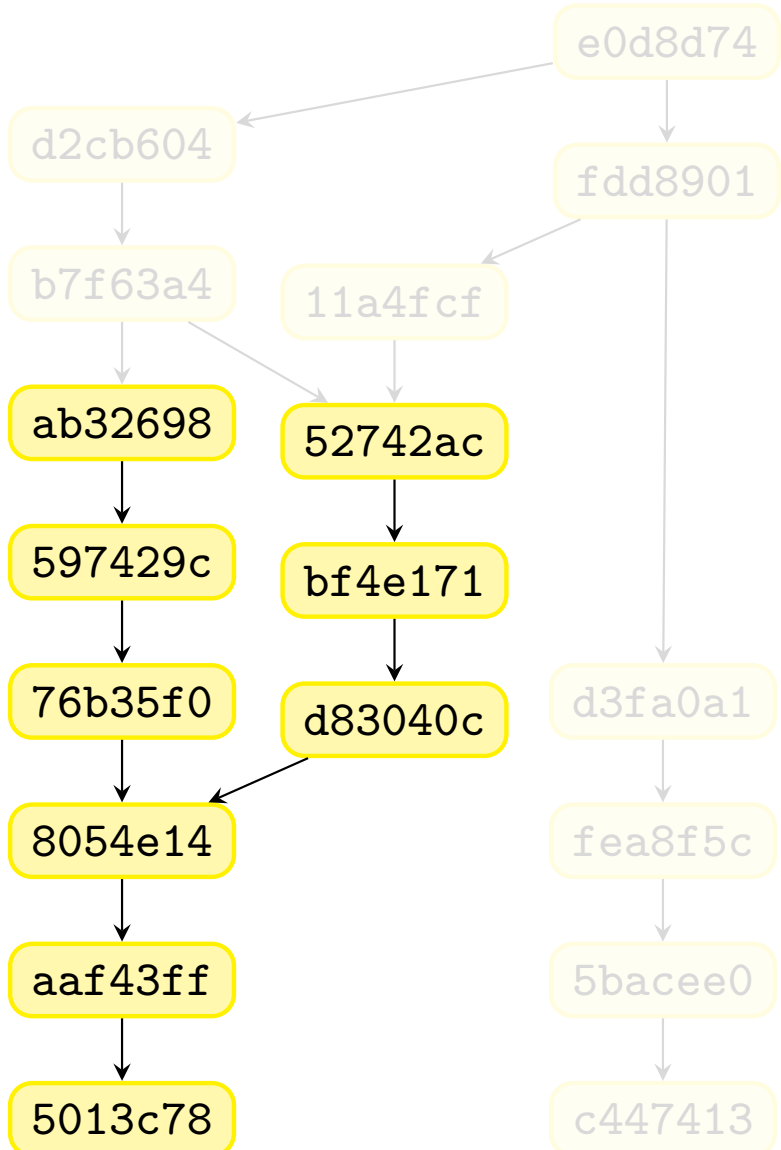
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



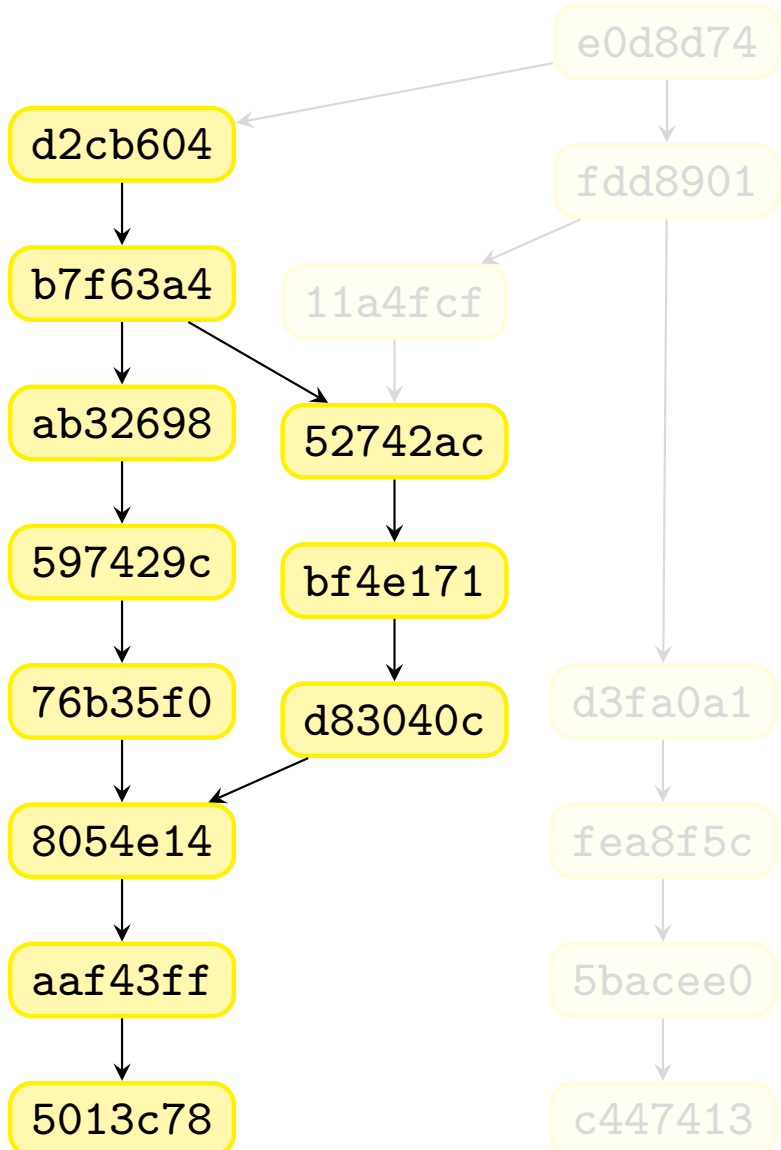
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



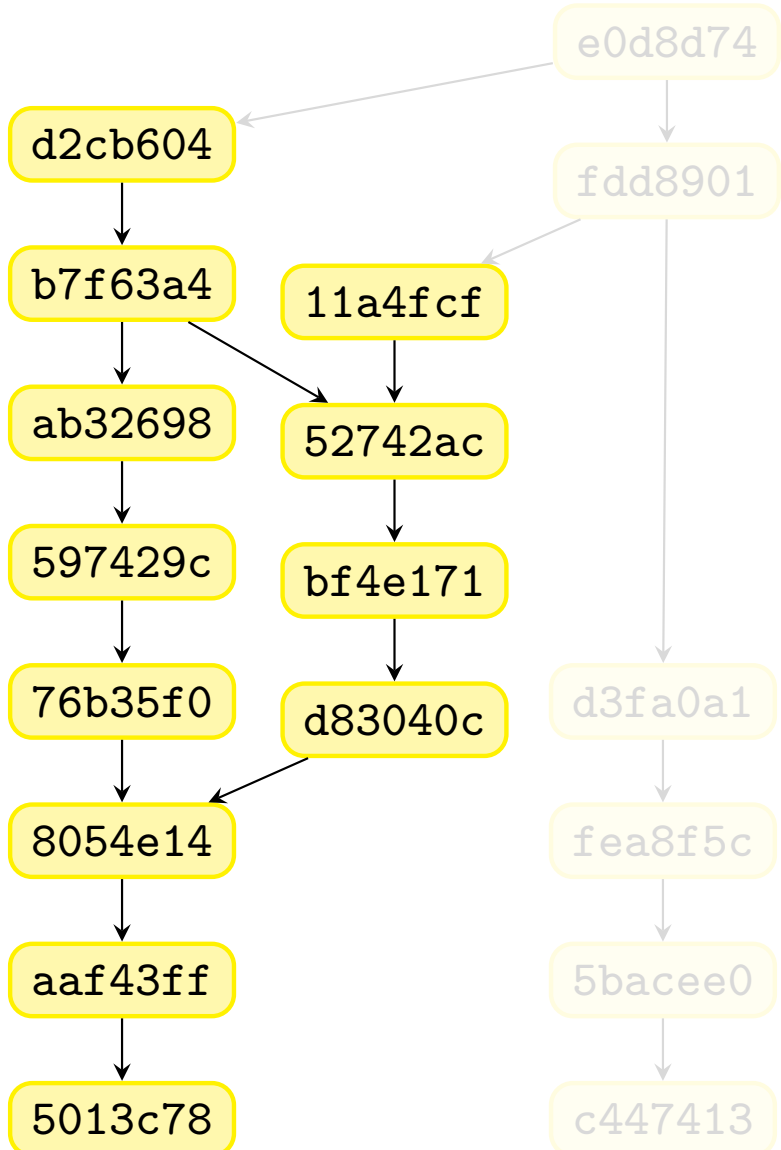
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



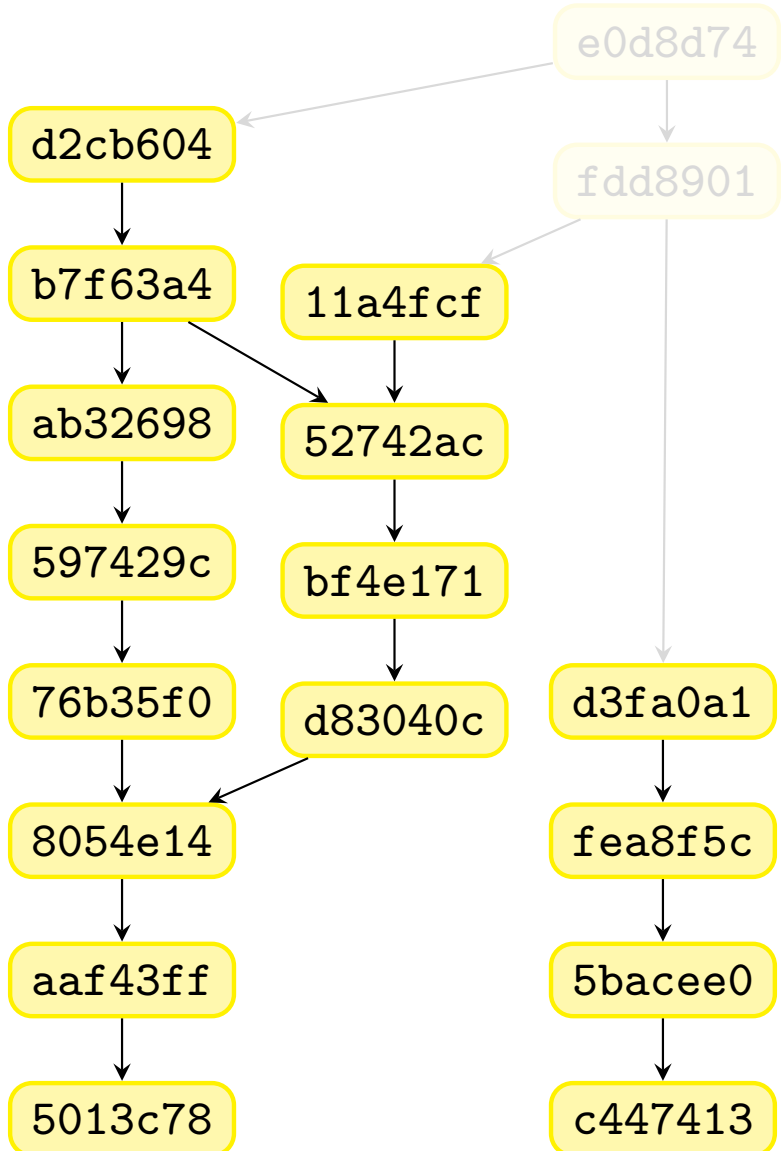
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



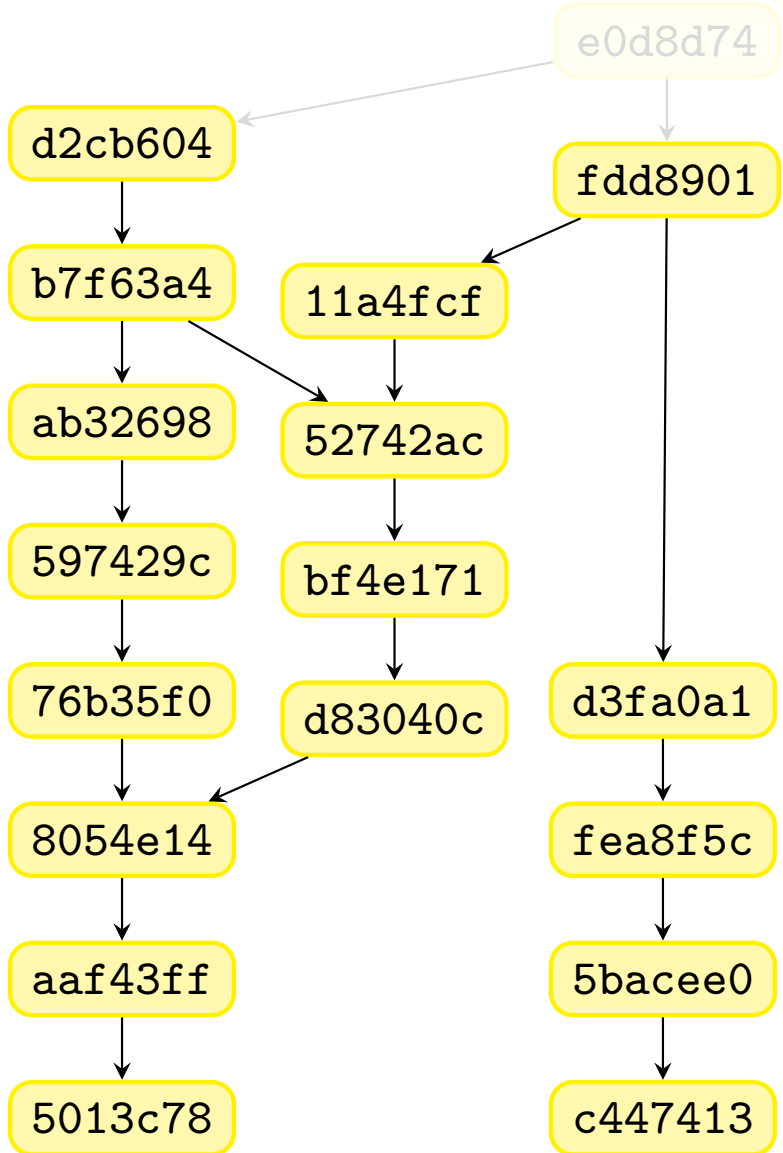
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



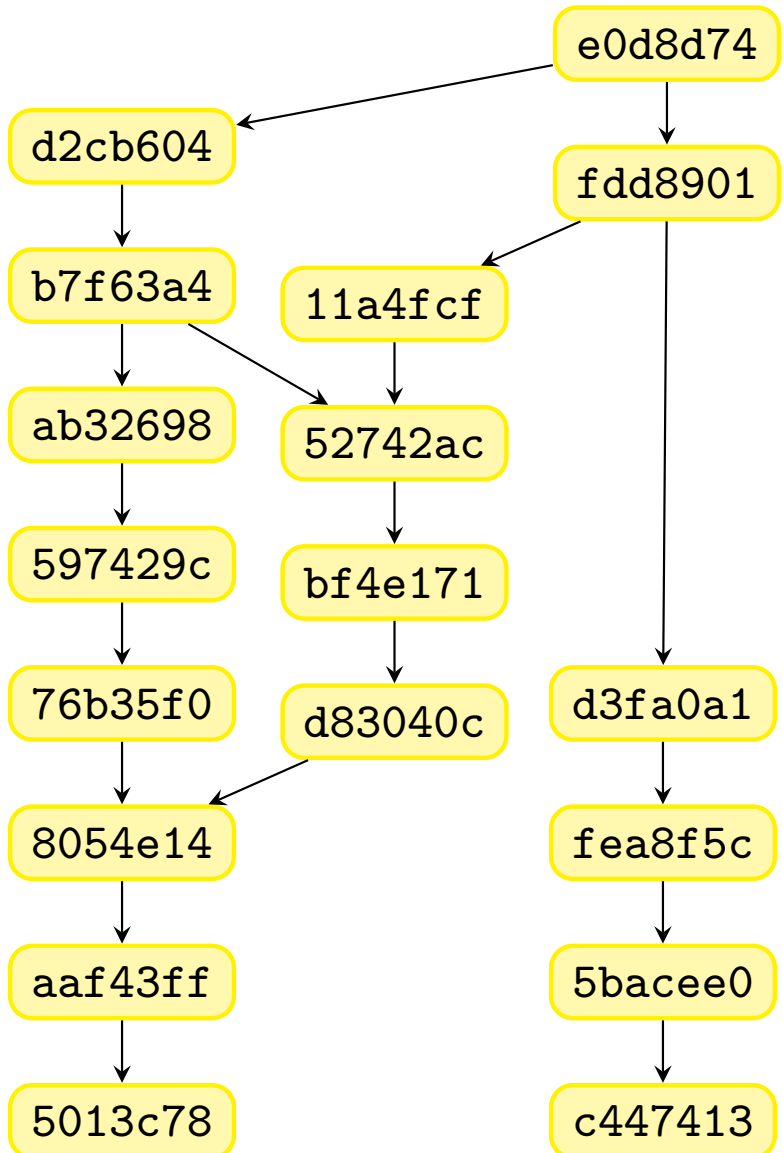
Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte



Commit-History

- gerichteter, azyklischer Graph der Commit-Geschichte

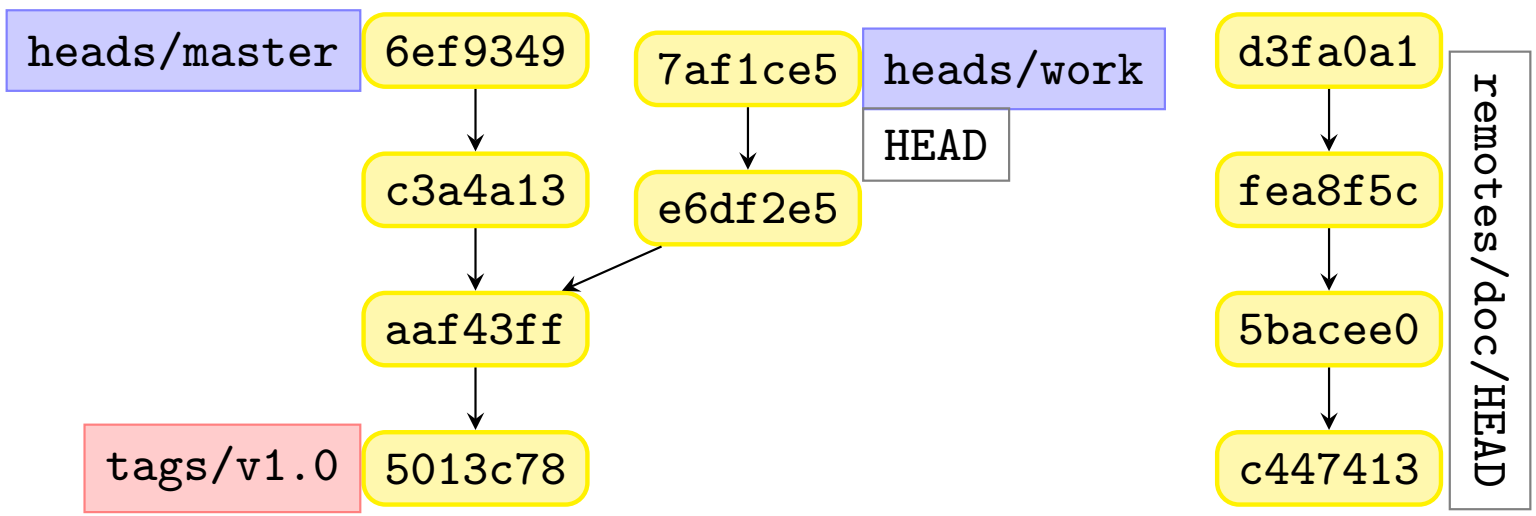


Referenzen

- SHA-1-IDs sind unhandlich, schwer zu merken und semantikarm
- Referenzen (*Refs*) sind selbst-definierbare Namen
- verweisen auf Commits
- oder auf andere Refs (→ symbolische Refs)
- wichtige Standardreferenz: *HEAD* – verweist auf aktuellen Commit
- wichtige „Arten“ von Refs:
  - Tags (→ `refs/tags/`)
  - Branches (→ `refs/heads/`)
  - Remotes (→ `refs/remotes/`)
  - ...

Referenzen

- SHA-1-IDs sind unhandlich, schwer zu merken und semantikarm
- Referenzen (*Refs*) sind selbst-definierbare Namen
- verweisen auf Commits
- oder auf andere Refs (→ symbolische Refs)
- wichtige Standardreferenz: *HEAD* – verweist auf aktuellen Commit
- wichtige „Arten“ von Refs:
  - Tags (→ `refs/tags/`)
  - Branches (→ `refs/heads/`)
  - Remotes (→ `refs/remotes/`)
  - ...



Einschub: Angabe von Commits – Revisionen

Revision benennt ein Commit-Objekt

- Angabe als SHA-1
  - Eingabe eines eindeutigen Präfixes genügt
  - mindestens 4 Zeichen
  - oft: 7 Zeichen (beliebte Voreinstellung)
- Angabe als Ref – Suchreihenfolge:
  1. *name* (*HEAD*, *FETCH_HEAD*, *ORIG_HEAD*, *MERGE_HEAD*)
  2. `refs/`*name*
  3. `refs/tags/`*name*
  4. `refs/heads/`*name*
  5. `refs/remotes/`*name*
  6. `refs/remotes/`*name*`/HEAD`
- Angabe über die Commit-Geschichte

Einschub: Angabe von Commits – Revisionen

Revision benennt ein Commit-Objekt

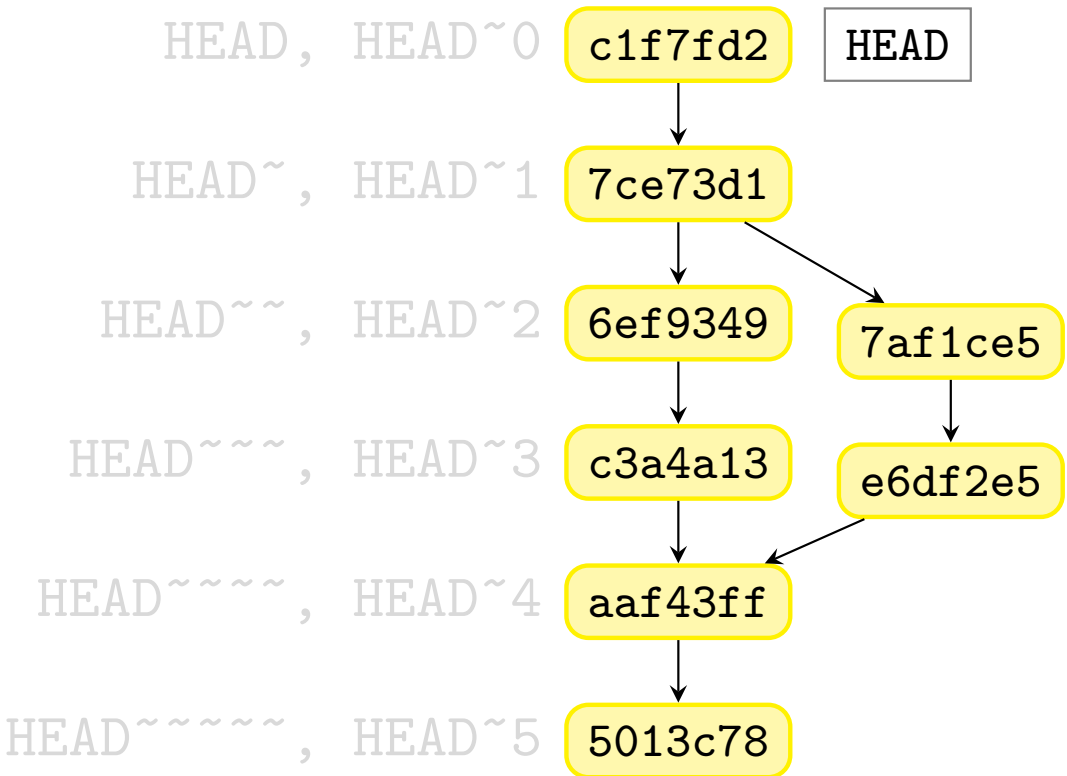
- Angabe als SHA-1
  - Eingabe eines eindeutigen Präfixes genügt
  - mindestens 4 Zeichen
  - oft: 7 Zeichen (beliebte Voreinstellung)
- Angabe als Ref – Suchreihenfolge:
  1. *name* (*HEAD*, *FETCH_HEAD*, *ORIG_HEAD*, *MERGE_HEAD*)
  2. `refs/`*name*
  3. `refs/tags/`*name*
  4. `refs/heads/`*name*
  5. `refs/remotes/`*name*
  6. `refs/remotes/`*name*`/HEAD`
- Angabe über die Commit-Geschichte

Einschub: Angabe von Commits – Revisionen

Geschichtstraverse mit der Tilde (~)

Sei `x` eine Revision.

- `x~` bezeichnet den ersten Parent von `x`
- `x~`*n* bezeichnet den ersten Parent der *n*-ten Generation

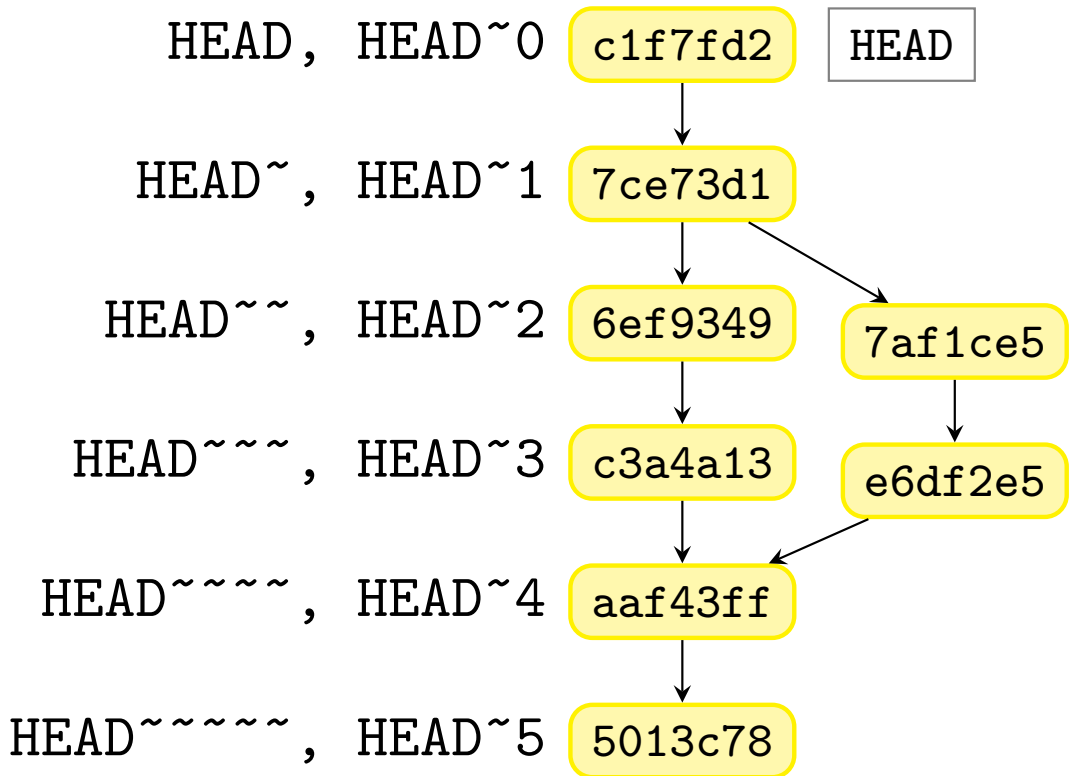


Einschub: Angabe von Commits – Revisionen

Geschichtstraverse mit der Tilde (~)

Sei `x` eine Revision.

- `x~` bezeichnet den ersten Parent von `x`
- `x~`*n* bezeichnet den ersten Parent der *n*-ten Generation

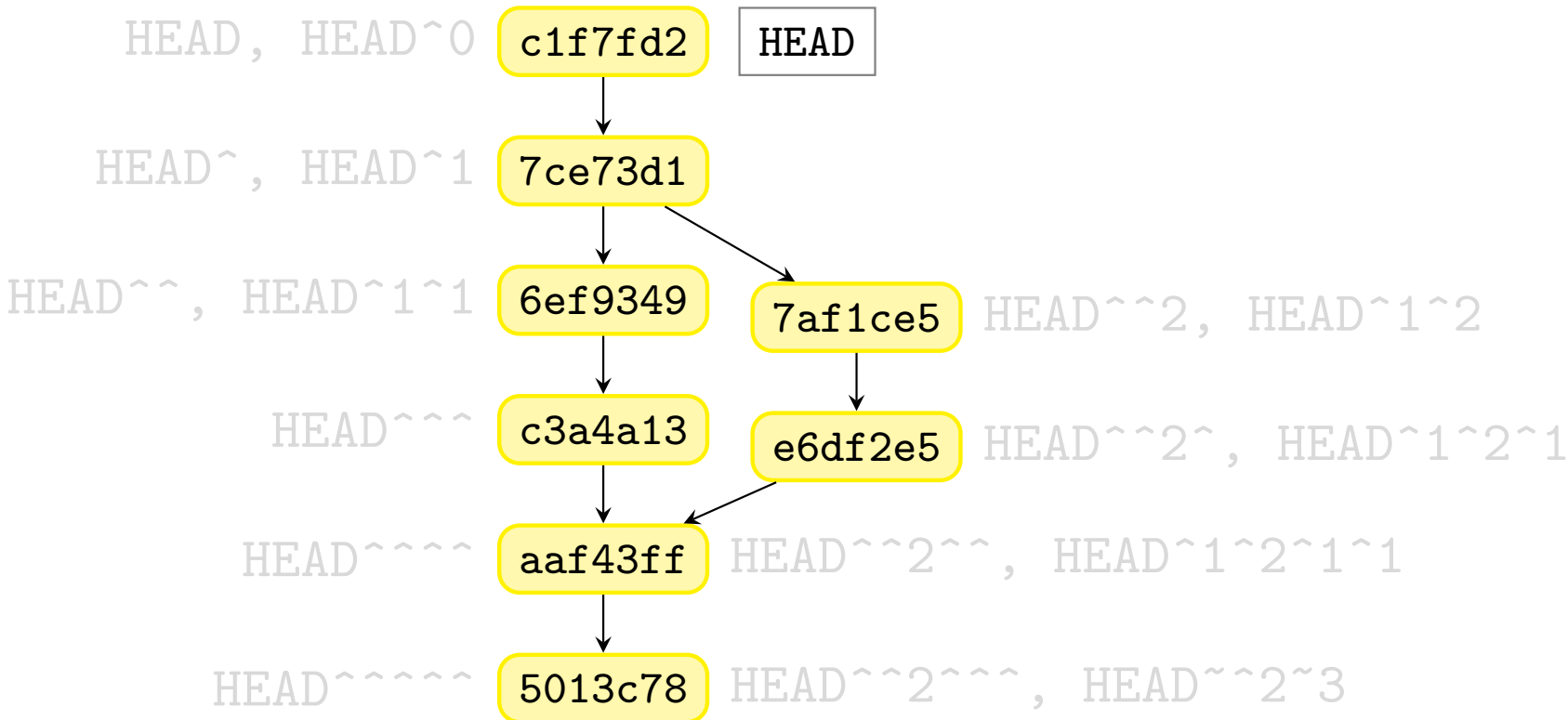


Einschub: Angabe von Commits – Revisionen

Geschichtstraverse mit Hut (^)

Sei `x` eine Revision.

- `x^` bezeichnet den ersten Parent von `x`
- `x^n` bezeichnet den *n*-ten (direkten) Parent von `x`

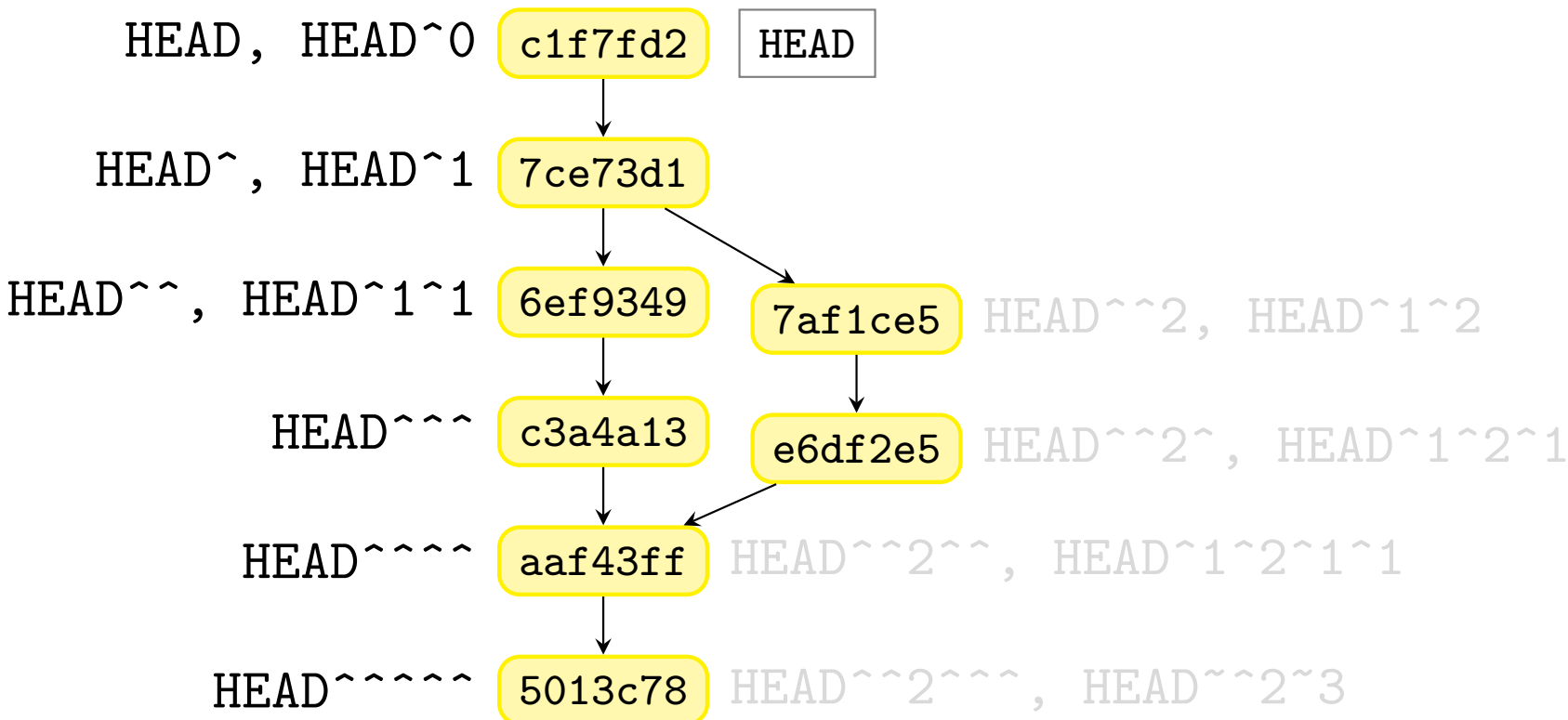


Einschub: Angabe von Commits – Revisionen

Geschichtstraverse mit Hut (^)

Sei `x` eine Revision.

- `x^` bezeichnet den ersten Parent von `x`
- `x^n` bezeichnet den *n*-ten (direkten) Parent von `x`

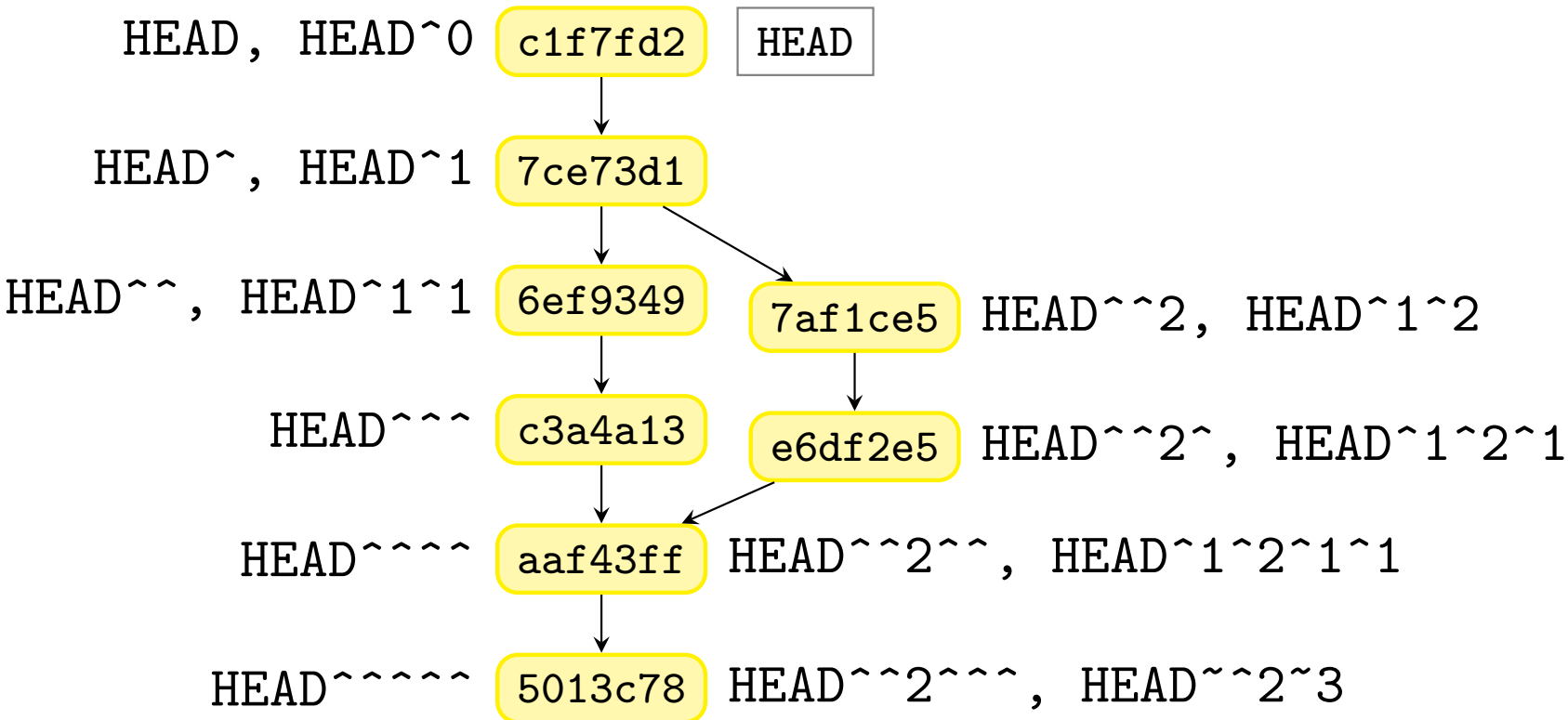


Einschub: Angabe von Commits – Revisionen

Geschichtstraverse mit Hut (^)

Sei `x` eine Revision.

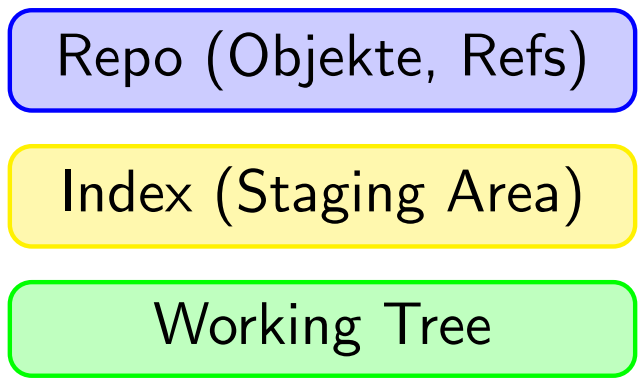
- `x^` bezeichnet den ersten Parent von `x`
- `x^n` bezeichnet den *n*-ten (direkten) Parent von `x`



Index

Index (auch: *Cache*, *Staging Area*)

- Schicht zwischen Arbeitsverzeichnis und dem eigentlichen Repo
- Tree, der irgendwann ins Repo geschrieben wird und von einem Commit-Objekt benutzt wird
- existiert implizit in allen VCS, in Git explizit nutzbar
- Vorteil: volle Kontrolle über einzuspeisende Daten



Teil III

Getting started

Installation

Paketiert:

- Debian: `aptitude install git-core`
  (eventuell auch: `git-arch`, `git-cvs`, `git-daemon-run`, `git-doc`, `git-email`, `git-gui`, `git-svn`, `gitk`, `gitweb`)
- Gentoo: `emerge git`
- RedHat/Fedora: `yum install git`
- ArchLinux: `pacman -S git`
- Microsoft Windows:
  - msysgit
  - über Cygwin
  - GitExtensions
  - TortoiseGit
- OS X: Installer

Installation

aus dem Quellcode, ein Beispiel

1. Installieren von:
   - `zlib` mit Headerfiles („`libz-dev`“)!
   - `asciidoc` (≥ 8.2) für Dokumentation
   - `libcurl` inkl. Headerfiles für HTTP-Support
   - `libsvn-perl` für Git-SVN
   - `tcl` (≥ 8.5) für GUIs
   - `gettext` für internationalisierte GUIs
2. Herunterladen und ins Quellverzeichnis wechseln
   - `wget http://kernel.org/pub/.../git-1.6.2.tar.bz2`
     `tar xvf git-1.6.2.tar.bz2 && cd git-1.6.2/`
   - `git clone git://git.kernel.org/pub/scm/git/git.git &&`
     `cd git` (oder `http://`)
3. `./configure --prefix=/usr/local/` (Defaultpräfix ist `~/bin/`)
4. `make -j2 all man html info pdf`
5. `su -c 'make install install-man install-html`
   `install-info install-pdf'`

Einrichtung

Konfiguration pro Systemnutzer: `~/.gitconfig`

Wichtig — sich Git vorstellen:

- `git config --global user.name 'Vorname Nachname'`
- `git config --global user.email 'user@example.com'`

Subjektiv, aber erwähnenswert (Doku):

- `git config --global color.ui auto`
- `git config --global pager.status yes`
- `git config --global diff.renames copy`
- `git config --global diff.renameLimit 0`
- `git config --global pack.threads 0`
- `git config --global rerere.enabled true`
- `git config --global alias.k '!gitk'`
- `git config --global core.editor '`*`editor`*`'`

Repo erzeugen

1. Initialisieren

- leeres Repo erzeugen
  1. `mkdir` *`Projekt`* (falls es noch nicht existiert)
  2. `cd` *`Projekt`*
  3. `git init`
- in *`Projekt/`* wird `.git/`-Verzeichnis erzeugt

`branches/`	`hooks/`
`info/`	`objects/`
`refs/`	`HEAD`
`config`	`description`

- es gibt keine Objekte und keine Refs
- der Index ist leer
- `git init --shared=...` für „Tuning“ der Repo-Permissions

Repo erzeugen

1. Initialisieren

- leeres Repo erzeugen
  1. `mkdir` *`Projekt`* (falls es noch nicht existiert)
  2. `cd` *`Projekt`*
  3. `git init`
- in *`Projekt/`* wird `.git/`-Verzeichnis erzeugt

`branches/`	`hooks/`
`info/`	`objects/`
`refs/`	`HEAD`
`config`	`description`

- es gibt keine Objekte und keine Refs
- der Index ist leer
- `git init --shared=...` für „Tuning“ der Repo-Permissions

Repo erzeugen

1. Initialisieren

- leeres Repo erzeugen
  1. `mkdir` *`Projekt`* (falls es noch nicht existiert)
  2. `cd` *`Projekt`*
  3. `git init`
- in *`Projekt/`* wird `.git/`-Verzeichnis erzeugt

`branches/`	`hooks/`
`info/`	`objects/`
`refs/`	`HEAD`
`config`	`description`

- es gibt keine Objekte und keine Refs
- der Index ist leer
- `git init --shared=...` für „Tuning“ der Repo-Permissions

Repo erzeugen

2. Informieren

1. wir haben Dateien ins Verzeichnis (Working Tree) gelegt
2. `git status`

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
#    committed)
#
#       empty
#       small.sh
#       small/
nothing added to commit but untracked files present (use
"git add" to track)
```

Repo erzeugen

3. Dateien dem Index vorstellen

1. `git add empty small.sh`
2. `git status`

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   empty
#       new file:   small.sh
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
#    committed)
#
#       small/
```

Repo erzeugen

4. Commit erzeugen

`git commit`

1. Editor öffnet sich:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
[...]
```

2. gebe ein: *Some basic files*
3. speichere und beende Editor

```
[master (root-commit) edbd6e5] Some basic files
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 empty
 create mode 100755 small.sh
```

Repo erzeugen

5. Vergewissern

1. `git log`

```
commit edbd6e5aae715ea0807c31478b096701a09d3bfb
Author: Vorname Nachname <vnach@example.com>
Date:   Thu Mar 12 00:11:17 2009 +0100

    Some basic files
```

2. `git status`

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
#    committed)
#
#       small/
nothing added to commit but untracked files present (use
"git add" to track)
```

eben angerissene Grundlagen

`git init` erzeugt leeres Repo

`git status` zeigt Status an inklusive Hilfe, wie man etwas daran ändert

`git add` teilt zu übernehmende Dateien/Änderungen dem Index mit

`git commit` erzeugt ein Commit-Objekt

`git log` zeigt die Commit-Geschichte an

letzten Commit berichtigen

`small/` inklusive Inhalt soll noch in den getätigten Commit hinein

1. `git add small` (arbeitet rekursiv)
2. `git status`

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   small/empty-file
#       new file:   small/hello
#
```

3. `git commit --amend`

```
[master c477d69] Some basic files
 2 files changed, 6 insertions(+), 0 deletions(-)
 create mode 100644 empty
 create mode 100755 small.sh
 create mode 100644 small/empty-file
 create mode 100644 small/hello
```

Änderungen

Wir ändern `small.sh` und `small/hello`.

1. `git diff`

```
diff --git a/small.sh b/small.sh
index 97d9fed..300c36c 100755
--- a/small.sh
+++ b/small.sh
@@ -2,4 +2,5 @@
 for i in "$@"
 do
     convert "$i" -scale 0.5 "small/$i"
+    echo "$i"
 done
diff --git a/small/hello b/small/hello
index 802992c..8095a18 100644
--- a/small/hello
+++ b/small/hello
@@ -1 +1 @@
-Hello world
+Hallo, Welt!
```

Änderungen

Änderungen an `small/hello` sollen „committet“ werden, also in den Index damit.

1. `git add small/hello`
2. `git status`

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   small/hello
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be
#    committed)
#   (use "git checkout -- <file>..." to discard changes
#    in working directory)
#
#       modified:   small.sh
#
```

Änderungen

Änderungen an `small/hello` sollen „committet“ werden.

1. `git diff`

```
diff --git a/small.sh b/small.sh
index 97d9fed..300c36c 100755
--- a/small.sh
+++ b/small.sh
@@ -2,4 +2,5 @@
 for i in "$@"
 do
     convert "$i" -scale 0.5 "small/$i"
+    echo "$i"
 done
```

2. `git commit`

```
[master d730247] Germanize small/hello
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Änderungen

Commit anzeigen

1. `git show`

```
commit d730247ae77903356171696009 1b4c288b74fcc6
Author: Vorname Nachname <vnach@example.com>
Date:   Thu Mar 12 02:58:35 2009 +0100

    Germanize small/hello

    Nicht jeder ist des Englischen maechtig.

diff --git a/small/hello b/small/hello
index 802992c..8095a18 100644
--- a/small/hello
+++ b/small/hello
@@ -1 +1 @@
-Hello world
+Hallo, Welt!
```

Änderungen

Geschichte anzeigen

1. `git log`

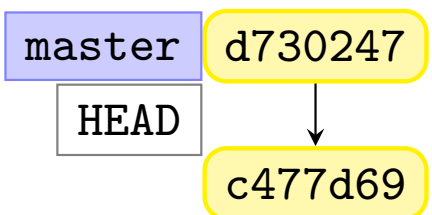
```
commit d730247ae77903356171696009 1b4c288b74fcc6
Author: Vorname Nachname <vnach@example.com>
Date:   Thu Mar 12 02:58:35 2009 +0100

    Germanize small/hello

    Nicht jeder ist des Englischen maechtig.

commit c477d69160b916b066e3f878c52eae7973a4b6b9
Author: Vorname Nachname <vnach@example.com>
Date:   Thu Mar 12 00:11:17 2009 +0100

    Some basic files
```



eben angerissene Grundlagen

`git init`, `git status`, `git add`, `git commit`, `git log` +

`git commit --amend` erlaubt Veränderung des letzten Commits, SHA-1 ändert sich

`git diff` zeigt (ohne weitere Argumente) Änderungen (*Diff*) zwischen Working Tree und Index an

`git show` zeigt Objekte an

Bisherige Grundlagen sollten ausreichen, um Git

- lokal
- für sich
- mit linearer Commit-Geschichte (wie CVS oder SVN)

nutzen zu können.

eben angerissene Grundlagen

`git init`, `git status`, `git add`, `git commit`, `git log` +

`git commit --amend` erlaubt Veränderung des letzten Commits, SHA-1 ändert sich

`git diff` zeigt (ohne weitere Argumente) Änderungen (*Diff*) zwischen Working Tree und Index an

`git show` zeigt Objekte an

Bisherige Grundlagen sollten ausreichen, um Git

- lokal
- für sich
- mit linearer Commit-Geschichte (wie CVS oder SVN)

nutzen zu können.

Klonen von Repos

- `git clone git://repo.or.cz/git.git`
  erzeugt Klon des git-Repos in neues Verzeichnis `git/`
- `git clone git://repo.or.cz/git.git meins`
  erzeugt Klon des git-Repos in neues Verzeichnis `meins/`
- *Klon* heißt Kopie aller Objekte, d. h. der *gesamten* Geschichte
- standardmäßig werden
  - Tags übernommen
  - aktiver Branch und HEAD übernommen
  - Branches (`refs/heads/`*name*) $\longrightarrow$
    Remote-Branches (`refs/remotes/origin/`*name*)

git://?

`git://` ist Git-eigenes Transport-Protokoll.
Transport auch möglich über

- `rsync://example.com/pfad/repo.git/`
- `http://example.com/pfad/repo.git/`
- `https://example.com/pfad/repo.git/`
- `ssh://user@example.com/pfad/repo.git/`
- `user@example.com:/pfad/repo.git/` (SSH)
- `/pfad/repo.git/` (lokal)

Branches anzeigen

1. `git branch`

```
* master
```

2. `git branch -va` (verbose, all)

```
* master          6462146 Merge branch 'maint'
  origin/HEAD     6462146 Merge branch 'maint'
  origin/html     ea6a764 Autogenerated HTML docs for v1.6.2-149-g6462
  origin/maint    c2aca7c Update draft release notes for 1.6.2.1
  origin/man      fae7dc0 Autogenerated manpages for v1.6.2-149-g6462
  origin/master   6462146 Merge branch 'maint'
  origin/next     c3f22eb Merge branch 'kg/http-auth' into next
  origin/pu       a5aa0c6 Merge branch 'fg/push-default' into pu
  origin/todo     a58b4f2 What's in -- first mass graduation post 1.6.2
```

Begriffe:

- `origin` ist *Remote* (steht für `git://repo.or.cz/git.git`)
- `origin/master` ist *Remote-Branch*
- `master` ist *Branch* (sogar *Tracking-Branch* für `origin/master`)

Branch-Geschichte anzeigen

1. `git log --pretty=oneline --abbrev-commit --first-parent`

```
6462146 Merge branch 'maint'
500ff11 Merge branch 'mg/maint-submodule-normalize-path'
aec8130 Merge branch 'jc/maint-1.6.0-keep-pack'
[...]
7d59cee test: do not LoadModule log_config_module unconditionally
e47eec8 git-instaweb: fix lighttpd configuration on cygwin
4d5d398 Include log_config module in apache.conf
c7cb12b Typo and language fixes for git-checkout.txt
15112c9 connect.c: remove a few globals by using git_config callback
[...]
```

2. `git log --pretty=oneline --abbrev-commit --first-parent origin/next`

```
c3f22eb Merge branch 'kg/http-auth' into next
220d65a Merge branch 'master' into next
4909f65 Merge branch 'mh/cvsimport-tests' into next
[...]
05ac6b3 improve missing repository error message
aec0c1b git-archive: add --output=<file> to send output to a file
734cd57 Improve error message for git-filter-branch
[...]
```

zu neuen und alten Branches wechseln

1. `git checkout -b test`

```
Switched to a new branch "test"
```

2. `git branch -v`

```
  master 6462146 Merge branch 'maint'
* test   6462146 Merge branch 'maint'
```

3. Änderungen machen, `git add`, `git commit`
4. `git branch -v`

```
  master 6462146 Merge branch 'maint'
* test   48e1178 Change test file
```

5. `git log --pretty=oneline --abbrev-commit master..`

```
48e1178 Change test file
e97987f Add test file
```

6. `git checkout master`

```
Switched to branch "master"
```

7. `git branch -v`

```
* master 6462146 Merge branch 'maint'
  test   48e1178 Change test file
```

zu neuen und alten Branches wechseln

1. `git checkout -b test`

   ```
   Switched to a new branch "test"
   ```

2. `git branch -v`

   ```
     master 6462146 Merge branch 'maint'
   * test   6462146 Merge branch 'maint'
   ```

3. Änderungen machen, `git add`, `git commit`

4. `git branch -v`

   ```
     master 6462146 Merge branch 'maint'
   * test   48e1178 Change test file
   ```

5. `git log --pretty=oneline --abbrev-commit master..`

   ```
   48e1178 Change test file
   e97987f Add test file
   ```

6. `git checkout master`

   ```
   Switched to branch "master"
   ```

7. `git branch -v`

   ```
   * master 6462146 Merge branch 'maint'
     test   48e1178 Change test file
   ```

Einschub: Revisionsbereiche

Interessant für Befehle wie `git log`

`rev`	alle von `rev` erreichbaren Commits (inkl. `rev`)
`r1 r2`	alle von `r1` oder `r2` erreichbaren Commits
`^r1 r2`	alle von `r1` erreichbaren Commits, abzüglich der, die von `r2` aus erreichbar sind
`r1..r2`	entspricht `^r1 r2`
`r1...r2`	symmetrische Differenz der von `r1` und `r2` erreichbaren Commits
`rev^@`	alle von `rev` erreichbaren Commits, ohne `rev` selbst
`rev^!`	nur `rev`

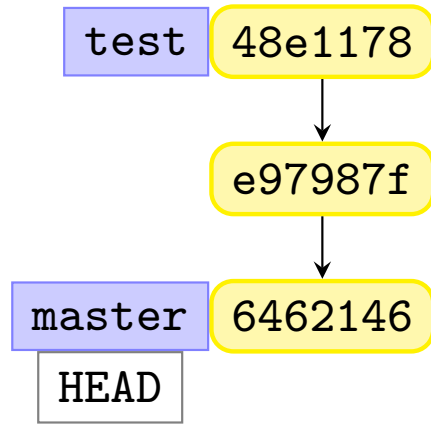
`master..` ist Kurzform von `master..HEAD`

Merges

Trivialform: Fast-forward

Sei `master` von eben der aktive Branch.

```
1 git merge test
```

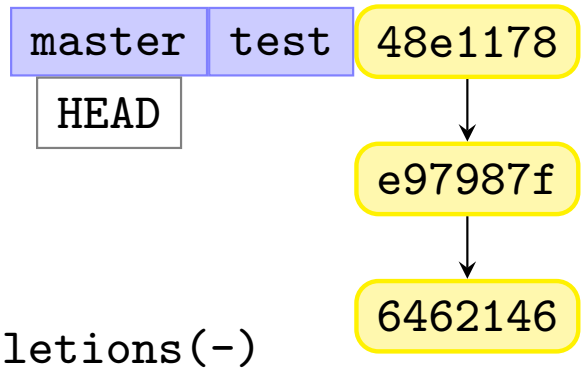


Merge	Vereinigung von Branches
Remote	hier: der in HEAD zu mergende Branch
Merge-Base	erster gemeinsamer Commit zweier Branches
Fast-forward	Wenn HEAD *Merge-Base* von HEAD und Remote ist, dann setze HEAD auf Remote.

Merges

Trivialform: Fast-forward

Sei `master` von eben der aktive Branch.



1. `git merge test`

```
Updating 6462146..48e1178
Fast forward
 test |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 test
```

2. `git log --decorate --pretty=oneline --abbrev-commit master@{1}..master`

```
48e1178 (refs/heads/test, refs/heads/master) Change test file
e97987f Add test file
```

Merge Vereinigung von Branches

Remote hier: der in HEAD zu mergende Branch

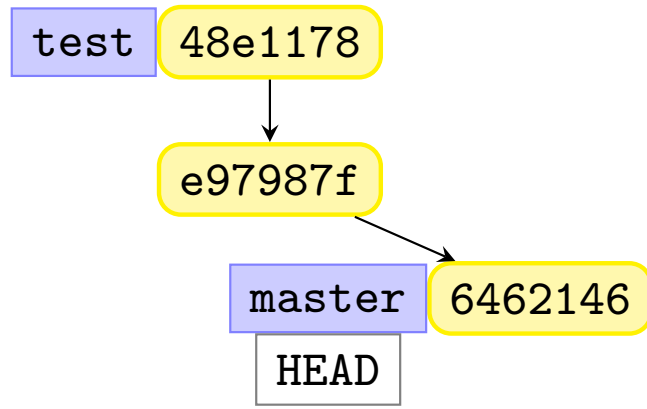
Merge-Base erster gemeinsamer Commit zweier Branches

Fast-forward Wenn HEAD *Merge-Base* von HEAD und Remote ist, dann setze HEAD auf Remote.

Merges

wenn Fast-forward nicht möglich ist

Sei `master` von vorhin der aktive Branch.

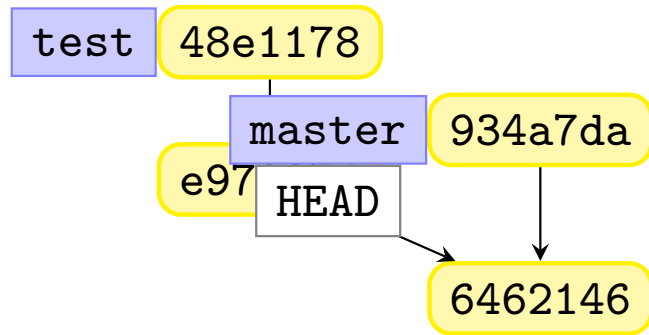


Merges

wenn Fast-forward nicht möglich ist

Sei `master` von vorhin der aktive Branch.

1. Änderungen, `git add`, `git commit`

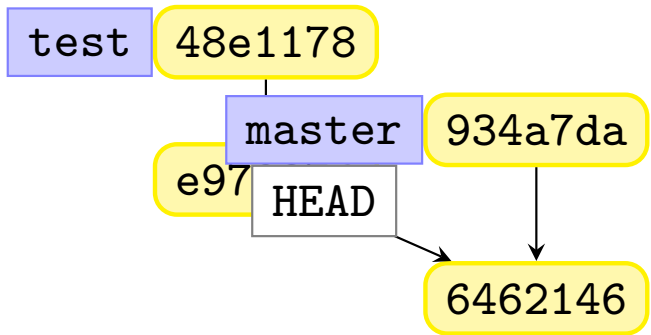


Merges

wenn Fast-forward nicht möglich ist

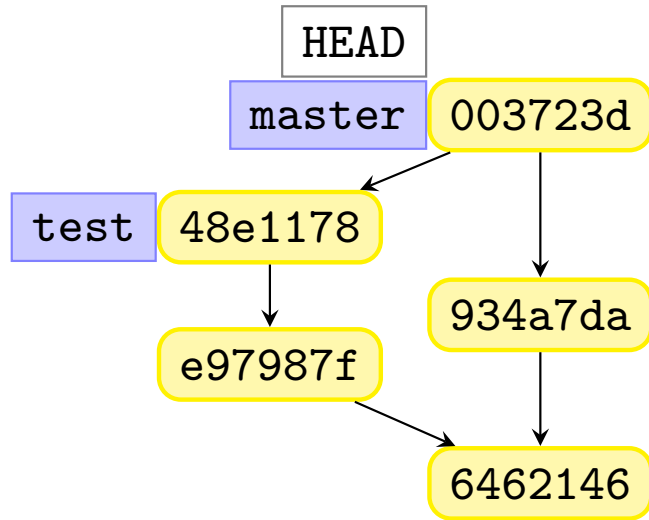
Sei `master` von vorhin der aktive Branch.

1. Änderungen, `git add`, `git commit`
2. `git merge test`



Merges

wenn Fast-forward nicht möglich ist



Sei `master` von vorhin der aktive Branch.

1. Änderungen, `git add`, `git commit`
2. `git merge test`

```
Merge made by recursive.
 test |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 test
```

3. `git log --graph --decorate --pretty=oneline --abbrev-commit \ master@{2}..master`

```
*   003723d (refs/heads/master) Merge branch 'test'
|\
| * 48e1178 (refs/heads/test) Change test file
| * e97987f Add test file
* 934a7da Add foo file
```

Merge-Commit-Objekt erstellt.

Konflikte

Keine Panik!

Sei `master` von vorhin der aktive Branch.

1. Änderungen, `git add`, `git commit`
2. `git merge test`

```
Auto-merging test
CONFLICT (add/add): Merge conflict in test
Automatic merge failed; fix conflicts and then commit the result.
```

3. `git status`

```
test: needs merge
# On branch master
#
# Changed but not updated:
[...]
#
#       unmerged:   test
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Konflikte

Was ist passiert?

- unterschiedliche Änderungen in HEAD und Remote an sich berührenden Stellen (gleiche Dateien, ähnliche Positionen)
- Git bzw. die angewendete Merge-Strategie weiß nicht, wie Konflikt aufzulösen ist
- betroffene Datei als *unmerged* markiert
- Konfliktmarkierungen an betroffene Stellen eingefügt

```
cat test
<<<<<<< HEAD:test
4
=======
2
>>>>>>> test:test
```

Konflikte inspizieren

aus zwei Perspektiven

- `git diff` (`git diff --cc`)

```
diff --cc test
index b8626c4,0cfbf08..0000000
--- a/test
+++ b/test
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:test
 +4
++=======
+ 2
++>>>>>>> test:test
```

beabsichtigter Merge

- `git diff --merge`

```
diff --cc test
index 0cfbf08,0000000..b8626c4
mode 100644,000000..100644
--- a/test
+++ b/test
@@@ -1,1 -1,0 +1,1 @@@
- 2
++4
```

- `git log --merge`

Konflikte inspizieren

HEAD-Perspektive

- `git diff --ours`

```
* Unmerged path test
diff --git a/test b/test
index b8626c4..cbf3eab 100644
--- a/test
+++ b/test
@@ -1 +1,5 @@
+<<<<<<< HEAD:test
 4
+=======
+2
+>>>>>>> test:test
```

Remote-Perspektive

- `git diff --theirs`

```
* Unmerged path test
diff --git a/test b/test
index 0cfbf08..cbf3eab 100644
--- a/test
+++ b/test
@@ -1 +1,5 @@
+<<<<<<< HEAD:test
+4
+=======
 2
+>>>>>>> test:test
```

Selbst ausprobieren: `git diff --base`

Konflikte lösen

1. *unmerged*-markierte Dateien bearbeiten
2. `git add test` (dies dem Index mitteilen)
3. schauen, ob alles ok
   - `git status`
   - `git diff HEAD`
   - `git diff MERGE_HEAD`
4. `git commit`

Problem gelöst!
Oder nicht? Merge abbrechen mit `git reset --hard`

Anmerkung

Je nach Merge-Strategie muss eine automatische Konfliktlösung nicht immer ein sinnvolles Ergebnis haben, bspw. beim Mergen eines Refactoring-Branches. Immer genau hinschauen und notfalls nach dem Merge den Commit mit `git commit --amend` berichtigen.

Konflikte lösen

1. *unmerged*-markierte Dateien bearbeiten
2. `git add test` (dies dem Index mitteilen)
3. schauen, ob alles ok
   - `git status`
   - `git diff HEAD`
   - `git diff MERGE_HEAD`
4. `git commit`

Problem gelöst!

Oder nicht? Merge abbrechen mit `git reset --hard`

Anmerkung

Je nach Merge-Strategie muss eine automatische Konfliktlösung nicht immer ein sinnvolles Ergebnis haben, bspw. beim Mergen eines Refactoring-Branches. Immer genau hinschauen und notfalls nach dem Merge den Commit mit `git commit --amend` berichtigen.

Konflikte lösen

1. *unmerged*-markierte Dateien bearbeiten
2. `git add test` (dies dem Index mitteilen)
3. schauen, ob alles ok
   - `git status`
   - `git diff HEAD`
   - `git diff MERGE_HEAD`
4. `git commit`

Problem gelöst!
Oder nicht? Merge abbrechen mit `git reset --hard`

Anmerkung

Je nach Merge-Strategie muss eine automatische Konfliktlösung nicht immer ein sinnvolles Ergebnis haben, bspw. beim Mergen eines Refactoring-Branches. Immer genau hinschauen und notfalls nach dem Merge den Commit mit `git commit --amend` berichtigen.

Konflikte lösen

1. *unmerged*-markierte Dateien bearbeiten
2. `git add test` (dies dem Index mitteilen)
3. schauen, ob alles ok
   - `git status`
   - `git diff HEAD`
   - `git diff MERGE_HEAD`
4. `git commit`

Problem gelöst!
Oder nicht? Merge abbrechen mit `git reset --hard`

Anmerkung

Je nach Merge-Strategie muss eine automatische Konfliktlösung nicht immer ein sinnvolles Ergebnis haben, bspw. beim Mergen eines Refactoring-Branches. Immer genau hinschauen und notfalls nach dem Merge den Commit mit `git commit --amend` berichtigen.

Warum Branching und Merging?

- *Entwicklungszweige*: Unterscheidung von stabilen und experimentellen Zweigen
- *Topic-Branches („Themenzweige“)*: unabhängig von anderen Änderungen ein Feature in einem Branch entwickeln, testen, und wenn stabil genug mergen
- *experimentieren*: Was wäre wenn . . . ?
- *Hilfe beim verteilten Arbeiten*: im Prinzip ist jeder Klon ein weiterer Branch, der später gemerged werden kann

weitere wichtige Befehle

`git branch` Branches verwalten (anzeigen, löschen, umbenennen, erreichbare Commits finden)

`git tag` Tags verwalten (leicht- und schwergewichtige)

`git checkout` „Auschecken“ von Branches oder Commits (*detached HEAD*)

`git reset` (Zurück)setzen von HEAD, Working Tree, Index

`git rebase` Rebasen von Branches auf neue Grundlage

`git mv` Umbenennen/Bewegen von Dateien oder Verzeichnissen

`git rm` Löschen von Dateien oder Verzeichnissen

`git diff` Diffs anzeigen (Branches, Dateien), vgl.

- `git diff --cached`
- `git diff` *`Rev1..Rev2`*
- `git diff` *`Rev -- Datei`*
- `git diff` *`Rev:dortDatei -- hierDatei`*
- `git diff` *`Rev1:Datei1 Rev2:Datei2`*

git remote

Verwaltung von Remotes (fremden Repos)

- `git remote -v`
- `git remote add -f` *`Name Url`*
- `git remote rename` *`alt neu`*
- `git remote rm` *`Name`*
- `git remote show` *`Name`*
- `git remote show -n` *`Name`* (Offline)
- `git remote prune` *`Name`*
- `git remote update`

`git fetch` und `git pull`

`git fetch` holt

- Branches und Tags von fernen Repos,
- die Objekte dahinter

`git pull`

- Faustregel: `pull` = `fetch` + X
  - normales `git pull`: X = `merge`
  - `git pull --rebase`: X = `rebase` $\rightarrow$ lineare Geschichte
- `git pull` *`Repo Refspec`*

`git push`

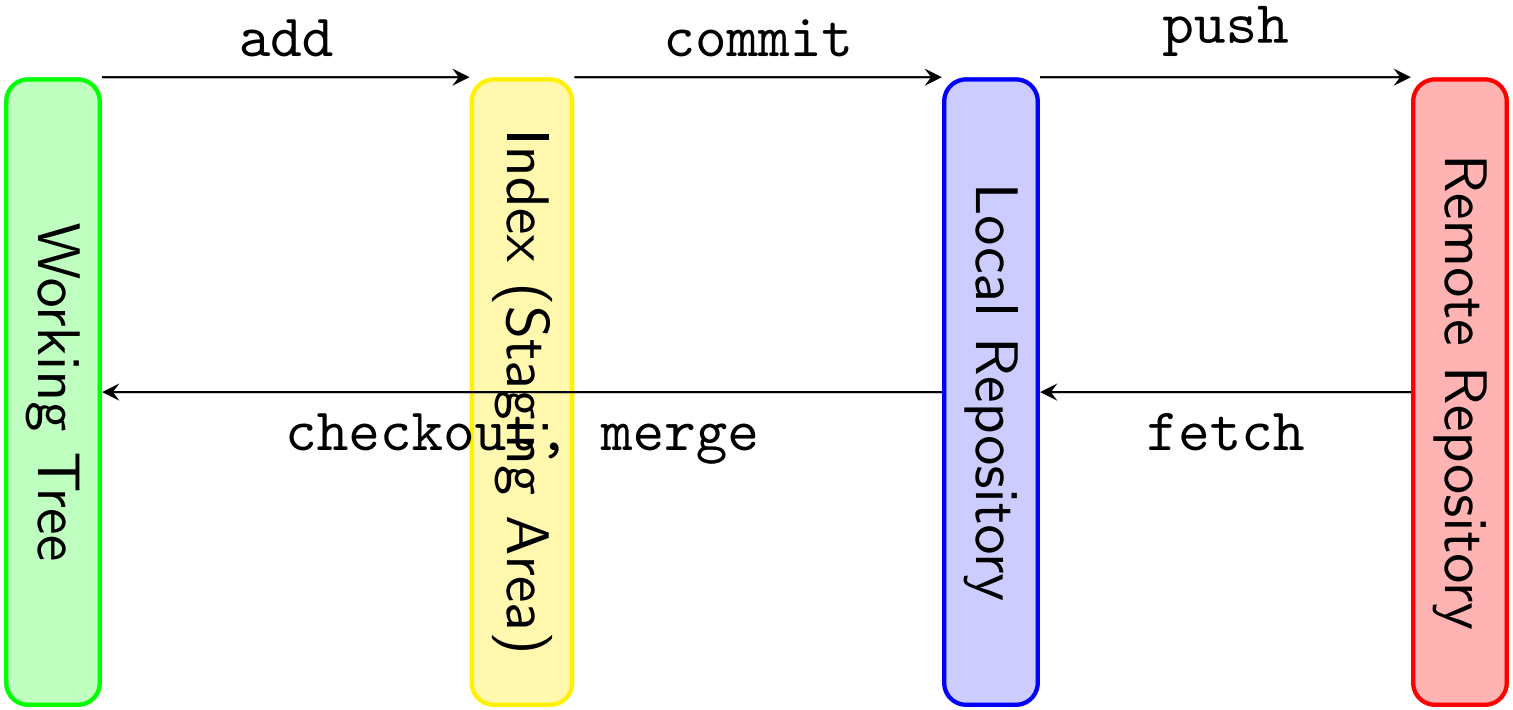
`git push` lädt

- Objekte
- Branches
- und auf Wunsch Tags (`--tags`)

hoch.

- Gegenstück zu `git fetch`
- keine Merges auf Remote-Seite
- `git push` *Repo lokalerBranch*:*fernerBranch*
- Löschen: `git push` *Repo lokalerBranch*:*fernerBranch*

Zusammenfassung



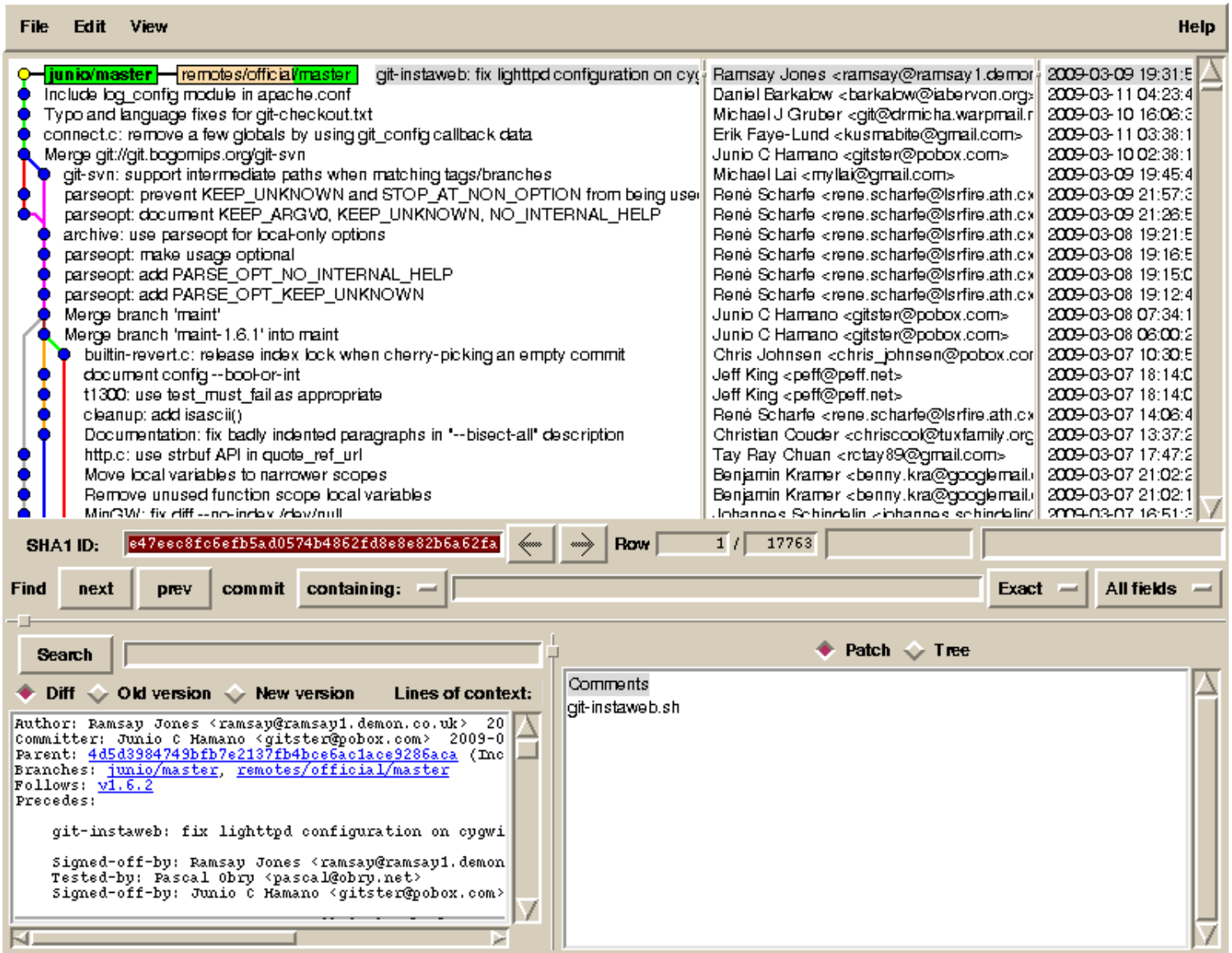
to be continued...

Was zum klicken...

- erste Anlaufstelle: GitWiki
- dort `GUI-Feature-Matrix`, enthalten:
  - gitk
  - git-gui
  - tig*
  - git-forest*
  - QGit
  - Giggle
  - gitview
  - git-forest*
  - gct
  - pmpu
  - Qct
  - git-cola
  - GitNub
  - GitX
  - Pyrite
  - gitSafe
  - teamGit
  - Git Extensions
  - TortoiseGit
  - gitg

Was zum klicken...

Screenshot: gitk



Teil IV

Verteilte Arbeitsabläufe

Öffentliches Eine-Person-Projekt

Grundlage

- allgemeiner Ablauf schon gezeigt
- einseitige Synchronisation: `git push`
  - über SSH: eventuell `git-shell` als restriktive Login-Shell
  - über HTTP: mit WebDAV
- öffentlich ziehbar über `git://` oder HTTP
- Öffentlichkeit via Git-Hosting:
  - GitHub – Hosting und soziales Netzwerk
  - Gitorious – etwas schlichter
  - repo.or.cz – `gitweb`-basiert
  - SourceForge – all-in-one Open-Source-Hoster
  - Savannah – ebenso
  - ...
  - oder eigener Server?
- Tracking-Branches sinnvoll

Öffentliches Eine-Person-Projekt

eigener Server

- Pull-Fähigkeit relativ einfach einzurichten ...
  - HTTP mittels `git update-server-info`
  - Git-Protokoll mittels `git-daemon`
- Repo auf Server sollte „*bare*“ (nackt) sein
  - d. h. es gibt keinen Working Tree
  - `git init --bare`
  - `git clone --bare` *`Quellrepo`*
  - `cd Quellrepo`
    `cp -a .git` *`/Pfad/zum/Ziel`*`.git`
    `cd` *`/Pfad/zum/Ziel`*`.git`
    `git config core.bare true`
- „*non-bare*“ Repos sind etwas kniffliger

Zentraler Server

Teamarbeit mit geschlossenen Teams

- technisch gibt es keinen „zentralen Server“
- aber ein *offizielles Repo* wurde untereinander etabliert
- Prinzip wie eben, aber jeder im Team hat Push-Rechte
- gelöst über Benutzerverwaltung des Transports (SSH, WebDAV)
- Git selbst kennt keine Benutzer- oder Rechteverwaltung

Patch-basierte Zusammenarbeit

offene Teams oder einzelne Zuarbeiten bei bis zu mittelgroßen Projekten

Zuarbeiter hat tolles Feature im Topic-Branch

1. `git rebase -i origin/master`
   um Topic-Branch auf aktuelle Grundlage zu rebasen
2. `git format-patch -o` *`Pfad/`* `origin/master..topic`
   um e-mail-artige Patch-Dateien in *`Pfad/`* zu erzeugen
3. diese Dateien anschauen und ggf. bearbeiten
4. `git send-email` *`Pfad/`*
   um die E-Mails zu versenden

`git send-email` kann die Patches auch direkt aus den Commits versenden. Nett ist beispielsweise

```
git send-email --compose --annotate origin/master..topic
```

Patch-basierte Zusammenarbeit

Patch-Format von `git format-patch`

- ```
  From <SHA-1> <Erstelldatum>
  From: <Vorname Nachname <E-Mail>>
  Date: <Autor-Datum>
  Subject: [PATCH 1/2] <erste Zeile der Commit-Message>

  <Rest der Commit-Message>
  ---
   <Diffstat>

  <Diffs>
  --
  <Git-Version>
  ```

- zwischen `---` und Diffstat sind Kommentare erlaubt, die Git nicht verarbeitet.
- sicherheitshalber diese Kommentare mit Space/Tab einrücken
  ```

Patch-basierte Zusammenarbeit

die Empfängerseite

Patches kommen beim Maintainer oder auf Mailingliste an

1. einfacher Code Review
2. eventuell Diskussion mit Verbesserungsvorschlägen → neue Patch-Serie
3. sobald ok, speichert Maintainer Patches in `mbox`/`Maildir` (Anmerkung: MUA sollte das können.)
4. `git am` *Mails*
   um Patches anzuwenden; für jeden Patch wird ein entsprechender Commit erzeugt

Oder...

- `git am -3` *Mails*
  falls Patch nicht sauber anwendbar ist → 3-Wege-Merge
- `git am -i` *Mails*
  für den interaktiven Modus, um bspw. Commit-Messages zu ändern

Patch-basierte Zusammenarbeit

Anmerkungen

- ohne Vortäuschung einheitlicher Identität und Zeit hat jeder Teilnehmer andere SHAs nach Anwenden der gleichen Patches
- nicht immer optimal
- sinnvoll: Patch-Einspeisung von zentraler, vertrauenswürdiger Stelle

hierarchische Systeme für Großprojekte

- offiziell etabliertes Repo
- wenige, vertrauenswürdige Delegierte
- Delegierte sind für definierte Teile des Projekts verantwortlich („Subsystem-Maintainer“)
- Delegierte erhalten patch-basierte Zuarbeiten
- reife Zuarbeiten werden ins offizielle Repo integriert via
  - direkten Push-Zugriff
  - Pull-Request an Maintainer
  - Bundles an Maintainer

Individuelle Projekte

- kleines Team, man kennt die anderen
- jeder hat eigene Vorlieben
- diese werden in einem Hauptbranch (`master`) gesammelt
- jeder hat die anderen als Remotes im Repo
- Sammlung per:
  - `git pull` bzw. `git merge` bestimmter Topic-Branches
  - Rosinen herauspicken (`git cherry-pick`)

Das Anarchie-Viertel: Mob-Branches

- Repo hat Branch `mob` oder frei-erstellbare Branches `mobs/`*`topic`*
- in die darf *jeder* pushen
- Maintainer sammelt die besten Commits/Branches per `git cherry-pick` oder `git merge`
- u. U. müssen Mob-Branches ab und zu „aufgeräumt“ werden (`git reset --hard`, `git branch -D`, `git rebase -i`)

Kleine Nützlichkeiten

- Reflog
- Aliases
- Hooks
- `git mergetool`
- `git rerere`
- `git tag`
- `.gitignore` vs `.git/info/exclude`
- `git add -p`
- `git cherry-pick`, `git revert`
- `git rebase -i`
- `git bisect`
- `git stash`
- clean/smudge-Filter
- Shallow Clone (`git clone --depth`)
- Submodules, Subprojects
- `git blame`
- `git grep`
- `git diff --color-words`, `git diff --patience`

Git-SVN

Git nutzen, SVN bedienen

- `git svn clone http://svn.example.com/prog/trunk prog`
- `cd prog`
- `git svn show-ignore >> .git/info/exclude`
  um selbe Dateien zu ignorieren
- normal mit Git arbeiten, lediglich auf lineare History beschränkt
- `git svn rebase`
  um Änderungen auf letzten SVN-Commit zu rebasen (in Git)
- `git svn dcommit`
  um Änderungen auf den SVN-Server zu committen

Weitere nützliche Programme

TopGit

- Git-Patch-Management „done right“
- Gegensatz zu Rebase: Geschichte bleibt erhalten
- `tg create t/`*`topic dep1 dep2`*
- jeder *Patch* ist ein Topic-Branch, kann also aus mehreren Commits bestehen
- Abhängigkeiten zwischen Patches werden verwaltet

MetaStore

- Dateiattributierungen mit Git nutzen
- verwaltet: Besitzer, Gruppe, Rechte, Xattrs, Mtime

Git-Hilfe

- Kommandodokumentation
  - `git help` zeigt allgemeine Hilfe
  - `git help` *`Kommando`* zeigt Manual Page
  - `git` *`Kommando`* `-h` zeigt Kurzhilfe
- Benutzerhandbuch dabei (`info`, `html`, `pdf`)
- Kurz-HOWTOs als Text im Git-Sourcecode unter `Documentation/howto`
- Suchmaschinen $\longrightarrow$ Blogs, GitWiki
- kommerzieller Git-Support oder der nächste Git-Guru
- IRC: FreeNode `#git`
- letzter Ausweg Mailingliste: `git at vger.kernel.org`

Das war's!

Vielen Dank für die Aufmerksamkeit!

Fragen?