git-und-die-wolke Documentation

Release 0.1

Christoph Piechula & Christopher Pahl

June 14, 2013


git

CHAPTER

ONE

INHALT

1.1 Einfuehrung

1.1.1 Fragen

1.1.2 Vortragstil

1.1.3 Aufteilung

- `git`-Basics (ca. 70 min)
- `git`-Annex (ca. 20 min)
- `git`-Hub (ca. 50 min.)
- `git`-Übung (ca. 30 min.)
- `git`-Fragen (ca. 10 min.)

1.1.4 Early-Adaptors-Umfrage

1.1.5 Warum git? Warum nicht X?

Weiter mit: *Git Basics: Ab in die Shell*

1.2 Git Basics: Ab in die Shell

1.2.1 Was ist git?

- Ein Versionsverwaltungssystem
- Ein Protokoll
- Ein abstraktes Filesystem
- Linus: `A distributed stupid content tracker`

Was ist es nicht?

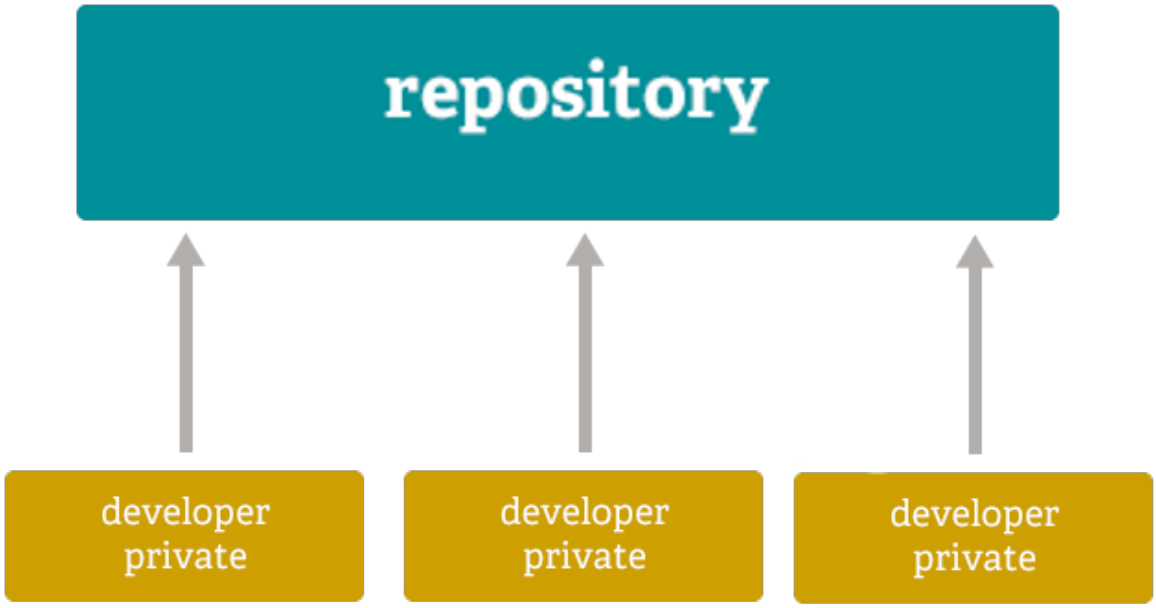
- CVS (`WWCVSND`)
- SVN (`CVS done right.`)

Erklärung:

Linus on GoogleTalk.

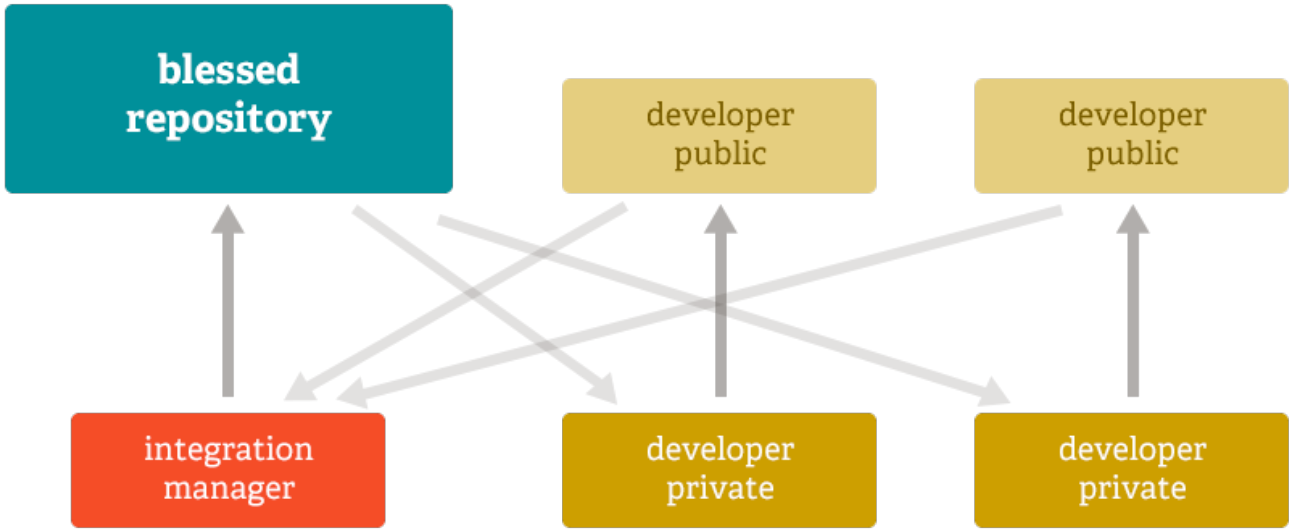
1.2.2 Zentraler Workflow

- Zentraler Workflow wie bei CVS
- Viele Entwickler arbeiten mit einem Repository
- Jeder Entwickler hat nur eine unvollständige Kopie



1.2.3 Dezentraler Workflow

- `git` typischer Ablauf.
- Jeder Entwickler hat ein Repository
- Jeder Entwickler hat eine volle Kopie



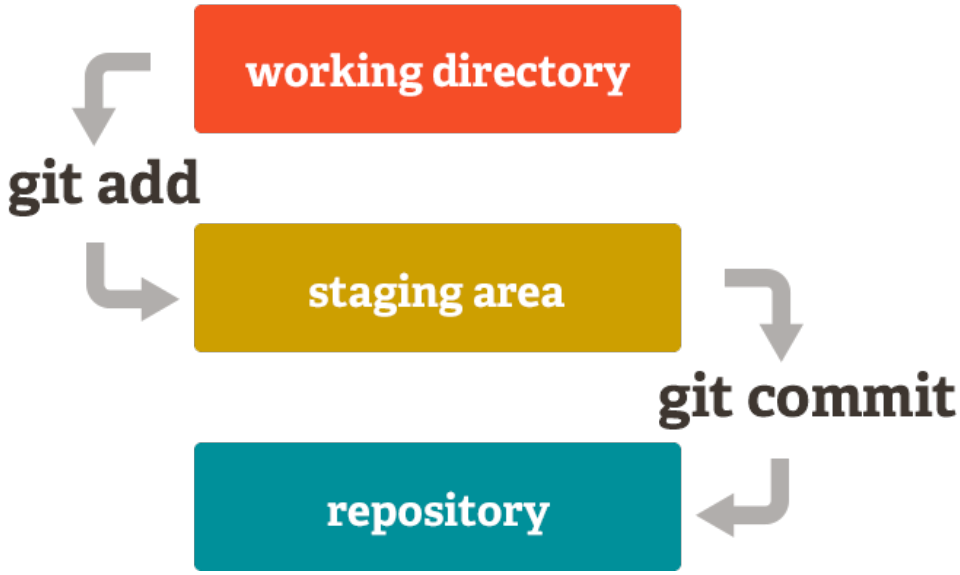
1.2.4 `git init`

- Lege ein neues Git-Repository in `your-repo` an.

```
$ mkdir your-repo && cd your-repo
$ git init .
$ ls --all
.  ..  .git
$ tree .git
.git
-- branches
-- config
-- HEAD
-- index
-- [...]
-- logs
|   -- HEAD
|   -- refs
-- objects
-- refs
```

1.2.5 `git add`

```
$ git add [your-file-or-dir-here]
```



Alle Bereiche interaktiv als HTML:

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

1.2.6 `git commit`

```
$ echo "Hello Phil!" > README
$ git add README

$ git status
# On branch master
# Changes to be committed:
#   new file:   README

$ git commit --all --message "commit message"  # ausgechrieben
$ git commit -am "commit message"              # oder kürzer
$ git commit -a                                # lange messages
[Editor öffnet sich]

$ git status
# On branch master
nothing to commit, working directory clean
```

1.2.7 Was ist ein `diff`?

- Ein `diff` ist die Änderung zwischen zwei Änderungen.
- Errr... Wat?

```
# Zeige alle Änderungen seit dem letzten Commit
$ git diff
diff --git a/TODO.list b/TODO.list
index e6c2b18..a2fe0bc 100644
--- a/TODO.list
+++ b/TODO.list
@@ -1,21 +1,20 @@
```

```
+ Hinweise in der Versionshistorie verstecken
- Zettelchen schreiben

# Bestimmten Commit zeigen
$ git show a2fe0bc
<dasselbe wie oben>
```

1.2.8 Freunde von `git commit`

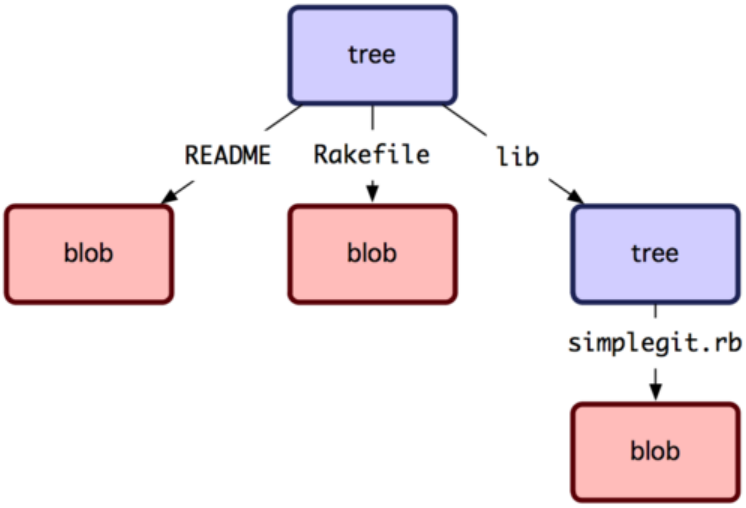
Früher oder später will man etwas berichtigen

```
# Letzte Commit-Messages berichtigen
# to amend == berichtigen.
$ git commit --amend

# Änderungen an einem file zurücksetzen
# Working Tree -> Unmodified
$ git checkout -- your_file.txt

# "git add" rückgängig machen
# Index -> Working Tree
$ git reset your_file.txt

$ git stash       # Änderungen kurz wegsichern
$ git stash pop   # ... später wieder hervorholen
```

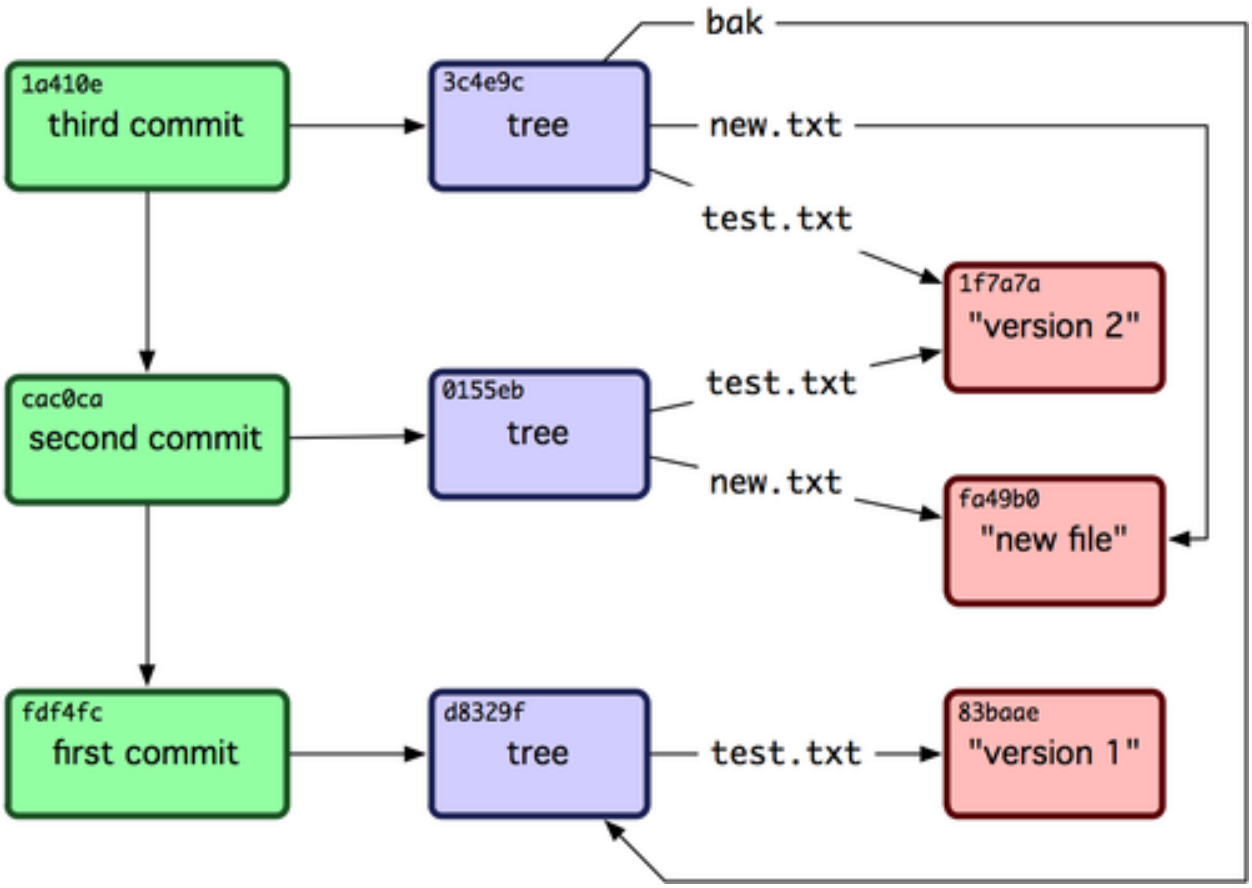
1.2.9 Die Objektdatenbank #1

Vier unterschiedliche Objekttypen:

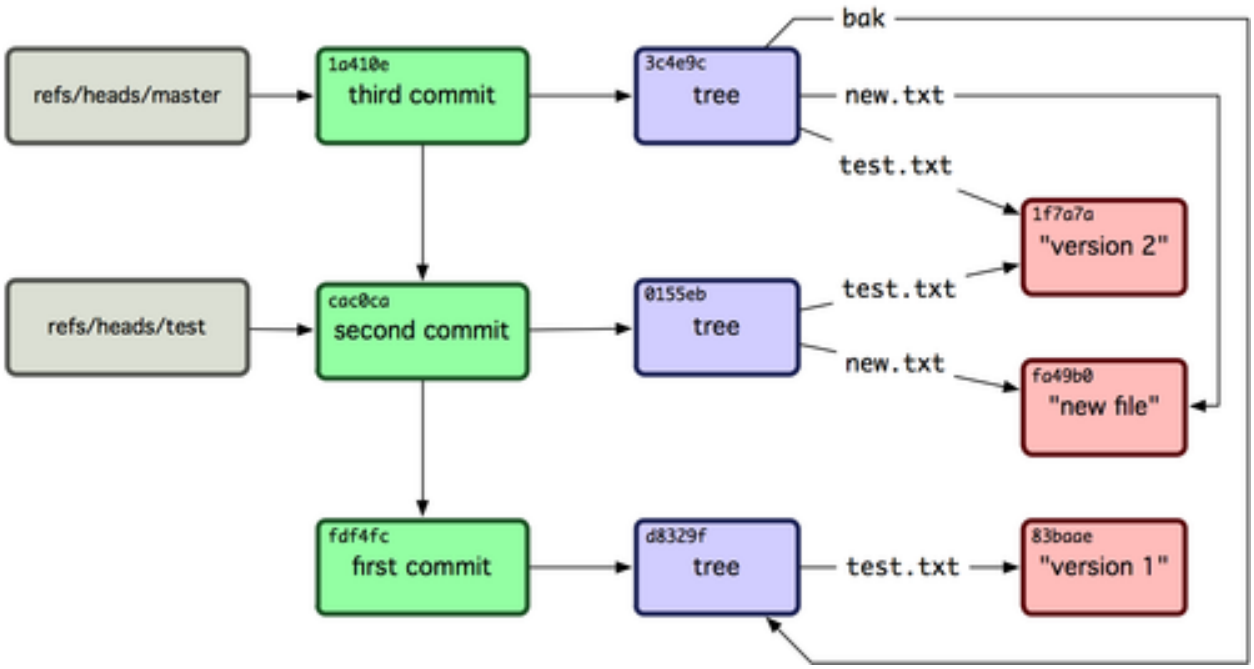
- Blobs (Dateien)
- Trees (Verzeichnisse)
- Commits (Änderungen)
- Referenzen (Branches oder Tags)



1.2.10 Die Objektdatenbank #2



1.2.11 Die Objektdatenbank #3



1.2.12 Git Branching

1.2.13 Branches #1

Branches erstellt man mit:

```
$ git checkout -b <branch-name>
```

In bestehende branches wechseln:

```
$ git checkout <branch-name>
```

Branches auflisten:

```
$ git branch --all
```

1.2.14 Branches #2

Branches führt man zusammen mit:

```
$ git merge <target-branch>
```

- Dabei können böse Dinge passieren.
- Dinge die `git`-Anfänger zu CVS-Usern werden lässt.
- Es können **Merge-Conflicts** entstehen.
- Was passiert wenn in beiden `branches` dasselbe File geändert wurde?
  - Andere Zeile? `git` merged es automatisch.

  - Selbe Zeile? Uh-oh.

1.2.15 git clone

- Klone ein Repository

```
$ git clone git://github.com/studentkittens/git-und-die-wolke.git

Cloning into 'git-und-die-wolke'...
remote: Counting objects: 94, done.
remote: Compressing objects: 100% (72/72), done.
remote: Total 94 (delta 36), reused 72 (delta 16)
Receiving objects: 100% (94/94), 5.70 MiB | 1.60 MiB/s, done.
Resolving deltas: 100% (36/36), done.
```

- URL-Schema Beispiele:

```
git://github.com/qitta/foozel.git         → Git [Read only]
git@github.com:sahib/rmlint.git           → SSH [Preferred]
https://github.com/tmarc/advanced-ios.git → HTTPS [Notlösung]
git clone file:///opt/git/project.git     → Lokal
```

1.2.16 git remote

Entfernte Repositories verwalten:

```
# Alle remotes auflisten
$ git remote -v
origin  git@github.com:studentkittens/git-und-die-wolke.git (fetch)
origin  git@github.com:studentkittens/git-und-die-wolke.git (push)

# Neues remote adden
$ git remote add nullcat git@nullcat.de
$ git remote -v
...
nullcat git@nullcat.de (fetch)
nullcat git@nullcat.de (push)

# Bestehendes remote verändern
$ git remote set-url nullcat https://git.nullcat.de
```

1.2.17 git push

```
$ git push [<remote> [<local-branch>]]

$ git push
$ git push origin
$ git push origin master
```

1.2.18 `git pull`

- Das logische Äquivalent zu `git push`.
- Zieht Änderungen von einem **remote**.

```
$ git pull <remote> <remote-branch>
```

- Auch hier können **Merge-Conflicts** entstehen.
- Vor einem `git push` sollte man immer ein `git pull` machen.

1.2.19 Hilfe?!

- Das ist ja alles schön und gut…
- …aber ich versteh kein Wort.
- Hier wirst du geholfen:
  - manpages:

    ```
    $ git help <commando>
    $ git help tutorial
    ```

  - http://www.git-scm.com/documentation
  - http://de.gitready.com/
  - Es gibt eine Menge Bücher.

1.2.20 `git bisect #1`

Source:

```
bool is_odd(int number) {
    return !number % 2; /* Wrong! */
}
```

```
int main(int argc, char *argv[]) {
    printf("Odd numbers of arguments? %d!\n",
        is_odd(argc - 1) ? "Yes" : "No");
}
```

Test case:

```
void test_is_odd(void) {
    for(int i = -20; i < 20; ++i) {
        assert((is_odd(i) != 0) == (i % 2 != 0));
    }
}
```

1.2.21 git bisect #2

```
Find by binary search the change that introduced a bug
```

Aufgabe:

- Finde heraus wann ein Fehler eingeführt wurde.
- Schaue dir an was damals geändert wurde.
- Leite daraus ab was der Fehler ist.

Funktionsweise:

- Festlegen eines good/bad commits
- Auschecken der Mitte, Testen, Links oder Rechts weitersuchen.

1.2.22 git bisect #3

```
$ git bisect start HEAD HEAD^^^
$ git bisect run make test
# ... viel output von $(make test) ...
5145c8 is the first bad commit
'bisect run' erfolgreich ausgeführt
$ git bisect reset    # Kehre zur normalen Arbeit zurück
$ git show 5145c8     # Zeige unterschiede im bad commit
commit 5145c8781e30057c8e2058d1c361363e213a17f4
Date:   Fri May 3 15:47:38 2013 +0200

    Made is_odd() better looking

diff --git a/is_odd.c b/is_odd.c

 bool is_odd(int number)
 {
-    return number % 2 == 1;
+    return !number % 2;
 }
```

1.2.23 git bisect #4

Was lernt man draus?

- Immer kleine Commits machen!
- Nehmt euch Zeit für eine *sinnvolle* Commit-Messages! Schlechte Beispiele (*):
  - **Some changes** - Riesiger diff.
  - **minor changes** - Complete Rewrite.
  - **Merge.** - Manuelles Merging.
- `git bisect` ist ein gutes Argument für Unit-Tests.

* (*Noch mehr davon:* http://whatthecommit.com/)

1.2.24 Suchen und Beschuldigen

Suche `background:` in allen `.css` Dateien.

```
$ git grep -n 'background:' -- '*.css'
src/custom.css:56: background: -webkit-radial-gradient(#9cf, #369);
src/custom.css:57: background:    -moz-radial-gradient(#9cf, #369);
src/custom.css:58: background:         radial-gradient(#9cf, #369);
```

Herausfinden wer wann etwas geändert hat:

```
$ git blame -L 56,58 src/custom.css
# SHA1   (Autor LN) Content
77a79bbc (Elch  56) background: -webkit-radial-gradient(#9cf, #369);
64ac73cb (Katze 57) background:    -moz-radial-gradient(#9cf, #369);
77a79bbc (Elch  58) background:         radial-gradient(#9cf, #369);
```

→ Der Autor `Katze` ist für den Mozilla-Support zuständig.

1.2.25 `git tag`

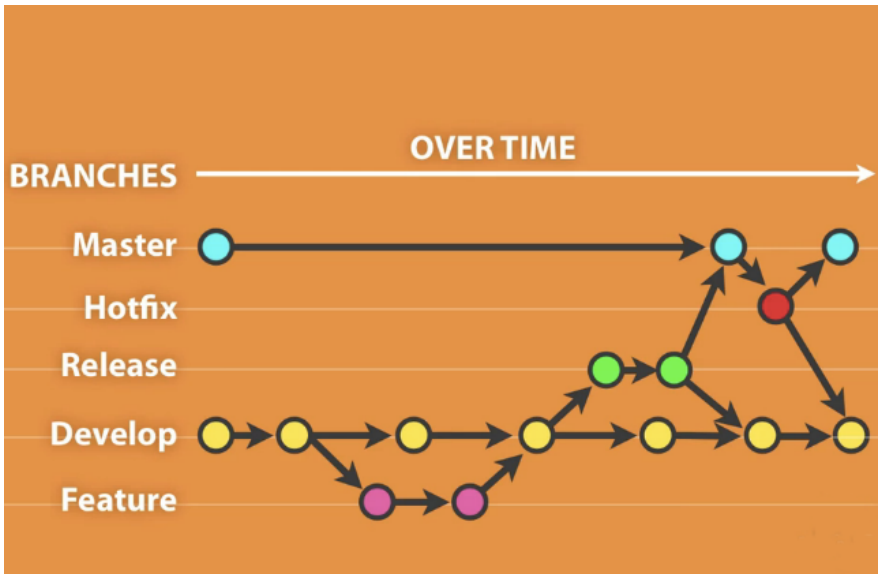
- Manchmal muss man einen Commit *taggen*.
- Wie `branches`, nur *fest*.
- Beispielsweise mit einer Version: **1.2 beta**

```
# Neuen Tag anlegen
git tag "1.2 beta"

# Alle Tags auflisten
git tag

# Anderes Tag löschen.
git tag -d "1.2 beta"

# Tags "veröffentlichen"
git push origin <local-tag-name>
```

1.2.26 Das GitFlow Branching Modell

—

—


OVER TIME
BRANCHES
Master
Hotfix
Release
Develop
Feature


BROKEN, MASTER IS

1.2.27 Tooling

Plugins

- GVim Fugitive Plugin
- Eclipse EGit
- Netbeans (bereits integriert)

Standalone Tools

- gitg (Linux / Gnome)
- giggle (Portabel / Gnome)
- tig (Linux / ncurses)
- gitk (bereits in git enthalten)
- GitHub Windows Client

1.2.28 Best Practices #1

- `.gitignore` nutzen (und `git clean`!).
  - Keinen auto generierten Code/Projektdateien committen.
  - Wenn nicht vermeidbar dann in eigenen Commit.
  - Für Dokumentation am besten eigenen Branch nutzen!
- Sinnvolle Commit-Messages.
  - Siehe Folie für `git bisect 4`.

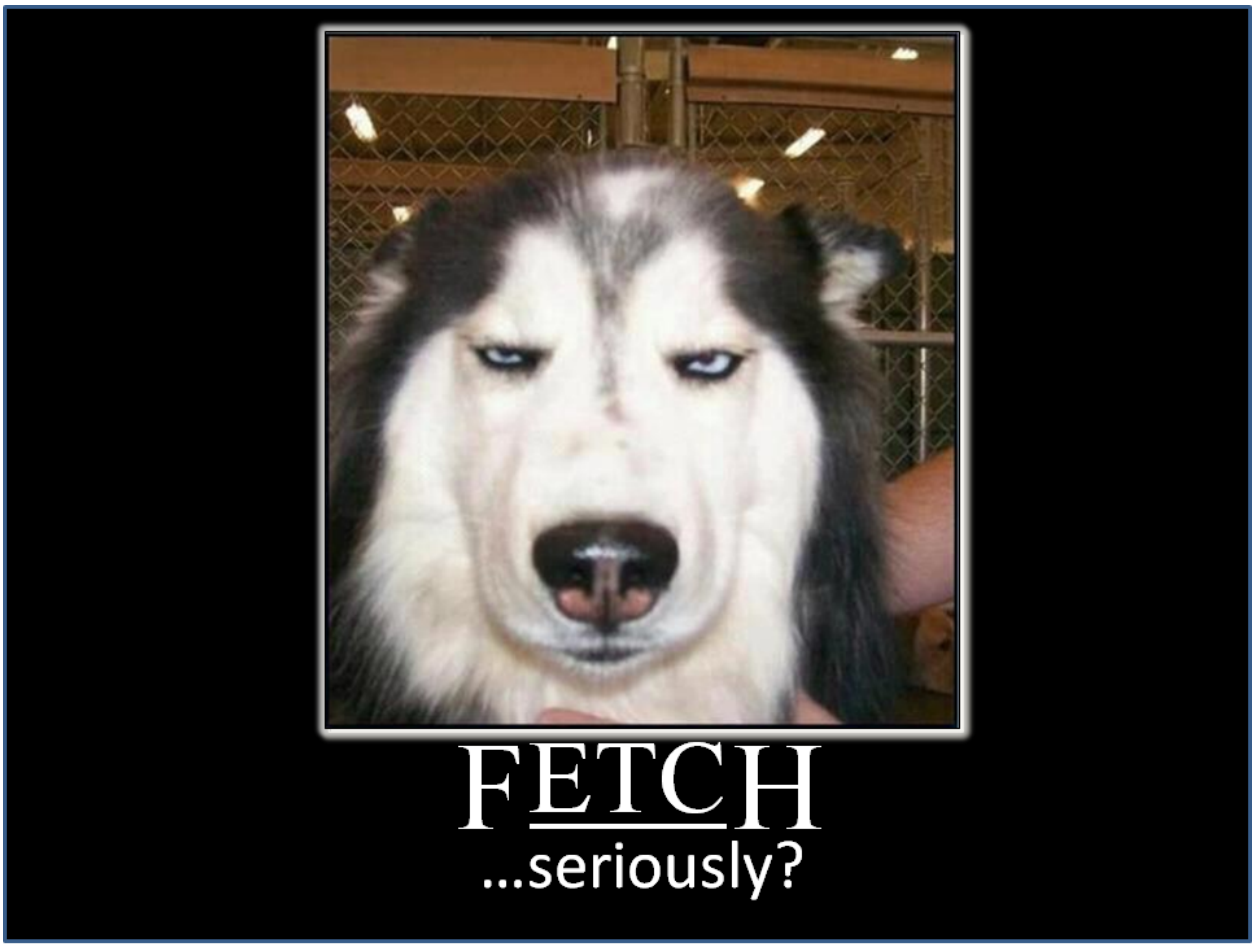
1.2.29 Best Practices #2

- Ein Feature == Ein Commit.
  - Macht Debugging/Übersicht einfacher.
- Review Code before commit.
  - Keine `Fixed up previous commit` Messages.
- Branches für Features nutzen.
  - Damit der `master` branch benutzbar bleibt.

1.2.30 ......

1.2.31 `git fetch`

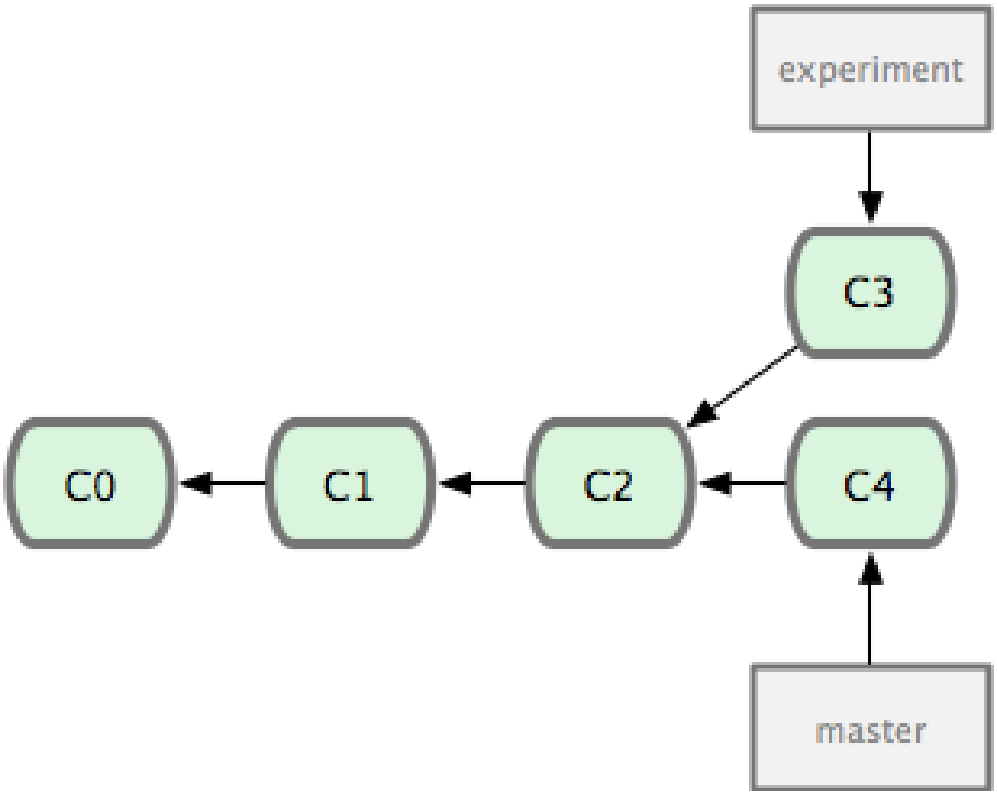
- `git pull` ist ein `git fetch && git merge`.
- Warum sollte man das wollen?
- Wenn man nicht will dass automatisch gemerged wird.
- Beispiel:


FETCH
...seriously?

```
$ git fetch origin
$ git checkout origin/master
$ # look around
$ # if satisfied:
$ git checkout master
$ git merge origin/master
```

1.2.32 `git rebase #1`

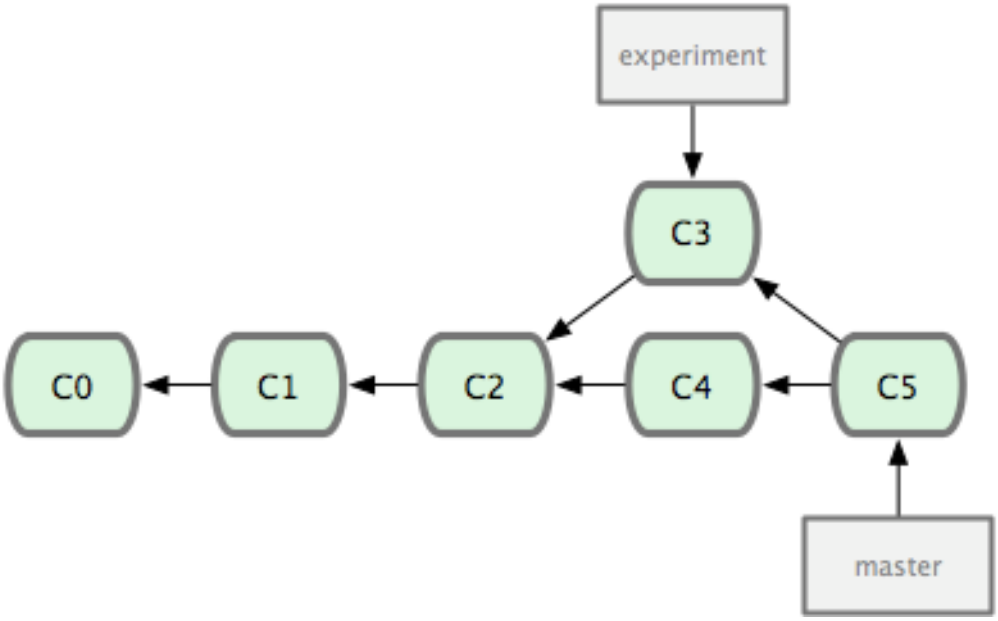
Ausgangszustand:



1.2.33 `git rebase #2`

Ohne Rebase, mit `git merge`:

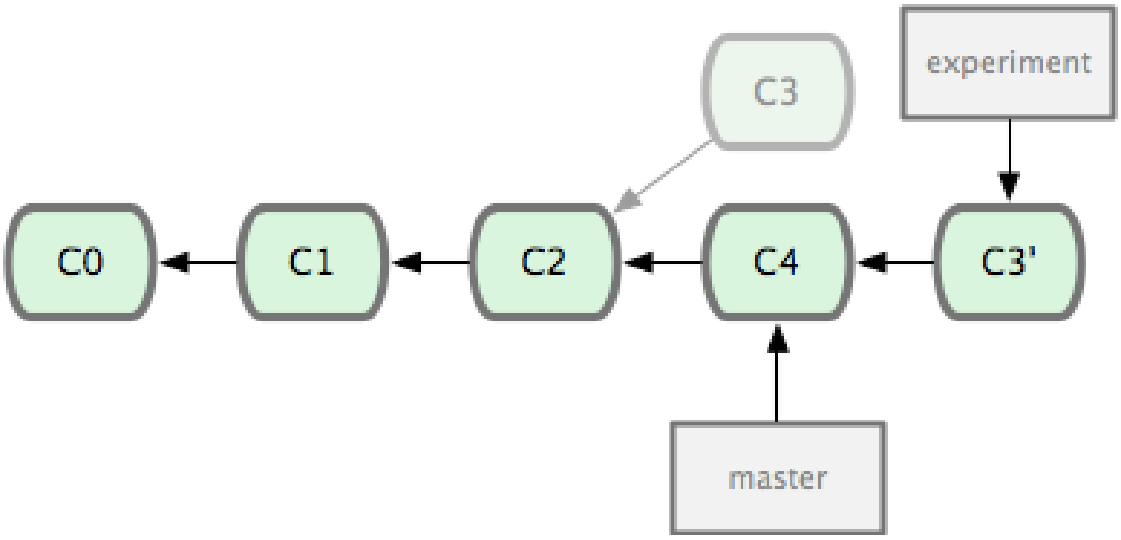
```
$ git checkout master
$ git merge experiment
```



1.2.34 `git rebase #3`

Mit Rebase:

```
$ git checkout experiment   # In 'experiment' wechseln
$ git rebase master         # Basis auf master verschieben
$ git checkout master       # In 'master' wechseln
$ git merge experiment      # Fast-Forward Merge zu 'experiment'
```



—

—


GIT REBASE
NO CONFLICTS!
DIYLOL.COM

1.2.35 Ein Exkurs zu Storage as a Service mit `git`

SaaS mit `git-annex`: *Git Annex: Dropbox fuer Harte*

1.3 Git Annex: Dropbox fuer Harte

1.3.1 `git annex` - Dropbox for geeks

1.3.2 Was ist `git-annex`?

Problem:

- Git eignet sich aufgrund seiner Funktionsweise nur bedingt für große Dateien
- Großen Dateien → hoher Ressourcenverbrauch

Abhilfe: annex

- Ein Wrapper um `git` herum, der git *„aufbohrt"*
- Nur Metadaten werden verwaltet
- Content wird nicht *„getrackt"*, git Features weiterhin vorhanden

1.3.3 Wie funktioniert das Ganze? #1

Repository anlegen

```
$ mkdir annex
$ cd annex
$ git init .
Initialized empty Git repository in /home/christoph/annex/.git/

$ git annex init 'repo on desktop'
init repo on desktop ok
(Recording state in git...)
```

Files hinzufügen

```
$ cp ~/debian-7.0.0-amd64-netinst.iso .
$ git annex add .
add debian-7.0.0-amd64-netinst.iso (checksum...) ok
add wallpaper-279066f0.jpg (checksum...) ok
(Recording state in git...)
```

1.3.4 Wie funktioniert das Ganze? #2

Dateien commiten.

```
$ git commit -am 'files added.'
[master (root-commit) 1dcad58] files added.
2 files changed, 2 insertions(+)
create mode 120000 debian-7.0.0-amd64-netinst.iso
create mode 120000 wallpaper-279066f0.jpg
```

Und nun? Let's sync!

1.3.5 Wie funktioniert das Ganze? #3

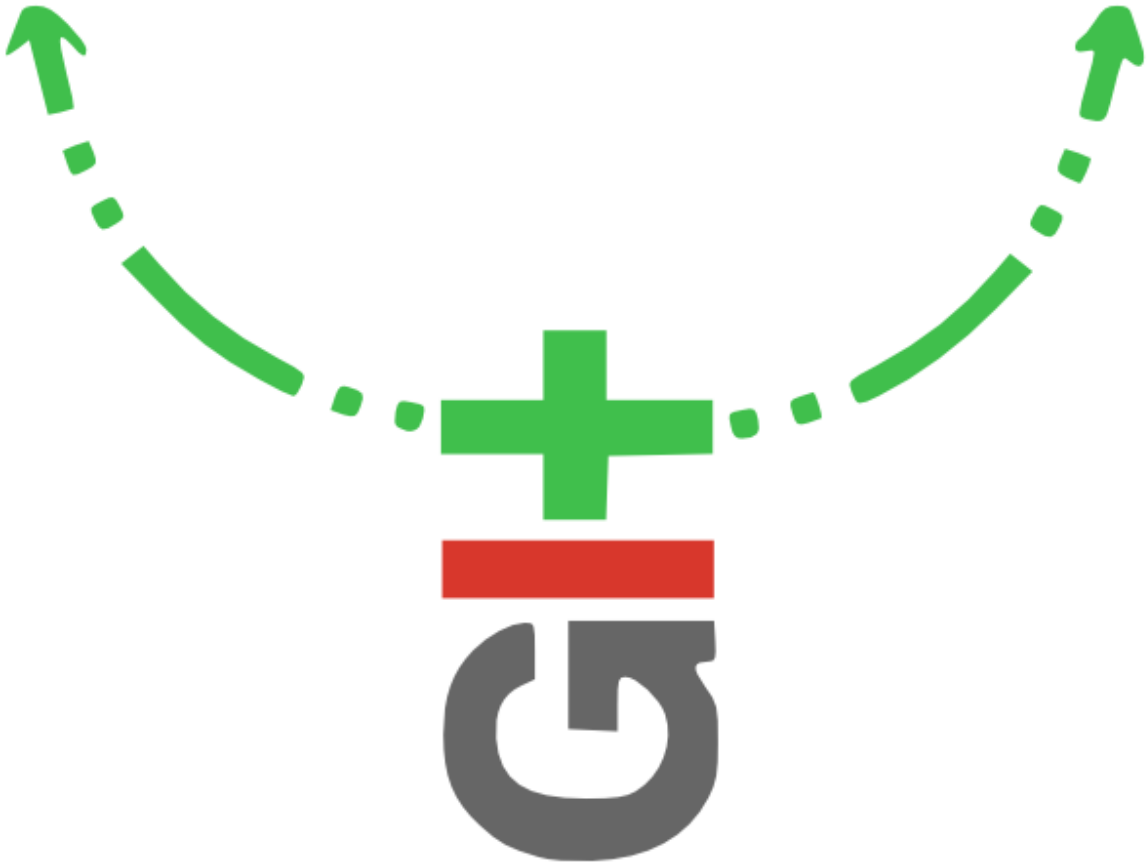
Dateien synchronisieren

```
$ git clone /home/christoph/annex/
Cloning into 'annex'...
done.

$ cd annex
$ ls
debian-7.0.0-amd64-netinst.iso  wallpaper-279066f0.jpg
$ feh wallpaper-279066f0.jpg
feh WARNING: wallpaper-279066f0.jpg does not exist - skipping
feh: No loadable images specified.
See 'feh --help' or 'man feh' for detailed usage information

$ git annex get wallpaper-279066f0.jpg
```

```
get wallpaper-279066f0.jpg (merging origin/git-annex into git-annex...)
(Recording state in git...)
(from origin...) ok
(Recording state in git...)
```

1.3.6 `git annex` Features

- Location Tracking
- Future Proofing
- Backup Copies
- Special Remotes
- Transfering Data
- Distributed Version Control



1.3.7 Alles okay?

njaaaa

- Noch recht stark in der Entwicklung

- Im Moment hauptsächlich was für Geeks

Aber

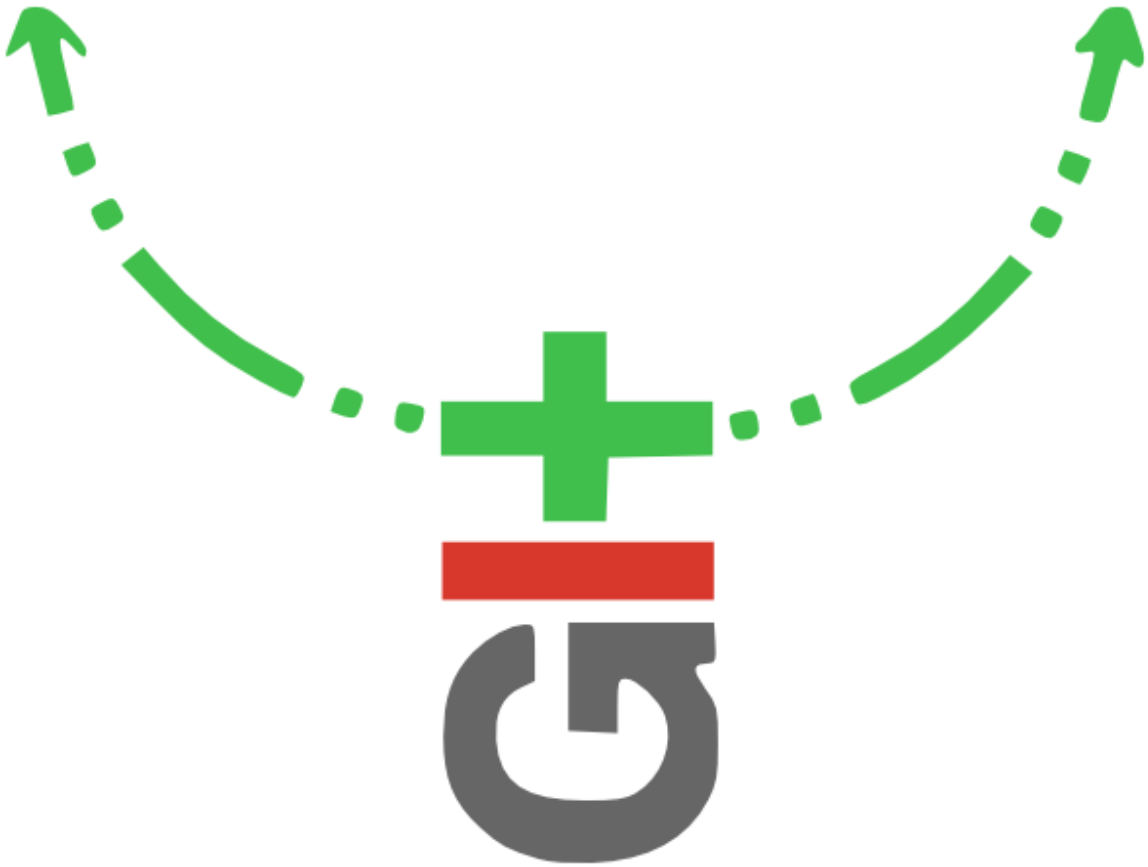
- Abhilfe in Arbeit → Webfrontend

1.3.8 Warum überhaupt das Ganze, es gibt doch Dropbox?

Interessante Features die es bisher so nicht gibt:

- Verschiedene *„cloud remotes'‘* nutzbar z.B. `box.com`, `rsync.net`, `Amazon S3`
- Kontrolle liegt beim Benutzer, nicht Storage Anbieter - interessant für Unternehmen mit kritischen Daten.
- Verschlüsselung, Vertrauensstufen, Sharing etc.
- Verschiedene „Repository Groups” definierbar und kombinierbar → verschiedene Szenarien abdeckbar.
- Praktisch viele Features die man von einer gutem Storage-Lösung erwartet

1.3.9 Power of git-annex for everybody



1.3.10 Nun zum Kernthema des Vortrags

Code Hosting mit Github: *Github: Ab in die Wolke*

1.4 Github: Ab in die Wolke

1.4.1 Was ist Github?

- Ein (Social-)Code-Hosting Dienst.
- Statistiken:
  - ~3,5 Millionen User

  - 6 Millionen Repositories
  - 158 Mitarbeiter
  - April 2008 mit 6000 Usern und 2500 Repos gestartet.
- Bekannte Projekte:
  - Erlang, PHP, Perl, Clojure
  - Mirros: Linux-Kernel, Ruby
  - Git itself

1.4.2 Octocat

1.4.3 Was kann Github?

- Für Open-Source kostenlos.
  - Einschränkungen: 1 GB Speicherplatz.
- Infrastructure as a Service
- Für Unternehmen gibt es `private repos`.
- Für große Unternehmen kann man die Github-Software lizenzieren.
- Eigener Pastebin mit Versionierung `Gist`.
- Volltextsuche über alle Repositories.

Demo: Nutzerprofil

- Git kennt nur den Namen und Email des Nutzers.

```
# Wird in ~/.gitconfig gespeichert
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- GitHub kennt unabhängig davon noch einen Account.
- Diesem sind `1..n` Repository zugeordnet.
- Zudem kann dieser User andere Projekte Forken, Bug-Reports schreiben u.v.m.

Demo: Fork

- Ein Fork ist ein `git clone` mit anderen Namen.
- Man klont `alice/example.git` zu `bob/example.git`.
- Dann macht man den Code den man geforkt hat kaputt.
- Wenn man fertig (mit der Welt) ist kommt ein **Pull Request**.

Demo: Pull Requests

Ablauf ohne GitHub:

```
# Auf Seite des Forkers (bob)
$ git request-pull HEAD^1 https://github.com/<bob>/repo.git
The following changes since commit 04ca9db3149956ed7670d699cb4b4328386b88e1:
  Sophisticated commit message. (2013-05-11 00:36:56 +0200)
are available in the git repository at:
  https://github.com/<bob>/repo.git master
$ git push
# Bob sendet diesen Text an Alice
# Auf Seite des Annehmers (alice):
$ git remote add bob https://github.com/<bob>/repo.git
$ git pull bob 04ca9d
```

Ablauf mit GitHub:

- `bob` macht über GitHub einen Pull Request.
- `alice` klickt auf `Confirme Merge`.

Demo: Organisationen

- Ein leichter Weg um Teams zu organisieren.
- Eine **Organisation** ist ein eigenständiger Nutzer.
- Grundlegender Ansatz bei Entwicklung mit mehreren Personen

Features:

Verwaltung von...

- Membern (ein GitHub User entspricht einem Member)
- Teams (Anlegen)
- Rechten (Pull, Push, Admin)

Demo: Online Blame/Annotate/Edit

Code lässt sich online:

- Browsen.
- Blamen.
- Historisch betrachten.
- Editieren.

Tipp: Auch Bilder, Dokumente und Videos sind previewbar.

Demo: Sonstiges #1

- **Issuetracker:**
  - Eingebauter Bugtracker.
- **Metriken:**
  - `Contributors`, `Commit Activity`, `Pulse`.
  - Beispiel.
- **Downloads:**
  - Gepushte Tags werden zu Downloads.
  - Beispiel: Anlegen von `1.2.0rc1`:

```
$ git tag 1.2.0rc1
$ git push origin 1.2.0rc1
```

Demo: Sonstiges #2

- **Wiki/Webpagehosting:**
  - Leicht erstellbares wiki.
  - `gh-pages` branch wird unter `<user>.github.io/<repo>` gehosted.
  - Beispiel: http://sahib.github.io/rmlint/
- **Soziales:**
  - Andere User kann man `followen`.
  - Andere Repos kann man `watchen`.
  - Anzeige von Aktivitäten anderer auf dem Dashboard.

1.4.4 Github-API

Möglichkeit um...

- ...GitHub in Anwendungen zu integrieren.
- ...Volltextsuche auf allen Repositories.
- ...Statisten.
- ...Activities. (Alternative zu `git hooks`)
- ...Aktionen zu triggern (z.B. Pull Request).

```
# Alle Repositories eines Users auflisten
$ curl -q https://api.github.com/users/qitta/repos \
  | grep 'full_name'
"full_name": "qitta/dotfiles",
"full_name": "qitta/foozel",
"full_name": "qitta/scripts",
```

1.4.5 `git hooks`

- Mechanismus um in wichtige git-commandos einzuhaken
- Meist kleine Shell-Scripte:

```
$ echo "echo I am a hook." > .git/hooks/pre-commit
$ git commit -am "some message"
I am a hook.
# Auf Zweig master
# Ihr Zweig ist vor 'origin/master' um 3 Versionen.
# ...
```

- Hooks werden durch bestimmte Namen identifiziert
  - `pre-commit, prepare-commit-msg, commit-msg, post-commit`
  - `pre-receive, update`

Demo: Cloud-Hooks

- Twitter

  Commit-Messages auf Twitter posten.
- TravisCI

  `make && make test`
- ReadTheDocs

  Generierung von Dokumentation.
- Bugzilla

  Linking von Bugs in Commit-Message.
- Email

  Bei Commit Email an Mailingliste schicken.

1.4.6 Meine Damen und Herren...

Die Haupattraktion des Tages: *Lasst die Spiele beginnen*

1.5 Lasst die Spiele beginnen

1.5.1 I. Git Basics

- Einfach VM starten.
- Terminal aufmachen.
- Aufgabe auf dem Blatt befolgen.
- Zeit: ca. 15 Minuten

1.5.2 II. Collaboration Game

- Tut euch in 2-3er Gruppen zusammen.
- Maximal 8 Gruppen.
- Jede Gruppe bekommt einen Task.
- Jede Gruppe legt sich einen GitHub Account an.
- Forkt das Projekt: https://github.com/studentkittens/git-python-project
- Clont das Projekt in eure VM.
- Der Task kann entweder durch Überlegen oder git Kommandos gelöst werden.
- Wenn fertig: pusht es zu euren Fork.
- Macht ein Pull Request zum Original Repository.
- Falls richtig erledigt wird die LED grüner werden.

1.5.3 III. Visualize!