Frank W. Bergmann
www.tuxad.com

xkcd...

https://xkcd.com/1597/

VCS – Geschichte/Überblick

- SCCS (70er) / RCS (80er) nur lokale Versionierung
- CVS (ca. 1990): Server (=shared access), intern RCS-Format
- Subversion/SVN (ca. 2000): DB statt RCS-Format, Umbenennen/Verschieben mit Historie, MetaData, mehr Protokolle
- Bitkeeper (ca. 1999): Verteilt, ursprünglich closed source, aber offen für OSS bis 2005, für Linux-Kernel bis 2005 genutzt
- Git (2005): Entwicklung durch Torvalds, da Bitkeeper nicht mehr "frei"
- Bazaar (ab 2005, Python): genutzt z.B. für Ubuntu
- Mercurial (2005), Python): häufig genutzt, Platz 2 nach Git

Vorteile SVN/CVS

- File-Locking möglich, da nur zentrales Directory
- Sub-Pfade: Verzeichnisses haben eigene .cvs/.svn-Einträge, git bietet nur Submodules
- Historie unabänderlich
- kein Garbage Collector (siehe reflogs)
- Aufzeichnung leerer Verzeichnisse
- Ressourcen schonender bei großen Binärdateien, da nur die letzte Kopie lokal vorhanden ist
- wegen stärkerer "Linearität" ist Arbeiten mit Timestamps möglich
- ebenso sequentielle Release-Nummern

Vorteile/Unterschiede Git

- Verteilt: commit/push auch offline, jeder Clone ist ein eigenständiges Repo!
- Hashes (z.B. bei Commits) - auch lokal möglich und ohne Server
- Branching
- Philosophie: commit early & often
- CLI Ausgabe mit Tipps
- Historie bleibt bei "Verlust" des Servers erhalten
- „Merge-Gedächtnis“: Historie ist direkt nachvollziehbar

Git objects/tree/blob

- Git arbeitet wie ein Dateisystem. Es kann Inhalte adressieren. Damit ist es wie ein FS ein Key-Value-Store.
- Die einzelnen gespeicherten Entitäten nennt man Objects.
- Es gibt unterschiedliche Typen. Ein Tree-Object ist dabei vergleichbar mit einem Verzeichnis und ein Blob-Object mit einer Datei.
- ("BLOB" als Abkürzung stammt vom Datentyp "BLOB" bei DBMS ab und kann demzufolge auch mit "Binary Large OBject" übersetzt werden.)
- Mehr Informationen zum Arbeiten auf den untersten Git-Ebenen mit Objects steht unter

  https://git-scm.com/book/de/v1/Git-Interna-Git-Objekte

Git reflog

- Im Gegensatz zu log, das commits ausgehend von HEAD, Tags etc. anzeigt, speichert reflog alle referenzierten commits im lokalen Repo. Die voreingestellte Expirationtime liegt bei 90 Tagen.
- Jedes Mal, wenn der Branch Tip aktualisiert wird, speichert Git die Information in dieser "temporary history".
- reflog wird auch oft das "Sicherheitsnetz" genannt. In Notfällen gilt daher die Regel: "Keep calm and use git reflog"
- Klassischer Anwendungsfall: Man hat irrtümlicherweise einen Branch gelöscht.
- Siehe

  https://stackoverflow.com/questions/5543280/how-do-i-get-the-deleted-branch-back-in-git

Konfiguration User

/home/johndoe/.gitconfig: "Globale" Konfiguration mit Grundeinstellungen für alle git-Repositories des Users.

Beispiel:

```
$ cat .gitconfig
[user]
        name = John Doe
        email = johndoe@example.com
[difftool "sourcetree"]
        cmd = opendiff \"$LOCAL\" \"$REMOTE\"
        path =
[pull]
        rebase = true
[push]
        default = simple
```

- Befehle: git config --global user.name "John Doe"
- git config --global user.email "x@y"

Konfiguration Projekt

Beispiel:

```
$ cd myproject
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = ssh://git@bitbucket.example.com:7999/hpim/myproject.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
[branch "B-SPRINT-10"]
        remote = origin
        merge = refs/heads/B-SPRINT-10
```

Git: Was versionieren?

- In der Datei .gitignore im Projektverzeichnis kann man Muster für Dateien angeben, die nicht versioniert werden dürfen wie *.o oder *.class.
- Beispiel:

```
$ cat .gitignore
*.patch
bin/logs
*.jar
# Intellij
.idea/
*.iml
*.iws
# Mac
.DS_Store
```

Clients

- CLI (automatisierbar, mächtig, flache Lernkurve)
- GUI (schneller Einstieg, bei komplexeren Problemen nicht nutzbar oder Nutzung komplizierter als in CLI)
- IDE (sehr bequem, Konfiguration evtl. komplizierter, oft nur eingeschränkt nutzbar)

Beispiele GUI Clients:

- Sourcetree https://www.sourcetreeapp.com/
- TortoiseGit https://tortoisegit.org
- SmartGit https://www.syntevo.com/smartgit/

Empfehlung: Einstieg mit Kommandozeile!

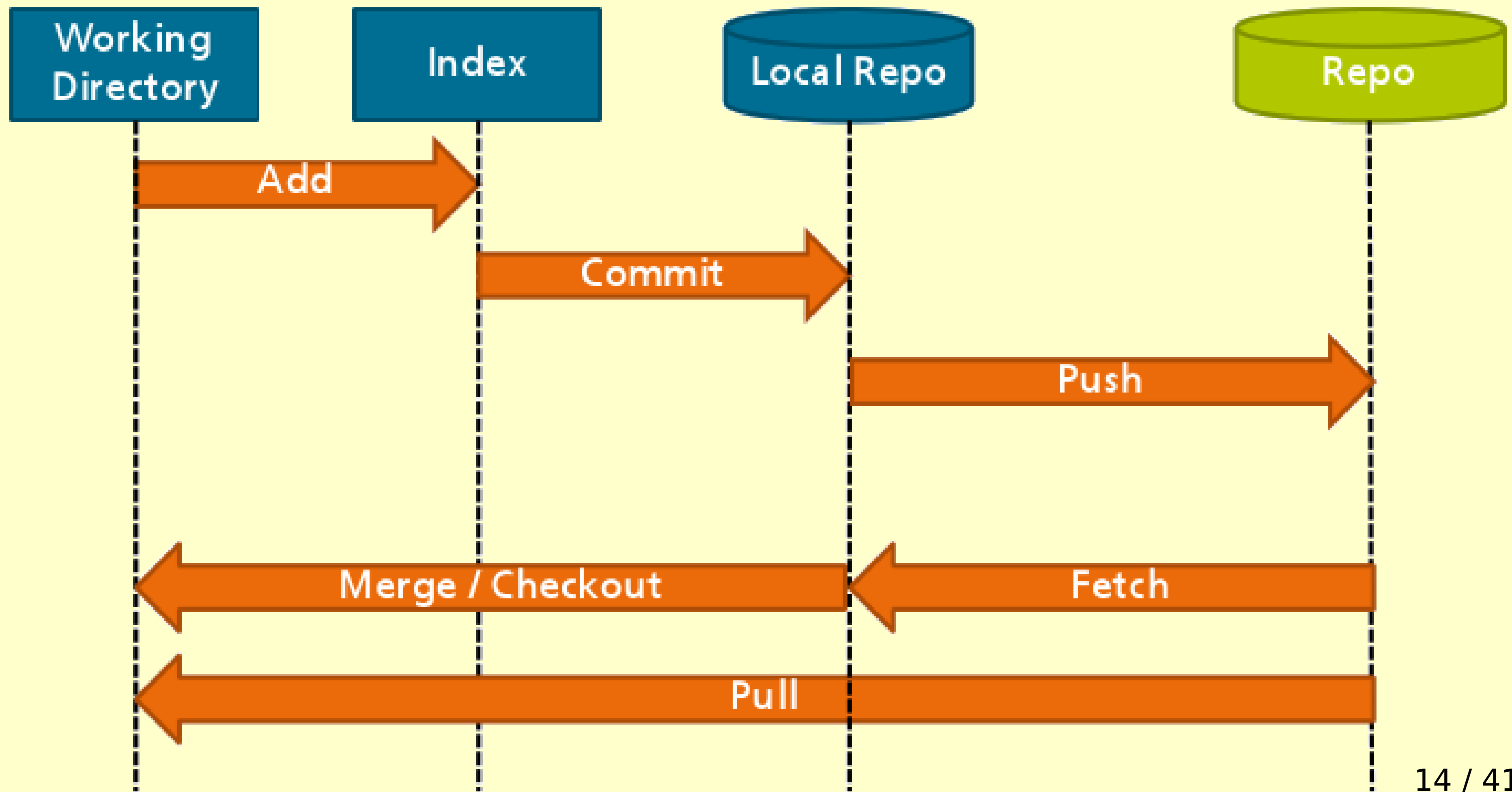
Erstes eigenes Repo (lokal) (1)

```
$ mkdir test; cd test

$ git init

Initialized empty Git repository in /home/johndoe/test/.git/

$ touch test

$ git add test

$ git commit

$ git status

On branch master
nothing to commit, working directory clean
```

Erstes eigenes Repo (lokal) (2)

```
$ git log

commit f982ae0f7708677392472519c1a28df0de1ce7a7
Author: John Doe <johndoe@localhost>
Date:   Sun Dec 03 13:18:02 2017 +0100
    1st commit

$ touch test2

$ git add test2

$ git status

On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   test2
```

Commit und Co: Wohin?



Clone Remote Repo

- Empfehlung: ssh mit Keys nutzen (mit oder ohne Passphrase)
- Erzeugung Key: ssh-keygen -t rsa -b 4096 *(mehrmals Return)*
- Public Key aus ~/.ssh/id_rsa.pub im Remote-Repo hinterlegen

  (evtl. mit Filter '*from="10.0.0.1,10.0.0.2"* ssh-rsa AAA...')
- $ **git clone ssh://git@git.example.com:7999/pim/project.git**

  ```
  echo -e 'Host git.example.com\n  KexAlgorithms +diffie-hellman-group1-sha1' >>.ssh/config
  ```
- SSH-Protokoll incl. Key-Authentifizierung wird von praktisch allen Diensten und jeder git-Server-Software unterstützt.
- Bei Verwendung einer Passphrase für den Schlüssel kann ein ssh-agent mit einem definierten Timeout zur Speicherung der Passphrase verwendet werden.

Exkurs: man-pages

- „man“ ist der Befehl zum Anzeigen von Hilfeseiten im ROFF-Format
- Der git-Befehl wird mit „-“ angehängt, z.B. „man git-checkout“
- Tastatursteuerung:
- „G“ bzw. „p“: Ende bzw. Anfang der man-page
- Pfeil ab bzw. auf: Eine Zeile runter/rauf rollen
- SPACE bzw. „b“: Seite vor-/rückwärts
- „/“ bzw. „?“: Vor- bzw. Rückwärtssuche
- „n“ bzw. „N“: Letzte Suche in gleicher/umgekehrter Richtung
- „q“: Verlassen
- „man man“ zeigt die Hilfe zu „man“ an

Git checkout

- **Schaltet auf einen (lokal vorhandenen) Branch um.**
- Vereinfachte Syntax:

  git checkout [-f] [-m] [<branch>]

  git checkout [-f] -b <new_branch>
- „-b new“ ist identisch mit: „git branch new; git checkout new“
- „-f“ steht für „force“ zum Ignorieren von Warnungen
- „git checkout“ ohne neuen Branch setzt den aktuellen Branch zurück
- „-m“ macht einen 3-Way-Merge vom alten in den neuen Branch (eher selten genutzt)
- „git checkout -b local_5.1 -t origin/5.1.x“ (setzt Tracking)

Git add/rm

- **Fügt Dateien zum Index/Staging Area hinzu bzw. entfernt sie**
- Vereinfachte Syntax:
- git add [-n] [-f] file
- git rm [-n] [-f] [-r] file
- „-n“ / „--dry-run“: Trockenlauf, treten Fehler auf?
- „-f“: Ignorieren von Fehlermeldungen, „forciere“ den Befehl
- „rm -r“: Erlaube rekursives Entfernen, wenn Verzeichnis

Git commit

- **Speichert Änderungen aus dem Index/Staging Area im (lokalen) Repo**
- Vereinfachte Syntax:

  git commit [-a] [-m MESSAGE] [-F file]
- „-a“: Dateien automatisch entfernen/hinzufügen (s.u.)
- „-m msg“: Kommentar/Beschreibung angeben
- „-F file“: Kommentar aus Datei „file“ lesen
- Commit wird auch nach „Merge“ und Behebung von Konflikten genutzt
- Auch genutzt nach Cherry-Picking und Konflikten

Git commit -a

- Beispiel:

  $ echo 1 >test

  $ git status

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
new file:   test2

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
working directory)
modified:   test
```

- „test“ wird ohne „-a“ bei commit nicht „mitgenommen“

Git pull

- **Aktualisiert aktuellen Branch mit Änderungen von Remote**
- Im einfachen Modus entspricht es „git fetch; git merge FETCH_HEAD“
- Mit Option „--rebase“ wird statt „git merge“ ein „git rebase“ gemacht.
- Beispiel, vorher:

```
      A---B---C master on origin
     /
D---E---F---G master
    ^
    origin/master in your repository
```

- Nachher:

```
      A---B---C origin/master    mit rebase:   A'--B'--C' master
     /         \                              /
D---E---F---G---H master         D---E---F---G
```

Git push

- **Aktualisiert Remote von lokal**
- Wichtige Option:
- „--tags“: aktualisiert auch Tags
- „-u“ / „--set-upstream“: Setze eine Upstream/Tracking Reference
- Beispiel dafür: Man hat lokal einen neuen Branch angelegt mittels „git checkout -b newbranch“. Wenn man nun versucht, diesen zu „pushen“, gibt es eine Fehlermeldung, dass es remote keinen dazugehörigen Branch gibt. Beispielhafte Behebung des Problems:
- git push --set-upstream origin mynewbranch
- Dieser Hinweis wird von git aber bei der Fehlermeldung mit ausgegeben!

Git fetch

- **Aktualisiert das lokale Repo von Remote**
- Tatsächlich ist „fetch“ das „Gegenteil“ von „push“, da hier in umgekehrter Richtung von „push“ aktualisiert wird – siehe Grafik auf Seite „Commit und Co“.
- „pull“ dagegen macht noch mehr, da es auch die Dateien im Dateisystem passend zum frisch aktualisierten Branch anpasst.
- Fetch kann wie auch push oder pull auf beliebige Remote Repos gemacht werden. Sollten also mehrere Remote Repos konfiguriert sein, so kann man sie mit allen drei Befehlen adressieren.

Git log

- **Zeige Commit-Logs an**
- Beispiele:
- „git log -2“: Zeige zwei letze Commits an
- „git log --pretty='%H %an %s' “: Zeige Commits zeilenweise mit Hash, Author und Comment an
- Beispiel mit Tags und Baumansicht:

  git log --decorate=short --graph --oneline --simplify-by-decoration
- Die 20. letzten Tags:

  git log --pretty='%h %d %s' |grep "tag: "|head -n 20

Git log - weitere Beispiele

```
$ git log --graph --all --pretty=format:'%C(auto)%h%C(auto)%d %s %C(dim white)(%aN, %ar)'
```

```
$ git log --graph --color --date-order --date=local --format="%C(auto)%h%Creset %C(blue bold)%ad%Creset %C(auto)%d%Creset %s"
```

```
$ git log --graph --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n''          %C(white)%s%C(reset) %C(bold white)—  %an%C(reset)' --abbrev-commit
```

```
$ git log --graph --pretty=format:"%x09%h | %<(10,trunc)%cd |%<(25,trunc)%d | %s" --date=short
```

```
$ git log --graph --pretty=format:'%Cred%h%Creset %ad %s %C(yellow)%d%Creset %C(bold blue)<%an>%Creset' --date=short
```

```
$ git log --decorate --graph --abbrev-commit --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset) %C(bold cyan)(committed: %cD)%C(reset) %C(auto)%d%C(reset)%n''          %C(white)%s%C(reset)%n''          %C(dim white)- %an <%ae> %C(reset) %C(dim white)(committer: %cn <%ce>)%C(reset)'
```

```
$ git log --decorate --graph --abbrev-commit --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(auto)%d%C(reset)%n'' %C(white)%s%C(reset) %C(dim white)- %an%C(reset)'
```

```
$ git log --decorate=short --graph --oneline --simplify-by-decoration --branches
```

```
$ git log --graph --pretty=oneline --abbrev-commit
```

```
$ git log --graph --oneline --branches
```

Git status

- **Status-Anzeige**
- Zeigt Dateien mit Unterschieden zwischen Index und HEAD
- Zeigt Dateien mit Unterschieden zwischen Dateisystem und Index
- Zeige Dateien im Dateisystem an, die nicht „tracked“ sind / nicht im Git erfasst sind
- Zeigt auch „Schwebezustand“ an wie beispielsweise ein Merge, der wegen Konflikten nicht beendet werden konnte
- Zeigt auch Schwebezustand bei Cherry-Picking an
- **Zeigt für den aktuellen Zustand auch mögliche Lösungsmöglichkeiten**

Git diff

- **Zeigt Unterschiede z.B. zwischen Commits an**
- Beispiel, unser lokales Repo:

```
$ git commit -am 'zweiter Commit'

$ git log --pretty='%H %an %s'

6584e1c413298b0cd466fe954c800a9f86ac3519 John Doe zweiter Commit
f982ae0f770867739247251 9c1a28df0de1ce7a7 John Doe 1st commit

$ git diff f982ae 6584e1

diff --git a/test b/test
index e69de29..d00491f 100644
--- a/test
+++ b/test
@@ -0,0 +1 @@
+1
```

- Die „1“ wurde eben mit „echo“ in die Datei geschrieben.

Git diff anderer Branch

- "git diff FROM TO" bzw. "git diff FROM..TO" funktioniert nicht nur mit Commits, sondern auch mit Branches. Hier wird dann jeweils HEAD des Branches genommen. Ebenso können Tags verwendet werden.
- Beispiel, Unterschied README zwischen aktuellem Branch und master:

```
$ git diff ..master -- README
```

- Zwischen develop und master:

```
$ git diff develop master README ChangeLog
```

- Der Trenner "--" vor dem/den Dateinamen ist meist nicht notwendig.

Git merge

- **Zusammenführen von Änderungen (zweier oder mehrerer Entwicklungs-Historien)**
- Beispiel:
- „git merge newbranch“: Fügt die Änderungen von newbranch dem aktuellen Branch hinzu
- Bei Merge-Konflikten wird eine entsprechende Meldung ausgegeben. Ebenso gibt „git status“ jederzeit Auskunft über den aktuellen Zustand.
- Nach Behebung der Merge-Konflikte müssen die betroffenen Dateien mit „git add“ markiert werden (bei anderen nicht nötig).
- Danach muss der Merge mittels „git commit“ finalisiert werden.
- Algorithmus wird von git bestimmt

Merge-Konflikte

- Beispiel einer Datei, die bei der ein automatischer Merge fehlgeschlagen ist und git Markierungen eingefügt hat:

```
If you have questions, please
<<<<<<< HEAD
open an issue
=======
ask your question in IRC.
>>>>>>> branch-a
```

- Ist branch-a gültig: Zeilen zwischen „<<<“ und „===“ löschen plus die Zeile „>>>“. Dann speichern, mit „add“ markieren und „commit“.
- Achtung: Es können mehrere Merge-Konflikte in einer Datei vorhanden sein! Vorsicht bei unterschiedlichen Zeilenenden CR/LF!
- git-scm.com/book/de/v1/Git-Branching-Einfaches-Branching-und-Merging

Git branch

- **Anzeigen, Anlegen oder Löschen von Branches**
- „-a“: Alle Branches, lokal und remote
- „-d“: Branch löschen (eher selten benötigt!)
- Beispiel anlegen (neuer Branch ist „billig“, 250 Bytes):

```
$ git branch newbranch
$ git branch -a
* master
  newbranch
$ git checkout newbranch
Switched to branch 'newbranch'
$ git branch -a
  master
* newbranch
```

- Kurzform: „git checkout -b newbranch“

Git tag

- **Anzeigen, Anlegen oder Löschen von Tags**
- Es gibt einfache Tags (Pointer auf Commits) und kommentierte (annotated) Tags.
- Letztere sind ein Objekt in Git mit Prüfsumme, Namen und E-Mail-Adresse, Datum und Kommentar.
- Annotated Tags sind generell empfohlen. Sie können auch signiert werden.
- Beispiel:
- git tag -a PIM-5.3.2.1 -m "PIM-5.3.2.1, Hotfix vom 2017-12-05" -f
- Quizfrage: Warum wurde hier „-f“ verwendet?

Git stash und apply/pop

- **„Stash“ „bunkert“ aktuelle Änderungen und „pop“ holt sie zurück**
- Beispiel:
- Man arbeitet auf einem Branch und hat auch Änderungen vorgenommen, die man aber noch nicht „committen“ möchte. Aber dann muss man den Branch wechseln. Um die lokalen Änderungen zwischenzuspeichern, nutzt man „stash“. Wenn man dann vom anderen Branch zurückgekehrt ist, kann man die Änderungen mit „stash pop“ oder „stash apply“ wieder zurückholen.

Git cherry-pick

- **Einzelne und bereits vorhandene Commits anwenden**
- Beispiele:
- In einem anderen Branch ist ein Bug bereits behoben worden. Man möchte aber nur diesen spezifischen Commit des Bugfixes anwenden, weil ein Merge zu dem Zeitpunkt noch nicht stattfinden darf.
- In einem anderen Repo (z.B. ein Fork der Software) ist ein Fehler behoben worden. Wenn das Repo als zusätzliches Remote-Repo hinzugefügt worden ist, können auch dessen Commits anhand ihrer Hashes einzeln herausgepickt werden können.

Git reset

- **Setze HEAD auf den angegebenen Status**
- „git reset“: Setzt Index-Einträge zurück. Gegenteil zu „add“.
- „git reset --hard HEAD^^^“: Setzt HEAD 3 Commits zurück.

Git - weitere Befehle

- bisect: Hilft bei iterativer Suche nach einem Commit mit Bug
- blame: Listet pro Zeile einer Datei den Commit dazu
- gc: Garbage Collector, löscht nicht benötigte Dateien und optimiert das lokale Repository (siehe Git reflog)
- submodule: Initialisieren, aktualisieren oder untersuchen von Submodules ("Unter-Repositories")

Git-Flow Konventionen

- Git-Flow ist eine Konvention zur Nutzung von Branches.
- Erstmals dokumentiert durch Vincent Driessen:
  http://nvie.com/posts/a-successful-git-branching-model/
- Mindestens fünf Arten von Branches:
- master: enthält stets die zuletzt releaste Version
- develop: enthält aktuelle Entwicklungsversion
- feature/topic branches: zur Entwicklung individueller Features
- hotfix branches: zur Implementierung dringender Bugfixes
- release branches: zum Vorbereiten eines Releases

Git stash/workspace/index/staging/repo

GIT CHEATSHEET
AN INTERACTION FROM NDP SOFTWARE

previous version»
escape a git mess, step-by-step»
discover character entities at &what»

en fr 简 es de

STASH
WORKSPACE
INDEX
LOCAL REPOSITORY
UPSTREAM REPOSITORY

diff <commit or branch>
commit -a [-m 'msg']
reset --soft HEAD^
reset --hard
checkout <branch>
checkout -b <name of new branch>
merge <commit or branch>
rebase <upstream>
cherry-pick <commit>
revert <commit>
diff --cached [<commit>]
commit [-m 'msg']
commit --amend
log
diff <commit> <commit>
branch
branch -d <branch>
branch --track <new>

git commit [-m 'msg'] Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

(c) Andrew Peterson 2009-2016 All Rights Reserved.

Git cheatsheet: http://ndpsoftware.com/git-cheatsheet.html

Undoing all kinds of mistakes

- Ein nützlicher Artikel über häufige Fehler, wie man sie behebt und sie vermeidet:

  https://git.seveas.net/undoing-all-kinds-of-mistakes.html

- Wichtig ist hier die Erklärung "Stop tracking a file": Wenn man eine Datei einem Repository hinzugefügt hat, die Datei aber nicht mehr "tracken" will, so sollte man kein einfaches "rm" machen.

Links

- https://git-scm.com/book/de/v1/Git-Branching-Einfaches-Branching-und-Merging
- https://git-scm.com/book/de/v1/Git-Branching-Rebasing
- https://git-scm.com/book/de/v1/Git-Interna-Git-Objekte
- http://nvie.com/posts/a-successful-git-branching-model/
- http://ndpsoftware.com/git-cheatsheet.html
- https://stackoverflow.com/questions/5543280/how-do-i-get-the-deleted-branch-back-in-git
- https://git.seveas.net/undoing-all-kinds-of-mistakes.html

Vielen Dank für‘s Zuhören.

Noch Fragen?
Auch per E-Mail: git@tuxad.com