Git und die Wolke Documentation

Release

Christoph Piechula, Christopher Pahl

June 14, 2013

KAPITEL

ONE

KURZBESCHREIBUNG

Stichpunktartig sind die Ziele unseres **Git und die Wolke**:

- Vermittlung von Basics in der Versionsverwaltung anhand von `git`.
- Weitergehende Techniken wie Workflows.
- Überleitung zum Cloud-Thema mit `git-annex`.
- Einführung in die Code-Hosting Plattform GitHub.
- Praxisübung mit `git` lokal und GitHub.

Die Einführung halten wir für nötig um überhaupt zu verstehen was Github ist und welche Möglichkeiten es bietet. So gesehen ist GitHub der Schwerpunkt des Vortrags, auch wenn das Thema nicht die meiste Zeit belegt.

Folgender grober Zeitplan für den 180-Minuten Vortrag sollte zudem ein Gefühl dafür geben welche Themen uns besonders am Herzen liegen:

Thema	**Dauer in min.**
`git`-Einführung Teil I	50
Kleine Pause	5
`git`-Einführung Teil I	20
Überblick: `git-annex`	15
Große Pause	15
Einführung in Github	45
Kleine Pause	5
Übung	40
	195

KAPITEL

TWO

VORWORT

2.1 Motivation

Das Thema `git` ist für ein Fach wie Cloud-Computing sicherlich erst einmal ungewöhnlich. Auf den zweiten Blick finden sich aber einige Parallelen:

- Verteiltes Protokoll
- Plattformen wie Github
- Möglichkeit von Cloud Storage mit `git annex`
- Integration in viele Dienste mit `git hooks` möglich:
  - Continious Integration mit TravisCI
  - automatisiertes Release-Management
  - u.v.m (mehr dazu im Inhalt)

Abgesehen davon finden wir einfach dass `git` (oder eine vergleichbares DVCS) zur Grundaustattung jedes Informatikers gehören sollte. Deswegen wollen wir den Workshop gewißermaßen zur Allgemeinbildung der Informatiker nutzen.

Zudem ist `git` nicht nur ein Tool dass man in seinem Repertoire hat, es führt auch zu einer ganz anderen, verteilteren Arbeitsweise, welche die Zusammenarbeit mehrerer Personen konsequent erleichtert.

Wie üblich bei uns wird der Vortrag relativ technisch, da wir der Meinung sind dass man als praktischer Informatiker so am meisten lernt. Deshalb haben wir der Authentizität wegen viele Code-Beispiele auf diesem Paper den Folien entnommen, um nicht zusätzliche Verwirrung zu stiften.

2.2 Ziel

Das Ziel des Vortrages ist nicht den einzelnen Studenten detaillierte `git` Kenntnisse zu vermitteln. Dies ist in der kurzen Zeit auch gar nicht machbar. Vielmehr geht es uns darum den Studenten einen Überblick über die Möglichkeiten von `git` und den damit verbundenen Cloud-Services zu verschaffen. Wir hoffen dass die Studenten später das erlernte Wissen zum gemeinsamen Arbeit untereinander nutzen können. Und auch dass `git` unter den Studenten endlich Dropbox für die Versionsverwaltung von Quellcode ablöst. Wir haben in dieser Beziehung leider schon schlechte Erfahrungen mit Kommilitonen sammeln müssen.

Ein weiteres Ziel ist natürlich auch dass der Vortrag uns selbst interessiert. Denn nur ein interessierter Vortragender kann den Stoff überzeugend und fachkompetent vermitteln.

2.3 Vortragsstil

Entsprechend dem Thema der Vorlesung ist der Vortragsstil Workshop-artig aufgebaut. Die Themen werden interaktiv auch immer in der Shell gezeigt um die Theorie aufzulockern und den Zuhörern ein Gefühl für die Materie zu geben.

KAPITEL

THREE

INHALT

3.1 Was ist git?

Von [Wikipedia]:

> Git ([t], engl. Blödmann) ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die ursprünglich für die Quellcode-Verwaltung des Linux-Kernels entwickelt wurde.

Detaillierte Beschreibung von Linus Torvalds:

> Linus Torvalds on GoogleTalk.

3.1.1 Übersicht der git-Befehle:

Befehle mit einem Sternchen werden im Vortrag behandelt.

```
* add        stellt Dateiinhalte zur Eintragung bereit
* bisect     Findet über eine Binärsuche die Änderungen,
             die einen Fehler verursacht haben
* branch     Zeigt an, erstellt oder entfernt Zweige
* checkout   Checkt Zweige oder Pfade im Arbeitszweig aus
* clone      Klont ein Projektarchiv in einem neuen Verzeichnis
* commit     Trägt Änderungen in das Projektarchiv ein
* diff       Zeigt Änderungen zwischen Versionen, Version und Arbeitszweig, etc. an
* fetch      Lädt Objekte und Referenzen von einem anderen Projektarchiv herunter
* grep       Stellt Zeilen dar, die einem Muster entsprechen
* init       Erstellt ein leeres Git-Projektarchiv oder
             initialisiert ein bestehendes neu
* log        Zeigt Versionshistorie an
* merge      Führt zwei oder mehr Entwicklungszweige zusammen
  mv         Verschiebt oder benennt eine Datei, ein Verzeichnis, oder eine
             symbolische Verknüpfung um
* pull       Fordert Objekte von einem externen Projektarchiv an und führt
             sie mit einem anderen Projektarchiv oder einem lokalen Zweig zusammen
* push       Aktualisiert externe Referenzen mitsamt den verbundenen Objekten
* rebase     Baut lokale Versionen auf einem aktuellerem externen Zweig neu auf
* reset      Setzt die aktuelle Zweigspitze (HEAD) zu einem spezifizierten Zustand
  rm         Löscht Dateien im Arbeitszweig und von der Bereitstellung
* show       Zeigt verschiedene Arten von Objekten an
* status     Zeigt den Zustand des Arbeitszweiges an
* tag        Erzeugt, listet auf, löscht oder verifiziert ein
             mit GPG signiertes Markierungsobjekt
```

Quelle: [GitHelp].

Im folgenden wird jeweils eine kurze Erklärung dieser Befehle gegeben. Die abgebildeten Quelltexte sind aufgrund der Abbildungstreue den Folien entnommen.

3.2 git init

`git init` ist der zentrale Befehle um ein git-repository anlegen. Ein Repository ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten.

Code Beispiel aus den Folien:

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

3.3 git clone

Klont ein Repository von einer entfernten oder lokale Quelle. Das neu erzeugte Repository ist aufgrund des dezentralen Ansatzes vollkommen unabhängig voneinander.

Code Beispiel aus den Folien:

```
$ 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.
```

Git versteht verschiedene Protokolle:

```
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
```

3.4 git add

Mit `add` kann ein File dem sogenannten Staging-Bereich bekannt gemacht werden. Der Staging-Bereich ist eine Ablage um einen sogennanten Commit vorzubereiten. Dazu später mehr.

Die Syntax von `git add` ist eine recht einfache:

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

Ein Übersicht über die verschiedenen Bereiche ist hier gegeben, die meisten davon werden später davon ausführlich erläutert:

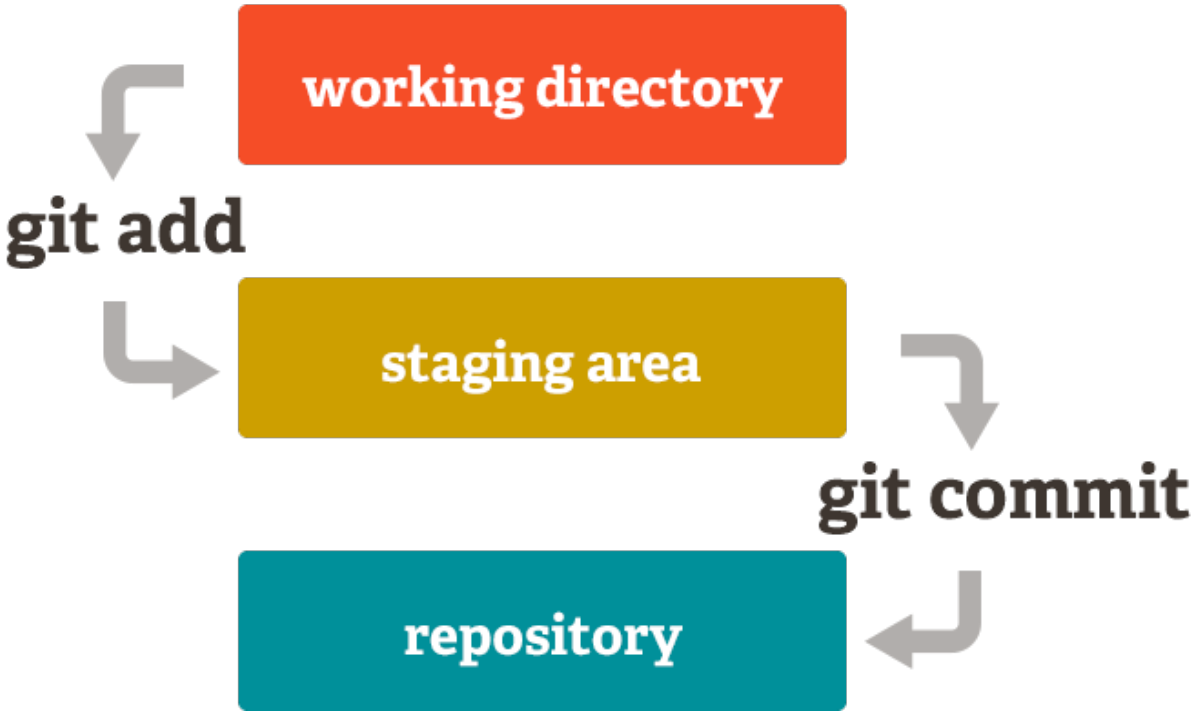

Abbildung 3.1: Übersicht git staging Bereich Quelle: [GitSCM]

Auch in interaktiver HTML-Form:

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

3.5 git commit

Ein Commit ist die kleinste Einheit in einem Repository. Er bildet einen Knoten im sogenannten Commit-Graphen. In einem Commit enthalten sind die Änderungen zu dem vorhergenden Commit, ein sogenanntes **Change-Set**.

Der Commit wird direkt aus dem Staging-Bereich erstellt.

Die Syntax von `git commit` ist die folgende:

```
# Anlegen einer Datei
$ echo "Hello Phil!" > README
```

```
# Bekannt machen im Staging-Bereich
$ git add README

# Anschauen des Staging-Bereichs
$ git status
# On branch master
# Changes to be committed:
#   new file:   README

# Verpacken des Staging-Bereiches in ein Changeset
$ git commit --all --message "commit message"  # ausgechrieben
$ git commit -am "commit message"              # oder kürzer
$ git commit -a                                # lange messages
[Editor öffnet sich]

# Nach dem Commit ist der Staging-Bereich wieder frei.
$ git status
# On branch master
nothing to commit, working directory clean
```

Grob kann man auch sagen dass ein Commit eine Änderung ist. Zugeorndet zu einem Commit ist eine Commit-Message. Diese wird vom Anwender verfasst und enthält eine Beschreibung der gemachten Änderungen. Diese sollte kurz, aber aussagekräftig sein.

3.6 Was ist ein `diff`?

Ein `diff` ist die Änderung zwischen zwei Änderungen. Die Änderungen werden dabei in einem definierten Format ausgegeben:

- Jede hinzugefügte Zeile wird mit einem **+** angefangen.
- Jede gelöschte Zeile wird mit einem **-** angefangen.
- Zudem werden darum herum Zeilennummern und andere unveränderte Zeilen angezeigt.
- Oben steht jeweils ein Header mit Information über das betreffende File.

```
# 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

# Bestimmte commit zeigen
$ git show a2fe0bc
<dasselbe wie oben>
```

3.7 Freunde von `git commit`

Es gibt einige Kommandos die oft in Verbindung mit `git commit` genutzt werden.

Berichtigung der letzten Commit-Message:

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

Hervolholen einer alten, in diesem Fall der letzten, Version:

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

Rückgängig machen von `git add`:

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

Kurzes Wegsichern von Änderungen auf einem "Änderungsstapel":

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

3.8 Die Objektdatenbank

`git init` legt ein `.git` Verzeichniss im Repository an, in das die verwalteten Objecte abgespeichert werden.

Die unterschiedlichen Objekttypen sind:

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

Beispiele aus der offiziellen git Dokumentation:

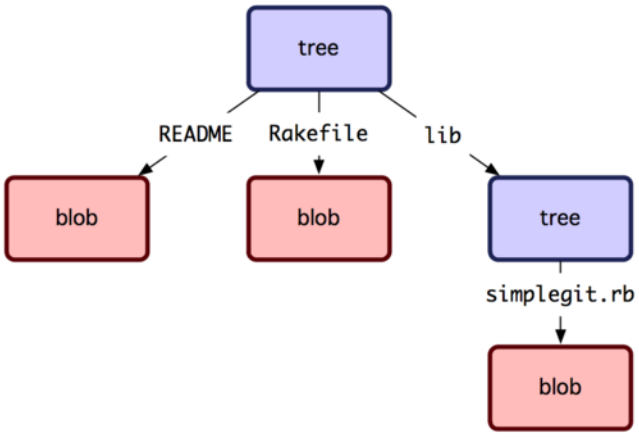

Abbildung 3.2: Einfaches Beispiel mit Blobs und Trees Quelle: [GitSCM]

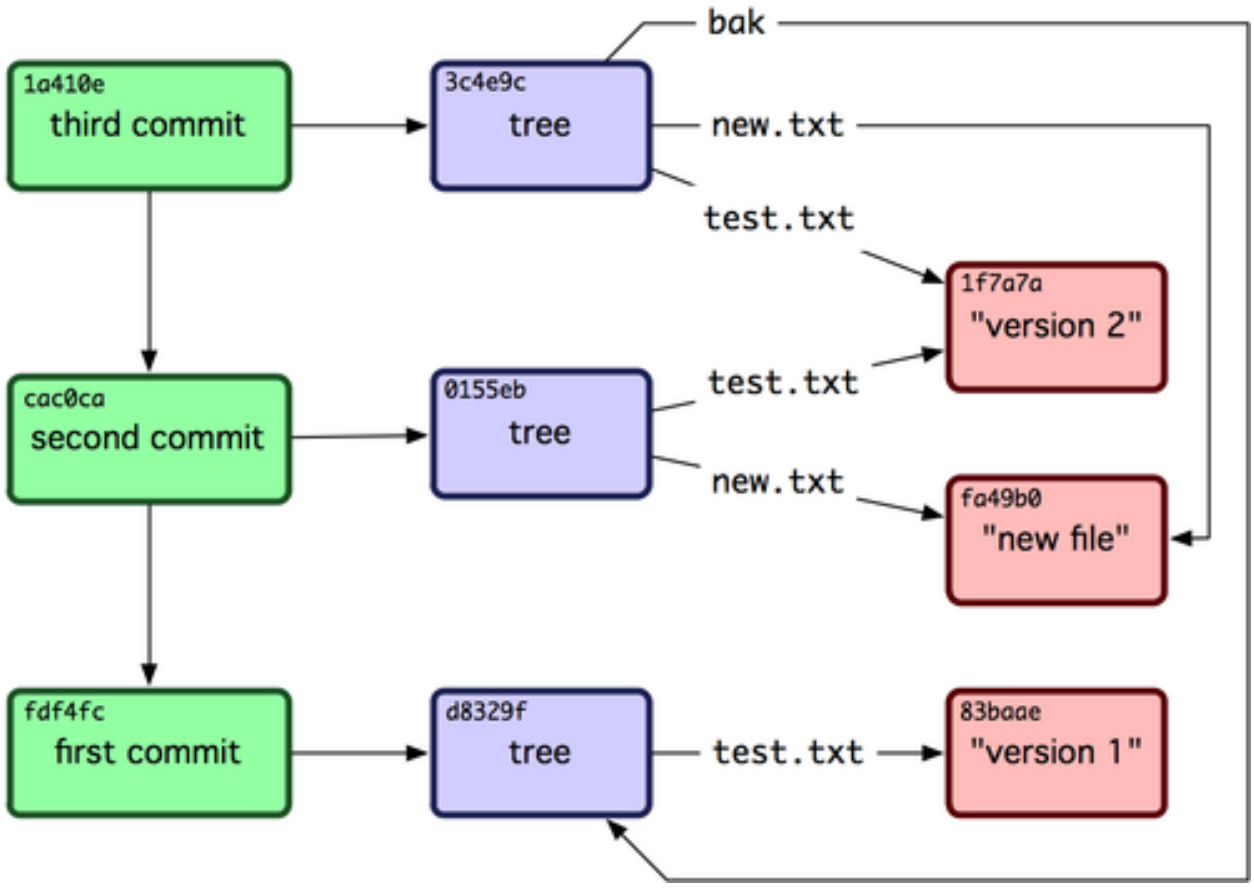

Abbildung 3.3: Erweiterung um Commit Objekte Quelle: [GitSCM]

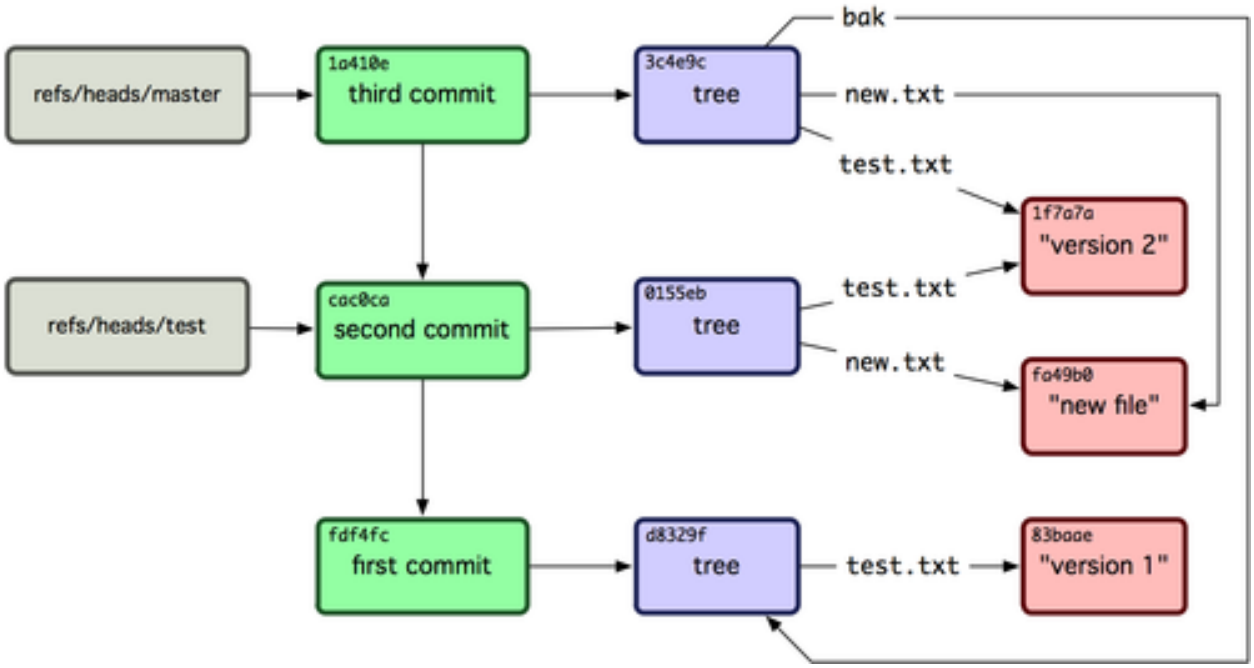

Abbildung 3.4: Erweiterung um Referenzen Quelle: [GitSCM]

3.9 Git Branching

Wie hoffentlich oben aus den Bildern hervorgegangen ist sind Branches lediglich Zeiger auf bestimmte Commits innerhalb des Baums. Ein Branch zeigt immer auf den aktuellsten Commit innerhalb eines Zweigs des Baums.

Aus einer weniger theorethischen Sichtweise sind Branches eine Möglichkeit um von einander abgetrennte Arbeitsbereiche zu schaffen. So werden beispielsweise einzelne Features in einem Projekt oft in seperaten Branches entwickelt. Diese werden später dann mit dem Hauptzweig **master** verschmolzen (mit `git merge`).

Unten stehend finden sich einige Beispiele zur Anwendung von Branches:

Branches erstellt man mit:

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

In bestehende branches wechseln:

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

Branches auflisten:

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

Branches führt man zusammen mit:

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

Bei einem Merge kann es zu sogneannten Merge-Conflicts kommen. Wenn beispielsweise in einer Datei in der gleichen Zeile in verschiedenen Branches etwas geändert wurde gibt es beim Mergen dieser einen Merge-Conflict. Dieser kann nicht automatisch von `git` zusammengeführt werden und man wird aufgefordert diesen manuell zu beheben. Zu diesem Zweck werden in der Datei **Merge-Marker** eingefügt welche die betroffene Stelle markieren.

Merges sind oft eine Quelle vielfacher Verwirrung bei `git`-Anfängern, weshalb wir interaktiv darauf besonders eingehen werden.

3.10 Remotes

Ein Remote ist bei `git` eine externe Quelle die auf ein Repository zeigt. Bisher haben wir `git`, bis auf `git clone` lokal genutzt, nun kommen remotes ins Spiel.

Die Vorgehensweise bei einem zentralem Versionsverwaltungsystem.

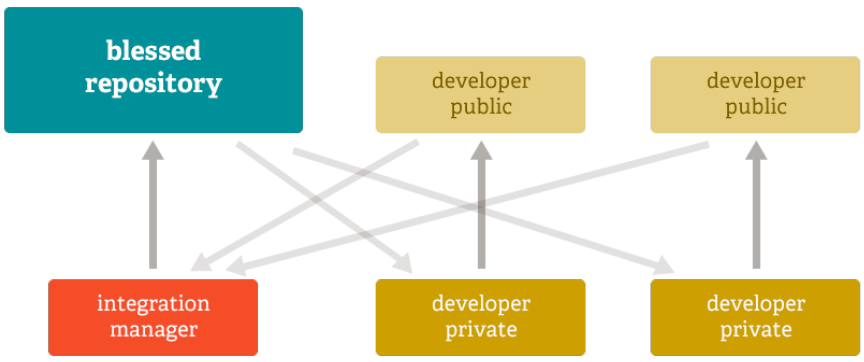

Abbildung 3.5: Workflow bei z.B. SVN
Quelle: [GitSCM]

Dezentrale Arbeitsweise im Kontrast dazu. In diesem Beispiel ist Alice beispielsweise eine Maintainerin und hat Commit-Rechte, Bob jedoch ist nur ein Entwickler der keine zugriffsrechte auf das zentrale Repository hat. Deshalb pullt er jeweils nur vom zentralen Repo die Änderungen, arbeitet auf seiner Codebasis und schickt Alice einen sogenannten `pull request`. Alice kann diesen nun integrieren und in das zentrale Repository einpflegen.

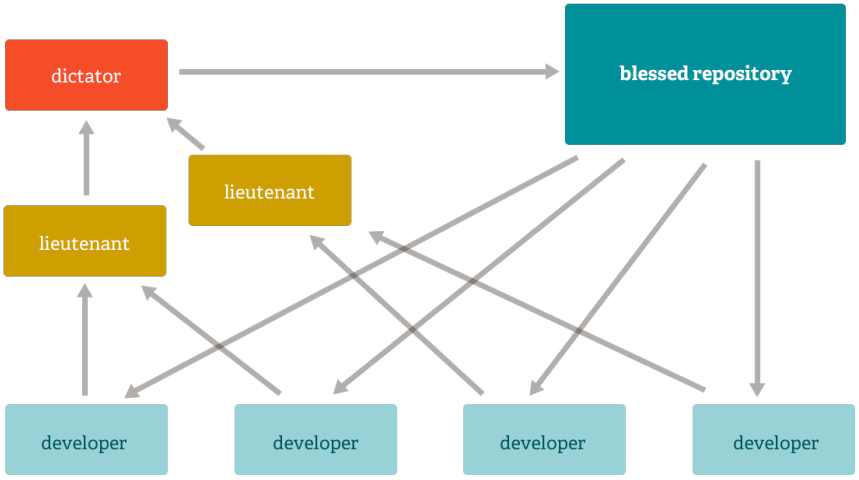

Abbildung 3.6: Dezentraler Workflow mit Dikator
Quelle: [GitSCM]

Bedienung von `git-remote`:

```
# 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
```

3.11 `git push`

`git push` ist einer der meistgenutzten git Befehle. Er dient dazu, seinen Commit auf eines externes Repository zu übertragen.

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

Wie man hier im Beispiel sieht, gibt es mehrere Möglichkeiten. Das `origin` ist in diesem Fall der Remote, von dem das Repository ursprünglich geklont wurde. Zusätzlich kann hier auch noch der zu pushende Branch als zweites Argument angegeben werden. In unserem Fall ist das der `master`.

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

3.12 git pull

`git pull` ist das Äquivalent zu `git push`. Es zieht Änderungen vom Remote, und `merged` diese mit dem aktuellen Branch.

Syntax von `git pull`:

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

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

3.13 git fetch

`git fetch` ist ein Teilbefehl des `convinience` Befehls `git pull`. Er dient Hauptsächlich dazu den Mergeschritt zu vermeiden, wenn man z.B. vorher den Code validieren will.

Beispiel:

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

3.14 git bisect

Aus der man-page von `git bisect(1)` ([GitBisectMan]):

```
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.

Fehlerhafter Beispielquelltext:

```
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");
}
```

Testcase:

```
void test_is_odd(void) {
    for(int i = -20; i < 20; ++i) {
        assert(is_odd(i) == (i % 2 == 1));
    }
}
```

Hier im Beispiel wird der bisect Befehl genutzt um die erste fehlerhafte Version des oben gezeigten Quelltextes zu finden.

```
$ 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;
 }
```

Daraus sind folgende Schlüsse zu ziehen:

- Immer kleine commits machen.
- Zeit nehmen für sinnvolle Commit Messages.
- `git bisect` ist ein gutes Argument für Unit-Tests.

3.15 `git tag`

Bei bestimmten Releases oder Meilensteinen ist es sinnvoll diese mit einem entsprechenden Schlagwort zu taggen. Für diesen Einsatzzweck ist `git tag` gedacht. `git tags` sind wie Branches mit dem Unterschied dass diese immer statisch auf einem bestimmten Commit zeigen, Branches im Gegensatz dazu zeigen immer auf den aktuellen Commit eines Branches.

Hier 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>
```

3.16 Workflow-Model

Wie in der Grafik zu sehen werden verschiedene Branches bei der Entwicklung nach diesem Modell verwendet. Der Master Branch, ist die Grundbasis für alle anderen Branches. Der Master Branch spiegelt stets ein funktionierendes Release wieder. Die eigentliche Arbeit geschieht im Develop Branch, hier ist auch immer die Entwicklerversion zu finden aus, der z.B. `nightly builds` hervorgehen. Neue Features werden in seperaten Feature Branch entwickelt und fließen nach erfolgreicher Testphase in den Develop Branch zurück. Release Candidates werden im Release Branch vorbereitet und fließen in den Master Branch ein. Bugfixes die nach einem offiziellen Release gefunden wurden, kommen in den Hotfix Branch und fließen direkt ins nächste Release und in den Develop Branch mit ein.

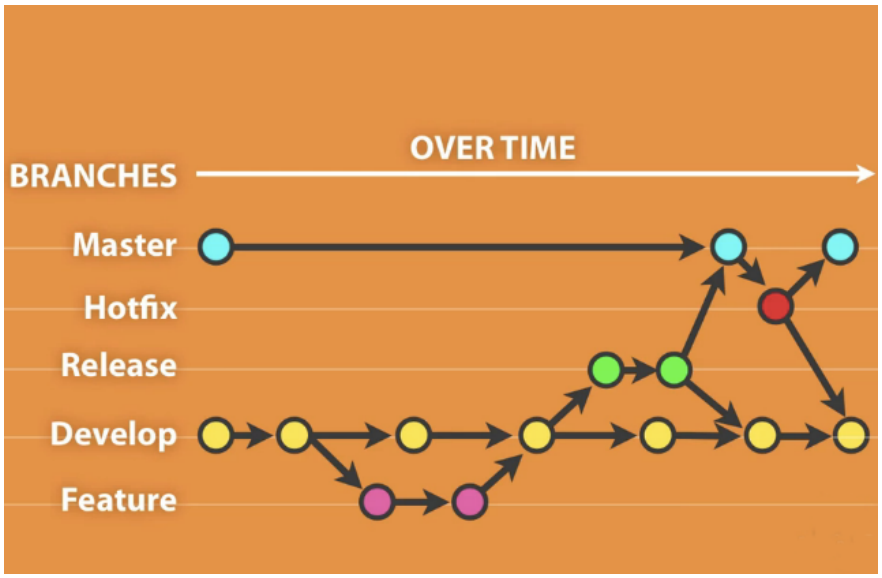

Abbildung 3.7: Darstellung des gitflow-modells. Quelle [GitFlow]

Der Vorteil dieser Vorgehensweise ist die gute Skalierung bei großen Projekten und Teams. Nachteilig muss man sagen, dass diese Vorgehensweise am Anfang sehr gewöhnungsbedürftig ist und von den Teammitgliedern gelebt werden muss.

3.17 Git Plugins und Werkzeuge

Hier eine kleine Zusammenstellung der beim weitesten verbreiteten Git Plugins und Werkzeuge:

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

3.18 Best Practices

Abschließend zu den git Grundlagen stichpunktartig noch ein paar Best Practices.

- `.gitignore` nutzen (und `git clean`!).
  - Keinen autogenerierten Code/Projektdateien committen.
  - Wenn nicht vermeidbar dann in eigenen Commit.
  - Für Dokumentation am besten eigenen Branch nutzen!
- Sinnvolle commit messages.
- 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.

3.19 `git rebase`

Zum Abschluß `git rebase`, einer der komplizierten git Befehle. Bildlich gesprochen dient Rebase dazu einen Zweig abzubrechen und anderswo anzuflanschen. Dies dient dazu, die Basis eines Branches auf neuen Änderungen aufzubauen.

Ausgangszustand:

Folgendes würde bei einem normalen Merge passieren:

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

Mit Rebase, hat man in unserem Beispiel den Vorteil dass kein zusätzlicher Merge Commit entsteht, der Branch wird praktisch direkt mit dem Master verschmolzen.

```
$ 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'
```

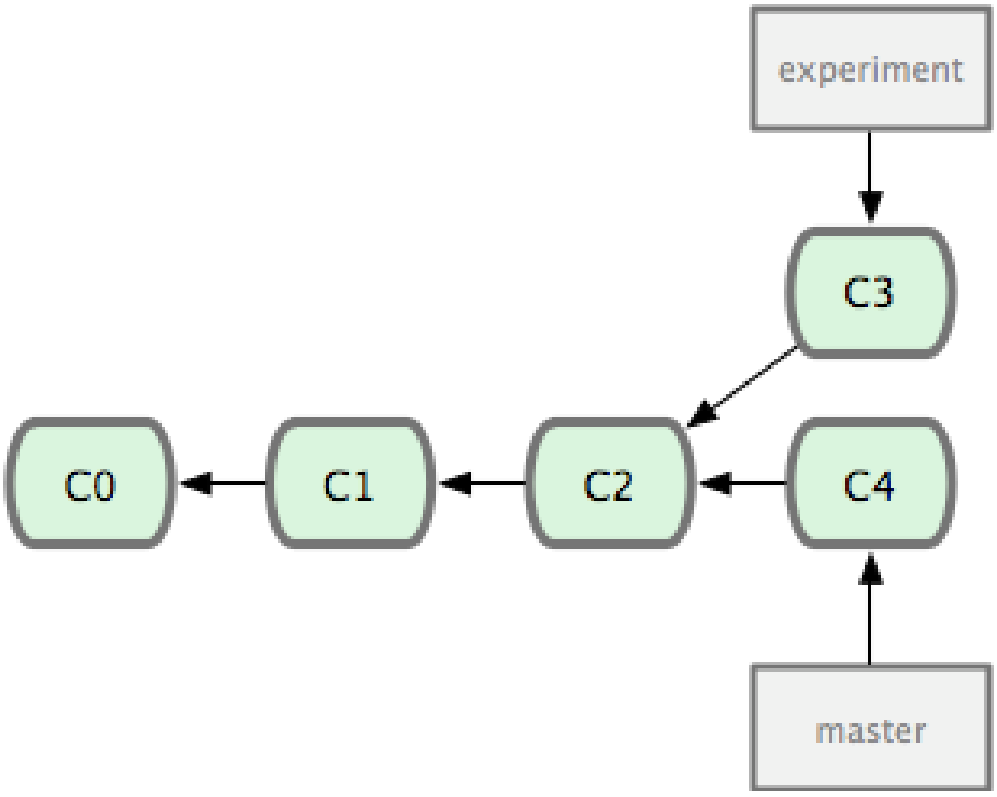

Abbildung 3.8: Ausgangszustand vor dem Merge/Rebase Quelle: [GitSCM]

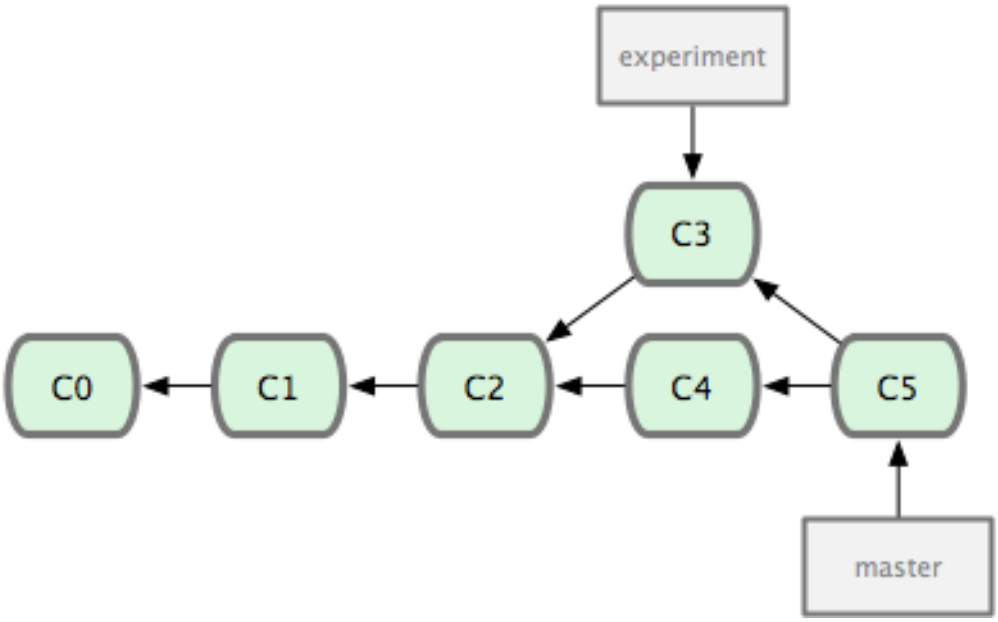

Abbildung 3.9: Zustand nach einem Merge Quelle: [GitSCM]

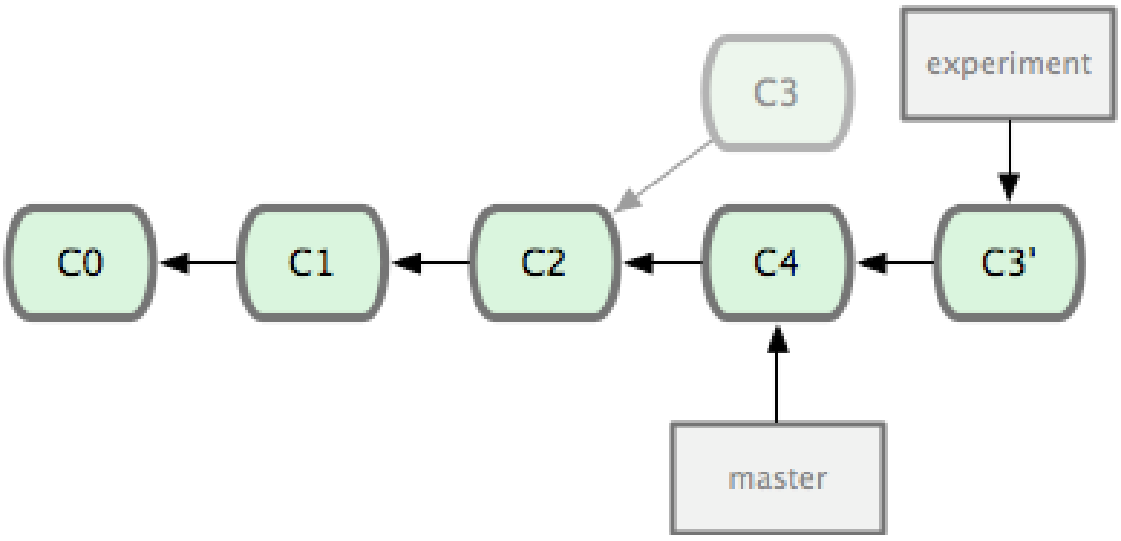

Abbildung 3.10: Zustand nach einem git rebase Quelle: [GitSCM]

3.20 Suchen und Beschuldigen

Suche `background:` in allen `.css` Dateien mit `git grep`.

```
$ 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);
```

Mit `git blame` kann man herausfinden wer zuletzt an den gesuchten Zeilen etwas geändert hat.

```
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);
```

Daraus kann man schließen dass der Autor **Katze** die Mozilla-Zeile eingefügt hat.

KAPITEL

FOUR

GIT ANNEX: DROPBOX FUER HARTE

4.1 Was ist `git-annex`?

Problem:

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

Abhilfe schafft hier `git annex` das ein Wrapper um `git` ist. Hierbei werden nur Metadaten verwaltet, keine Changesets der Binärdateien. Deshalb wird der Content nicht *getrackt*, aber für die Metadaten stehen alle git Features weiterhin bereit.

4.2 Wie funktioniert das Ganze?

Die Vorgehensweise beim Anlegen eines `git annex` Repository. Der Einfachkeit halber ist der selbsterklärende Code von den Folien kopiert worden.

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...)
```

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
```

Auf dem Gegenpart kann nun ein "Spiegel" des Repositories eingerichtet werden um wie bei Dropbox die Datein tatsächlich zu synchroniseren. Hierbei werden nur die Metadaten synchronisiert, wie beim ersten `git annex sync` zu sehen ist. Das Bild kann noch nicht geöffnet werden, da der Content erst mit `git annex get` besorgt werden muss.

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...)
```

4.3 Nachteile

Momentan ist `git annex` ist noch recht stark in der Entwicklung, daher kommt es momentan recht oft noch zu unerwarteten Verhalten. Deshalb ist `git annex` momentan noch etwas für Early Adaptors und andere Geeks.

Mittlerweile gibt es aber auch ein ästhetisches Webfrontend dass auch von normalen Usern benutzt werden kann. Mehr dazu weiter unten.

4.4 Warum überhaupt das Ganze, es gibt doch Dropbox?

Stichpunktartige Features die Dropbox nicht bietet:

- 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 Storagelösung erwartet

4.5 Frontend

Zur einfachere Benutzung gibt es auch ein Webfrontend dass die Bedienung von `git annex` massiv vereinfacht. Bevor wir hier übermäßig schreiben, verlinken wir einfach ein Frontend-Demo des Entwicklers:

http://downloads.kitenet.net/videos/git-annex/git-annex-xmpp-pairing.ogv

KAPITEL

FIVE

GITHUB: AB IN DIE WOLKE

5.1 Was ist Github?

Ein (Social-)Code-Hosting Dienst der 2008 gegründet wurde. Die Plattform ist mithilfe von Ruby on Rails und Erlang geschrieben.

Statisiken:

- ~3,5 Millionen User
- 6 Millionen Repositories
- 158 Mitarbeiter
- April 2008 mit 6000 Usern und 2500 repos gestartet.

Bekannte Projekte die auf Github gehostet sind:

- Erlang, PHP, Perl, Clojure
- Mirros: Linux-Kernel, Ruby
- Git selbst.

5.2 Was kann Github?

Github ist für OpenSource Projekte kostenlos. Die einzig existierende Einschränkung ist eine Begrenzung des Speicherplatzes auf 1 GB. Dies macht Github als Dropbox-Alternative uninteressant. Allerdings kann man sogar große Projekte wie Eclipse auf Github hosten.

Im Cloud Context ist Github ein Infrastructure as a Service as a Service.

Für Businesskunden gibt es entweder private Repositories, oder eine Lizenz der Software die auf Github läuft (das sogennante `Github Enterprise`). So ist es möglich dass Github in großen Unternehmen praktisch abgekapselt betrieben werden kann.

Zudem findet sich auf Github ein Pastebin-Klon namens `Gist` der Versionierung unterstüzt. Dieser bietet sich immer dann wenn ein volles Repository zu aufwendig wäre.

Zudem unterstützt die Plattform Volltextsuche auf alle Repositories.

Das Maskottchen von Github ist die Octocat:

Anmerkung: Der folgende Teil ist wieder direkt von den Folien, da diese bereits jetzt ausführlich sind und ausgiebig von URL-Verlinkungen Gebraucht macht. Daher macht es kaum Sinn diesen Teil neu zu verfassen.

Abbildung 5.1: Github Maskottchen. Quelle: [Octocat]

5.2.1 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, BugReports schreiben u.v.m

5.2.2 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**.

5.2.3 Demo: Pull Requests

Ablauf ohne Github:

```
# Auf Seite des Forkers (bob)
$ git request-pull HEAD^1 https://github.com/<bob>/repo.git > mail
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

# Auf Seite des Annehmers (alice)
$ git remote add bob https://github.com/<bob>/repo.git
$ git pull bob
```

Ablauf mit Github:

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

5.2.4 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)

5.2.5 Demo: Online Blame/Annotate/Edit

Code lässt sich online:

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

Auch Bilder, Dokumente und Videos sind previewbar.

5.2.6 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
```

5.2.7 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.

5.3 Github-API

Möglichkeit um...

- ...GitHub in Anwendungen zu integrieren.
- ...Volltextsuche auf allen Repositories.
- ...Statisken.
- ...Activities. (Alternative zu `git hooks`)
- ...Aktionen zu triggern (zb. Pull Requests).

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

5.4 `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`

5.4.1 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.

KAPITEL

SIX

ÜBUNG ZU GIT UND GITHUB

Die Übungen sollen dazu dienen das Gelernte spielerisch zu festigen. Sie bestehen aus Zwei Teilen:

1. **GitBasics**: Einfache Übungen die man lokal machen kann. Jeder der die Übung abschließt beommt einen Octocat-Sticker.
2. **GitHubGame**: Kleines Spiel bei dem Gruppen aus 2-3 Leuten gebildet werden. Diese bekommen dann einen Task bei dem sie eine in Python geschriebene Funktion berichtigen müssen. Dies kann entweder durch reine Überlegung geschehen, oder durch Verwendung von git tools.

Zum Zwecke der Übung wird eine Virtual Maschine ausgeteilt.

Genaueres kann in beigelegten `exercise.pdf` nachgelesen werden.

[Wikipedia] http://de.wikipedia.org/wiki/Git

[GitHelp] https://www.kernel.org/pub/software/scm/git/docs/

[GitBisectMan] https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html

[GitFlow] http://joefleming.net/2012/06/07/git-flow/

[Octocat] http://octodex.github.com/

[GitSCM] http://git-scm.com/documentation