Versionsverwaltung

5. Klasse TFO Brixen

Michael Mutschlechner

Wofür?

„Häh, das hat doch vorher funktioniert!“

- Nachvollziehbarkeit
  - Wie und wann hat sich ein Fehler ins System eingeschlichen?
- Entwicklerkoordinierung
  - Wie können mehrere Entwickler gleichzeitig an der Codebasis arbeiten?
- Releasemanagement & Archivierung
  - Wie markiere einen bestimmten Zustand als Version?
- Wiederherstellbarkeit
  - Wie bekomme ich mein Filesystem in einen alten bugfreien Zustand?

Snapshot-basiertes Versionsverwaltungssystem

- Stellt euch vor, es gäbe keine Versionsverwaltungssysteme und ihr würdet ohne Versionskontrolle an einem Projekt arbeiten.
- Um trotzdem Zwischenstände eurer Arbeit sichern zu können, bietet es sich an, den Projektordner zwischendurch einfach immer wieder mal zu kopieren:

```
> cp -a mein-projekt mein-projekt-20130507-1050 # Erste Sicherung
> # Jetzt wird an "mein-projekt" gearbeitet
> cp -a mein-projekt mein-projekt-20130508-0034 # Das Feature ist fertig und funktioniert, zweite Sicherung
```

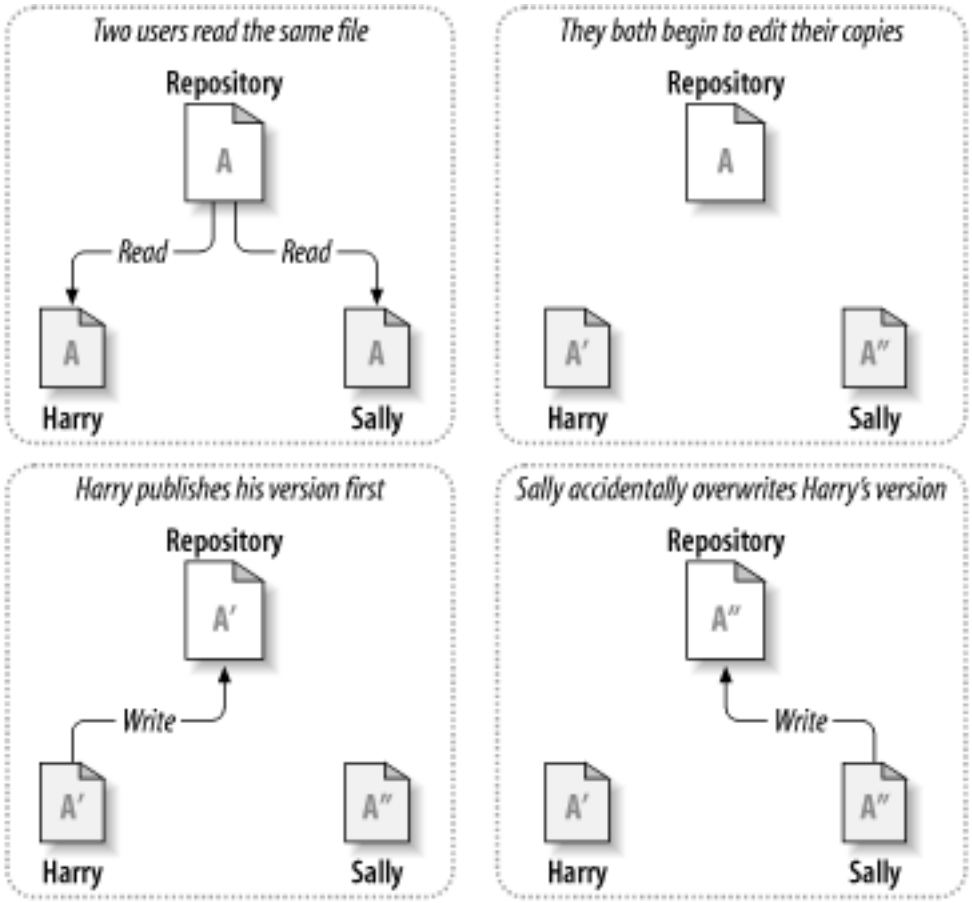
- Auf diese Weise könnt Ihr sogar Änderungen zwischen euren Versionsständen nachvollziehen, indem ihr einfach den Unterschied zwischen beiden Verzeichnissen ermittelt. Das Unix-Tool „diff“ liefert hier beispielsweise gute Dienste:

```
> diff mein-projekt-20130508-0034 mein-projekt
```

Eigenstudium

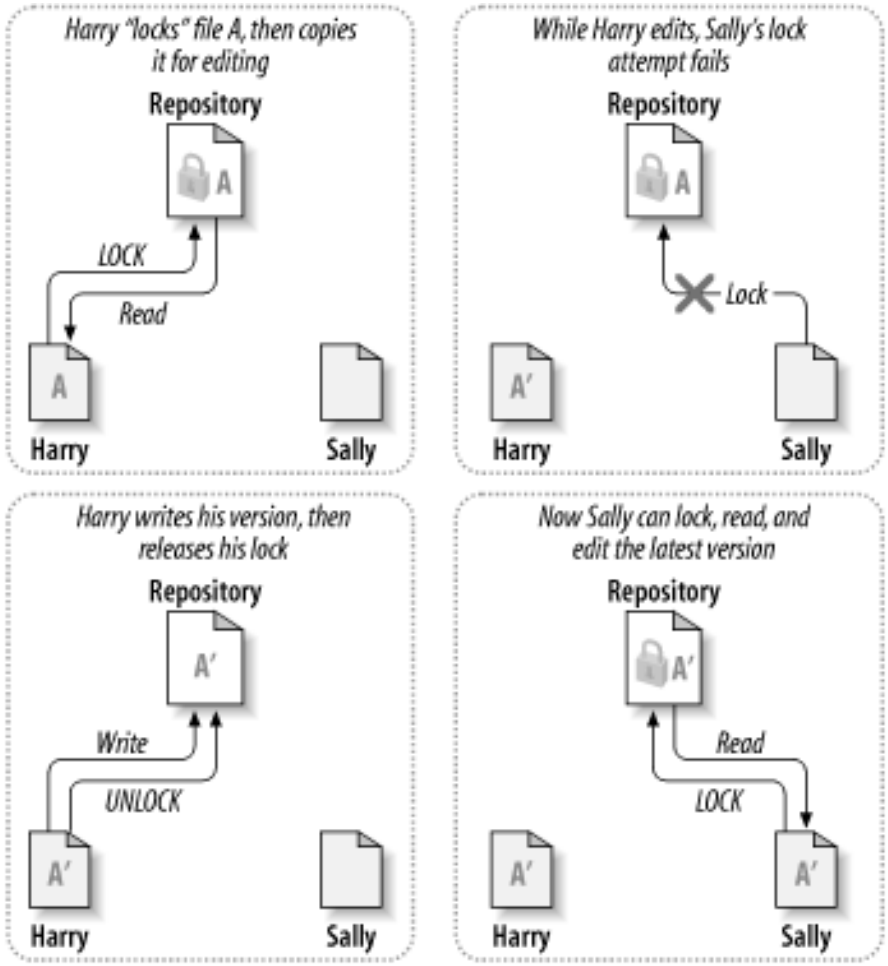
- http://www.computerhope.com/unix/udiff.htm

Das Problem der Entwicklung im Team



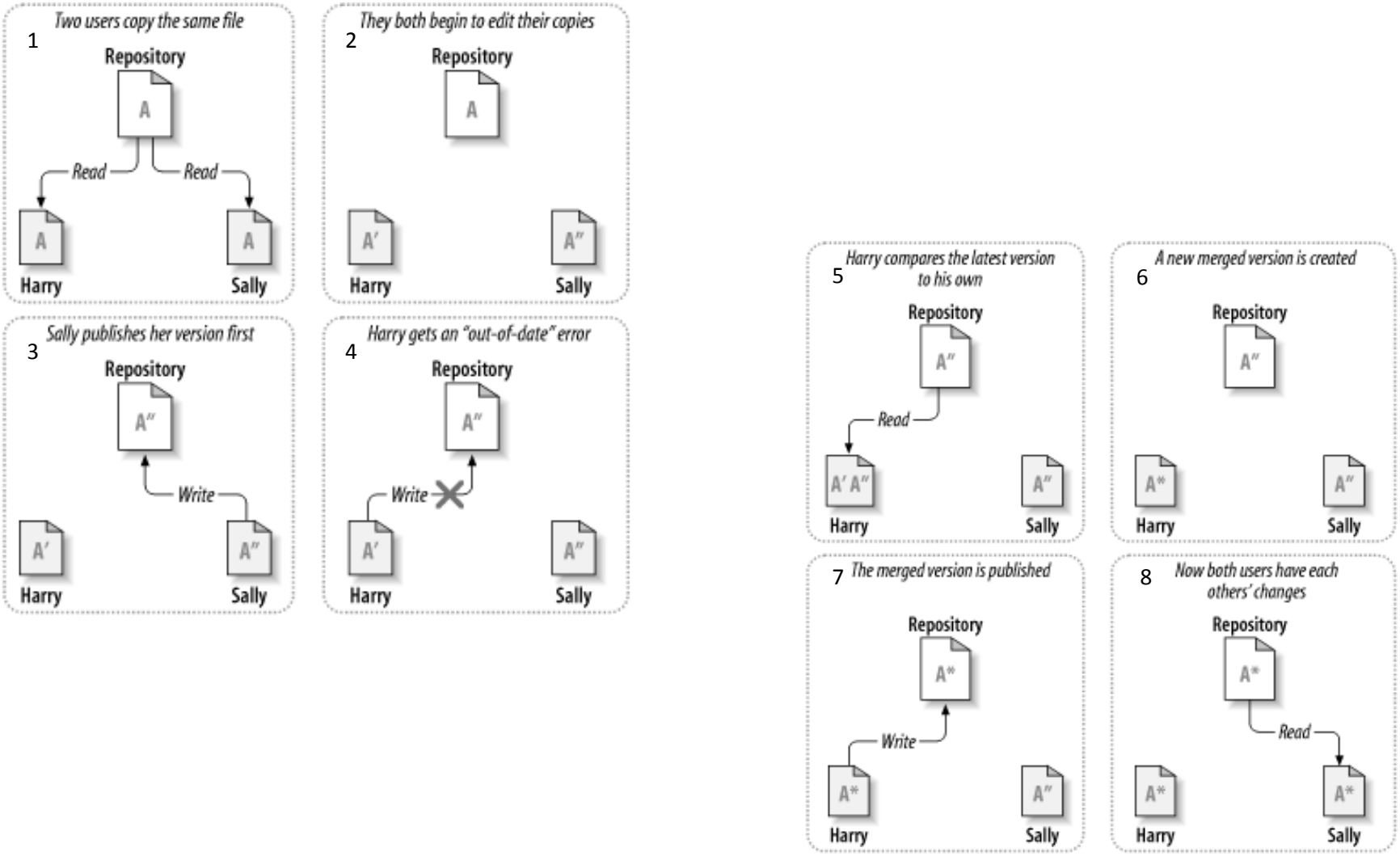
Lösungsansatz 1/2

- **Pessimistisches Locking**



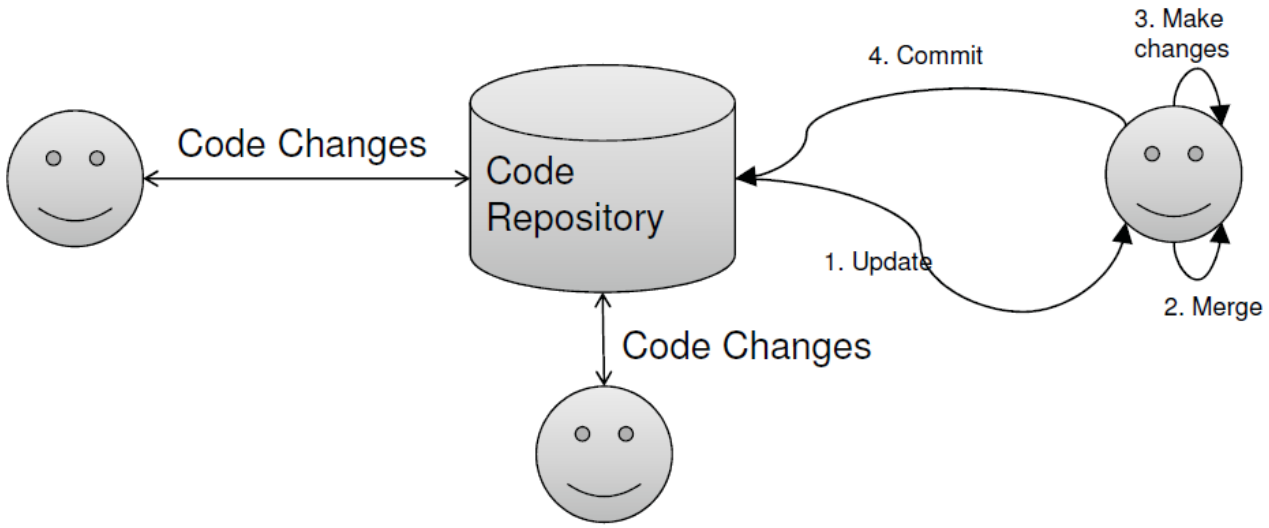
- Umständlich,
- Kein paralleles Arbeiten möglich!

Lösungsansatz 2/2



Arten von Version Control Systemen

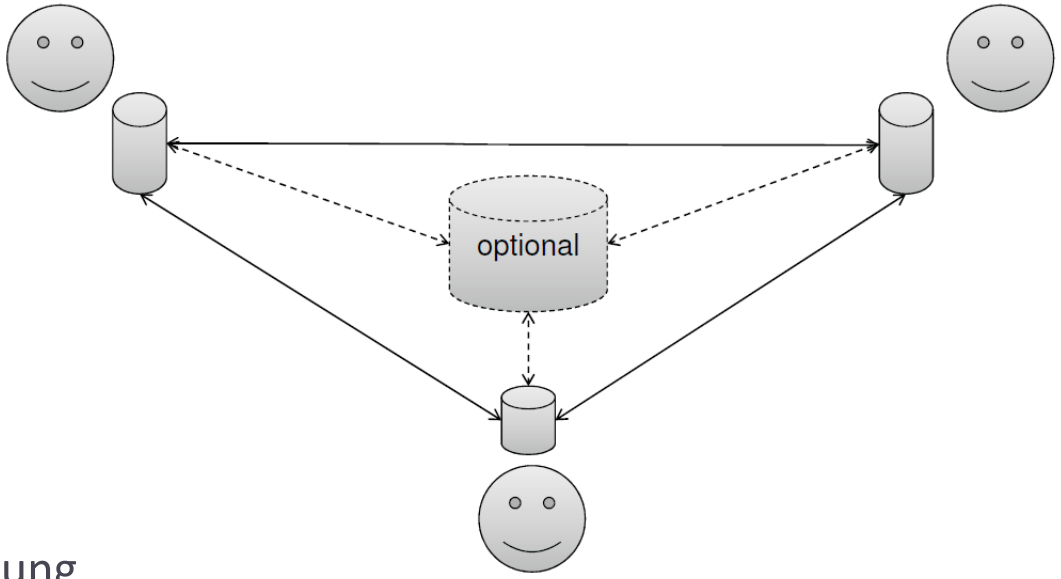
- **Zentrales VCS (z.B. CVS & SVN)**



- Vorteile:
  - Einfach
  - Jeder weiß mehr oder weniger darüber Bescheid, was andere, an einem Projekt Beteiligte gerade tun
  - Administratoren haben die Möglichkeit, detailliert festzulegen, wer was tun kann

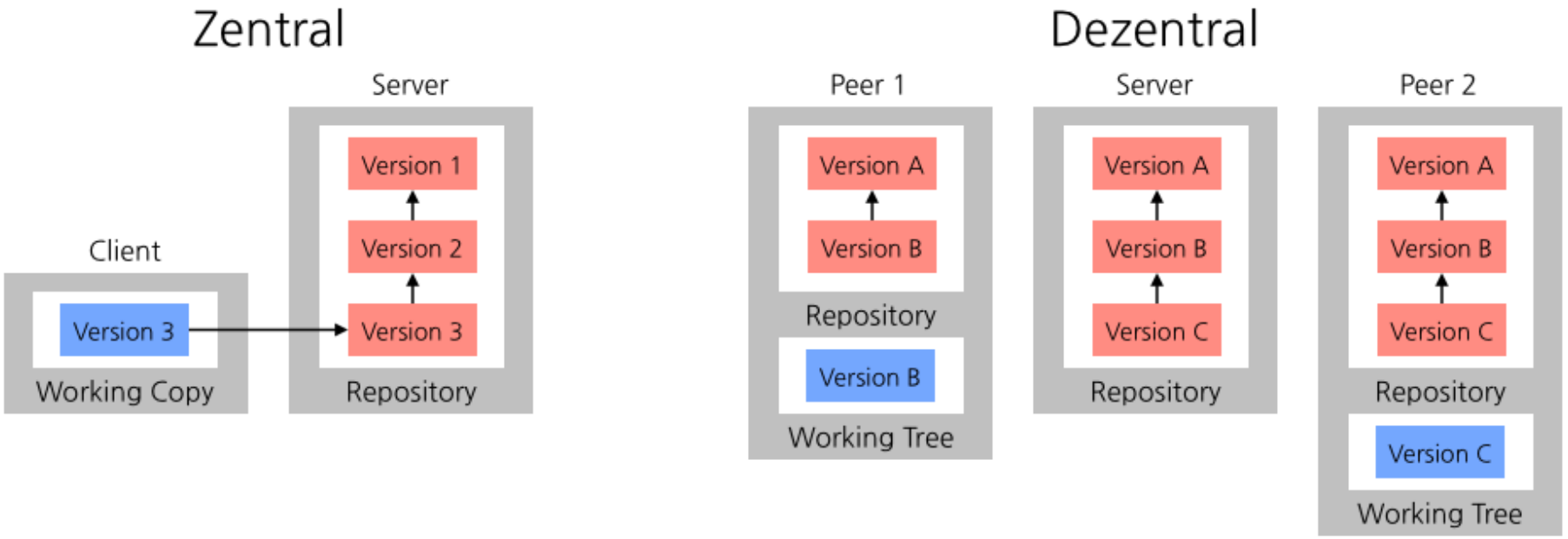
- Nachteile:
  - Single Point of Failure
  - Repository muss online sein

- **Dezentrales VCS (z.B. Git)**



- Vorteile:
  - Viele lokale Commits
  - Redundante Archivierung
  - Offline Arbeit möglich
- Nachteile:
  - Komplexer

Zentral vs. dezentral



Terminologie

- Commit Operation
  - Fügt Änderungen zum Repository hinzu
- Merge Operation
  - Mischt verschiedene Versionen
- Branch
  - Entwicklungszweig
- Tag
  - Markiert einen bestimmten Status, z.B. als Release Version

Tools

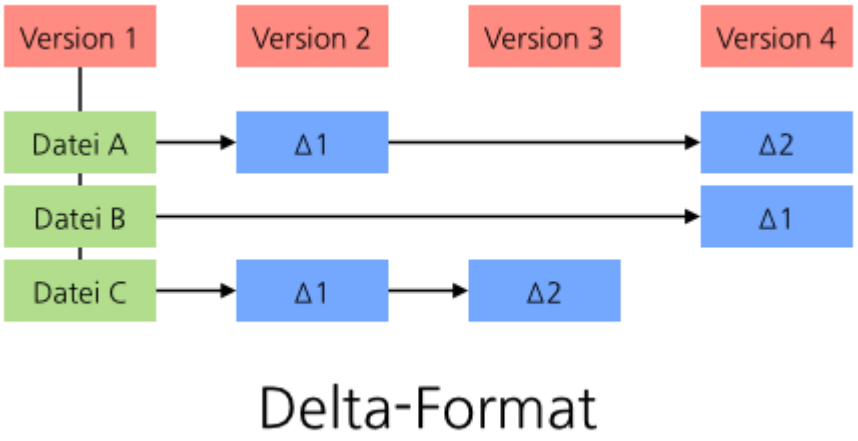
- Umfang
  - Interaktion mit Repositories (Commit, Branch, Tag, Merge,..)
  - Vergleich von Historie
  - Rollback, Sprung zwischen Branches
- Tool Arten
  - Command-line Tools
  - Integration mit IDEs
    - Mit Vorsicht zu genießen, da oft nicht so gut entwickelt wie Command-line Tools
  - Desktop Clients
  - Web-Apps wie GitHub, Trac,...
    - Bieten auch noch Bug-Tracking, etc.

Kleine Geschichte Populärer VCS

- Lokale VCS, z.B. SCCP (1972)
- CVS (1989 - 2008)
  - Entwicklungsstart 1989
  - In vielen OS Projekten eingesetzt
  - Wird nicht mehr weiterentwickelt
- SVN (2000 - ?)
  - Nachfolger von CVS
    - Sollte Fehler beheben
  - Eigene Schwächen
- GIT (2005 - ?)
  - Eigenentwicklung von Linux Erfinder Linus Torvald
  - Wird heute zur Linux Kernel Entwicklung eingesetzt

Concurrent Versions System (CVS)

- Zentrales Repository
  - Speichert Deltas
    - Nur Änderungen der Dateien werden gespeichert
- Umbenennungen und Dateibewegungen werden nicht versioniert
- Umgang mit Verzeichnissen kompliziert



Delta-Format

Subversion (SVN)

- Als Rewrite von CVS gedacht
- Schneller als CVS
  - Braucht aber mehr Speicher
- Gesamtversionsnummer „Revision“

Git

- Verteiltes VCS
- Lokale Commits
  - Performance Vorteile
  - Offline möglich
- Extrem populär durch GitHub: „social coding“

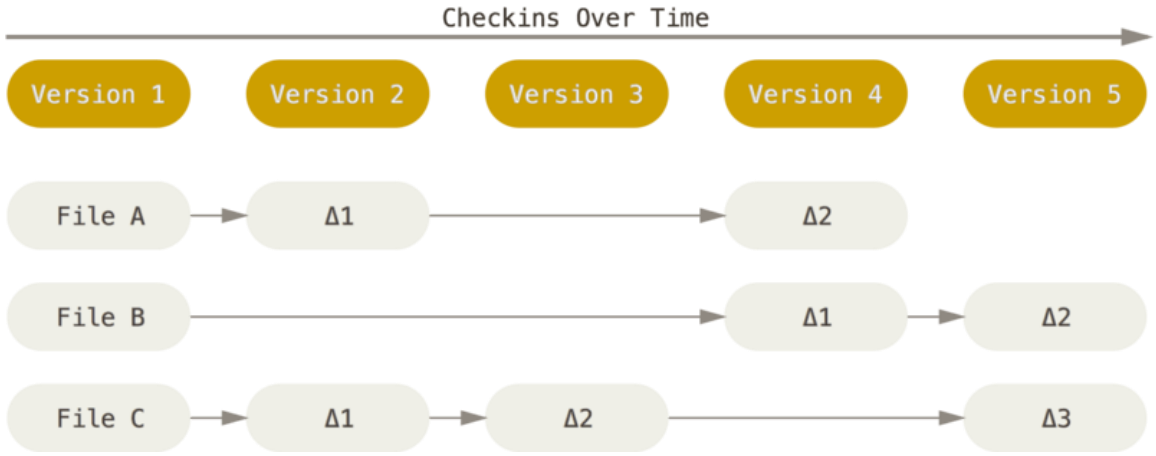
Git

- Git funktioniert nach genau demselben Prinzip wie das Beispiel auf Folie 3:
  - Ein Versionsstand in Git (in Git-Sprache „Commit“ genannt) ist ein vollständiger Schnappschuss des Projekts, in dem die vollständigen Inhalte aller Dateien gesichert werden. Um Speicherplatz zu sparen, speichert Git allerdings Dateien, die in mehreren Snapshots inhaltsgleich sind, nur einmal.

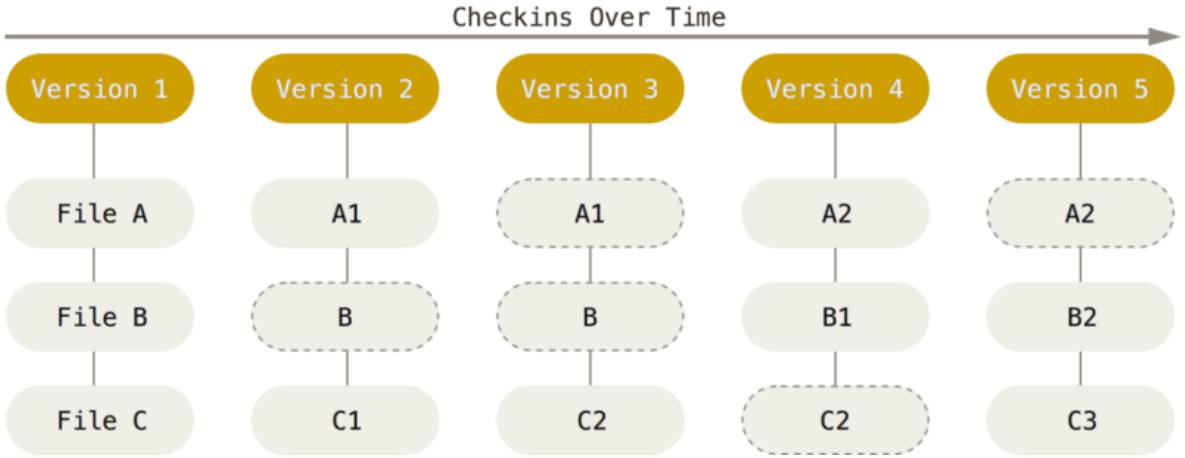
```
> cp -a mein-projekt mein-projekt-20130507-1050 # Erste Sicherung
> # Jetzt wird an "mein-projekt" gearbeitet
> cp -a mein-projekt mein-projekt-20130508-0034 # Das Feature ist
fertig und funktioniert, zweite Sicherung
```

GIT Versioning

- **CVS/SVN:**
  - Änderungen auf File-basis gespeichert



- **Git:**
  - Snapshots vom gesamten Filesystem pro Version



Snapshots

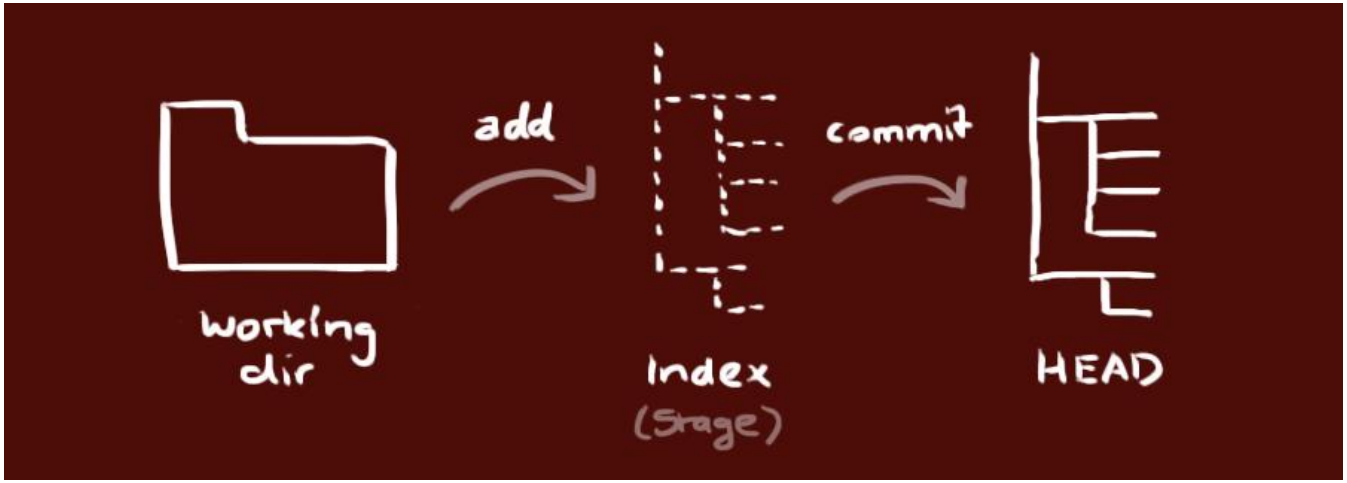
- Jedes Mal, wenn der gegenwärtige Status des Projektes als eine Version (in Git) gespeichert wird, sichert das System den Zustand sämtlicher Dateien in diesem Moment („Snapshot“) und speichert eine Referenz auf diesen Snapshot.
- Um dies möglichst effizient und schnell tun zu können, kopiert das System unveränderte Dateien nicht, sondern legt lediglich eine Verknüpfung zu der vorherigen Version der Datei an.

GIT Dezentrales System

- Bei **dezentralen Systemen** wie Git gibt es genau genommen keinen Server.
- Die komplette Versionsgeschichte wird lokal beim Client gespeichert.
- Dieser kann seine Versionsgeschichte dann optional mit denen auf anderen Rechnern abgleichen.

GIT Funktionsweise

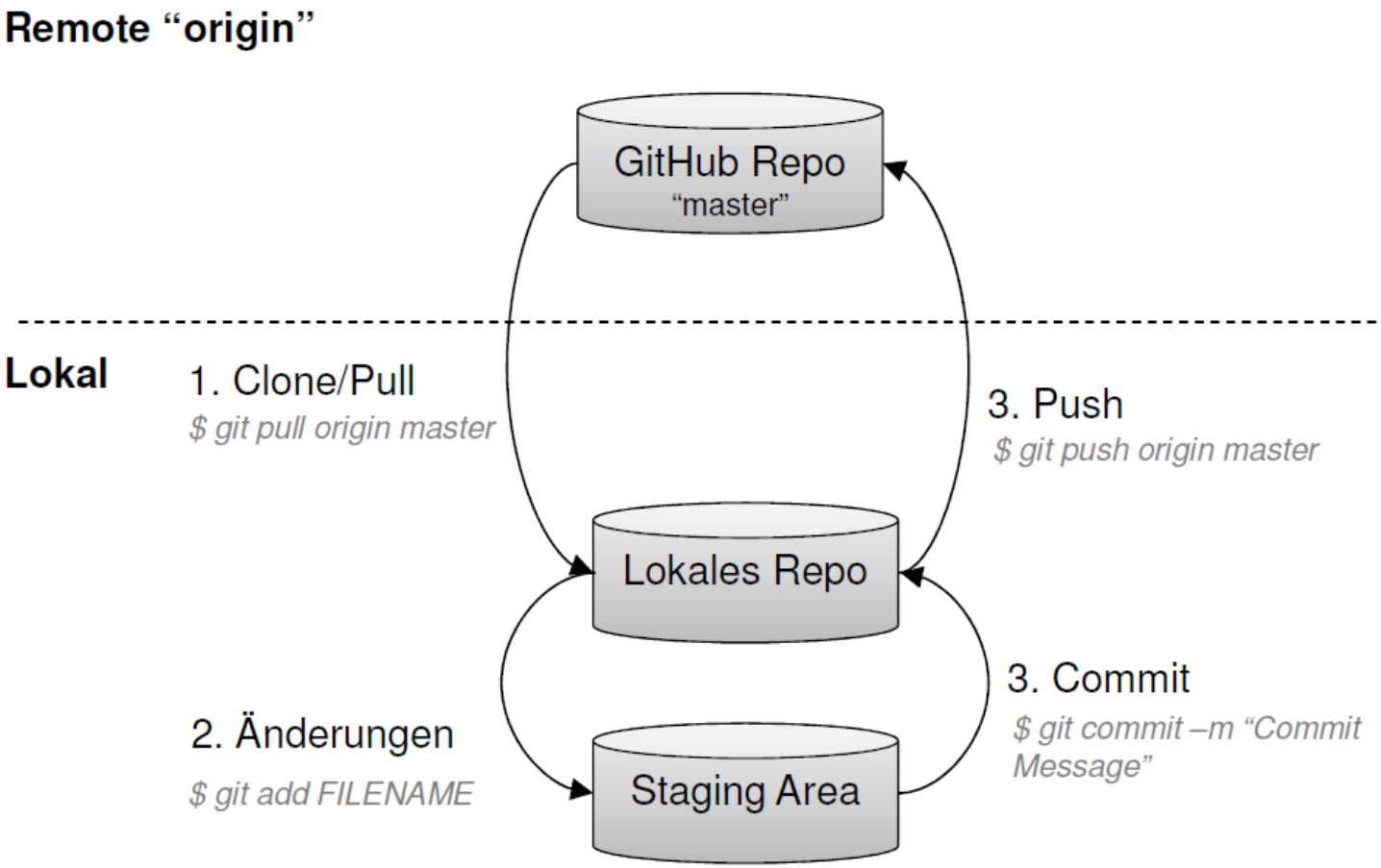
- Das lokales Repository besteht aus drei "Instanzen", die von git verwaltet werden.
  - Arbeitskopie
    - Enthält die echten Dateien
  - Index
    - Zwischenstufe
  - HEAD
    - Zeigt auf den letzten Commit



Spezielle Git Terminologie

- Commit
  - Speichert Änderungen im lokalen Repository
- Remote
  - Ein anderes Repository im Netzwerk
- Push
  - Lädt Änderungen auf ein remote Repository
  - Nur möglich wenn lokales und remote Repository vorher „gemerged“ wurden

GIT Zyklus


Remote "origin"
GitHub Repo
"master"
Lokal
1. Clone/Pull
$ git pull origin master
3. Push
$ git push origin master
Lokales Repo
3. Commit
$ git commit –m "Commit Message"
2. Änderungen
$ git add FILENAME
Staging Area

GIT Installation

- Installation unter Linux
  - Unter nahezu allen Linux-Distributionen einfach über die integrierte Paketverwaltung
  - Unter Ubuntu bzw. Debian: apt-get install git

- Konfiguration
  - Je nachdem, wie „git config“ aufgerufen wird, werden die Einstellungen entweder systemweit (unter Unix in der Datei /etc/gitconfig), für den jeweiligen Benutzer (im Benutzer-Verzeichnis unter ~/.gitconfig) oder pro Repository (dann in .git/config) gespeichert.
    - --global, -- system, --local
  - Git mitteilen, wer du überhaupt bist:

    > git config --global user.name 'Max Mustermann'

    > git config --global user.email 'm.mustermann@mittwald.de'
  - Einen alternativen Editor (anstelle von vi) nutzen:

    > git config --global core.editor 'vi'

  - Konfiguration kontrollieren

    > git config --list

GIT Erstellen von Repositories

- Ein neues Git-Repository kann mit dem Kommando „git init“ erstellt werden. Folgender Befehl erstellt ein neues Repository im aktuellen Verzeichnis:

```
> git init .
```

Ein Repository auschecken

- Um eine Arbeitskopie eines Repositories zu erstellen:

  > git clone /pfad/zum/repository

- Falls ein entferntes Repository verwendet wird:

  > git clone benutzername@host:/pfad/zum/repository (ssh)

  > git clone urlZumRepository (https)

- Anschließend können in dem Verzeichnis bereits existierende Dateien dem Repository hinzugefügt werden:

  > git add .

  > git commit –m 'Mein erster Commit.'

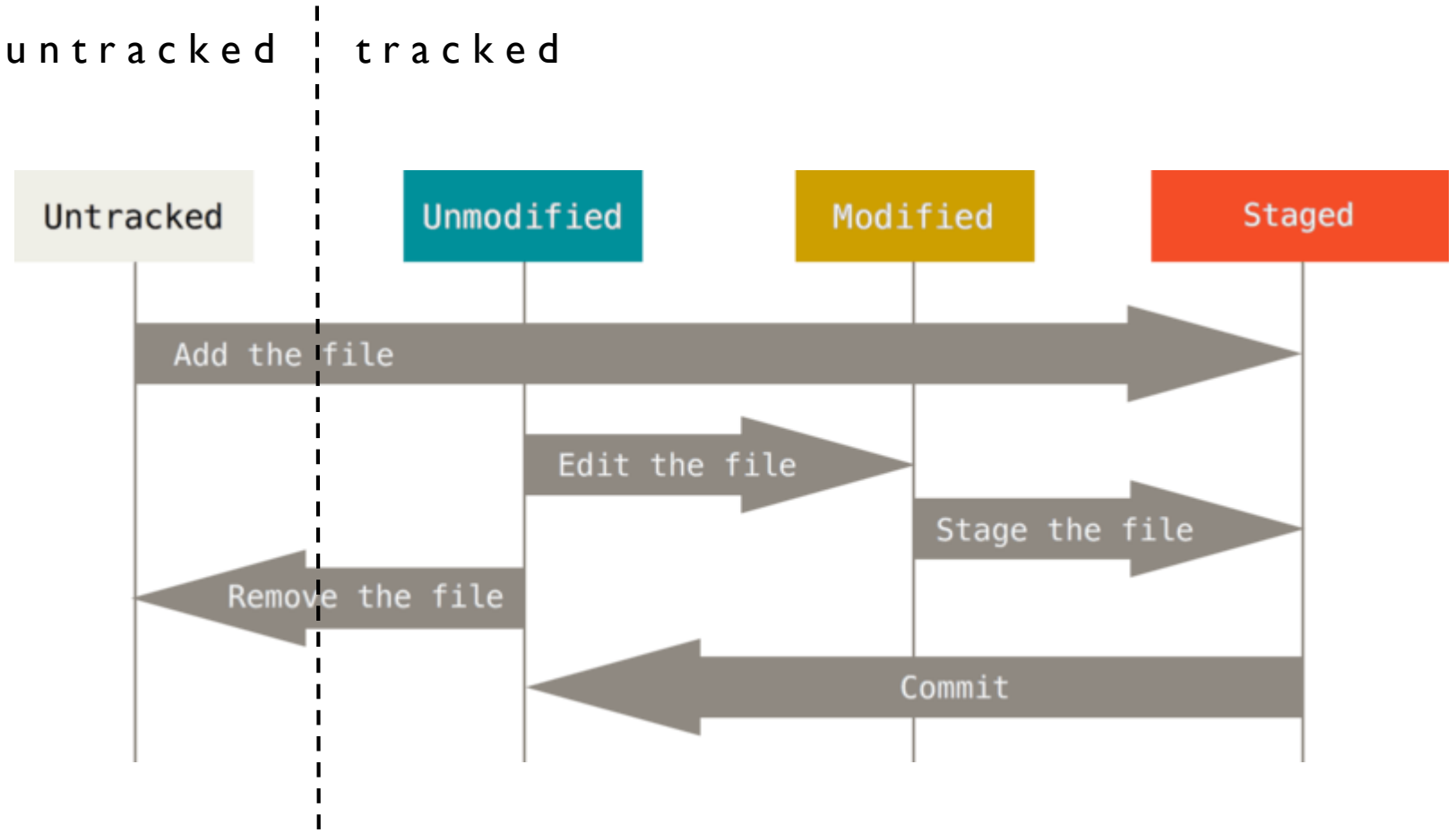
GIT Commits und Dateizustände

- Ein Git-Commit ist nichts anderes als ein Schnappschuss eines bestimmten Versionsstandes. Ein Commit zeichnet sich durch die folgenden Merkmale aus:
  - Ein Commit hat eine sogenannte Commit-ID. Die Commit-ID ist eine 160 Bit lange kryptografische Prüfsumme des aktuellen Zustands des Repositories.
  - Ein Commit hat in der Regel einen Vorgänger-Commit
    - „Parent“-Commit
  - Ein Commit hat eine Commit-Message, die beschreibt, was in dem jeweiligen Commit geändert wurde.
  - Ein Commit enthält außerdem ein Tree-Objekt, welches die im Repository enthaltenen Dateien beschreibt.

GIT Dateizustände

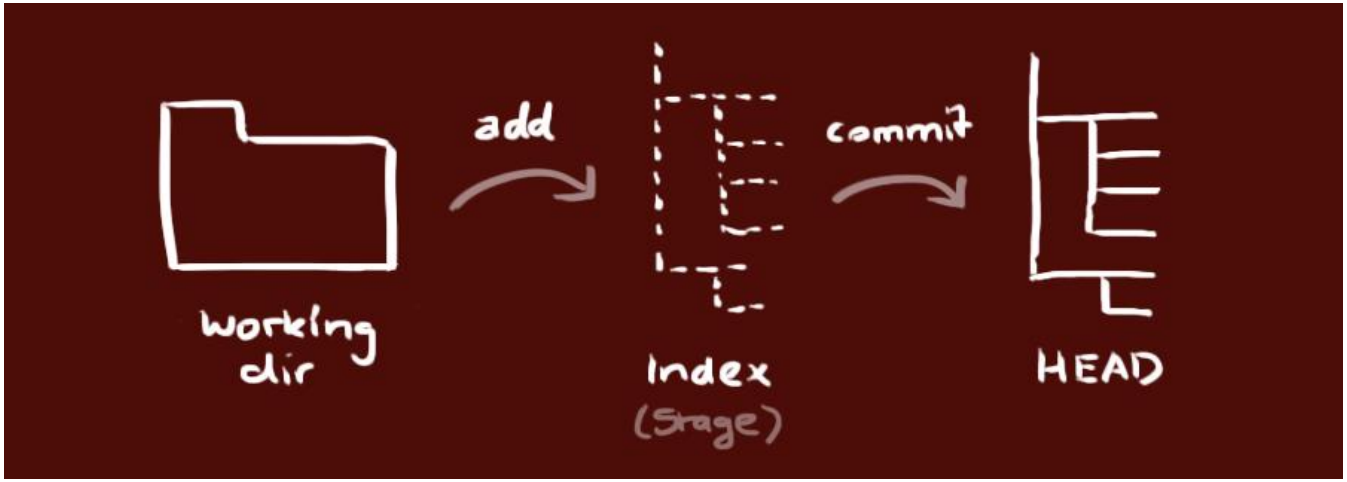
- Jede Datei in einem Git-Repository kann mehrere Zustände annehmen. Grundsätzlich unterscheidet Git zunächst zwischen „verfolgten“ (tracked) und „nicht verfolgten“ (untracked) Dateien. Alle Dateien, die bereits im letzten Commit enthalten waren, gelten automatisch als verfolgt. Jede verfolgte Datei kann wiederum einen der folgenden Zustände annehmen:
  - committed (unmodified)
    - in der lokalen Datenbank gesichert
  - verändert (modified)
    - Datei geändert, aber Änderung noch nicht committed
  - für den nächsten Commit vorgemerkt (staged)
    - geänderte Datei in ihrem gegenwärtigen Zustand für den nächsten Commit vorgemerkt

GIT Dateizustände

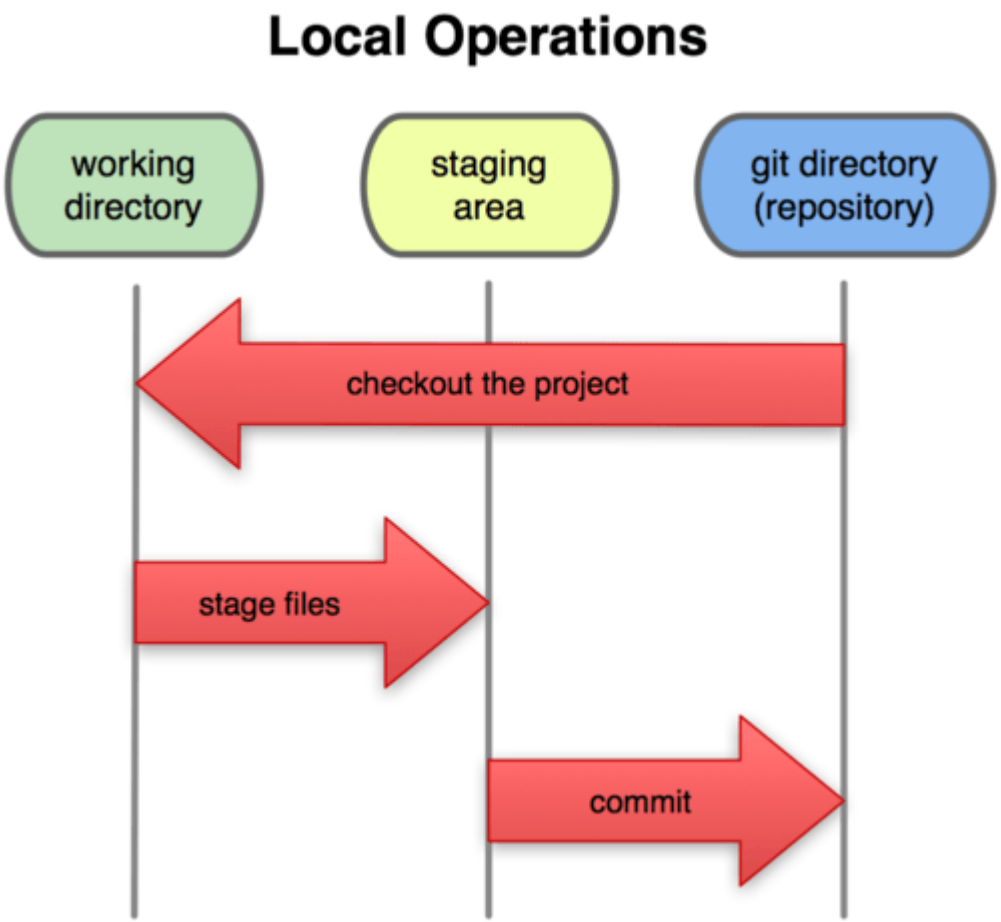


Verzeichnisse

- Das Git Verzeichnis ist der Ort, an dem Git Metadaten und die lokale Datenbank für ein Projekt gespeichert werden.
  - wichtigster Teil von Git
  - wird kopiert, wenn ein Repository von einem anderen Rechner geklont wird
- Das Arbeitsverzeichnis ist ein Checkout („Abbild“) einer spezifischen Version des Projektes.
  - Dateien werden aus der komprimierten Datenbank geholt und auf der Festplatte in einer Form gespeichert, die bearbeitet und modifiziert werden kann
- Die Staging Area ist einfach eine Datei (normalerweise im Git Verzeichnis), in der vorgemerkt wird, welche Änderungen der nächste Commit umfassen soll.
  - Wird auch als „Index“ bezeichnet



GIT Dateizustände



Git Arbeitsprozess

- Dateien im Arbeitsverzeichnis bearbeiten
- Dateien für den nächsten Commit markieren
  - Snapshots zur Staging Area hinzufügen
- Commit anlegen
  - die in der Staging Area vorgemerkten Snapshots werden dauerhaft im Git Verzeichnis (d.h. der lokalen Datenbank) gespeichert

GIT Commits erzeugen

- Der Zustand von Dateien kann mit dem „git status“-Befehl überprüft werden:

```
> git status
# On branch masternothing to commit (working directory clean)
```

- Wird nun eine neue Datei hinzugefügt, wird sie von Git zunächst als „untracked“ erkannt:

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

- Mit dem Kommando „git add“ können Dateien (oder ganze Ordner) zur Versionskontrolle hinzugefügt werden:

```
> git add README.txt
> git status
# On branch master
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#new file:   README.txt
```

GIT Commits erzeugen

- Die so hinzugefügten Dateien sind zwar noch in keinem Commit erfasst, aber zumindest schon mal für den nächsten Commit vorgemerkt. Mit dem Befehl „git commit“ kann aus den Dateien in der Staging Area schließlich ein neuer Commit erstellt werden:

```
> git commit –m 'Mein erster Commit.‘
[master (root-commit) 51fe4d0] Mein erster Commit.
1 file changed, 1 insertion(+)
create mode 100644 README.txt

> git status
# On branch master
nothing to commit (working directory clean)
```

Die Staging Area überspringen

- Die Staging Area ist manchmal ein bisschen umständlich
- Git stellt deshalb eine Alternative zur Verfügung, mit der die Staging Area übersprungen werden kann:
  - git commit -a
    - übernimmt automatisch alle Änderungen an denjenigen Dateien, die sich bereits unter Versionskontrolle befinden, in den Commit
    - vorheriges git add kann weggelassen werden

Git add und gitignore

- Wenn ein Verzeichnis als Pfad für git add angegeben wird, fügt git add alle Dateien in diesem Verzeichnis und allen Unterverzeichnissen rekursiv hinzu

- In der Regel befinden sich eine Reihe von Dateien im Projektverzeichnis, die man nicht versionieren bzw. im Repository haben will
  - z.B. automatisch generierte Dateien, wie Logdateien oder Binärdateien
- In solchen Fällen können in der Datei .gitignore alle Dateien oder Dateimuster angegeben werden, die ignoriert werden sollen

```
> cat .gitignore
*.[oa]
*~
```

Die erste Zeile weist Git an, alle Dateien zu ignorieren, die mit einem .o oder .a enden. Die zweite Zeile bewirkt, dass alle Dateien ignoriert werden, die mit einer Tilde (~) enden.

.gitignore Beispiel

```
# ein Kommentar - dieser wird ignoriert

# ignoriert alle Dateien, die mit .a enden
*.a

# nicht aber lib.a Dateien (obwohl obige Zeile *.a ignoriert)
!lib.a

# ignoriert alle Dateien im build/ Verzeichnis
build/

# ignoriert doc/notes.txt, aber nicht doc/server/arch.txt
doc/*.txt

# ignoriert alle .txt Dateien unterhalb des doc/ Verzeichnis
doc/**/*.txt
```

Dateien entfernen

- Um eine Datei aus der Git Versionskontrolle zu entfernen, muss diese aus der Staging Area entfernt werden und dann mit einem Commit bestätigt werden.
  - git rm datei
    - löscht die Datei auch aus dem Arbeitsverzeichnis

- Wenn die Datei nur aus dem Arbeitsverzeichnis gelöscht wird, wird sie mit git status in der Sektion „Changes not staged for commit“ angezeigt:

  > rm test.txt

  > git status

  On branch master

  Changes not staged for commit:

  (use "git add/rm <file>..." to update what will be committed) (use "git checkout -- <file>..." to discard changes in working directory)

  deleted: test.txt

  no changes added to commit (use "git add" and/or "git commit -a")

- Wenn Du jetzt git rm ausführst, wird diese Änderung für den nächsten Commit in der Staging Area vorgemerkt

Dateien verschieben

> git mv file_from file_to

- git status zeigt an, dass die Datei umbenannt wurde:

  > git mv README.txt README
  > git status
  On branch master
  Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

        renamed: README.txt -> README

- Alternative:

> mv README.txt README
> git rm README.txt
> git add README

GIT Änderungen hochladen

- Um Änderungen an das entfernte Repository (von dem man vorher geklont hat) zu senden:

```
> git push origin
```

Übung

- Erstelle ein Git-Repository in deinem Account auf dem linuxserver.
- Füge Dateien ein, verändere sie inhaltlich und erzeuge nach jeder Änderung einen Commit (2-3x)
- Probiere die Befehle zum Löschen und Verschieben von Dateien aus (alle Varianten)
- Teste den git commit –a Befehl

Übung

- Erstelle ein Git-Repository auf dem RaspberryPi
  - **git init --bare**
- Clone das Repository in deinen Account auf dem linuxserver
- Arbeite am repository
- Lade die Änderungen hoch und schaue dir die Ordnerstruktur auf dem RaspberryPi an.

git init --bare

- git init
  - Der angegebene Ordner wird zur working directory mit ausgecheckten Kopien der Repository-Dateien
  - Im .git Unterordner werden alle Git-relevanten Informationen gespeichert
  - Wird zum Arbeiten verwendet (add, rm, commit, …)

- git init --bare
  - Keine working directory
  - Typischerweise hat ein Ordner die .git Endung
  - Wird zum Teilen verwendet
    - Zentrale Stelle, die meistens/immer online ist und als Verteiler für die Dateien dient
    - Repository wird von Teilnehmern geklont, Modifikationen erfolgen immer lokal und werden dann wieder hochgelanden um sie allen zugänglich zu machen

Übung

- Erstelle ein Repository auf GitHub
- Erstelle darin eine neue Textdatei mit kurzem Inhalt
- Füge die Datei mit git add hinzu
- Ändere den Inhalt der Datei
- Führe git status aus. Was wird ausgegeben?
- Was passiert, wenn du jetzt einen commit anlegst?

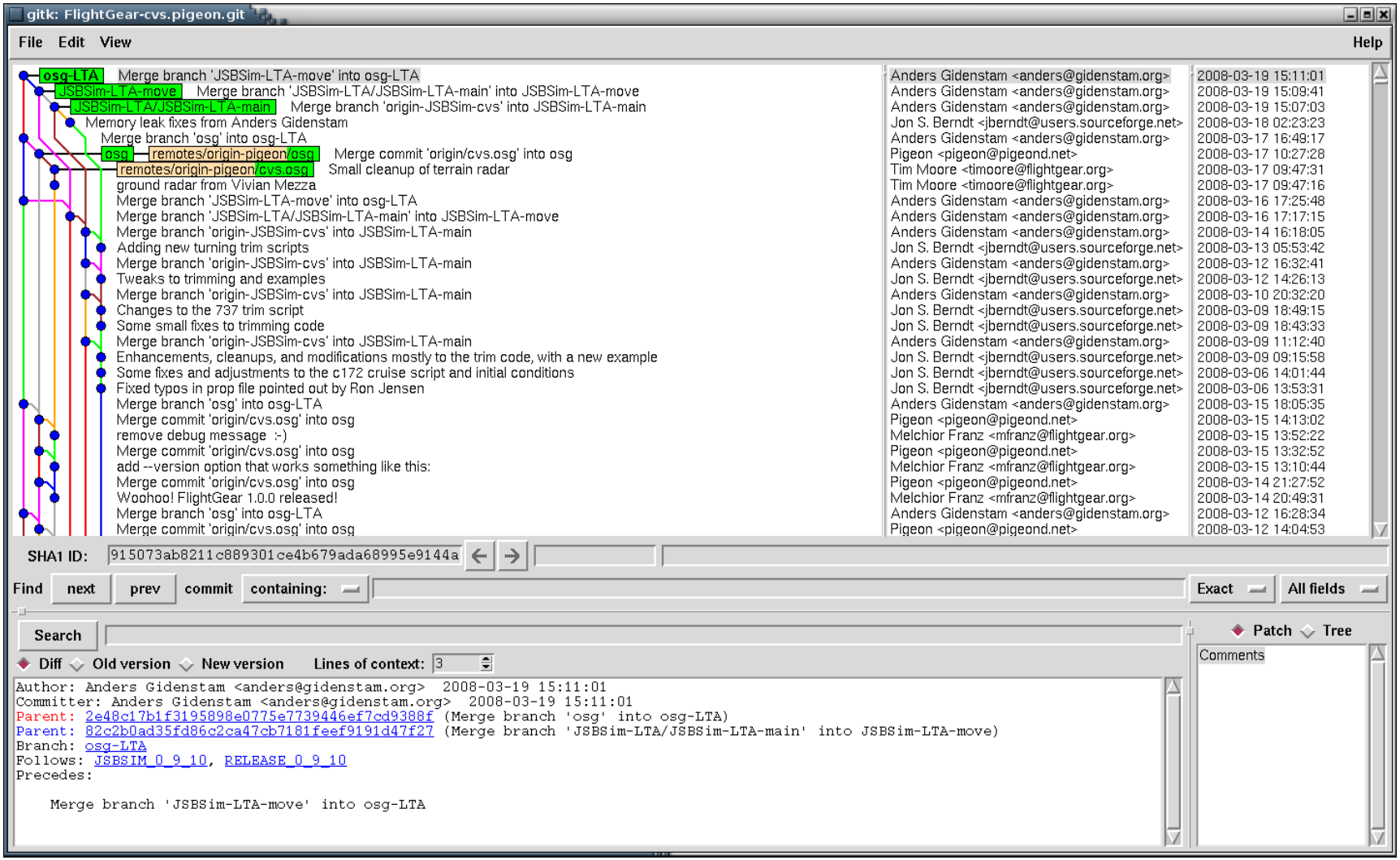
Die History betrachten

- Mit dem Kommando „git log“ kann die Versionsgeschichte betrachtet werden.
- Unter Linux und MacOS kann auch das Programm „gitk“ verwendet werden
  - Stellt die History grafisch dar
  - Unter Windows bieten TortoiseGit und Sourcetree ähnliche Funktionen
- Um eine Kurzform der History auf der Kommandozeile auszugeben, kann auch folgendes Kommando verwendet werden:

  > git log --oneline --decorate --graph –all
- Auch nützlich:

  > git log –p

gitk


gitk: FlightGear-cvs.pigeon.git
File Edit View Help
osg-LTA Merge branch 'JSBSim-LTA-move' into osg-LTA
JSBSim-LTA-move Merge branch 'JSBSim-LTA/JSBSim-LTA-main' into JSBSim-LTA-move
JSBSim-LTA/JSBSim-LTA-main Merge branch 'origin-JSBSim-cvs' into JSBSim-LTA-main
Memory leak fixes from Anders Gidenstam
Merge branch 'osg' into osg-LTA
osg remotes/origin-pigeon/osg Merge commit 'origin/cvs.osg' into osg
remotes/origin-pigeon/cvs.osg Small cleanup of terrain radar
ground radar from Vivian Mezza
Merge branch 'JSBSim-LTA-move' into osg-LTA
Merge branch 'JSBSim-LTA/JSBSim-LTA-main' into JSBSim-LTA-move
Merge branch 'origin-JSBSim-cvs' into JSBSim-LTA-main
Adding new turning trim scripts
Merge branch 'origin-JSBSim-cvs' into JSBSim-LTA-main
Tweaks to trimming and examples
Merge branch 'origin-JSBSim-cvs' into JSBSim-LTA-main
Changes to the 737 trim script
Some small fixes to trimming code
Merge branch 'origin-JSBSim-cvs' into JSBSim-LTA-main
Enhancements, cleanups, and modifications mostly to the trim code, with a new example
Some fixes and adjustments to the c172 cruise script and initial conditions
Fixed typos in prop file pointed out by Ron Jensen
Merge branch 'osg' into osg-LTA
Merge commit 'origin/cvs.osg' into osg
remove debug message :-)
Merge commit 'origin/cvs.osg' into osg
add --version option that works something like this:
Merge commit 'origin/cvs.osg' into osg
Woohoo! FlightGear 1.0.0 released!
Merge branch 'osg' into osg-LTA
Merge commit 'origin/cvs.osg' into osg
Anders Gidenstam <anders@gidenstam.org> 2008-03-19 15:11:01
Anders Gidenstam <anders@gidenstam.org> 2008-03-19 15:09:41
Anders Gidenstam <anders@gidenstam.org> 2008-03-19 15:07:03
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-18 02:23:23
Anders Gidenstam <anders@gidenstam.org> 2008-03-17 16:49:17
Pigeon <pigeon@pigeond.net> 2008-03-17 10:27:28
Tim Moore <timoore@flightgear.org> 2008-03-17 09:47:31
Tim Moore <timoore@flightgear.org> 2008-03-17 09:47:16
Anders Gidenstam <anders@gidenstam.org> 2008-03-16 17:25:48
Anders Gidenstam <anders@gidenstam.org> 2008-03-16 17:17:15
Anders Gidenstam <anders@gidenstam.org> 2008-03-14 16:18:05
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-13 05:53:42
Anders Gidenstam <anders@gidenstam.org> 2008-03-12 16:32:41
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-12 14:26:13
Anders Gidenstam <anders@gidenstam.org> 2008-03-10 20:32:20
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-09 18:49:15
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-09 18:43:33
Anders Gidenstam <anders@gidenstam.org> 2008-03-09 11:12:40
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-09 09:15:58
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-06 14:01:44
Jon S. Berndt <jberndt@users.sourceforge.net> 2008-03-06 13:53:31
Anders Gidenstam <anders@gidenstam.org> 2008-03-15 18:05:35
Pigeon <pigeon@pigeond.net> 2008-03-15 14:13:02
Melchior Franz <mfranz@flightgear.org> 2008-03-15 13:52:22
Pigeon <pigeon@pigeond.net> 2008-03-15 13:32:52
Melchior Franz <mfranz@flightgear.org> 2008-03-15 13:10:44
Pigeon <pigeon@pigeond.net> 2008-03-14 21:27:52
Melchior Franz <mfranz@flightgear.org> 2008-03-14 20:49:31
Anders Gidenstam <anders@gidenstam.org> 2008-03-12 16:28:34
Pigeon <pigeon@pigeond.net> 2008-03-12 14:04:53
SHA1 ID: 915073ab8211c889301ce4b679ada68995e9144a
Find next prev commit containing: Exact All fields
Search
Patch Tree
Diff Old version New version Lines of context: 3
Comments
Author: Anders Gidenstam <anders@gidenstam.org> 2008-03-19 15:11:01
Committer: Anders Gidenstam <anders@gidenstam.org> 2008-03-19 15:11:01
Parent: 2e48c17b1f3195898e0775e7739446ef7cd9388f (Merge branch 'osg' into osg-LTA)
Parent: 82c2b0ad35fd86c2ca47cb7181feef9191d47f27 (Merge branch 'JSBSim-LTA/JSBSim-LTA-main' into JSBSim-LTA-move)
Branch: osg-LTA
Follows: JSBSIM_0_9_10, RELEASE_0_9_10
Precedes:
Merge branch 'JSBSim-LTA-move' into osg-LTA

Änderungen rückgängig machen

- Vorab: Nicht immer alles lässt sich erneut wiederherstellen, was rückgängig gemacht wird!
  - Dies ist eine der wenigen Situationen in Git, in denen Daten bei falschen Eingriffen verloren gehen können

- Den letzten Commit ändern
  - Manchmal hat man einen Commit zu früh angelegt und möglicherweise vergessen, einige Dateien hinzuzufügen, oder eine falsche Commit Meldung verwendet.
    - > git commit –amend
    - Wenn beispielsweise ein Commit angelegt wurde und dann festgestellt wird, dass ein add vergessen worden ist:

      > git commit -m 'initial commit'

      > git add forgotten_file

      > git commit --amend
    - Diese drei Befehle legen einen einzigen neuen Commit an – der letzte Befehl ersetzt dabei das Ergebnis des ersten Befehls

Änderungen rückgängig machen

- Änderungen aus der Staging Area entfernen
  - git status liefert zugleich auch einen Hinweis dafür, wie Änderungen rückgängig gemacht werden können.
  - Beispiel: zwei Dateien wurden geändert und git add * ausgeführt. Es sollen aber zwei zwei seperate Commits angelegt werden. Wie kann eine der beiden Änderungen wieder aus der Staging Area genommen werden? git status gibt einen Hinweis:

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

        modified:   README.txt
        modified:   benchmarks.rb
```

Änderungen rückgängig machen

- Eine Änderung an einer Datei rückgängig machen
  - Wenn Änderungen an z.B. der Datei benchmarks.rb nicht beibehalten werden sollen, d.h., wenn sie in den Zustand zurückversetzt werden soll. in dem sie sich befand, als der letzte Commit angelegt wurde (oder das Repository geklont wurde).
    - > git status

      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: benchmarks.rb

    - > git checkout -- benchmarks.rb
  - Dieser Befehl ist potentiell gefährlich, da er Änderungen an einer Datei vollständig verwirft.
  - Alles was jemals in einem Commit in Git enthalten war, lässt sich aber fast immer wieder wiederherstellen.
    - Selbst Commits, die sich in gelöschten Branches befanden, oder Commits, die mit einem --amend Commit überschrieben wurden
    - Änderungen, die es nie in einen Commit geschafft haben, lassen sich wahrscheinlich nie wieder restaurieren

Externe Repositories

- „git remote“ zeigt an, welche externen Server für ein Projekt lokal konfiguriert wurden und listet die Kurzbezeichnungen für diese Remote Repository auf
- Wenn ein Repository geklont wurde, sollte mindestens origin stehen – welches der Standardname ist, den Git für denjenigen Server vergibt, von dem geklont wurde
- die Option “-v“ gibt für jeden Kurznamen auch die jeweilige URL an, die Git gespeichert hat:
  - > git remote -v
  - origin git://github.com/schacon/ticgit.git (fetch)
  - origin git://github.com/schacon/ticgit.git (push)

Externe Repositories

- Remote Repositories hinzufügen
- Um ein neues Remote Repository mit einem Kurznamen hinzuzufügen, wird git remote add [shortname] [url] ausgeführt:

```
> git remote
origin
> git remote add pb git://github.com/paulboone/ticgit.git
> git remote -v
origin  git://github.com/schacon/ticgit.git
pb  git://github.com/paulboone/ticgit.git
```

- Jetzt kann der Name pb anstelle der vollständigen URL in verschiedenen Befehlen verwendet werden. Wenn alle Informationen, die in pb, aber noch nicht im eigenen Repository verfügbar sind, heruntergeladen werden sollen, kann der Befehl git fetch pb verwendet werden:

```
> git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

- Pauls master Branch ist jetzt lokal auf dem Rechner als pb/master verfügbar – er kann mit einem der eigenen Branches zusammengeführt werden. Oder er wird auf einen lokalen Branch gewechselt, um damit zu arbeiten.

Externe Repositories

- Änderungen aus Remote Repositorys herunterladen und herunterladen inkl. zusammenführen
  - git fetch lädt alle Daten aus dem Remote Repository herunter, die noch nicht auf dem eigenen Rechner verfügbar sind. Danach kennt das eigene Repository Verweise auf alle Branches in dem Remote Repository
  - Wenn ein Repository geklont wurde, legt der Befehl automatisch einen Verweis auf dieses Repository unter dem Namen origin an. D.h. git fetch origin lädt alle Neuigkeiten herunter, die in dem Remote Repository von anderen hinzugefügt wurden, seit es geklont wurde (oder zuletzt git fetch ausgeführt wurde). Es ist wichtig, zu verstehen, dass der git fetch Befehl Daten lediglich in das lokale Repository lädt. Er führt sich mit den eigenen Commits in keiner Weise zusammen (mergt) oder modifiziert, woran gerade gearbeitet wird. D.h. es müssen die heruntergeladenen Änderungen eventuell anschließend selbst manuell mit den eigenen zusammengeführt werden.

  - Wenn allerdings ein Branch so aufgesetzt wurde, dass er einem Remote Branch „folgt“ (also einen „Tracking Branch“, später), dann kann der Befehl **git pull** verwendet werden, um automatisch neue Daten herunterzuladen und den externen Branch gleichzeitig mit dem aktuellen, lokalen Branch zusammenzuführen.
  - git clone setzt den lokalen master Branch deshalb standardmäßig so auf, dass er dem Remote master Branch des geklonten Repositorys folgt (sofern das Remote Repository einen master Branch hat). Wenn dann git pull ausgeführt wird, wird Git die neuen Commits aus dem externen Repository holen und versuchen, sie automatisch mit dem Code zusammenzuführen, an dem gerade gearbeitet wird.

Externe Repositories

- Änderungen in ein Remote Repository hochladen
  - Änderungen können in ein gemeinsam genutztes Repository hochgeladen werden (engl. „push“). Der Befehl dafür ist einfach:

    git push [remote-name] [branch-name].
  - Wenn der master Branch auf den origin Server hochgeladen werden soll, dann kann dieser Befehl verwendet werden:

    git push origin master
  - Funktioniert nur dann, wenn Schreibrechte für das jeweilige Repository vorhanden sind und niemand anders in der Zwischenzeit irgendwelche Änderungen hochgeladen hat.
    - Wenn zwei Leute ein Repository zur gleichen Zeit klonen, dann zuerst der eine seine Änderungen hochlädt und der zweite anschließend versucht, das gleiche zu tun, dann wird sein Versuch korrekterweise abgewiesen.
    - In dieser Situation muss man neue Änderungen zunächst herunterladen und mit seinen eigenen zusammenführen, um sie dann erst hochzuladen.

Externe Repositories

- Verweise auf externe Repositorys löschen und umbenennen
  - Wenn eine Referenz auf ein Remote Repository umbenannt werden soll:

    git remote rename
  - Beispiel: pb in paul umbenennen:

    > git remote rename pb paul

    > git remote

    origin

    paul
  - Wenn eine Referenz aus irgendeinem Grund entfernt werden soll (z.B. weil der Server umgezogen ist):

    > git remote rm paul

    > git remote

    origin

Tags

- Git kann bestimmte Punkte in der Historie als besonders wichtig markieren, also taggen
- Normalerweise verwendet man diese Funktionalität, um Release Versionen zu markieren (z.B. v1.0).

- Vorhandene Tags anzeigen

```
> git tag
v0.1
v1.3
```

  - Listet die Tags in alphabetischer Reihenfolge auf

  - Es kann auch nach Tags mit einem bestimmten Muster gesucht werden:

```
> git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Tags

- Neue Tags anlegen
  - Git kennt im wesentlichen zwei Typen von Tags:
    - einfache (engl. Lightweight) Tags
      - lediglich ein Zeiger auf einen bestimmten Commit
    - kommentierte (engl. annotated) Tags
      - werden als vollwertige Objekte in der Git Datenbank gespeichert
      - haben eine Checksumme, beinhalten Namen und E-Mail Adresse desjenigen, der den Tag angelegt hat, das jeweilige Datum sowie eine Meldung.
  - Kommentierte Tags
    - git tag -a

      > git tag -a v1.4 -m 'my version 1.4'
      > git tag
      v0.1
      v1.3
      v1.4
    - git show zeigt dann folgenden Tag zusammen mit dem jeweiligen Commit, auf den der Tag verweist, an:

      > git show v1.4
      tag v1.4
      Tagger: Scott Chacon <schacon@gee-mail.com>
      Date: Mon Feb 9 14:45:11 2009 -0800

      my version 1.4

      commit 15027957951b64cf874c3557a0f3547bd83b3ff6
      Merge: 4a447f7... a6b4c97...
      Author: Scott Chacon <schacon@gee-mail.com>
      Date: Sun Feb 8 19:02:46 2009 -0800

      Merge branch 'experiment'

Tags

- Einfache Tags
  - Für einen einfachen Tag wird im wesentlichen die jeweilige Commit Prüfsumme, und sonst keine andere Information, in einer Datei gespeichert. Um einen einfachen Tag anzulegen, verwendet man einfach keine der drei Optionen -a, -s und -m:

    > git tag v1.4-lw

    > git tag

    v0.1

    v1.3

    v1.4

    v1.4-lw

    v1.5

  - git show zeigt jetzt einfach den jeweiligen Commit:

    > git show v1.4-lw

    commit 15027957951b64cf874c3557a0f3547bd83b3ff6

    Merge: 4a447f7... a6b4c97...

    Author: Scott Chacon <schacon@gee-mail.com>

    Date: Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'

Tags

- Nachträglich taggen
  - Commits können jederzeit getaggt werden, auch lange Zeit nachdem sie angelegt wurden
  - Nehmen wir an, die Commit Historie sieht wie folgt aus:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

  - Nehmen wir an, dass vergessen wurde, Version v1.2 des Projekts zu taggen und dass dies der Commit „updated rakefile“ gewesen ist. Dieser Commit kann im Nachhinein getaggt werden, indem die Checksumme des Commits (oder ein eindeutiger Teil davon) am Ende des Befehls angegeben wird:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Tags

- Tags veröffentlichen
  - Der git push Befehl lädt Tags nicht von sich aus auf externe Server.
  - Tags müssen explizit auf einen externen Server hochgeladen werden, nachdem sie angelegt wurden.

```
> git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]       v1.5 -> v1.5
```

- Wenn viele Tags auf einmal hochgeladen werden sollen, kann dem git push Befehl außerdem die --tags Option übergeben werden

```
> git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]       v0.1 -> v0.1
 * [new tag]       v1.2 -> v1.2
 * [new tag]       v1.4 -> v1.4
 * [new tag]       v1.4-lw -> v1.4-lw
 * [new tag]       v1.5 -> v1.5
```

Branches

- Selbsttudium
  - https://git-scm.com/book/de/v1/Git-Branching

Hilfe

- Möglichkeiten:
  - > git help <verb>
  - > git <verb> --help
  - > man git-<verb>

- Beispiel:

  > git help config

Wichtigste Git Commandos

- $ git status
  - Zeigt den momentanen Stand im Zyklus an
- $ git add <file name>
  - Fügt File(s) zum Staging Area (Index) hinzu
- $ git commit –m <comment>
- $ git push <ziel>
  - Transferiert lokalen Stand zu Remote Repository
- $ git fetch <quelle>
  - Lädt Änderungen aus Quelle in lokales Repository
  - Integriert sie NICHT mit lokalem Stand
- $ git merge <commit>
  - Versucht den lokalen Stand mit dem ausgewählten commit zu mergen
- $ git pull <remote name> <branch>
  - Kombiniert fetch und merge
  - Beispiel: $ git pull origin master

Wichtigste Git Commandos

- $ git branch <branchname>
  - Erstellt neuen Branch

- $ git checkout <commit>
  - selektiert commit/branch an dem gearbeitet werden soll
  - D.h. es lädt den Stand des Commits in die Working Copy Ordner