GIT

LERNEN MIT BEISPIELEN

Eine Schritt für Schritt Anleitung für den GIT-Neuling

Andy Theiler
Xtreme Software GmbH
www.x3m.ch

Version: 3. Februar 2015

INHALTSVERZEICHNIS

EINLEITUNG

Für wen ist dieses Dokument geeignet?

Dieses Dokument richtet sich an den Git-Neuling. Wie der Titel "GIT - Lernen mit Beispielen" schon vermuten lässt, wird anhand von Beispielen das Handling mit Git erklärt. Es soll einen einfachen Einstieg in die Versionsverwaltung mit Git bieten.

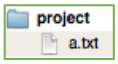
Der Hauptfokus liegt im Verstehen der Git-Konzepte und bei den Git-Befehlen. Auf die Funktionsweise von grafischen Git-Clients wird nicht eingegangen. Diese Anleitung ist in keinster Weise vollständig. Themen wie "Git downloaden" oder "Die Installation von Git" fehlen komplett!

GRUNDLAGEN

Ein Git Repository anlegen

Ein Git Repository kann in jedem beliebigen Verzeichnis erzeugt werden. Egal ob dort schon Dateien vorhanden sind oder das Verzeichnis leer ist.

In unserem Beispiel erstellen wir das neue Repository in dem bereits existierenden Verzeichnis „**project**". In diesem Verzeichnis ist die Datei „**a.txt**" bereits vorhanden.



Dazu öffnen wird das Terminal und wechseln in das Verzeichnis „project".

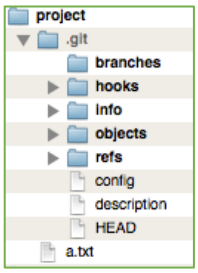
Befehl:

```
$ cd project
$ git init
```

Output:

```
Initialized empty Git repository in /Users/andy/Documents/project/.git/
```

Der **init** Befehl erzeugt ein Unterverzeichnis **.git**, in dem alle für das Git Repository benötigten Daten (Verzeichnisse und Dateien) erzeugt werden.



Zustand der Dateien überprüfen

Mit dem **status** Befehl kann jederzeit der Zustand der Dateien überprüft werden.

Befehl:

```
$ git status
```

Output:

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

Der **status** Befehl teilt uns mit, dass wir uns aktuell auf dem Branch „master" befinden (Mit Branches befassen wir uns später) und dass die Datei „**a.txt**" nicht unter der Versionskontrolle von Git ist.

Vielfach kriegen wir vom status Befehl weiterführende Informationen und Tips. In unserem Beispiel wird uns erklärt, wie die Datei „a.txt" der Versionskontrolle (engl. tracked) von Git hinzugefügt werden kann.

Wird der status Befehl mit dem -s (oder --short) Parameter ausgeführt, erscheint das Resultat in kompakter Form aber ohne die zusätzlichen Tips.

Befehl:

```
$ git status -s
```

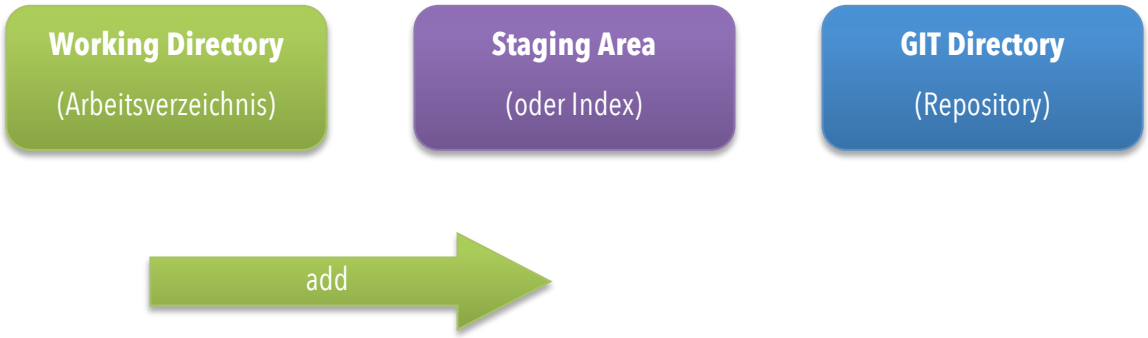
Output:

```
?? a.txt
```

?? bedeutet in unserem Fall, dass die Datei „untracked", also nicht unter der Versionskontrolle von Git ist

Datei zur Versionskontrolle hinzufügen

Wir werden nun die Datei „a.txt" in die „Staging Area" aufnehmen bzw. der Versionskontrolle hinzufügen. Danach überprüfen wir den Status der Datei.



Befehl:

```
$ git add a.txt
$ git status
```

Output:

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

Der Status unserer Datei hat von „untracked" auf „added" bzw. „staged" gewechselt und ist so für den Commit vorgemerkt. Was ein Commit ist, erkläre ich später.

Man verwendet **add** nicht nur um „neue" Dateien zur Versionskontrolle hinzuzufügen, sondern auch um bestehende Dateien (d.h. welche bereits von Git getracked werden) für den Commit vorzubereiten.

Kurz: hinzufügen heisst, die Datei aus dem „Working Directory" (das Arbeitsverzeichnis) in der „Staging Area" (oder „Index Area") registrieren.

Git denkt in Änderungen, nicht in Dateien!

Im Gegensatz zu anderen Versionierungssystemen[1] setzt Git den Fokus auf die Änderungen von Dateien. Wenn wir eine Datei in den „Stage Bereich" hinzufügen bedeutet das noch nicht, dass wir diese Datei auch in das Repository committen. Das folgende Beispiel soll dieses Verhalten demonstrieren.

Wir ändern den Inhalt der Datei „a.txt" und lassen uns den Status anzeigen:

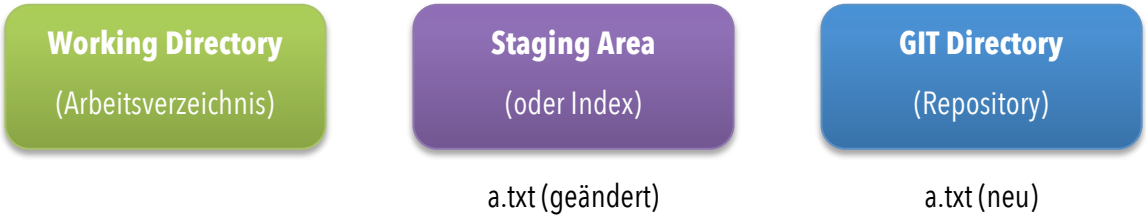
Befehl:

```
$ git status
```

Output:

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

Wir bemerken, dass die Datei „a.txt" zwei mal aufgelistet wird! Die erste Änderung (hinzufügen der neuen Datei) ist gestaged und bereit für den Commit. Die letzte Änderung hingegen ist (noch) nicht gestaged. Wenn wir nun einen Commit durchführen würden, wird nur die erste Änderung im Repository gespeichert.



[1] Beispiel Subversion: sobald eine Datei der Versionskontrolle hinzugefügt wurde, wird sie fortan getracked.

Einen Commit erzeugen

Committen bedeutet, den aktuellen Status des Projektes als eine Version im „GIT Verzeichnis" (d.h. in der lokalen Datenbank) dauerhaft zu speichern. Es werden nur Dateien berücksichtig, welche in der „Staging Area" vorgemerkt sind.

Auszug aus dem Buch „Pro Git" von Scott Chacon:

„Im Gegensatz zu anderen Versionierungssystemen sichert Git 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 Git unveränderte Dateien nicht, sondern legt lediglich eine Verknüpfung zu der vorherigen Version der Datei an."

Wir führen nun den Commit aus und prüfen dann erneut den Status.

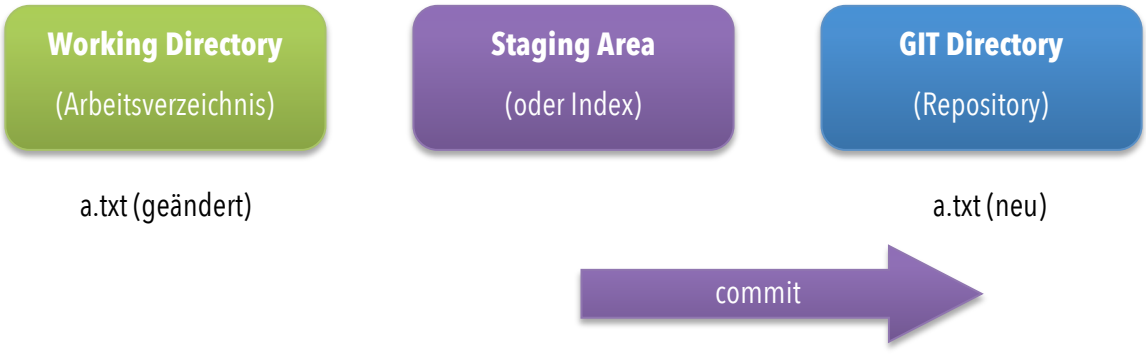
Befehl:

```
$ git commit -m "Mein erster Commit"
$ git status
```

Output:

```
$ git commit -m "Mein erster Commit"
[master (root-commit) 5372dce] Mein erster Commit
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt

$ git status
# On branch master
# 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:   a.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Der **status** Befehl sagt uns, dass die Datei "a.txt" den Zustand "modified" hat, aber noch nicht gestaged ist.



Dateien entfernen

Dazu betrachten wir folgende 2 Anwendungsfälle:

Datei "nur" aus der Versionskontrolle entfernen

Wir wollen eine Datei aus der Staging Area entfernen (die Datei soll von Git nicht mehr getracked werden), ohne sie im Arbeitsverzeichnis zu löschen. Dazu müssen wir nach dem **rm** Befehl den Parameter **--chached** verwenden:

Befehl:

```
$ git rm --cached a.txt
$ git status
```

Output:

```
$ git rm --cached a.txt
rm 'a.txt'

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    a.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       a.txt
```

Der status Befehl bestätigt uns, dass die Datei "a.txt" in der Staging Area als "deleted" gekennzeichnet wurde und dass die Datei im Arbeitsverzeichnis von Git nicht mehr getracked bzw. nicht mehr unter der Versionskontrolle ist.

Beim nächsten Commit würde die Datei "a.txt" aus dem Repository entfernt.

Datei aus der Versionskontrolle entfernen "und" lokal löschen

Wie möchten eine Datei sowohl aus der Versionskontrolle wie auch aus dem Arbeitsverzeichnis löschen. Da unsere Datei nach dem vorherigen Beispiel von Git nicht mehr getracked wird, werden wir als erstes diese wieder stagen und gleich wieder löschen.

Befehl:

```
$ git add a.txt
$ git rm a.txt
```

Output:

```
error: 'a.txt' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
```

Offensichtlich hat Git was gegen die Löschung. Git warnt uns, dass sich die Datei "a.txt" in der Staging Area befindet aber noch nicht committed wurde. Wenn uns das egal ist, dann führen wir den Befehl aus, wie ihn uns Git vorschlägt; nämlich mit dem **-f** Parameter.

Das machen wir auch, und schicken gleich den Commit hinterher:

Befehl:

```
$ git rm -f a.txt
$ git commit -m "Datei a.txt entfernt"
```

Output:

```
$ git rm -f a.txt
rm 'a.txt'

$ git commit -m "Datei a.txt entfernt"
[master 1f52f19] Datei a.txt entfernt
 1 file changed, 1 deletion(-)
 delete mode 100644 a.txt
```

Nach diesen Aktionen wurde die Datei "a.txt" wie gewünscht sowohl im Arbeitsverzeichnis wie auch aus dem Repository gelöscht.

Die Commit Historie

Nach dem obigen Beispiel könnte man meinen, dass die Datei "a.txt" unwiederruflich aus dem Git Repository entfernt wurde. Warum diese Vermutung glücklicherweise falsch ist, merken wir, wenn wir uns die Commit Historie anschauen.

Befehl:

```
$ git log
```

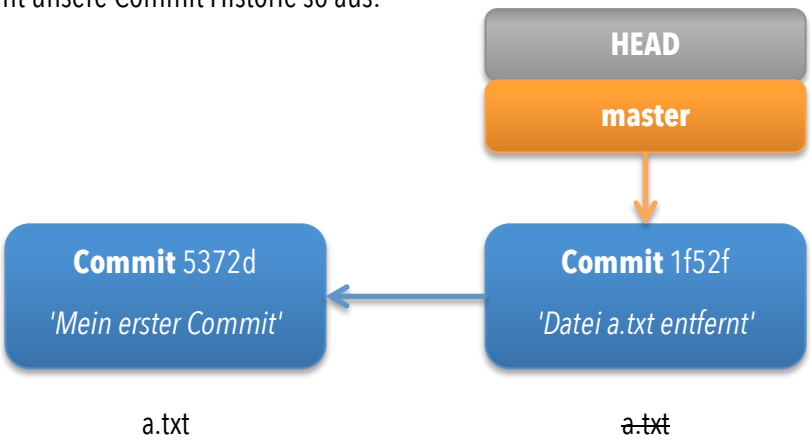
Output:

```
commit 1f52f19f82e1956dd86d98ec6d72e45475840aed
Author: Andy Theiler <andy.theiler@x3m.ch>
Commit: Andy Theiler <andy.theiler@x3m.ch>

    Datei a.txt entfernt

commit 5372dce1031958c238d8bc2004abccb190a1474e
Author: Andy Theiler <andy.theiler@x3m.ch>
Commit: Andy Theiler <andy.theiler@x3m.ch>

    Mein erster Commit
```

Als erstes fällt auf, dass die Historie aus den 2 Commits: "Mein erster Commit" und "Datei a.txt entfernt" besteht. Somit ist sichergestellt, dass die Datei "a.txt" nicht verloren ist, sondern sich im Snapshot des ersten Commits befindet.

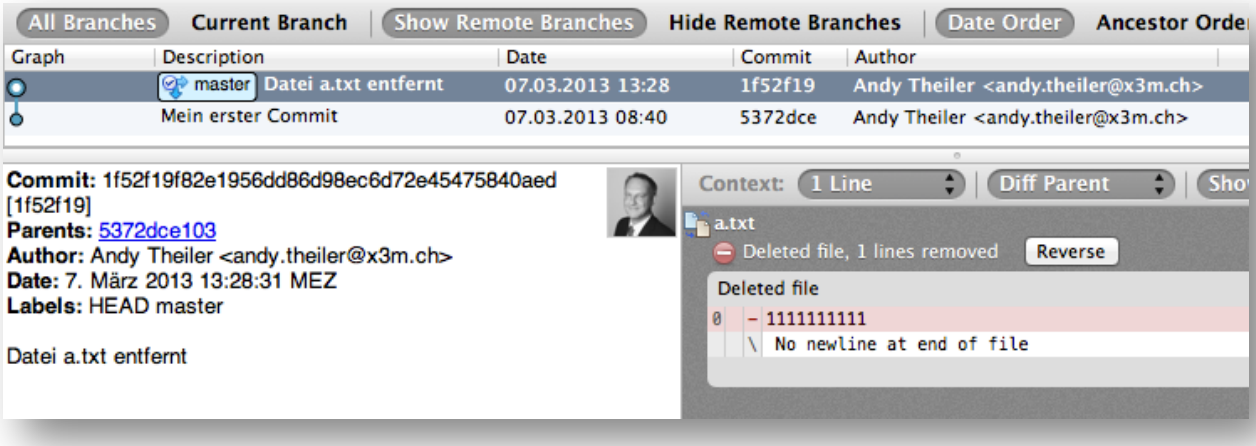
Weiter bemerken wird, dass jeder Commit mit einem 40stelligen Hexadezimalstring gekennzeichnet wird. Git generiert beim Commiten von Dateien und Verzeichnissen aus deren Inhalte einen 40stelligen SHA1-Hashwert. Denn Informationen über Dateien referenziert Git nicht nach ihren Dateinahmen sonder nach dem generierten Hash-Wert.

Grafisch dargestellt sieht unsere Commit Historie so aus:



Was ich bisher noch nicht erklärt habe: Der "**HEAD**" ist eine Referenz (einen simplen Zeiger) auf den aktuellen Branch[2]. In unserem Beispiel haben wir nur den Branch "master";

Ich verwende auf meinem Mac den Git-Client[3] "SourceTree", welche mir die Historie folgendermassen anzeigt:



Git erlaubt uns die Art der Ausgabe des **log** Befehls selbst zu bestimmen. Nachfolgend einige Beispiele.

Ein Commit pro Zeile ausgeben

Befehl:

```
$ git log --pretty=oneline
```

Output:

```
1f52f19f82e1956dd86d98ec6d72e45475840aed Datei a.txt entfernt
5372dce1031958c238d8bc2004abccb190a1474e Mein erster Commit
```

Ausgabe selbst formatieren

Befehl:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

Output:

```
1f52f19 - Andy Theiler, 2 hours ago : Datei a.txt entfernt
5372dce - Andy Theiler, 7 hours ago : Mein erster Commit
```

---

[2] Siehe Kapitel "Was ist ein Branch?"

[3] Siehe im Anhang im Kapitel "Grafische Git-Clients"

Änderungen je Commit anzeigen lassen, aber nur der letzte Eintrag

Befehl:

```
$ git log -p -1
```

Output:

```
commit 1f52f19f82e1956dd86d98ec6d72e45475840aed
Author: Andy Theiler <andy.theiler@x3m.ch>
Commit: Andy Theiler <andy.theiler@x3m.ch>

    Datei a.txt entfernt

diff --git a/a.txt b/a.txt
deleted file mode 100644
index aa75558..0000000
--- a/a.txt
+++ /dev/null
@@ -1 +0,0 @@
-1111111111
\ No newline at end of file
```

Änderungen mit Statistiken

Befehl:

```
$ git log --stat
```

Output:

```
commit 1f52f19f82e1956dd86d98ec6d72e45475840aed
Author: Andy Theiler <andy.theiler@x3m.ch>
Commit: Andy Theiler <andy.theiler@x3m.ch>

    Datei a.txt entfernt

 a.txt |    1 -
 1 file changed, 1 deletion(-)

commit 5372dce1031958c238d8bc2004abccb190a1474e
Author: Andy Theiler <andy.theiler@x3m.ch>
Commit: Andy Theiler <andy.theiler@x3m.ch>

    Mein erster Commit

 a.txt |    1 +
 1 file changed, 1 insertion(+)
```

Alle möglichen Ausgabe-Optionen sind hier beschrieben: http://git-scm.com/docs/git-log

Dateien aus dem Repository wiederherstellen

In die (Commit-) Vergangenheit zu springen ist ganz einfach. Mit dem **checkout** Befehl kann jeder beliebige Commit vom Repository in das Arbeitsverzeichnis kopiert werden. So wollen wir unsere gelöschte Datei "a.txt" aus dem 1. Commit wieder zurückholen. Die Commit-Nummer ermitteln wir wie eben erklärt mit **git log**. Die ersten 5 Stellen genügen[4], um den Commit eindeutig zu identifizieren.

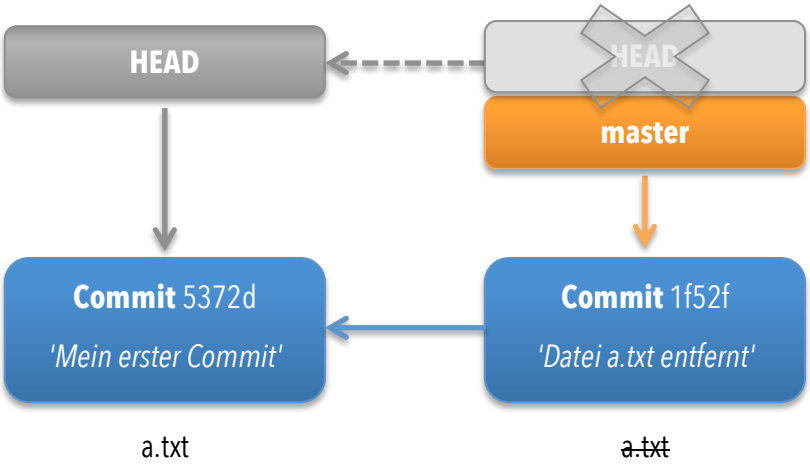
Befehl:

```
$ git checkout 5372d
```

Output:

```
Note: checking out '5372d'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 5372d... Mein erster Commit
```

Jetzt ist unsere vermisste Datei zwar wieder da, doch die Ausgabe des Checkouts ist nicht das, was wir uns erhofft haben: Git hat für uns einen anonymen Branch erstellt, genannt "detached HEAD", weil Git nicht in der Lage ist, dem Checkout einen Branchnamen zuzuordnen:



[4] mehr zum Thema Kurz-Form von SHA-1 Hashes: http://git-scm.com/book/en/Git-Tools-Revision-Selection

Hätten wir beim Checkout statt einen Commit (5372d) die Datei a.txt angegeben, würde der HEAD auf seinem Platz bleiben.

Weil sich der Csommit bei einem losgelösten (detached) etwas anderst verhält[5], machen wir unseren Checkout wieder rückgängig. Dann, wie eben besprochen, gezielt die Datei "a.txt" aus dem Commit 5372d holen:

Befehl:

```
$ git checkout master
$ git checkout 5372d a.txt
$ git status
```

Output:

```
$ git checkout master
Previous HEAD position was 5372d... Mein erster Commit
Switched to branch 'master'


$ git checkout 5372d a.txt

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a.txt
#
```

Geschafft: Die Datei a.txt ist wieder im Arbeitsverzeichnis und auch gleich in der Staging Area als "added" bzw. als "new" markiert.

Bevor wir die Datei wieder in das Repository commiten, fügen wir der Datei eine Zeile hinzu und speichern sie ab.

Befehl:

```
$ git add a.txt
$ git commit -m "Datei a.txt wieder hinzugefuegt"
```

Output:

```
[master 6d3008e] Datei a.txt wieder hinzugefuegt
1 file changed, 2 insertions(+)
create mode 100644 a.txt
```

[5] Commits würden dem unbenannten Branch hinzugefügt: http://marklodato.github.com/visual-git-guide/index-de.html#detached

Schauen wir uns die neue Commit-Historie an.

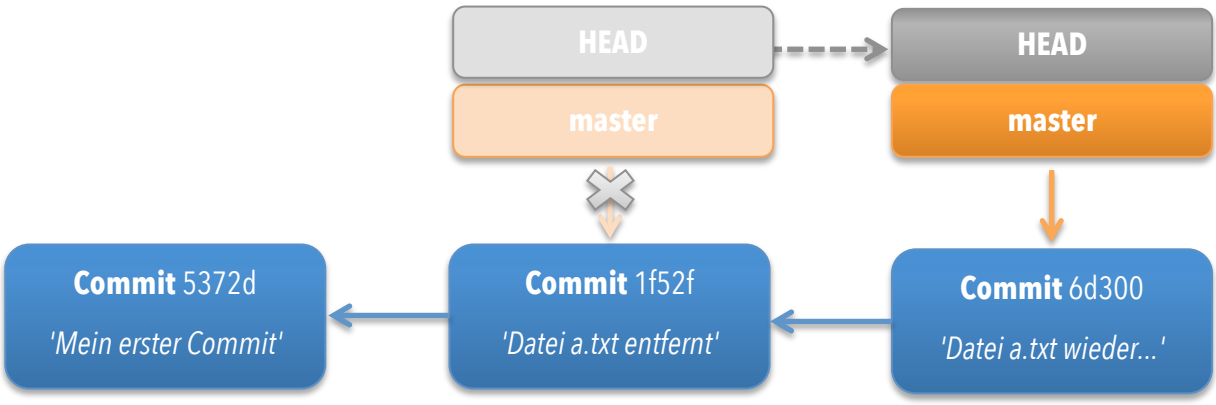
Befehl:

```
$ git log --pretty=format:"%h - %an: %s"
```

Output:

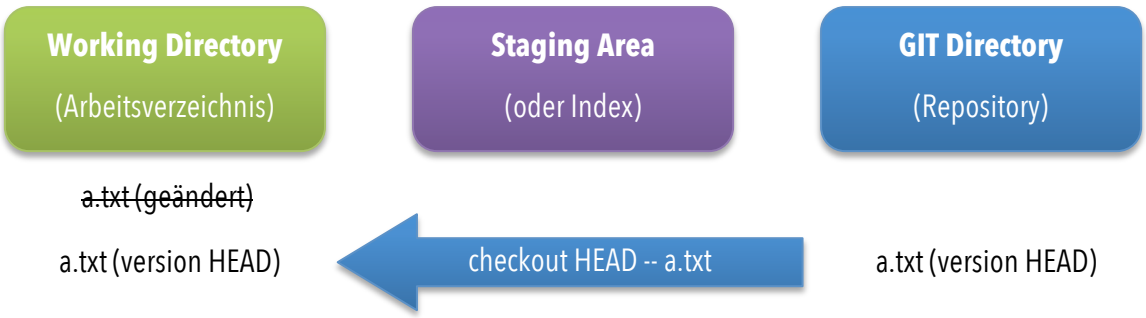
```
6d3008e - Andy Theiler: Datei a.txt wieder hinzugefuegt
1f52f19 - Andy Theiler: Datei a.txt entfernt
5372dce - Andy Theiler: Mein erster Commit
```

Der "HEAD" wurde auf den neusten Commit verschoben:



Lokale Änderungen rückgängig machen

Ich habe an der Datei "a.txt" diverse Änderungen gemacht, welche ich nun aber wieder verwerfen und wieder den Zustand aus dem letzten Commit (d.h. vom HEAD) herstellen will.



Befehl:

```
$ git checkout HEAD -- a.txt
$ git status
```

Output:

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

Tip

Die doppelten Bindestriche "**--**" werden verwendet, um den Steuerteil der Befehlszeile von einer Liste mit Operanden, wie etwa Dateinamen, zu trennen. Diese sind nicht immer nötig! Im obigen Beispiel hätten wir die doppelten Bindestriche auch weglassen können.

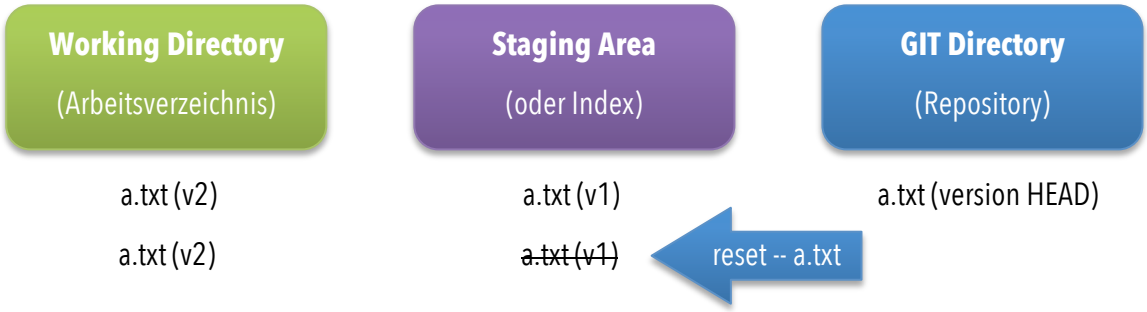
Falls im Repository sowohl eine Datei wie auch ein Tag mit dem Namen "a.txt" existiert würde, bekämen wir unterschiedliche Ergebnisse:

```
# Das Tag namens "a.txt" auschecken
$ git checkout a.txt

# Die Datei namens "a.txt" auschecken
$git checkout -- a.txt
```

Änderungen in der Stage Area rückgängig machen

Eine einzelne Datei

Ich habe in der Datei "a.txt" Änderungen gemacht und diese dem Stage Bereich hinzugefügt. Nun will ich aber die dem Stage hinzugefügte Datei wieder entfernen - also den **add** Befehl rückgängig machen.



Befehl:

```
$ git status
$ git reset -- a.txt
$ git status
```

Output:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   a.txt
#

$ git reset -- a.txt
Unstaged changes after reset:
M       a.txt

$ git status
# On branch master
# 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:   a.txt
#
```

Die nach dem **add** Befehl durchgeführten Fileänderungen bleiben nach dem **reset** im Arbeitsverzeichnis erhalten.

Alle Dateien aus der Staging Area entfernen

Um nicht nur eine bestimmte, sondern alle Dateien aus der Stage Area zu entfernen, verwenden wir den **reset** Befehl ohne Parameter.

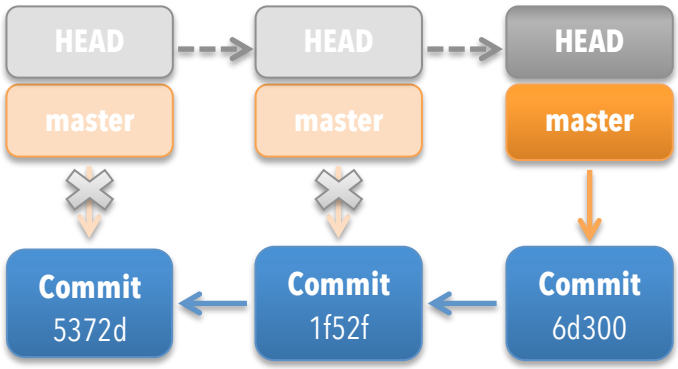
Befehl:

```
$ git reset
```

BRANCHING

Was ist ein Branch?

Ein Branch (dt. Zweig) in Git ist nichts anderes als einen simplen Zeiger auf einen Commit. Mit dem ersten Commit (initial commit) erstellt uns Git einen Standard-Branch namens "master". Mit jedem weiteren Commit bewegt sich der Branch automatisch vorwärts:

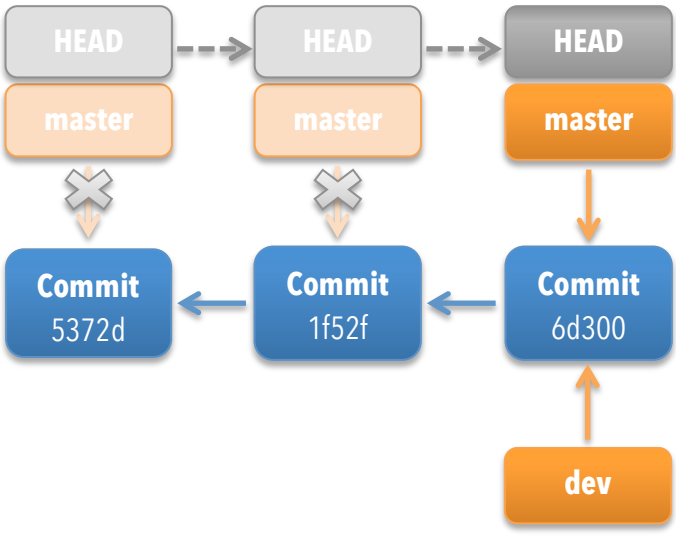


Wenn wir nun einen neuen Branch erstellen - wir nennen ihn "dev" - wird ein neuer Zeiger auf den gleichen Commit erstellt, auf dem wir aktuell arbeiten. An welchem aktuellen lokalen Branch wir gerade arbeiten, erkennt Git am **HEAD** Zeiger.

Einen neuen Branch erstellen wir mit dem **branch** Befehl:

```
$ git branch dev
```

Wie in der Abbildung ersichtlich, hat Git einen neuen Branch erzeugt, jedoch nicht zu diesem gewechselt:



Branch wechseln

Mit dem **checkout** Befehl können wir zu einem anderen Branch wechseln.

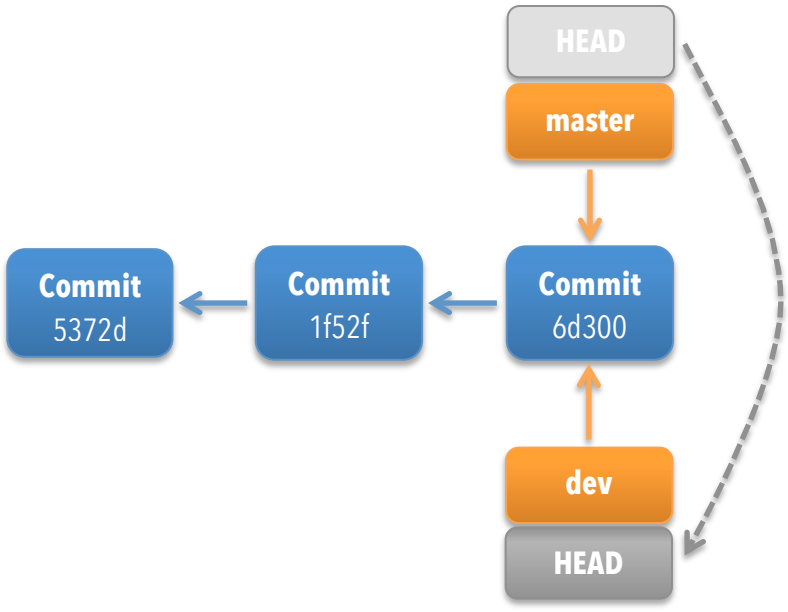
Befehl:

```
$ git checkout dev
$ git status
```

Output:

```
M       a.txt
Switched to branch 'dev'

$ git status
# On branch dev
# 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:   a.txt
#
```

Der status Befehl bestätigt uns den Branchwechsel. Weil der "dev" Branch auf den gleichen Commit zeigt wie der "master" Branch, ist die lokale Datei "a.txt" noch immer da und unverändert als "modified" markiert:



TIP: Die Brancherstellung (**branch dev**) und den Wechsel in den neuen Branch (**checkout dev**) kann man auch in einem Befehl durchführen:

```
$ git checkout -b dev
```

Um das Branching-Konzept von Git besser zu verstehen, erstellen wir eine neue Datei "b.txt" und commiten diese ins Repository.

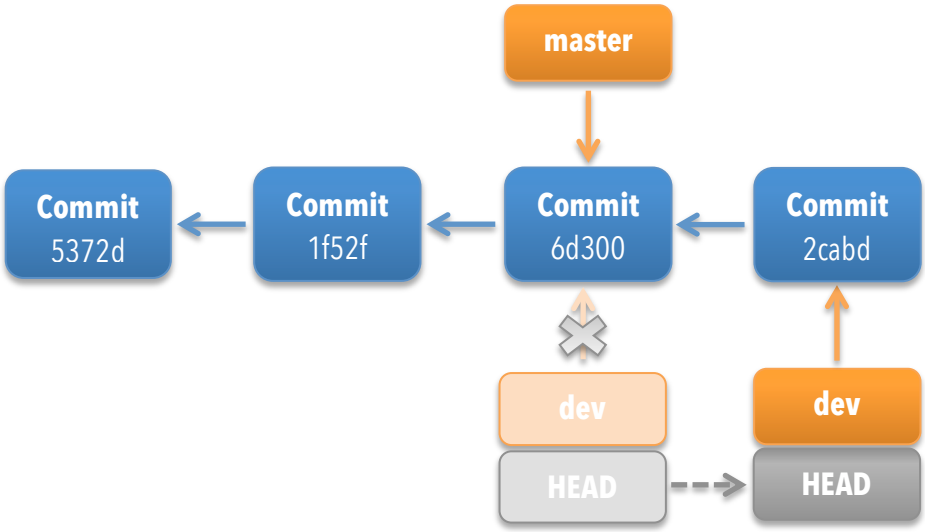
Befehl:

```
$ git add b.txt
$ git commit -m "neue Datei b.txt"
```

Output:

```
[dev 2cabdf7] neue Datei b.txt
 1 file changed, 3 insertions(+)
 create mode 100644 b.txt
```

In der Abbildung sehen wir, dass durch den Commit der HEAD-Zeiger für den "dev" Branch sich vorwärts vorschoben hat. Der "master" Branch hingegen zeigt noch immer auf seinen letzten Commit:

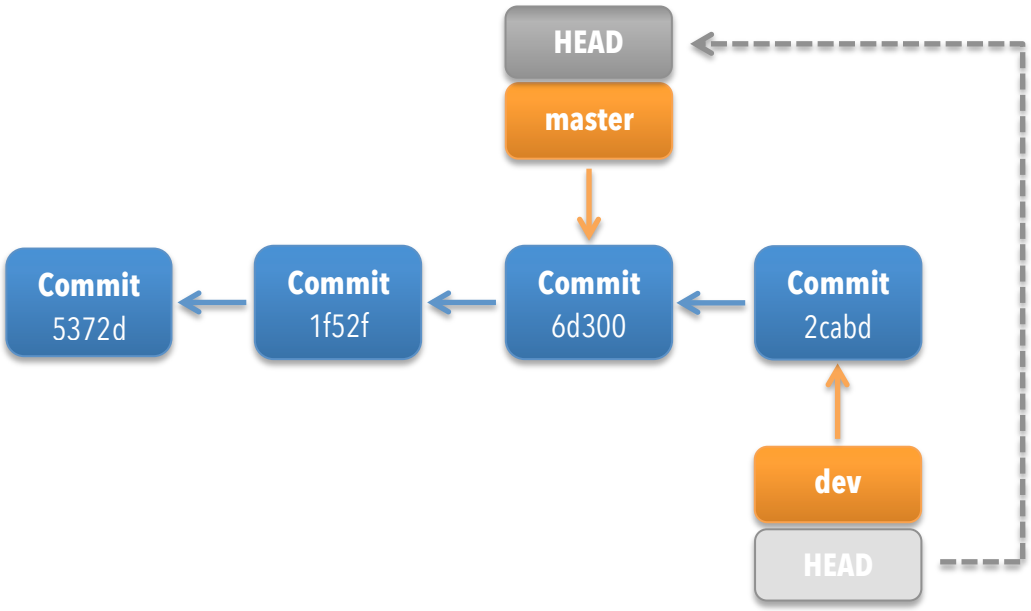


Nun wechseln wir wieder zurück auf den "master" Branch.

Befehl:

```
$ git checkout master
```

Die Abbildung zeigt das Ergebnis an:

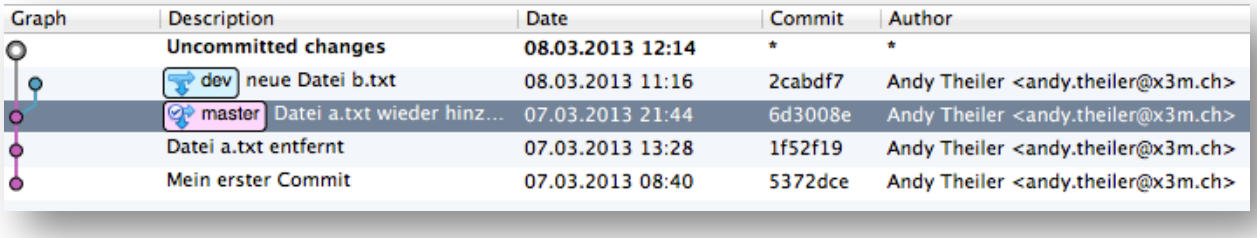


Der HEAD-Zeiger hat sich zurück zum "master" Branch verschoben.

ACHTUNG: Alle Daten im Arbeitsverzeichnis haben wieder den Stand des letzten Commits, also der Stand wie damals "vor" dem Wechsel in den "dev" Branch. D.h. die Datei "b.txt" aus dem "dev " Branch ist aus dem Arbeitsverzeichnis verschwunden. **Damit lokale Änderungen nicht verloren gehen, empfehle ich dringendst, diese vor dem Branchwechsel zu committen!**

Völlig unabhängig und ungestört können wir nun auf dem „master" Branch unsere Entwicklung weiterführen, mit der Gewissheit, dass wir jederzeit in den „dev" Branch wechseln können. Cool, oder?

Mein Git-Client "SourceTree" zeigt die neue Situation so an:



Nun will ich noch demonstrieren, was passiert, wenn wir die Datei „a.txt" ändern und im aktuellen "master" Branch committen.

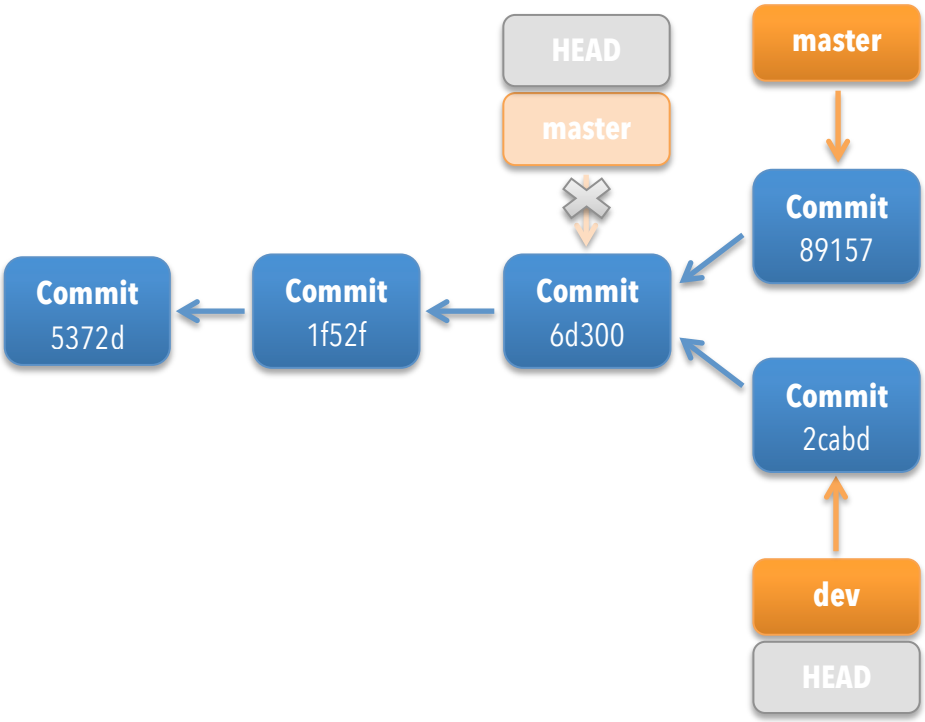
Befehl:

```
$ git add a.txt
$ git commit –m "Datei a.txt schon wieder mutiert"
```

Output:

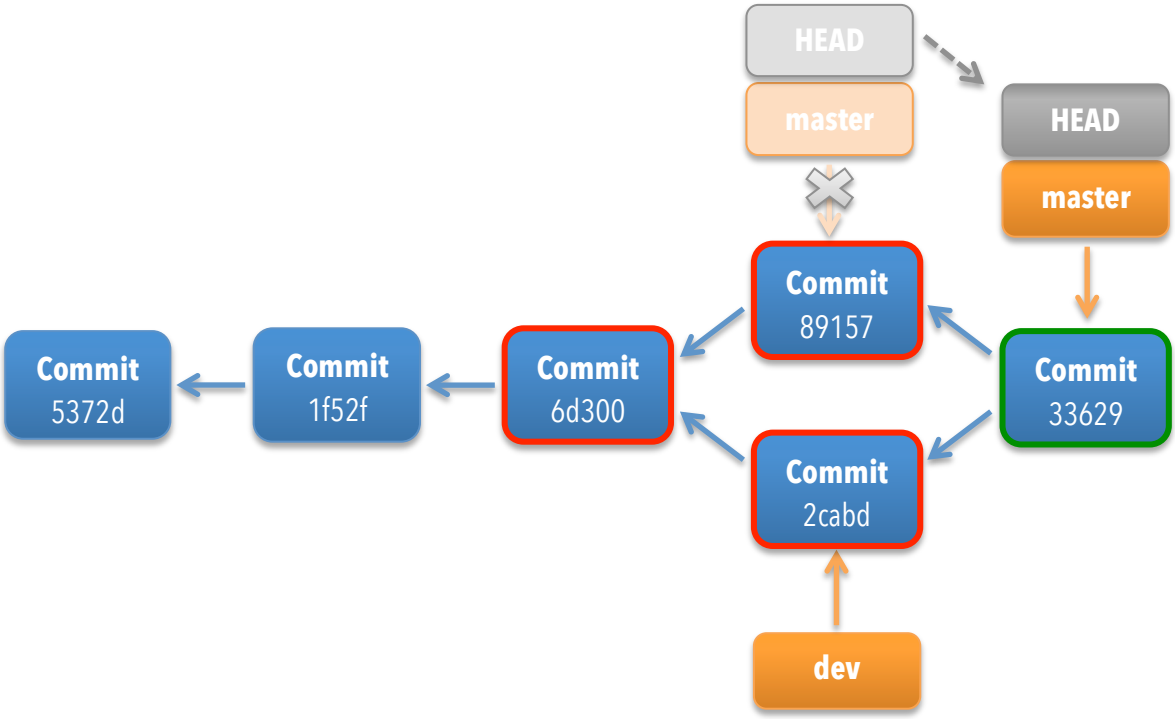
```
[master 8915790] Datei a.txt schon wieder mutiert
 1 file changed, 4 insertions(+), 1 deletion(–)
```

Wie in der Abbildung ersichtlich, existieren jetzt zwei völlig voneinander unabhängige Branches. Das Zusammenführen von Branches nennt sich "Merging" und wird in einem eigenen Kapitel behandelt.



MERGING

Beim mergen werden 2 Branches wieder zu einem Branch vereint. Git verwendet für den Merge die aktuellen Commits der Branches (#89157 und #2cabd) sowie der allgemeine Nachfolger (#6d300) der beiden (rot markiert) und erstellt einen neuen Commit (grün markiert):



Mergen

In unserem Beispiel haben wir unsere Weiterentwicklung im "dev" Branch abgeschlossen und wollen diese nun in den "master" Branch integrieren. Dazu wechseln wir (falls wir uns nicht schon dort befinden) in den "master" Branch und geben führen den **merge** Befehl aus:

Befehl:

```
$ git checkout master
$ git merge dev
```

Output:

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

Mergekonflikte

Weil die Datei "b.txt" sowohl im "master" wie auch im "dev" Branch verändert wurde, meldet uns Git einen Mergekonflikt. Git hat in diesem Fall **keinen** Merge-Commit erstellt sondern den Merge-Prozess gestoppt, damit wir den Konflikt beseitigen können.

Dazu editieren wir die Datei "b.txt" und stellen fest, dass Git der Datei sogenannte Konfliktmarker (<<<<, ==== und >>>>) hinzugefügt hat. Diese helfen uns bei der Konflikbeseitigung.

```
<<<<<<< HEAD
master änderungen  hier
master änderungen da
=======
dev weiterentwicklung zeile 1
dev weiterentwicklung zeile 2
>>>>>>> dev
```

Wir beseitigen die Marker und ändern die Datei, so wie wir sie gerne hätten. Nach dem stagen führen wir den Commit aus, um dem Merge-Prozess doch noch erfolgreich zu beenden.

Befehl:

```
$ git add b.txt
$ git commit –m 'dev nach master mergen'
```

Output:

```
[master 33629ec] dev nach master mergen
```

Visuelle Diff/Merge-Tools

Generell empfehle ich den Einsatz von visuellen Diff/Merge-Tools . Das Mac-Tool "Kaleidoscope" ist eines der besten seines Faches, jedoch auch sehr teuer! Das von Apple mitgelieferte "FileMerge" Tool genügt in den meisten Fällen auch. Ich selber verwende "DiffMerge" (http://www.sourcegear.com/diffmerge/)

Konfiguration DiffMerge

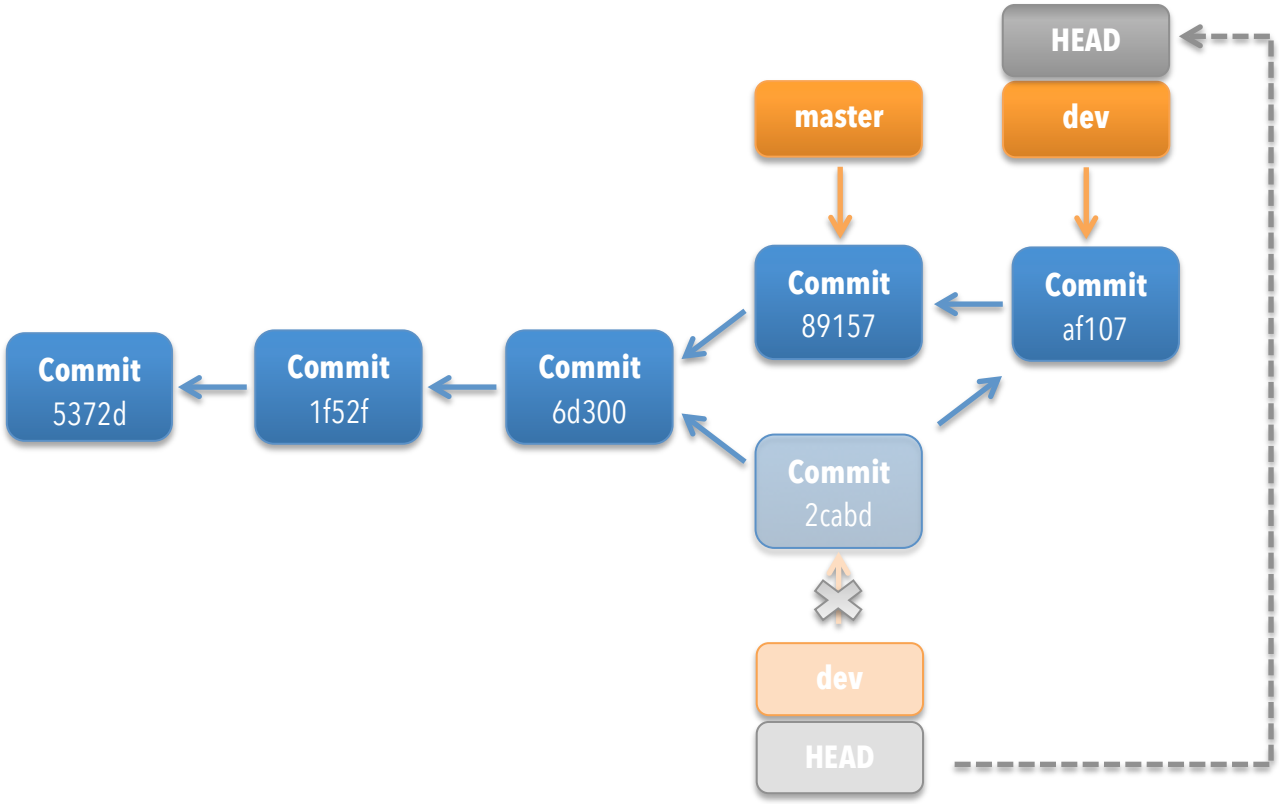
Um DiffMerge in Git als das default Diff- und Merge-Tool zu verwenden, müssen folgend Befehle ausgeführt werden:

```
$ git config --global diff.tool diffmerge
$ git config --global difftool.diffmerge.cmd 'diffmerge "$LOCAL" "$REMOTE"'
$ git config --global merge.tool diffmerge
$ git config --global mergetool.diffmerge.cmd 'diffmerge --merge --result="$MERGED"
"$LOCAL" "$(if test -f "$BASE"; then echo "$BASE"; else echo "$LOCAL"; fi)" "$REMOTE"'
$ git config --global mergetool.diffmerge.trustExitCode true
```

Rebasing

Der **merge** Befehl ist nicht die einzige Möglichkeit, Branches zusammen zu führen. Der **rebase** Befehl wendet alle Commits des aktuellen Branches auf dem andere Branch erneut an.

Hätten wir bei unserem vorherigen Beispiel anstelle von merge, den rebase Befehl benutzt, sähe die Commit-Historie folgendermassen aus:



Was ist passiert?

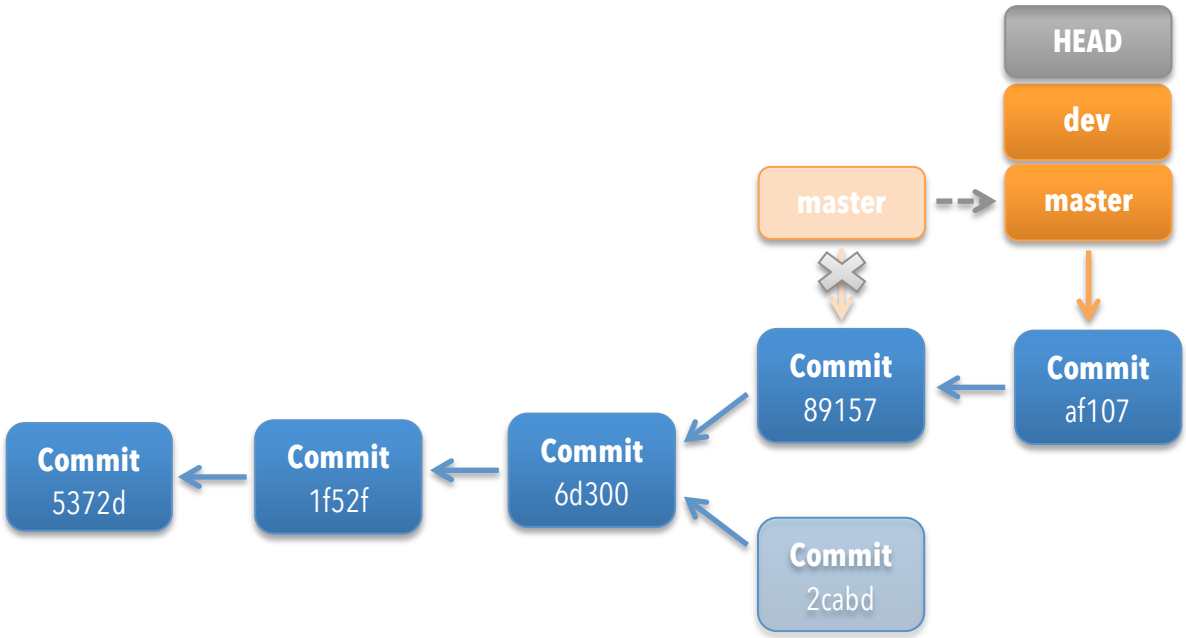
Der rebase Befehl hat den Commit (#2cabd) vom "dev" Branch auf dem "master" Branch nachgeführt (#af107) und den HEAD-Zeiger dorthin verschoben.

Fast Forward Merge

Jetzt wechseln wir wieder auf den "master" Branch und führen einen merge durch.

Weil der aktuelle Commit (#89157) ein Vorfahren des "dev" Commit ist, nennt man diese Art von merge ein"**Fast-Forward-Merge**". d.h. die Referenz des "master" Branches wird einfach auf den "dev" Branch verschoben.

```
$ git checkout master
$ git merge dev
```



ANHANG

Anleitungen

Hier eine kleine Auswahl an weiterführender Literatur:

Für Einsteiger

- **TIP:** Git - Der einfache Einstieg: http://rogerdudler.github.com/git-guide/index.de.html
- **TIP:** GT IMMERSION: http://gitimmersion.com/
- Git from the bottom up: http://ftp.newartisans.com/pub/git.from.bottom.up.pdf

Für Alle

- Offizielle Git Dokumentation: http://git-scm.com/doc
- **TIP:** Das offizielle Pro Git Buch: http://git-scm.com/book
- A Visual Git Reference: http://marklodato.github.com/visual-git-guide/index-en.html

Cheat Sheets

- **TIP:** Interaktives Cheat Sheet: http://ndpsoftware.com/git-cheatsheet.html
- http://rogerdudler.github.com/git-guide/files/git_cheat_sheet.pdf
- http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf

Grafische Git-Clients

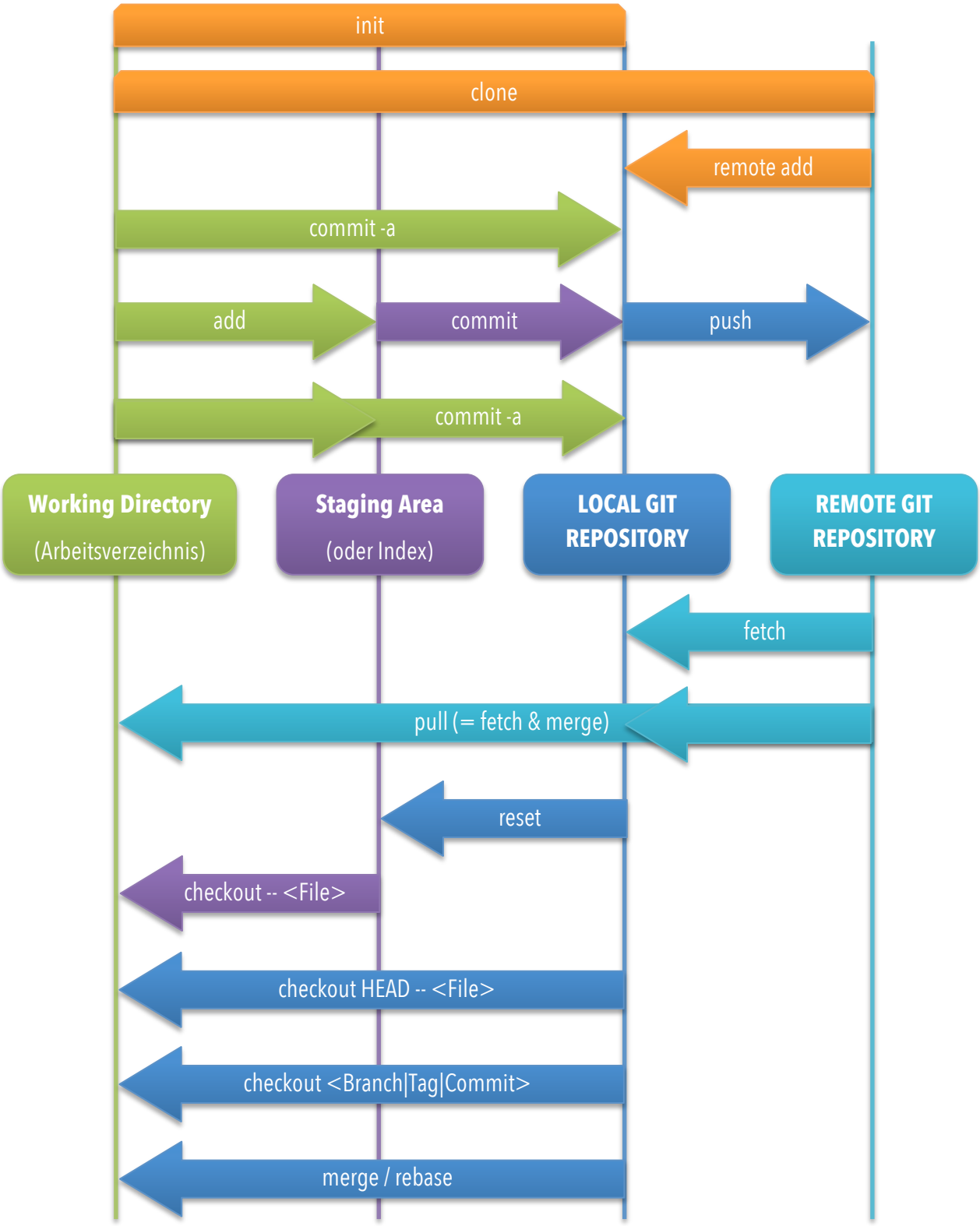
Für MAC

- **MEIN TIP:** SourceTree von Atlassian: http://www.atlassian.com/software/sourcetree/overview (kostenlos)
- GitX: http://gitx.laullon.com/ (kostenlos, Opensource)
- Tower2: http://www.git-tower.com/
- GitHub Mac: http://mac.github.com/ (kostenlos)
- Gitbox: http://www.gitboxapp.com/

Für WINDOWS

- **MEIN TIP:** SmartGit: http://www.syntevo.com/smartgithg/index.html (für nicht-kommerzielle Projekte kostenlos)
- tortoisegit: https://code.google.com/p/tortoisegit/
- GitHub Windows: http://windows.github.com/ (kostenlos)
- gitextensions: https://code.google.com/p/gitextensions/ (kostenlos, Opensource)
- GitEye: http://www.collab.net/downloads/giteye

VISUELLE REFERENZ



CHEAT SHEET

Repository erstellen

```
git init
  Repository erstellen

git clone <repository> <directory>
  Repository in ein Verzeichnis kopieren
```

Dateien in der Stage Area

```
git add <files>
git add .
  Alle lokalen Dateien hinzufügen

git mv <source> <destinatio>
  Datei/Verzeichnis verschieben/umbenennen

git rm <files>
git rm --cached <files>
  Datei "nur" aus der Staging Area
  entfernen (Datei lokal behalten)
```

Informationen

```
git status
git log
git branch
git blame <files>
git tag -l
git diff <ref1> <ref2>
```

Änderungen rückgänig machen

```
git checkout <ref> -- <files>
  Datei(en) aus Repo ins Arbeitsverzeichnis
  kopieren (überschreiben)

git reset -- <files>
  add Befehl rückgängig machen

git reset
  alle Änderungen in der Staging Area
  rückgängig machen

git commit -a --amend
  letzter Commit ersetzen
```

Commits

```
git commit [-a] -m 'Kommentar'
  -a = veränderte (und getrackte) Dateien
  autom. committen (d.h. ohne add Befehl)

git push [remote]
  ins Remote Repo committen
```

Projekt aktualisieren

```
git fetch
git fetch <ref>
git pull
  = fetch & merge
```

Branches

```
git checkout <branchname>
  Branch wechseln

git merge <branchname>
git rebase <branchname>
  in den aktuellen Branch mergen/rebasen

git checkout -b <branchname>
  Branch erstellen und dorthin wechseln

git branch <branchname>
  Branch erstellen

git branch -D <branchname>
  Branch löschen

git branch
  Liste aller Branches
```

Tag's

```
git tag <tagname>
  aktuelle Version taggen

git tag -l
  Liste aller Tag's

git tag -d <tagname>
  Tag löschen
```

Nützliche Befehle

```
git config color.ui true
  farbige Konsolenausgabe

git config format.pretty oneline
  Logausgabe 1 Zeile pro Commit

gitk
  visueller Git-Client

git log --pretty=format:"%h - %ai: %s"
  formatierte Logausgabe

git init --bare
git clone --bare <repository> <dir>
  ein "nacktes" (= ohne Arbeitsverzeichnis)
  Repository erstellen
```