



The Embedded Solution Company

GIT - Versionsverwaltung

Über GIT

- Entwickelt von Linus Torvalds für Linux
 - Lizenzänderung bei der zuvor genutzten kommerziellen Versionsverwaltungssoftware Bitkeeper
 - Neue Lizenz war nicht mehr „offen genug“ für die Verwaltung von Linux
- Gesprochen: „git“ mit echtem „g“ wie in „get“
 - nicht „dschid“ wie in „gin“
 - Git ist auch ein englischer Begriff für Blödmann, Idiot; Torvalds meinte, das passt ganz gut zu ihm

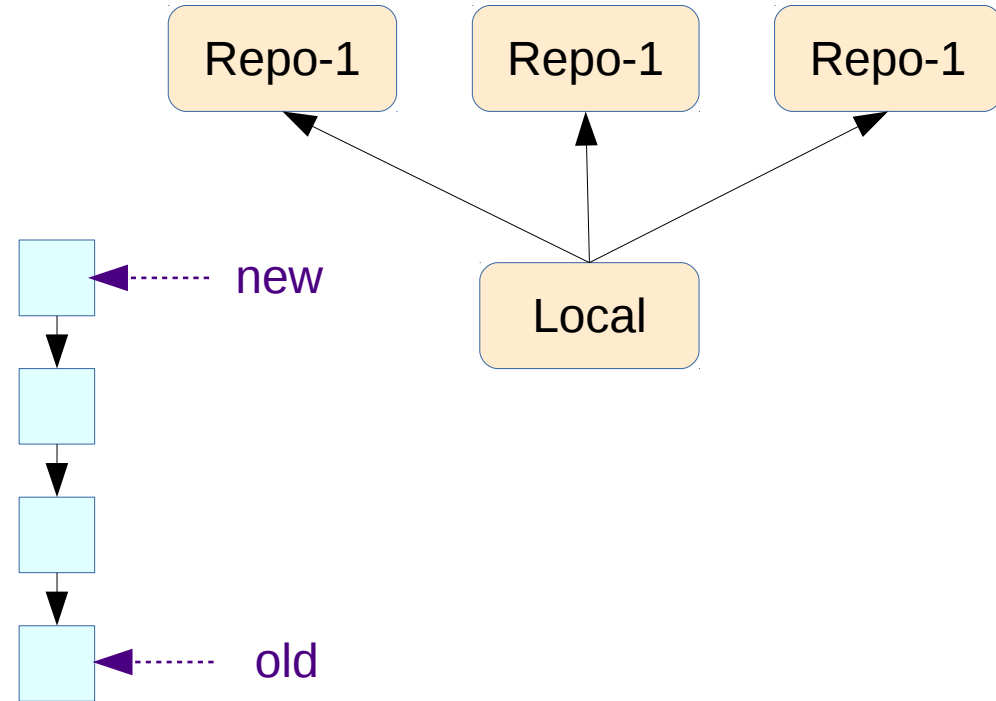


Ziel einer Versionsverwaltung

- Platzsparende Archivierung aller Versionen
 - Auch Varianten (z.B. weitergepflegte alte Versionen)
- Zugriff auf alle alten Versionen
- Unterschiede zwischen den Versionen anzeigen
- Leichtes Einpflegen neuer Versionen
- Auflösung von Konflikten bei paralleler Entwicklung
 - z.B. widersprüchliche Ergänzungen von zwei verschiedenen Entwicklern
- Nachvollziehbarkeit von Änderungen
 - Wer hat wann was geändert?
- Im Open-Source-Bereich: Veröffentlichung (z.B. github)
 - Kunden an der Entwicklung beteiligen

Grundsätzliches zu GIT

- Verteiltes Versionsverwaltungssystem
 - Lokale Kopie aller Repositories
 - Arbeit überwiegend lokal (dezentral)
- Change-Sets, nicht filebasiert
 - Falls zu einem Thema Änderungen in mehreren Files notwendig sind, werden diese dennoch nur als ein Commit gespeichert
- Sicht immer von neu nach alt
 - Oben sind immer die neuesten Versionen
- Jedes Objekt bekommt einen SHA1-Hash
 - Eindeutige 40-stellige Hex-Zahl
 - Dient zur Referenzierung der Objekte
 - Die Angabe der ersten Stellen reicht meist



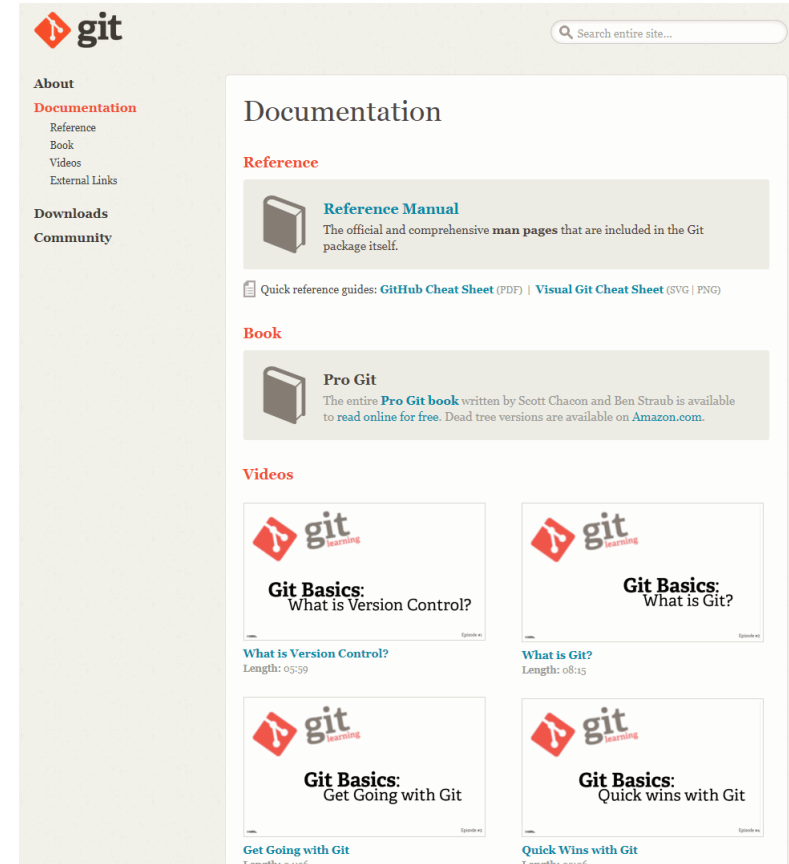
11ce98dc56dad793a8d95df43a8f6fda6c2af923
→ 11ce98

Allgemeine Befehlsstruktur

- `git [<global-options>] command [<options>] [<arguments>]`
 - Das Kommando `<command>` mit seinen Argumenten `<arguments>` wird ausgeführt
 - Mit `<options>` kann die Wirkungsweise des Befehls beeinflusst werden
 - `<global-options>` sind Optionen, die bei allen GIT-Befehlen zur Verfügung stehen; sie werden vor `<command>` angegeben; Beispiele:
 - Die Hilfe seitenweise durchblättern: `git --paginate help`
 - Das log nicht blättern, sondern gesamt am Stück ausgeben `git --no-pager log`
- Viele GIT-Befehle kennen die Option `--dry-run`
 - Es wird nur aufgelistet, was gemacht wird, aber nicht wirklich durchgeführt; Beispiel:
 - Liste auf, welche Files in den Commit übernommen werden: `git commit --dry-run`

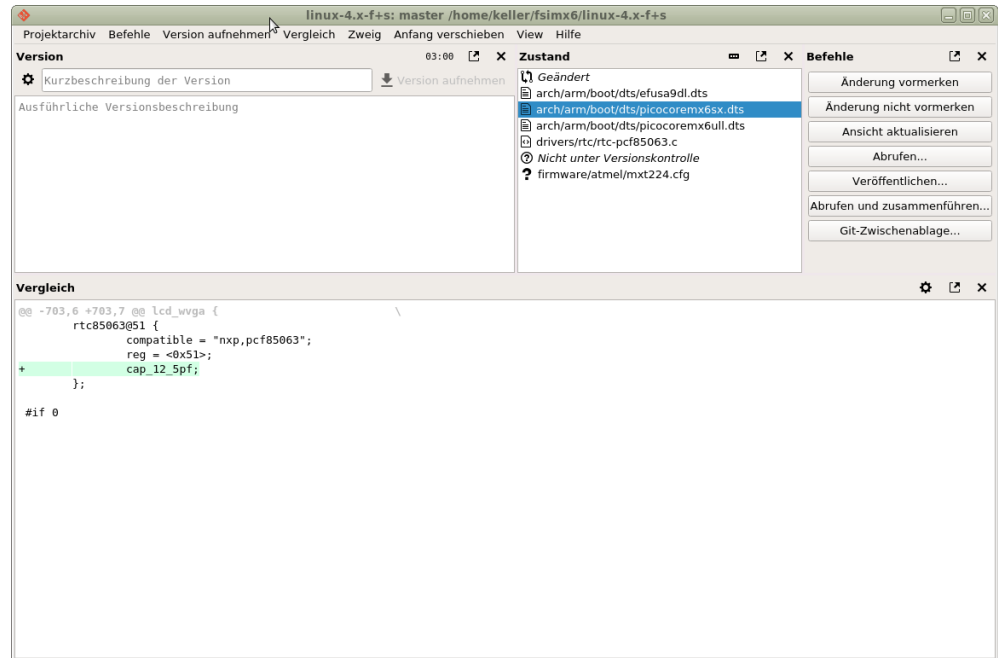
Dokumentation

- <http://git-scm.com/documentation>
 - Reference Manual
 - Git-Book
 - Videos
 - Man-Pages



Tools

- Command line: `git <command>`
 - Voller Funktionsumfang, überall verfügbar
 - Hilfe: `git help <command>`
- Versionsgraph: `gitk [--all]`
- Commits, Patches: `git cola`
- Teilweises Einchecken sehr einfach
- Nur Windows: `tortoise-git`
- Weitere grafische Tools
 - GitEye, git-gui, qgit, gitg

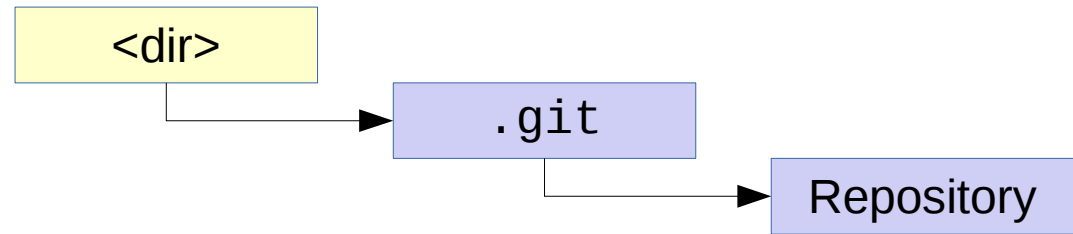


GIT-Umgebung einrichten

- Name, Email angeben
- Bevorzugten Editor angeben
 - emacs, kate, pluma, kwrite, vi/vim, gedit, joe, ...
- Bevorzugtes diff-Tool angeben
 - kdiff3, kompare, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, diffuse, opendiff, p4merge, araxis
- Globale Settings und projektspezifische Settings möglich
 - z.B. git cola, Menü Projektarchiv → Einstellungen
- Auflisten der globalen Settings
 - `git config --global -list`
- Einstellen neuer Werte
 - `git config --global user.name "Hartmut Keller"`
 - `git config --global user.email keller@fs-net.de`
 - `git config --global gui.editor emacs`
 - `git config --global diff.tool meld`
 - `git config --global merge.tool meld`
 - `git config --global core.pager "less -R"`

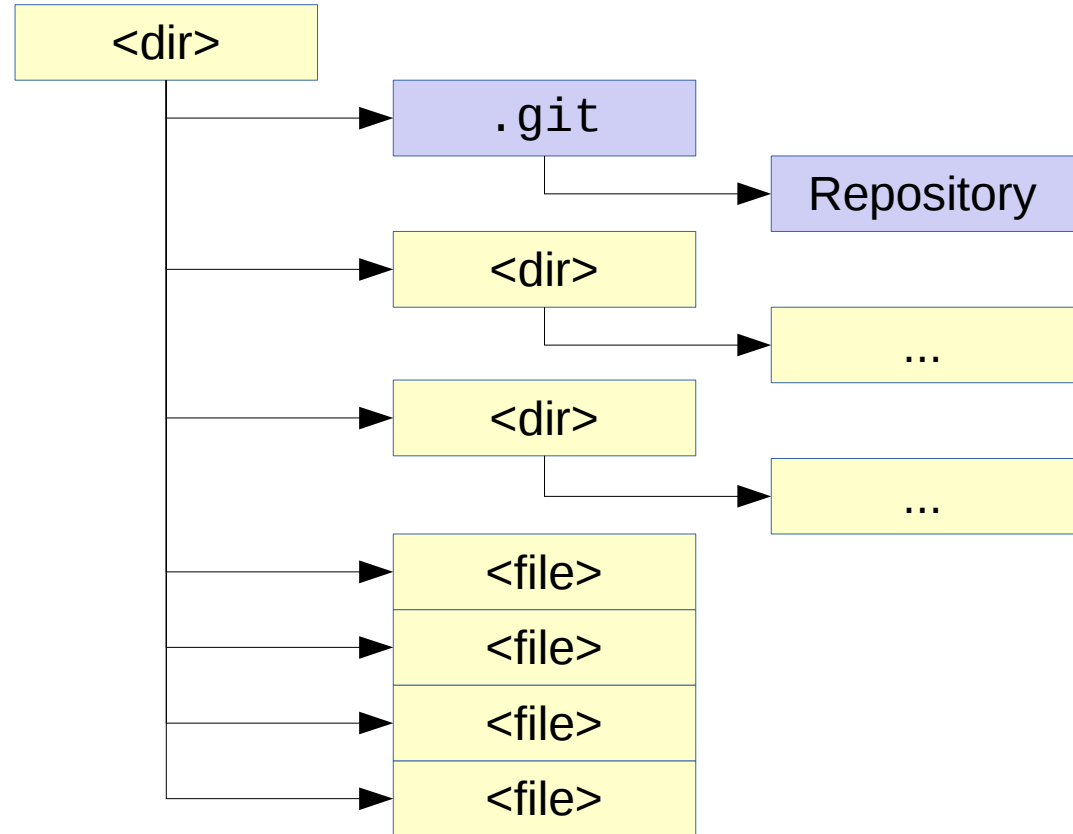
Neues Repository anlegen: git init

- Neues Verzeichnis mit leerem GIT anlegen
 - `git init <dir>`
- GIT in einem bestehenden Verzeichnis anlegen
 - `git init`
- Erzeugt
 - Unterverzeichnis `.git`
 - Enthält alles, inklusive Repository
 - Default-Branch `master`



Weitere Files hinzufügen

- Files hinzufügen
 - `git add <dir> ...`
 - `git add <file> ...`
- Commit
 - `git commit`



Beispiel: Neues GIT

- `mkdir demo`
- `cd demo`
- `git init`
- `emacs README.txt`
- `git add README.txt`
- `git commit`

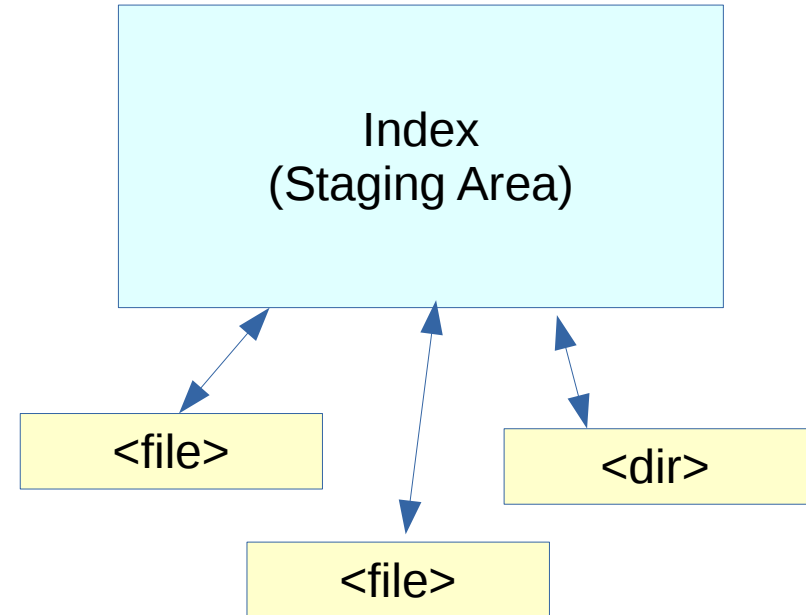
This is the root for the demo repository.

Root for demo repository

This is a simple root entry for the GIT demo repository.

Index (Staging Area)

- Im Index wird eine Änderung zusammengestellt
 - Neue Files, gelöschte Files, geänderte Files
 - Auch andere Zugriffsrechte sind Änderungen
 - Mehrere Dateien möglich (Change-Set)
 - Ergibt am Ende aber nur einen Commit
- Datei zum Index hinzufügen (beliebig oft)
 - `git add <path>`
- Datei aus dem Index entfernen
 - `git reset <path>`
- Am Ende Change-Set als Commit ins Repository übernehmen
 - `git commit`



Beispiel: Dateien hinzufügen (1)

- emacs main.c
- emacs inc.h
- emacs inc.c
- emacs Makefile

```
#ifndef __INC_H__
#define __INC_H__

int inc(int x);

#endif /* !__INC_H__ */
```

```
int inc(int x)
{
    return x + 1;
}
```

```
#include <stdio.h>
#include "inc.h"

int main(int argc, char *argv[])
{
    int a, b;

    a = 5;
    b = inc(a);
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```
CC = gcc
CFLAGS += -Wall

OBJS = main.o inc.o

demo: $(OBJS)
    $(CC) $(CFLAGS) $^ -o $@

$(OBJS): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@

clean:
    rm -f demo $(OBJS)
```

Beispiel: Dateien hinzufügen (2)

- `git status`

Auf Branch master
Unversionierte Dateien:
(benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

```
Makefile
inc.c
inc.h
main.c
```

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
(benutzen Sie "git add" zum Versionieren)

- `git add main.c inc.c inc.h Makefile`

- `git status`

Auf Branch master
zum Commit vorgemerkte Änderungen:
(benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

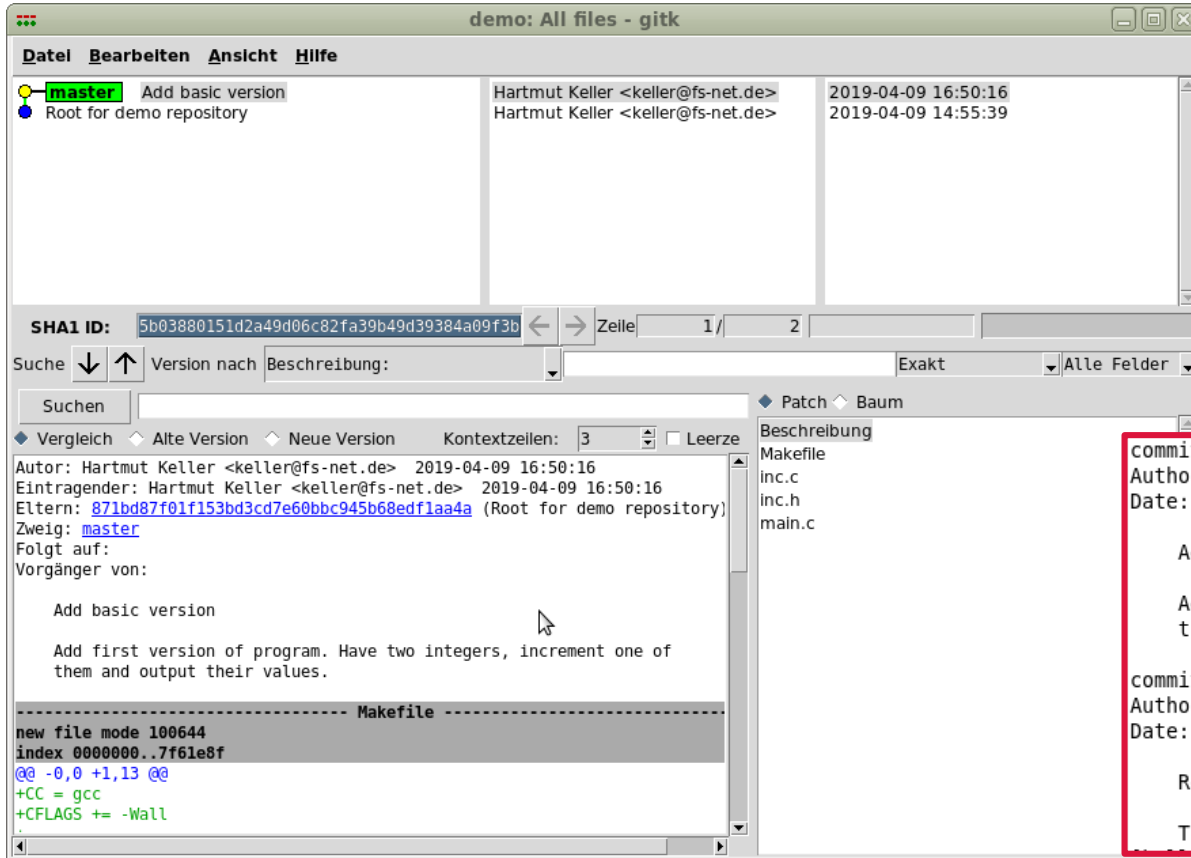
```
neue Datei:  Makefile
neue Datei:  inc.c
neue Datei:  inc.h
neue Datei:  main.c
```

- `git commit`

Add basic version

Add first version of program. Have two integers, increment one of them and output their values.

Beispiel: Anzeigen der Historie



• gitk

• git log

```
commit 5b03880151d2a49d06c82fa39b49d39384a09f3b
Author: Hartmut Keller <keller@fs-net.de>
Date: Tue Apr 9 16:50:16 2019 +0200

    Add basic version

    Add first version of program. Have two integers, increment one of
    them and output their values.

commit 871bd87f01f153bd3cd7e60bbc945b68edf1aa4a
Author: Hartmut Keller <keller@fs-net.de>
Date: Tue Apr 9 14:55:39 2019 +0200

    Root for demo repository

    This is a simple root entry for the GIT demo repository
```

Commit

- `git commit [-m <message>]`
 - Change-Set ins Repository aufnehmen
 - Wird die Commit-Message nicht direkt mit -m angegeben, wird der Editor zum Eingeben der Commit-Message geöffnet (sinnvoll bei längeren Messages)
- 1. Zeile: möglichst prägnant und aussagekräftig
 - Nicht: “Improve Code”, sondern “Add parameter x for driver y”
 - max. 50 Zeichen
 - Wird automatisch ins Changelog übernommen
 - Wird automatisch in E-Mail-Betreff übernommen
 - Wird automatisch in Patch-Dateinamen übernommen
 - Etc.
- Eine Leerzeile
- Beliebig viele Kommentarzeilen
 - max. 72 Zeichen
 - Damit < 80 Zeichen, auch wenn noch was vorangestellt wird, z.B. ‘>’ bei Zitat in einer Mail

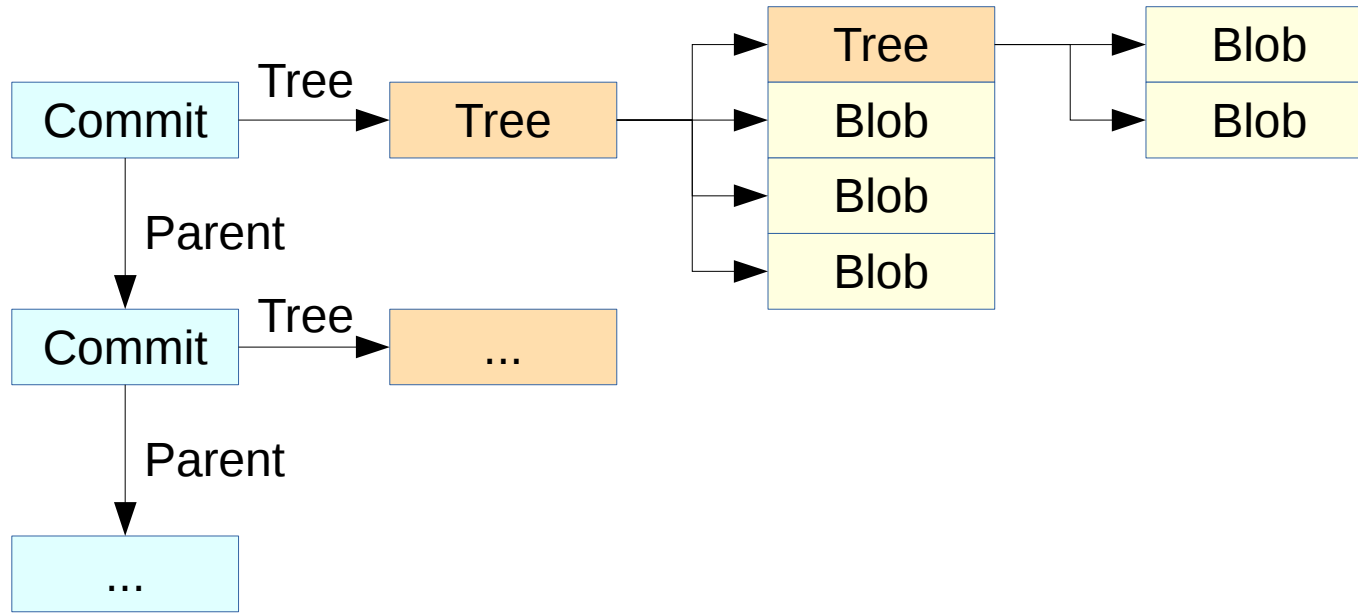
Aufbau der Commit-Message

- Commit-Message soll beschreiben, was dieser Commit erledigt
 - “Add support for second USB port”, nicht “Added support for second USB port”
 - Keine Bug-Beschreibung, also nicht “Second USB port does not work”
 - Höchstens “Fix for bug #1234 (Second USB port does not work)”
 - Noch besser: “Add support for second USB port (fixes bug #1234)”
- Möglichst den Bugtracker-Eintrag angeben, der dadurch gelöst wird
 - Umgekehrt beim Auflösen des Bugtracker-Eintrags im Bugtracker den GIT-Hash angeben und die Commit-Message übernehmen
- Commits möglichst themenbezogen
 - Keine Themen mischen, je ein Commit für ein Thema, z.B. einen Bug
 - Keine Trivial-Commits (“Fix typo”)
 - Entweder den Schreibfehler mal zu einem anderen Commit in diesem File hinzufügen
 - Oder ein Commit, der mehrere Schreibfehler in mehreren Files korrigiert
 - Komplexe Commits ggf. In mehrere einzelne Änderungen aufsplitten

Was passiert bei einem Commit?

- Jede Datei wird als Blob-Objekt abgelegt
 - Eindeutiger SHA1-Hash wird berechnet (40-stellige Hex-Zahl)
 - Datei wird komprimiert
 - Datei wird unter `.git/objects/12/34567890123456789012345678901234567890` abgelegt
- Für das Verzeichnis wird ein Tree-Objekt erzeugt
 - Enthält die Blob-Hashes für jede Datei im Verzeichnis
 - Enthält weitere Tree-Hashes für jedes Unterverzeichnis
 - SHA1-Hash wird berechnet, Tree wird komprimiert und als Objekt abgelegt
- Es wird ein Commit-Objekt erzeugt
 - Enthält die Commit-Message, Datum sowie Name und E-Mail des Committers
 - Enthält den Tree-Hash des zugehörigen Dateibaums
 - Enthält den Commit-Hash des Parents
 - SHA1-Hash wird berechnet, Commit wird komprimiert und als Objekt abgelegt

Dadurch entsteht Historie der Commits



- Nur neue/veränderte Dateien bekommen ein neues Blob-Objekt
- Nur veränderte Unterverzeichnisse bekommen ein neues Tree-Objekt

Beispiel: GIT-Objekte

- `find .git/objects -type f`

```
.git/objects/04/a6118a063b2379047288936d6ff5c46b76ef7b
.git/objects/87/1bd87f01f153bd3cd7e60bbc945b68edf1aa4a
.git/objects/5b/03880151d2a49d06c82fa39b49d39384a09f3b
.git/objects/16/70ae4150189cd3076cbc9f27f8d54ffa3e6acb
.git/objects/d4/a7780269842676029470c79742e2121632ef14
.git/objects/f3/631907a79d079ad1839b15dc979fd48e6031cc
.git/objects/54/b1cca7278b3612a98b040239b10b66e7afab01
.git/objects/2b/51a13344a01fc7969eefeeabb5171be8367877
.git/objects/7f/61e8f976ebbaa122370b0179b508e63c0fd57f
```

- `git cat-file -p 5b0388`

- `git cat-file -p f36319`

```
tree f3631907a79d079ad1839b15dc979fd48e6031cc
parent 871bd87f01f153bd3cd7e60bbc945b68edf1aa4a
author Hartmut Keller <keller@fs-net.de> 1554821416 +0200
committer Hartmut Keller <keller@fs-net.de> 1554821416 +0200

Add basic version

Add first version of program. Have two integers, increment one of
them and output their values.
```

- `git cat-file -p 04a611`

```
int inc(int x)
{
    return x + 1;
}
```

```
100644 blob 7f61e8f976ebbaa122370b0179b508e63c0fd57f    Makefile
100644 blob 1670ae4150189cd3076cbc9f27f8d54ffa3e6acb    README.txt
100644 blob 04a6118a063b2379047288936d6ff5c46b76ef7b    inc.c
100644 blob 54b1cca7278b3612a98b040239b10b66e7afab01    inc.h
100644 blob d4a7780269842676029470c79742e2121632ef14    main.c
```

Beispiel: Erzeugte Dateien ignorieren

- make

```
gcc -c -Wall main.c -o main.o  
gcc -c -Wall inc.c -o inc.o  
gcc -Wall main.o inc.o -o demo
```

- git status

```
Auf Branch master  
Unversionierte Dateien:  
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)  
  
    demo  
    inc.o  
    main.o  
  
nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
```

- emacs .gitignore

```
demo  
*.o  
*~
```

- git status

```
Auf Branch master  
Unversionierte Dateien:  
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)  
  
    .gitignore  
  
nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
```

Dateien ignorieren

- In der Datei `.gitignore` können Dateinamen angegeben werden, die von GIT nicht beachtet werden sollen
 - Gilt auch für alle Unterverzeichnisse, es sei denn dort gibt es wieder ein eigenes `.gitignore`
 - Wildcards `*`, `?`, `[a..x]`, `{...,...}`, etc. werden wie auf der Shell aufgelöst
 - Pfadnamen können vorangestellt werden; ein Pfad der mit `/` beginnt, ist relativ zum Verzeichnis von `.gitignore` selbst
 - Zeilen, die mit `!` beginnen, invertieren die Logik, werden also explizit *nicht* ignoriert
 - Kommentarzeilen beginnen mit einem `#`
 - `.git` wird schon automatisch ignoriert
- Beispiele
 - Alle Files ignorieren, die mit Punkt beginnen: `.*`
 - Aber `.gitignore` selbst nicht ignorieren: `!.gitignore`
 - Nur Files auf der obersten Ebene ignorieren, die mit Punkt anfangen: `/*.*`
 - Files ignorieren, die mit einem Punkt anfangen und in irgendeinem Unterverzeichnis `dir` stehen: `dir/*.*`
 - Nur Files mit Punkt ignorieren, die im `dir`-Verzeichnis der obersten Ebene stehen: `/dir/*.*`
 - Das komplette Verzeichnis `dir` auf der obersten Ebene ignorieren: `/dir/`
- Hinzufügen einer eigentlich ignorierten Datei
 - `git add --force <path>`

Beispiel: Weitere Änderungen

- Bei `inc()` ein zweites Argument hinzufügen
 - emacs `inc.c`
 - emacs `inc.h`
- Bei `main()` optional Wert für `a` als Argument einlesen
 - emacs `main.c`

```
int inc(int x, int i)
{
    return x + i;
}
```

```
#ifndef __INC_H__
#define __INC_H__

int inc(int x, int i);

#endif /* !__INC_H__ */
```

```
#include <stdio.h>
#include <stdlib.h> /* strtol() */
#include "inc.h"

int main(int argc, char *argv[])
{
    int a, b;

    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;

    b = inc(a, 1);
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

Beispiel: Unterschiede anzeigen (1)

git status

```
diff --git a/main.c b/main.c
index d4a7780..7499d22 100644
--- a/main.c
+++ b/main.c
@@ -1,12 +1,14 @@
#include <stdio.h>
+#include <stdlib.h>          /* strtol() */
#include "inc.h"

int main(int argc, char *argv[])
{
    int a, b;

-   a = 5;
-   b = inc(a);
+   a = (argc > 1) ? strtol(argv[1], NULL, 0): 5;
+
+   b = inc(a, 1);
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```
Auf Branch master
Änderungen, die nicht zum Commit vorgemerkt sind:
(benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
(benutzen Sie "git checkout -- <Datei>...", um die Änderungen im Arbeitsverzeichnis zu verwerfen)

    geändert:    inc.c
    geändert:    inc.h
    geändert:    main.c

Unversionierte Dateien:
(benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

    .gitignore

keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git commit -a")
```

git diff main.c

Beispiel: Unterschiede anzeigen (2)

git difftool main.c

```
W0hF9Z_main.c — main.c
Meld Datei Bearbeiten Änderungen Ansicht
Speichern Rückgängig
W0hF9Z_main.c — main.c
W0hF9Z_main.c main.c

#include <stdio.h>
#include "inc.h"

int main(int argc, char *argv[])
{
    int a, b;

    a = 5;
    b = inc(a);
    printf("a=%d, b=%d\n", a, b);

    return 0;
}

#include <stdio.h>
#include <stdlib.h> /* strtol() */
#include "inc.h"

int main(int argc, char *argv[])
{
    int a, b;

    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;
    b = inc(a, 1);
    printf("a=%d, b=%d\n", a, b);

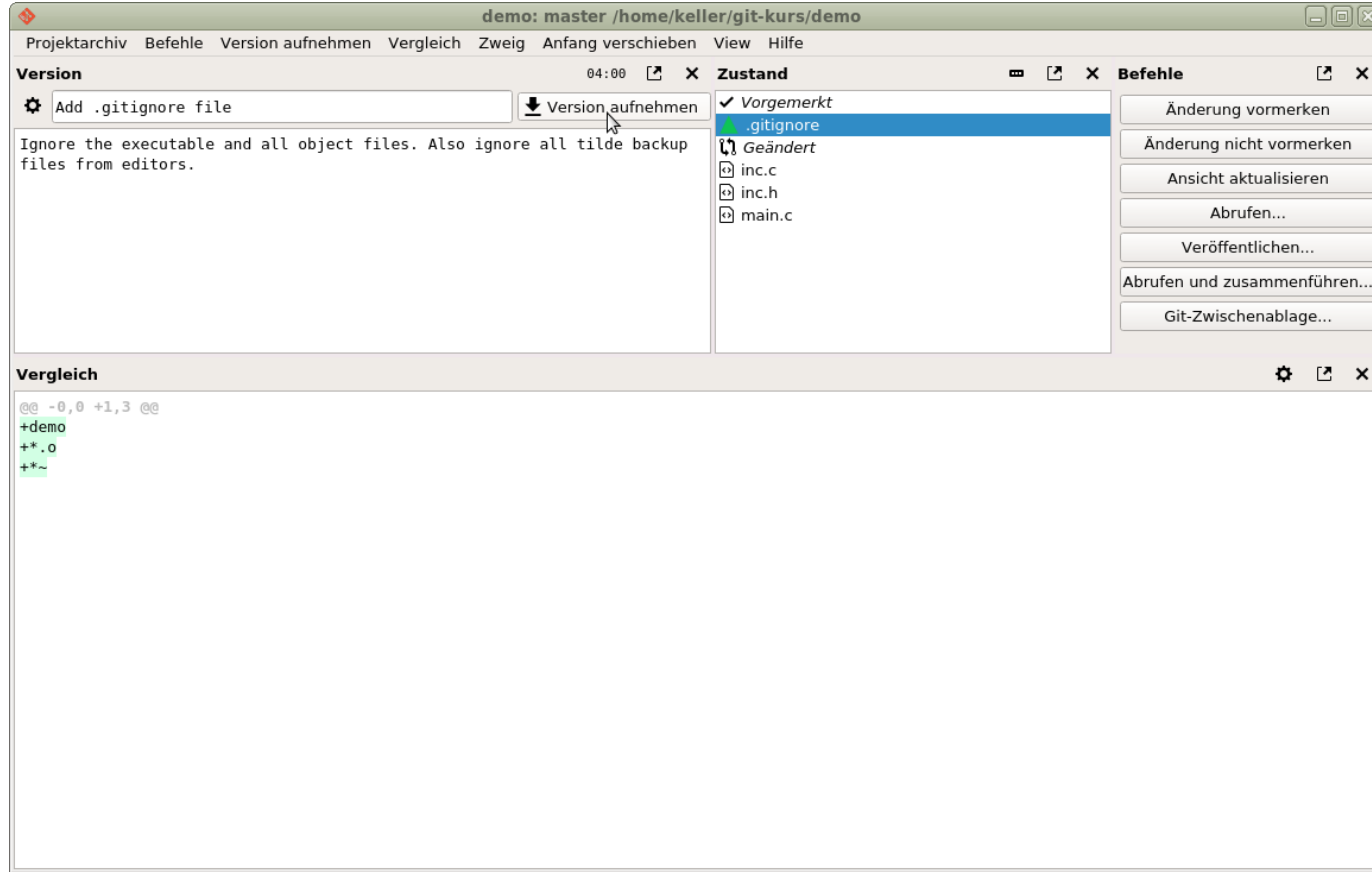
    return 0;
}

Zeile 2, Spalte 1 EINF
```

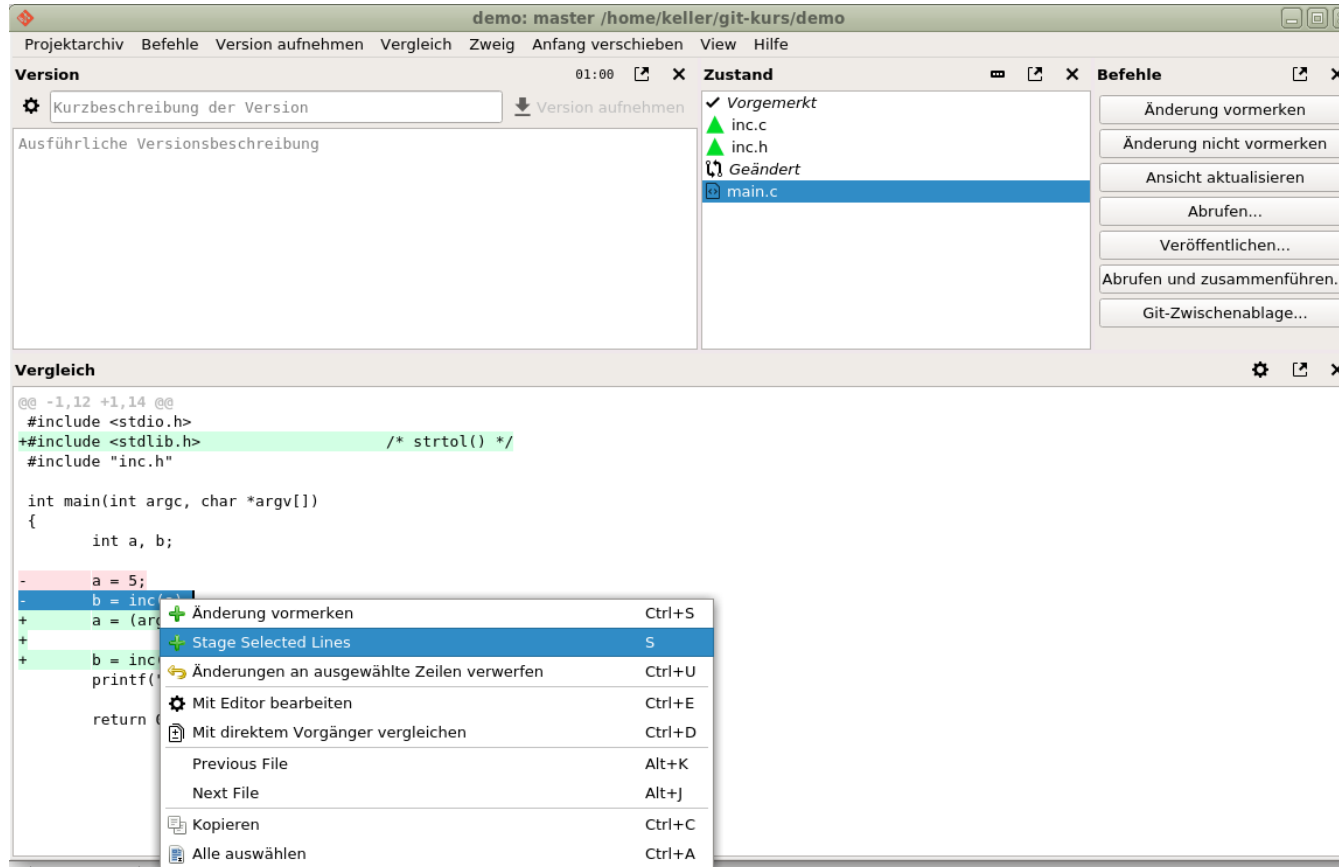
Beispiel: Themenbezogene Commits

- Wir haben jetzt drei Änderungen
 1. `.gitignore` hinzugefügt (\rightarrow `.gitignore`)
 2. Änderung von `inc()` mit zweitem Argument (\rightarrow `inc.h`, `inc.c`, `main.c`)
 3. Optionales Einlesen des Anfangswerts über Kommandozeile (\rightarrow `main.c`)
 - Alle Änderungen sind unabhängig voneinander
 - Jede Änderung sollte ein eigener Commit werden
 - Problem: in `main.c` sind sowohl Teile von 2. als auch von 3. enthalten
- \rightarrow Selektives Bereitstellen von Änderungen (Staging)
- Feature nur in `git cola` enthalten

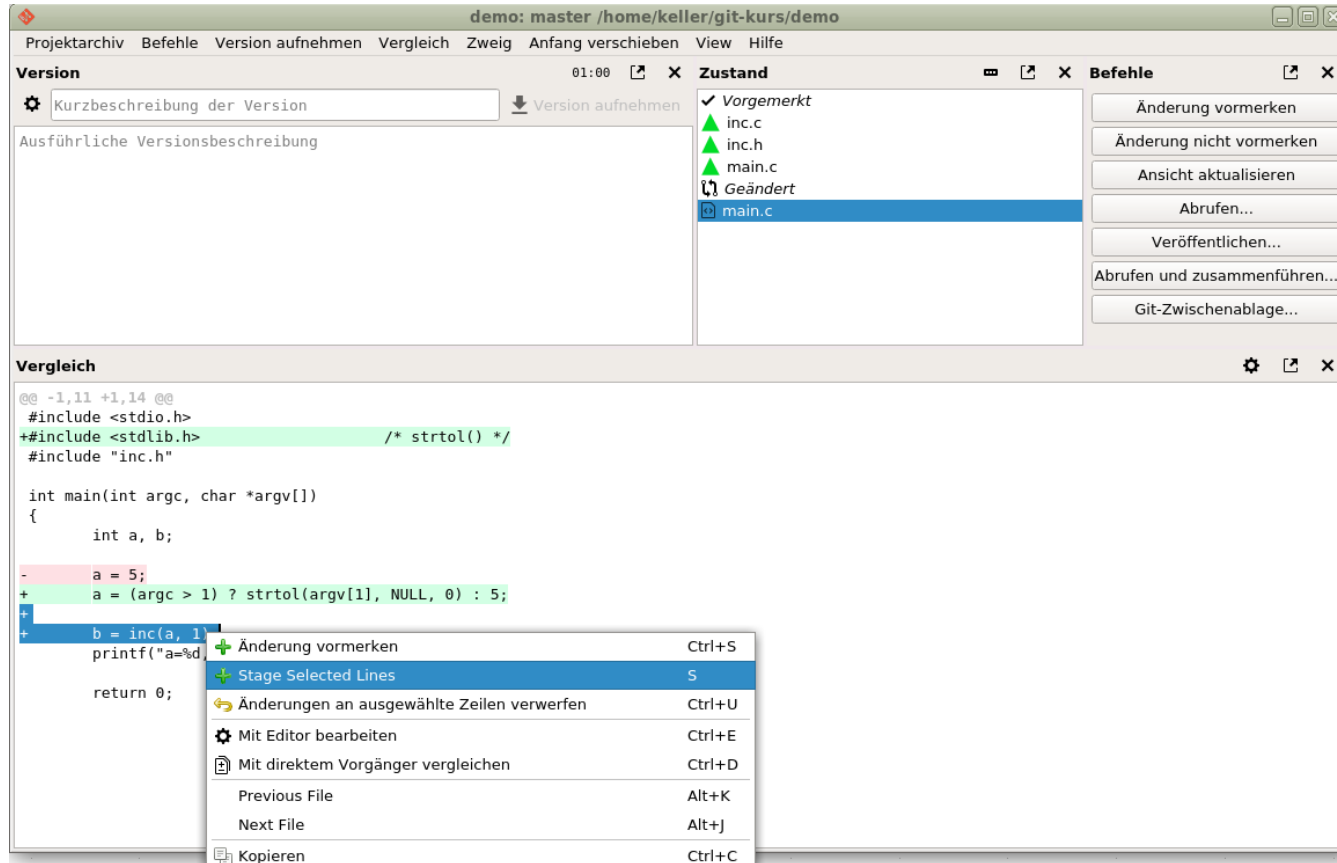
Beispiel: git cola (1)



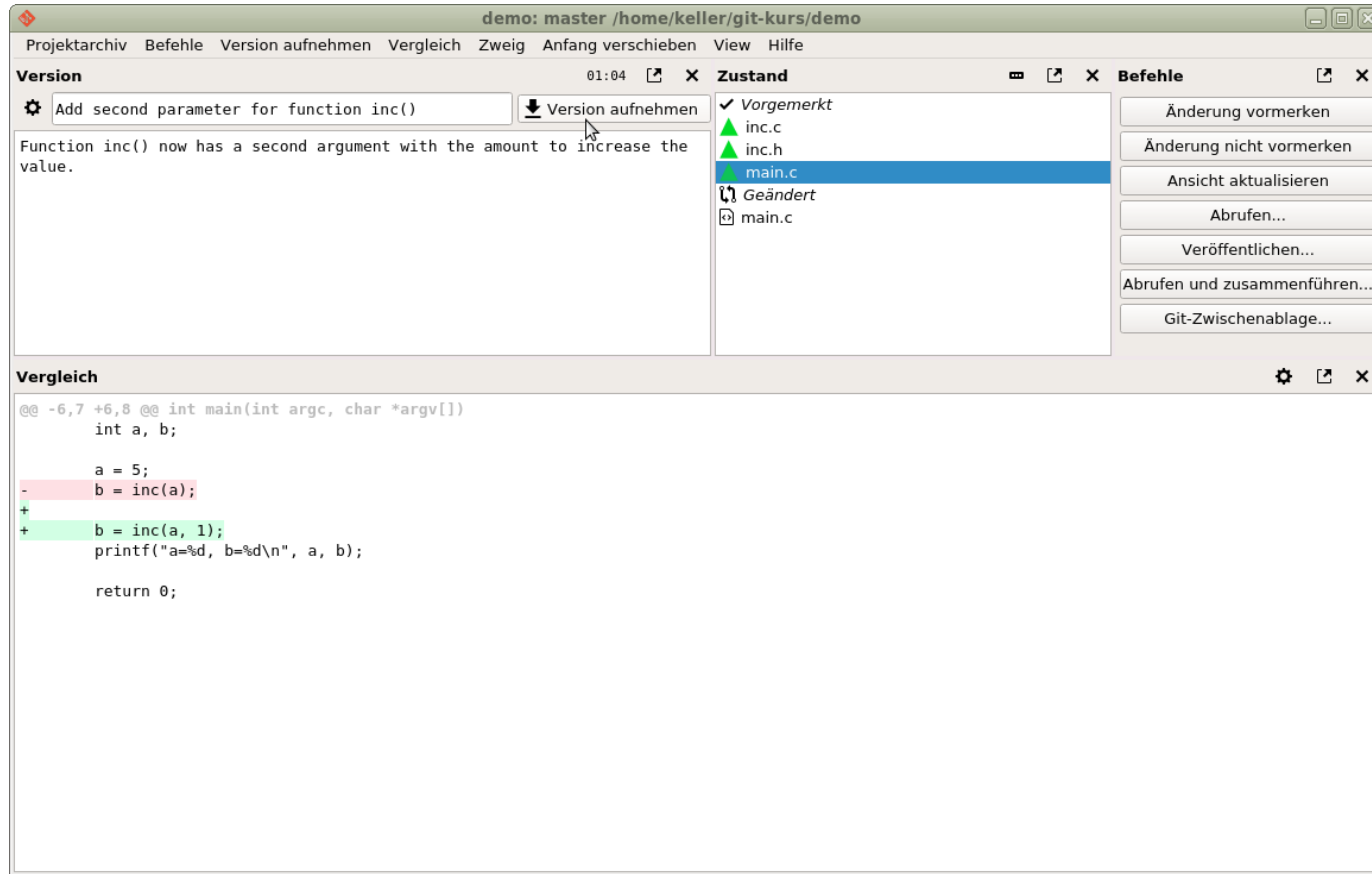
Beispiel: git cola (2a)



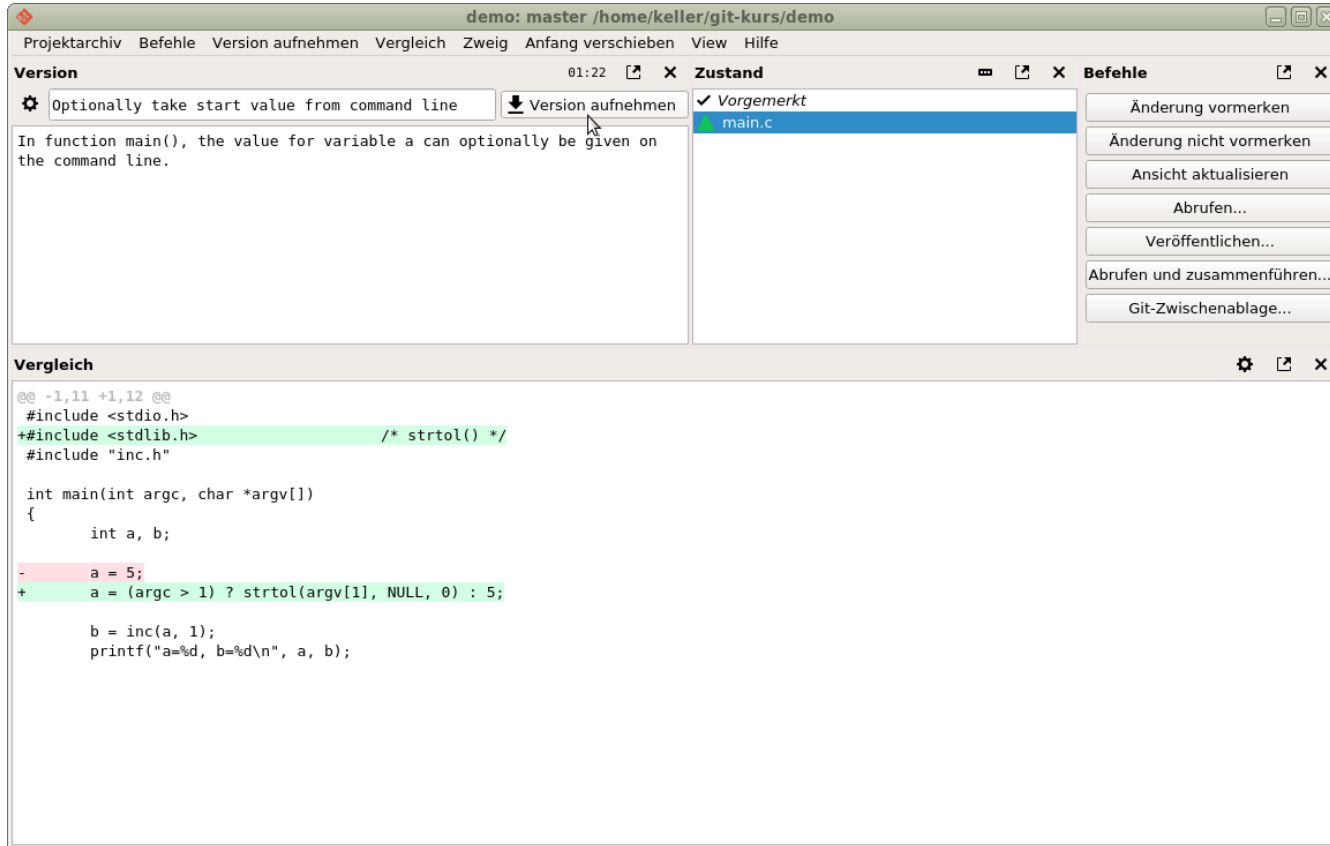
Beispiel: git cola (2b)



Beispiel: git cola (2c)




Beispiel: git cola (3)



Beispiel: File umbenennen (1)

- `main.c` soll nach `demo.c` umbenannt werden

- `mv main.c demo.c`
- `git add main.c`
- `git status`




```
Auf Branch master
zum Commit vorgemerkte Änderungen:
(benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

    gelöscht:      main.c

Unversionierte Dateien:
(benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

    demo.c
```

- `git add demo.c`
- `git status`



```
Auf Branch master
zum Commit vorgemerkte Änderungen:
(benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

    umbenannt:      main.c -> demo.c
```


Beispiel: File umbenennen (2)

- emacs Makefile

```
CC = gcc
CFLAGS += -Wall

OBJS = demo.o inc.o

demo: $(OBJS)
    $(CC) $(CFLAGS) $^ -o $@

$(OBJS): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@

clean:
    rm -f demo $(OBJS)
```

- git add Makefile

- git status

Auf Branch master
zum Commit vorgemerkte Änderungen:
(benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

geändert:	Makefile
umbenannt:	main.c -> demo.c

- git commit

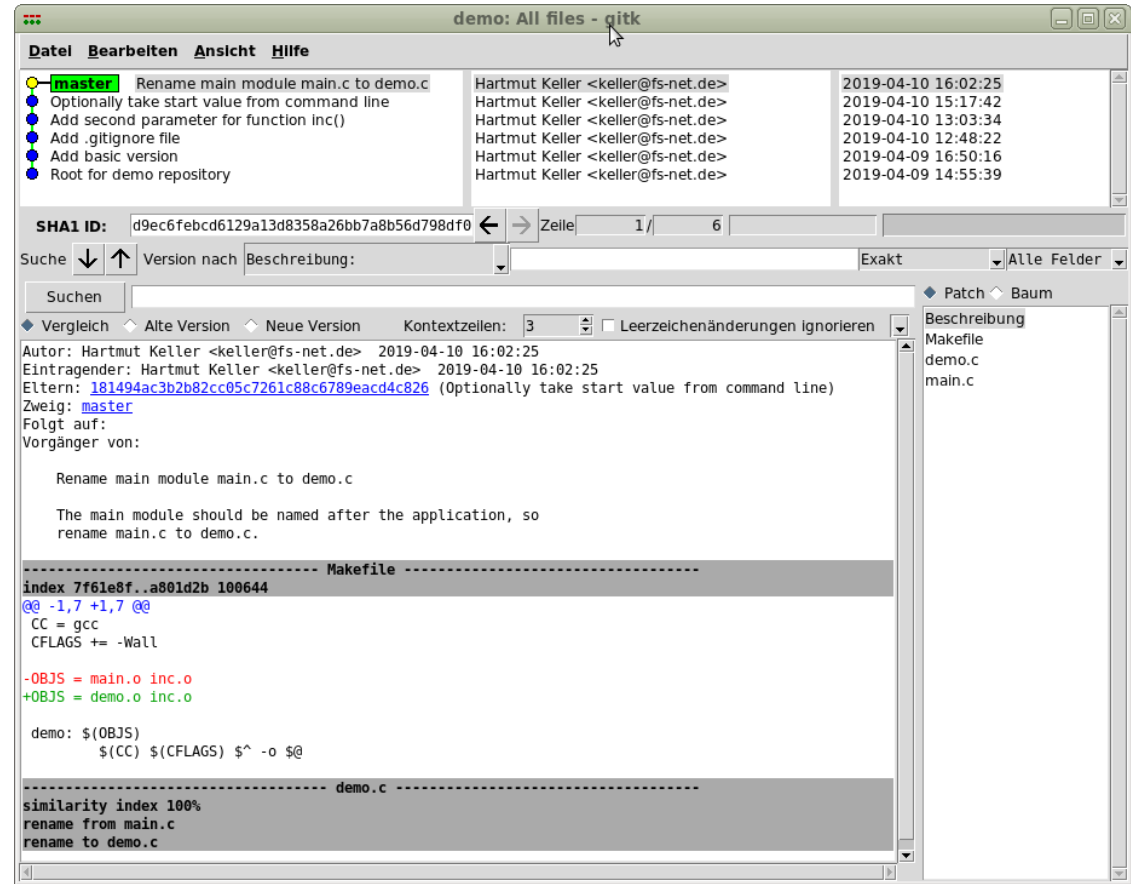
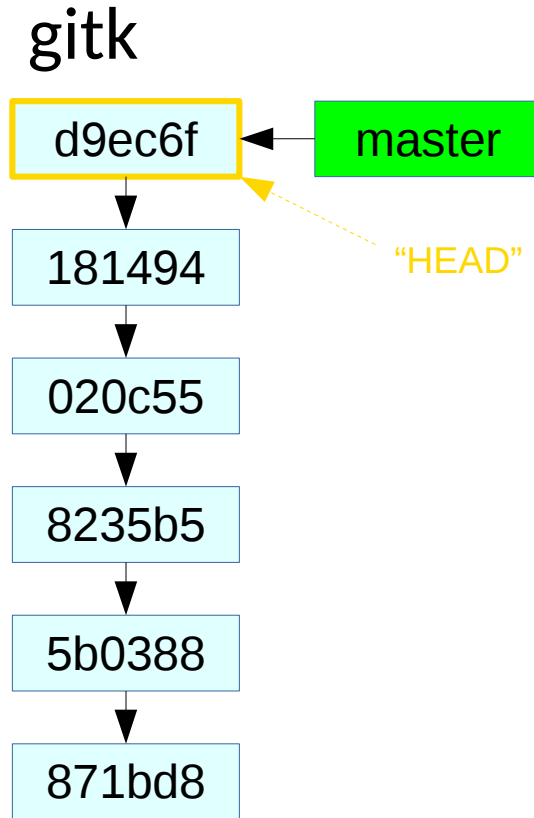
Rename main module main.c to demo.c

The main module should be named after the application, so
rename main.c to demo.c.

Dateien umbenennen und löschen

- Datei löschen
 - `git rm <path>`
 - Ersetzt
 - `rm <path>`
 - `git add <path>`
- Datei umbenennen
 - `git mv <old> <new>`
 - Ersetzt
 - `mv <old> <new>`
 - `git add <old> <new>`
 - GIT erkennt die Umbenennung heuristisch (Ähnlichkeit höher als eine Schwelle)
 - Nicht zuverlässig, wenn die Datei noch zusätzlich verändert ist
 - Umbenennung nicht im Repository gespeichert, sondern erst bei Auswertung, z.B. `git log`

Beispiel: Ältere Versionen (1)



Beispiel: Ältere Versionen (2)

- Ältere Version auschecken
 - Git checkout 5b0388
 - Geht nur, wenn Working Directory sauber ist
 - Keine veränderten Files dürfen durch die Aktion überschrieben werden.

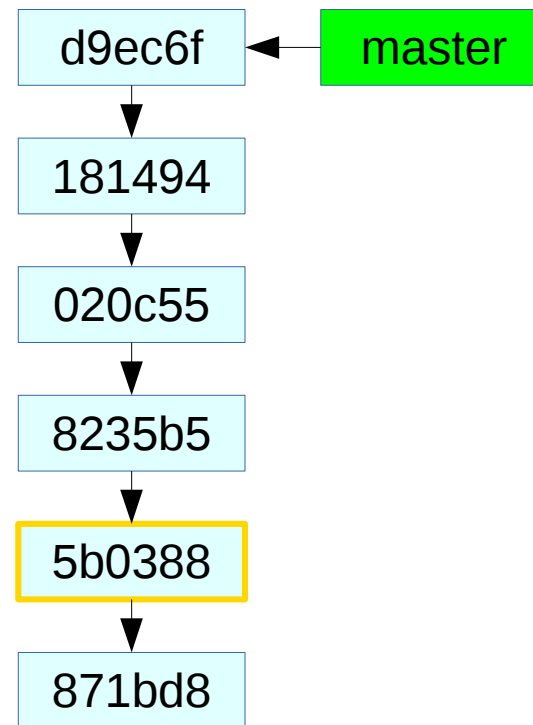
Hinweis: Checke '5b0388' aus.

Sie befinden sich im Zustand eines 'lösgelösten HEAD'. Sie können sich umschauen, experimentelle Änderungen vornehmen und diese committen, und Sie können alle möglichen Commits, die Sie in diesem Zustand machen, ohne Auswirkungen auf irgendeinen Branch zu verwerfen, indem Sie einen weiteren Checkout durchführen.

Wenn Sie einen neuen Branch erstellen möchten, um Ihre erstellten Commits zu behalten, können Sie das (jetzt oder später) durch einen weiteren Checkout mit der Option -b tun. Beispiel:

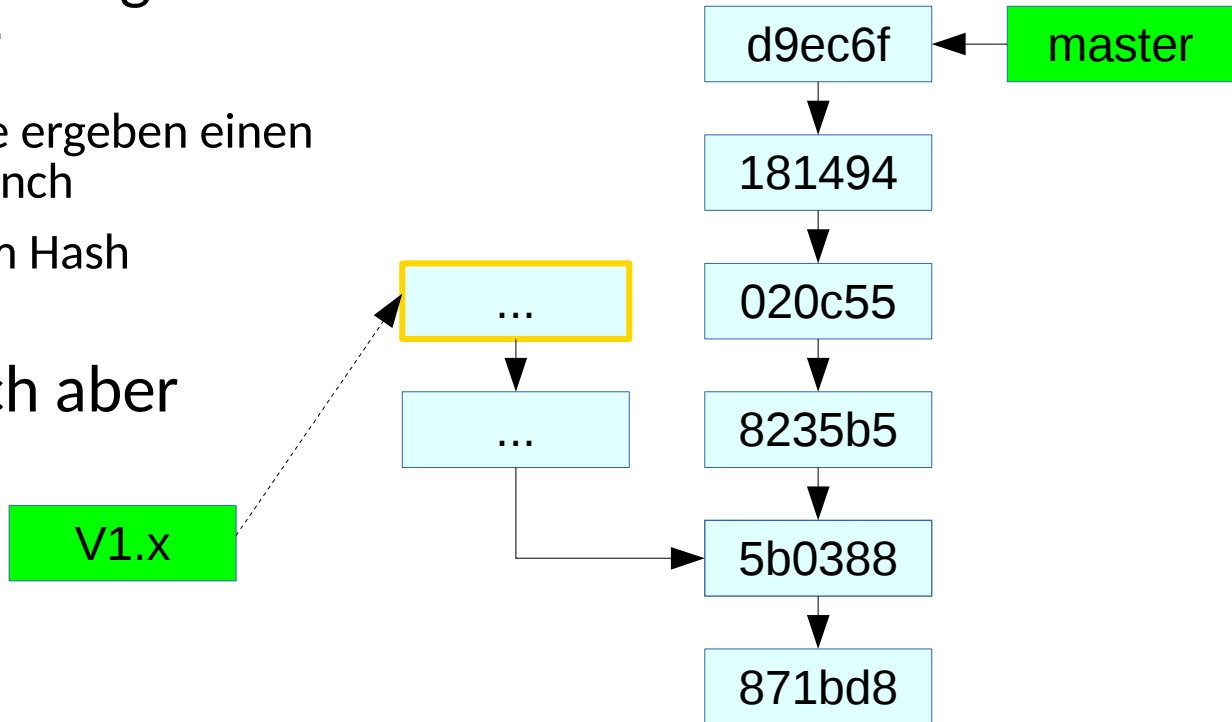
```
git checkout -b <neuer-Branchname>
```

HEAD ist jetzt bei 5b03880... Add basic version



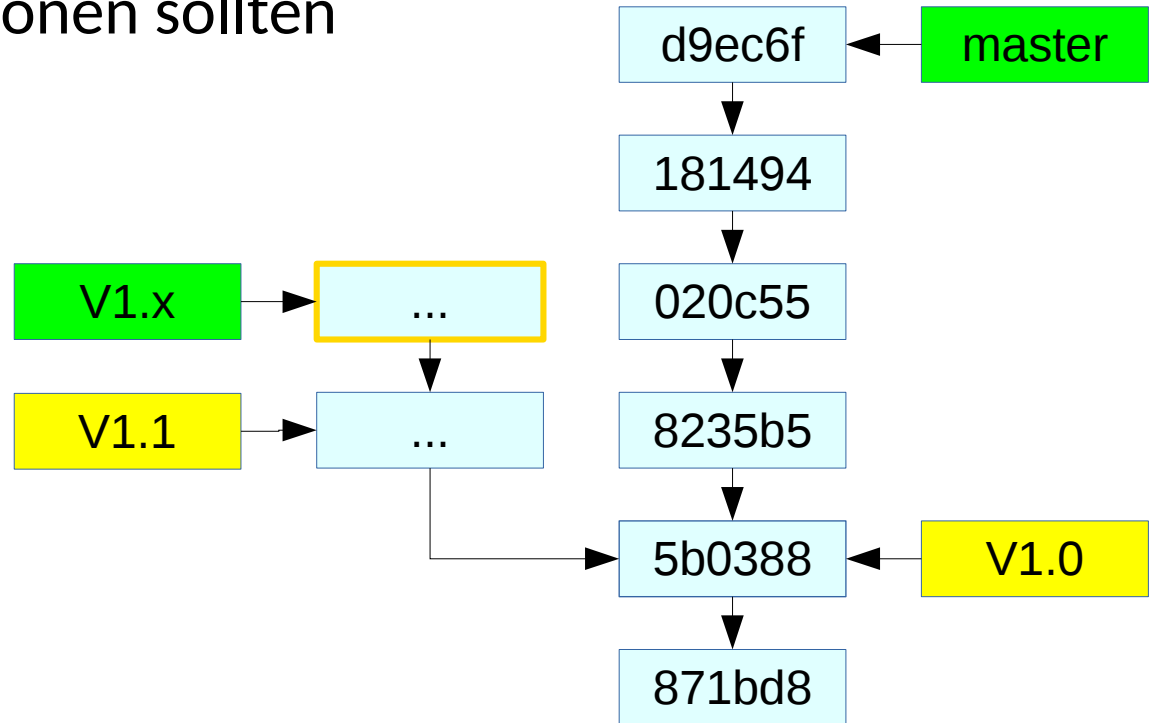
Beispiel: Ältere Versionen (3)

- Achtung! Man ist jetzt losgelöst vom Branch master
 - Commits an dieser Stelle ergeben einen neuen unbenannten Branch
 - Darauf kann nur mit dem Hash zugegriffen werden
- Man kann dem Branch aber einen Namen geben
 - `git branch V1.x`



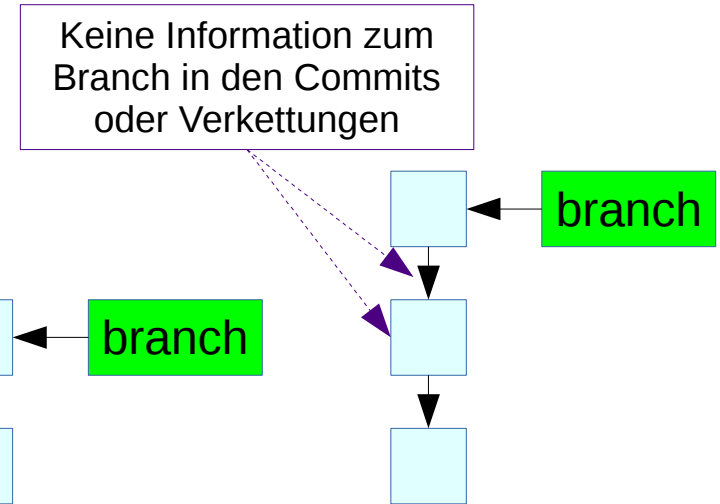
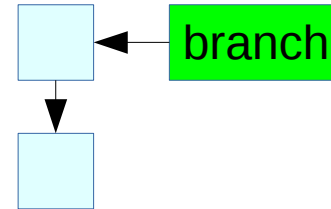
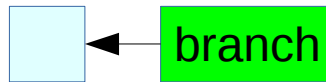
Beispiel: Markierung hinzufügen

- Bisher Zugriff auf Commits nur über Hash
- Häufig referenzierte Versionen sollten einen Namen bekommen
 - `git tag V1.0 5b0388`
 - `git tag V1.1 ...`



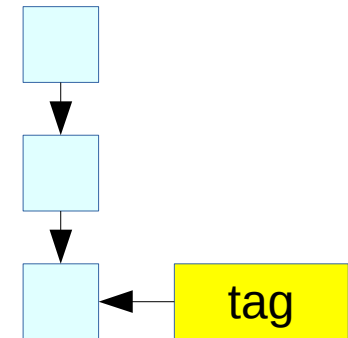
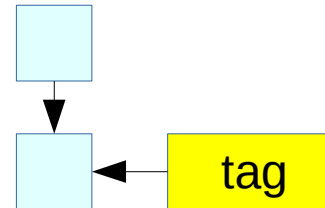
Zweige (Branches)

- `git branch <name> [<commit>]`
 - Erzeugt einen Branch an der aktuellen Position oder am angegebenen Commit
- `git branch -m <old> <new>`
 - Benennt einen Branch von <old> nach <new> um
- `git branch -d <name>`
 - Löscht den Branch (nur den Zugriff, die Commits bleiben erhalten)
- `git branch [-a]`
 - Listet Branches auf (mit -a auch die Tracking-Branches)
- Ein Branch ist ein Zugriff (Handle) auf den Baum, der mit neu angelegten Commits mitwandert



Markierungen (Tags)

- `git tag <name> [<hash>]`
 - Aktuellen bzw. angegebenen Commit mit der Markierung <name> versehen
- `git tag -d <name>`
 - Löscht den Tag <name>
- `git tag`
 - Listet alle Tags des Repositorie
- Ein Tag ist ein Zugriff auf den Baum, der immer an einer Stelle verbleibt



Beispiel: Zugriff auf Referenzen

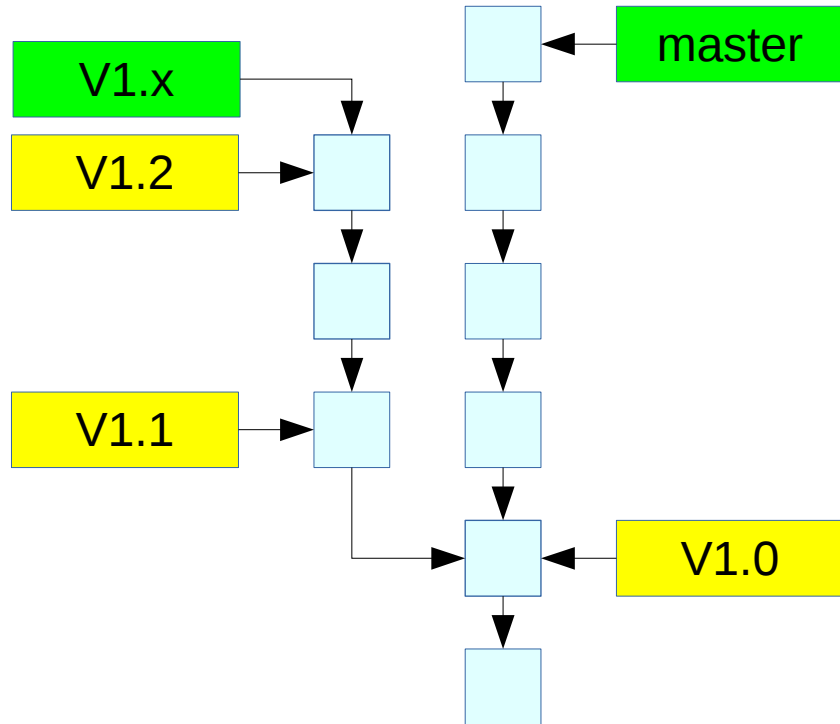
- `git checkout master`
 - Checkt Branch `master` aus (neuester Commit)
- `git checkout V1.x`
 - Checkt Branch `V1.x` aus (neuester Commit)
- `git checkout <hash>`
 - Checkt den angegebenen Commit aus (losgelöst)
- `git checkout V1.0`
 - Checkt den Commit mit Tag `V1.0` aus (losgelöst)
- `git diff V1.0 -- main.c`
 - Listet Unterschiede zwischen `V1.0` und aktueller Version auf
- `git diff V1.0 V1.x -- main.c`
 - Listet Unterschiede zwischen Commit mit Tag `V1.0` und neuestem Commit in Branch `V1.x` auf

Zugriff auf Referenzen

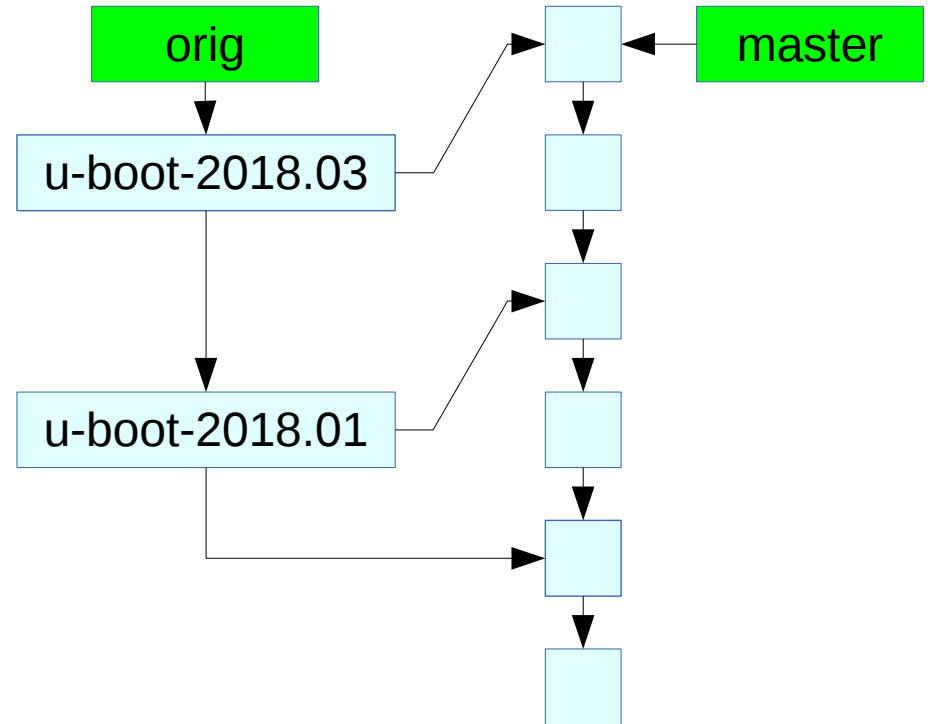
- Über Hash, Tag, Branch oder HEAD
- Modifikatoren
 - $^{\langle n \rangle}$
 - Parent, z.B. HEAD²: zweiter Parent von HEAD
 - $\sim \langle n \rangle$
 - n-ter Vorfahre, z.B. HEAD^{~3}: dritter Commit vor HEAD
- Bereiche
 - $\langle a \rangle .. \langle b \rangle$
 - Alle Commits, die von $\langle b \rangle$ aber nicht von $\langle a \rangle$ erreichbar sind, also $\langle b \rangle$, $\langle b \rangle^{\sim 1}$, $\langle b \rangle^{\sim 2}$, bis einen Commit vor $\langle a \rangle$
 - z.B. git log HEAD^{~3}..HEAD: Liste nur die drei neuesten Commits

Dauerhafte Versions-Branches

- Release Branch
 - Weiterpflege einer Release-Version



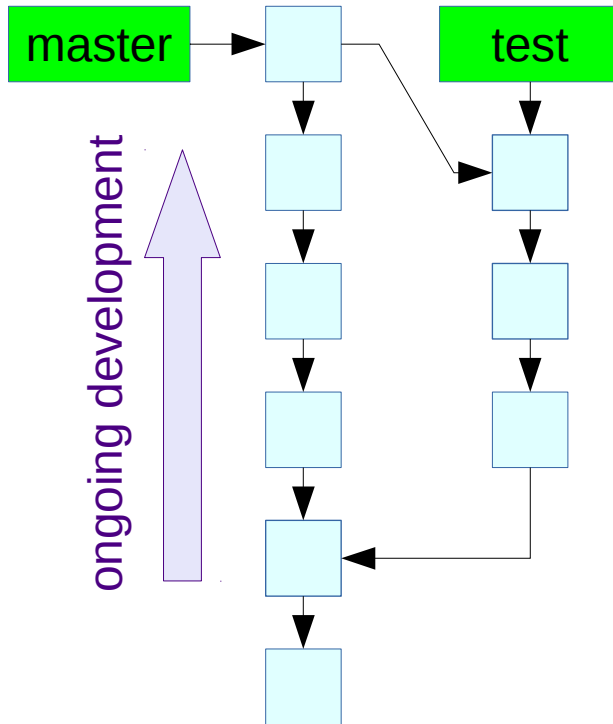
- Vendor Branch
 - Einbringen von Upstream-/Mainline-Versionen



Temporäre Entwicklungs-Branches

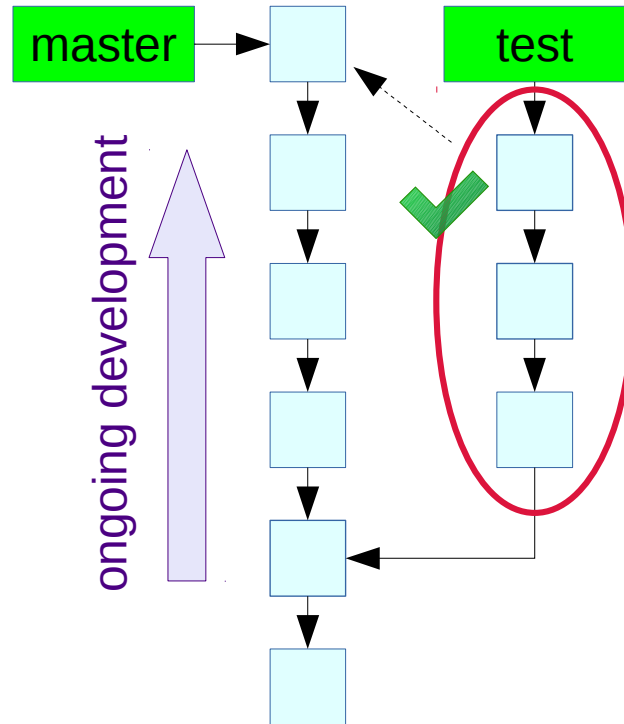
1. Parallelentwicklung

- Stände werden nur zusammengeführt (merge)



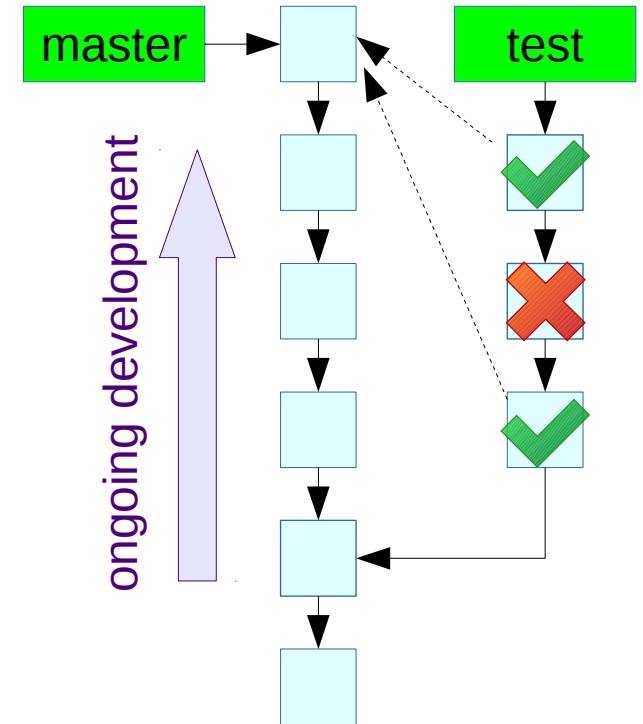
2. Weiterentwicklung

- Alles soll in die neueste Version übernommen werden (rebase)



3. Test/Versuch

- Keine oder nur wenige Teile werden übernommen (cherry-pick)



Beispiel: Branch "test" anlegen (1)

- git checkout master
- git branch test
- git checkout test
- git branch
- emacs demo.c
- git add demo.c
- git commit

```
master  
* test
```

```
#include <stdio.h>  
#include <stdlib.h> /* strtol() */  
#include "inc.h"  
  
int main(int argc, char *argv[])  
{  
    int a;  
  
    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;  
    printf("a=%d, a+1=%d\n", a, inc(a, 1));  
  
    return 0;  
}
```

```
Eliminate variable b in main()  
Variable b is not necessary, drop it.
```

Beispiel: Branch "test" anlegen (2)

- emacs inc.c

```
int inc(int x, int i)
{
    return x + i;          /* Return the sum */
}
```

- git add inc.c

- git commit

```
Explain return value of inc()
Add a comment to explain the return value of function inc().
```

Beispiel: Branch "test" anlegen (3)

- emacs Makefile
- git add Makefile
- git commit

```
CC = gcc
CFLAGS += -Wall

OBJS = demo.o inc.o

demo: $(OBJS)
    $(CC) $(CFLAGS) $^ -o $@

$(OBJS): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@

.PHONY: clean distclean
clean:
    rm -f demo $(OBJS)

distclean: clean
    rm -f *~
```

Add distclean to Makefile

In addition to clean, distclean also removes all editor backup files. Make clear that clean and distclean do not create files.

Beispiel: Branch “master” entwickelt sich weiter (1)

- git checkout master
- emacs inc.c
- emacs inc.h
- emacs demo.c
- git add inc.c inc.h demo.c
- git commit

Add description comment in all source files
Add a small description of the program to all C source code files.

```
/* Small demo program to show GIT features */  
  
int inc(int x, int i)  
{  
    return x + i;  
}
```

```
/* Small demo program to show GIT features */  
  
#ifndef __INC_H__  
#define __INC_H__  
  
int inc(int x, int i);  
  
#endif /* !__INC_H__ */
```

```
/* Small demo program to show GIT features */  
  
#include <stdio.h>  
#include <stdlib.h> /* strtol() */  
#include "inc.h"  
  
int main(int argc, char *argv[])  
{  
    int a, b;  
  
    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;  
  
    b = inc(a, 1);  
    printf("a=%d, b=%d\n", a, b);  
  
    return 0;  
}
```


Beispiel: Branch “master” entwickelt sich weiter (2)

- emacs inc.h

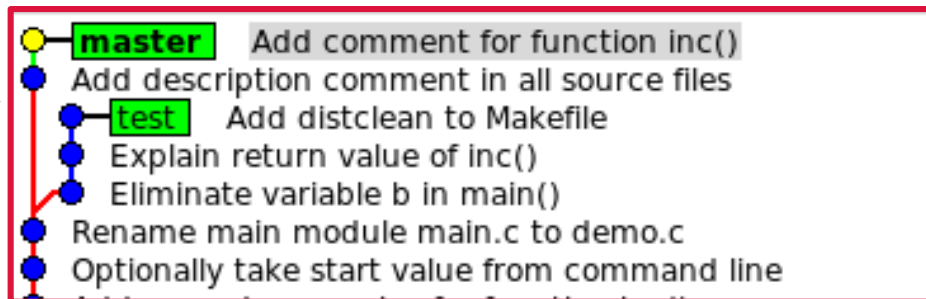
```
/* Small demo program to show GIT features */  
  
#ifndef __INC_H__  
#define __INC_H__  
  
/* Increase first value by second value and return result */  
int inc(int x, int i);  
  
#endif /* ! INC_H */
```

git add inc.h

- git commit

Add comment for function inc()
In inc.h, explain the function of inc() by adding a comment.

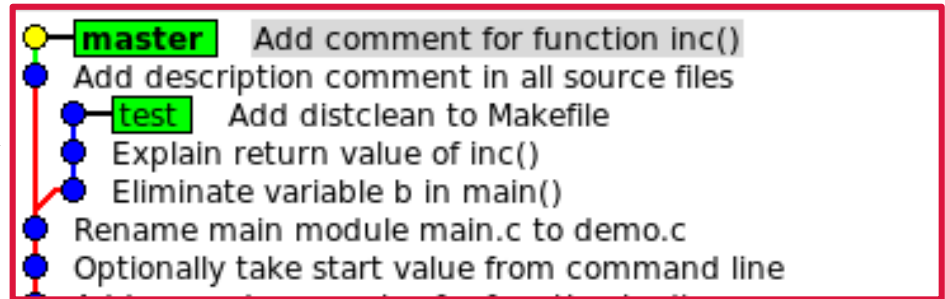
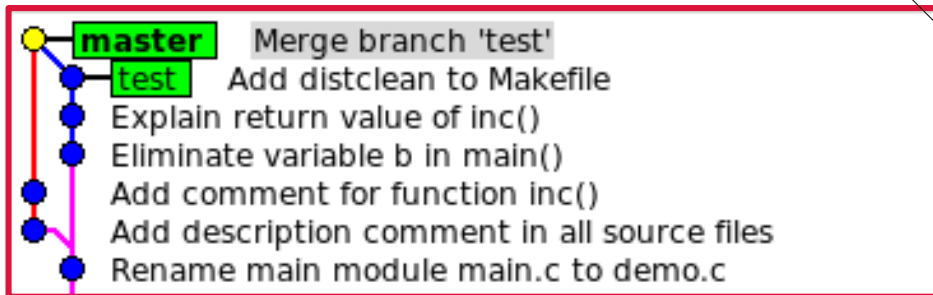
- gitk --all



Beispiel: Variante 1: Merge

- Branch test vereinigen

- gitk --all
- git merge test
- gitk --all



automatischer Merge von inc.c
automatischer Merge von demo.c
Merge made by the 'recursive' strategy.
Makefile | 4 ++++
demo.c | 5 ++--
inc.c | 2 +-
3 files changed, 7 insertions(+), 4 deletions(-)

Merge branch 'test'
Bring in all features of test development.

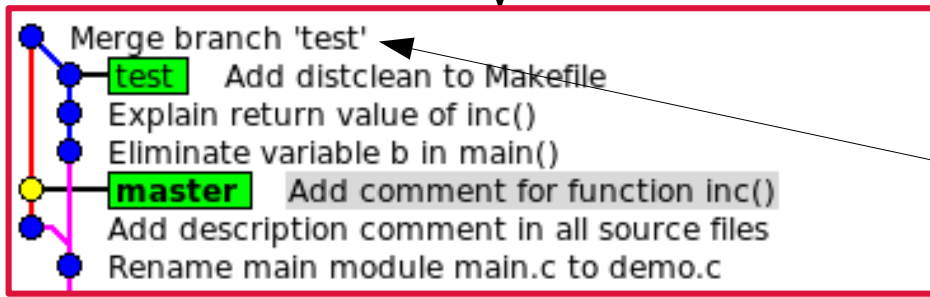
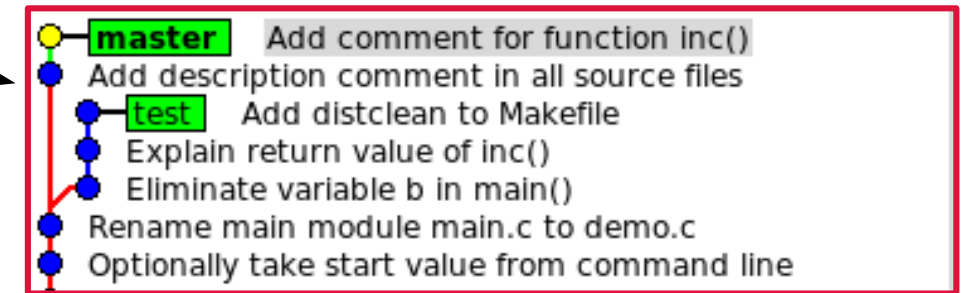
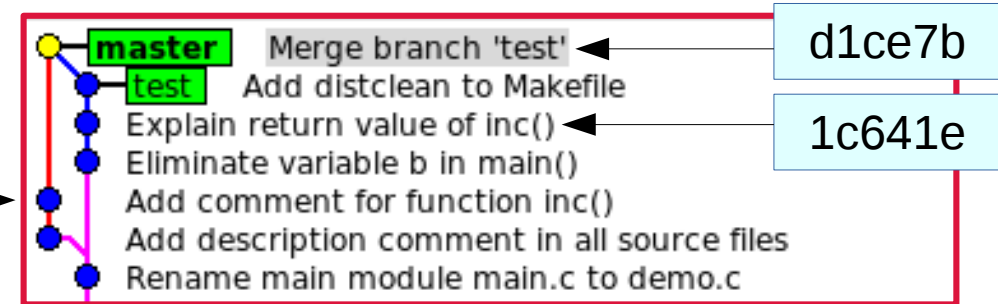
Beispiel: Zurücksetzen von Branch "master"

- Branch master auf den Zustand vor dem Cherry-Picking zurücksetzen

- gitk --all
 - git reset --hard 1c641e
 - gitk --all

- Commits bleiben erhalten

- gitk --all d1ce7b

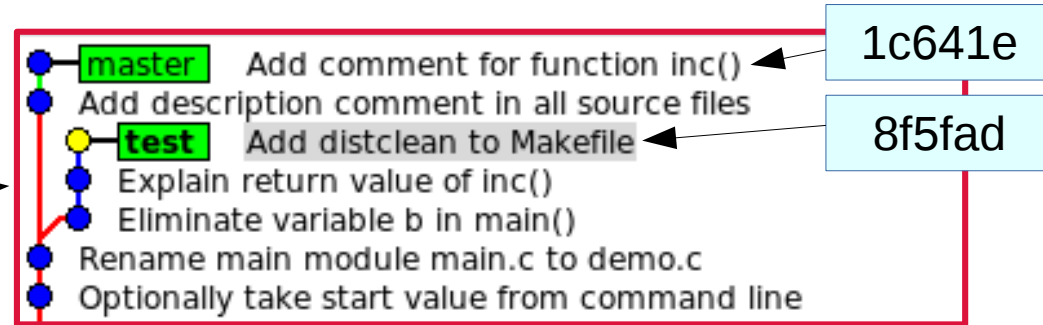


“Baumelnder” Eintrag (Dangling Commit)

- Nur über Hash erreichbar

Beispiel: Variante 2: Rebase (1)

- In den Branch wechseln, der bewegt wird
 - `git checkout test`
 - `gitk --all`
- Auf einen anderen Commit umsetzen
 - `git rebase master`
 - `gitk --all`



Zunächst wird der Branch zurückgespult, um Ihre Änderungen darauf neu anzuwenden ...

Wende an: Eliminate variable b in main()

Wende an: Explain return value of inc()

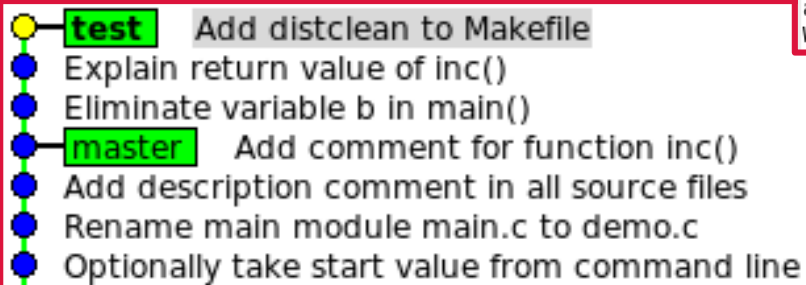
Verwende Informationen aus der Staging-Area, um ein Basisverzeichnis nachzustellen ...

M inc.c

Falle zurück zum Patchen der Basis und zum 3-Wege-Merge ...

automatischer Merge von inc.c

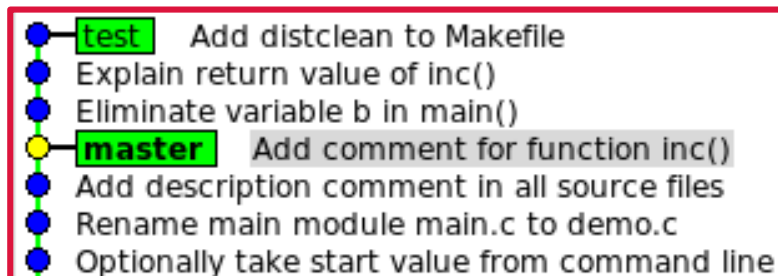
Wende an: Add distclean to Makefile



Beispiel: Variante 2: Rebase (2)

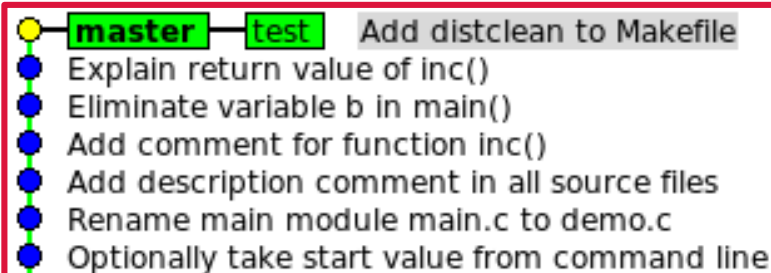
- Branch master vorspulen (Fast Forward)

- `git checkout master`
- `gitk -all`



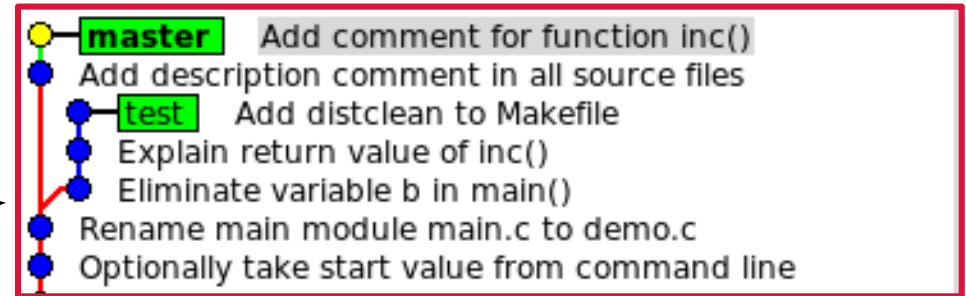
```
Aktualisiere 1c641ee..bac02f8
Fast-forward
 Makefile | 4 ++++
 demo.c   | 5 ++--
 inc.c    | 2 +-
 3 files changed, 7 insertions(+), 4 deletions(-)
```

- `git merge test`
- `gitk --all`



Zurücksetzen von “master” und “test”

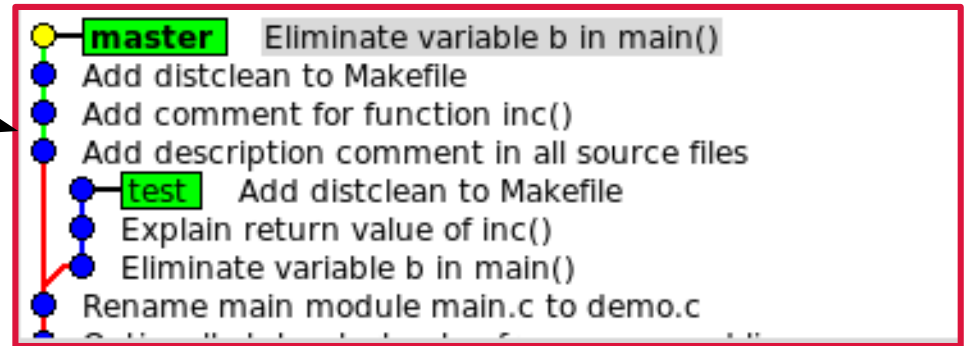
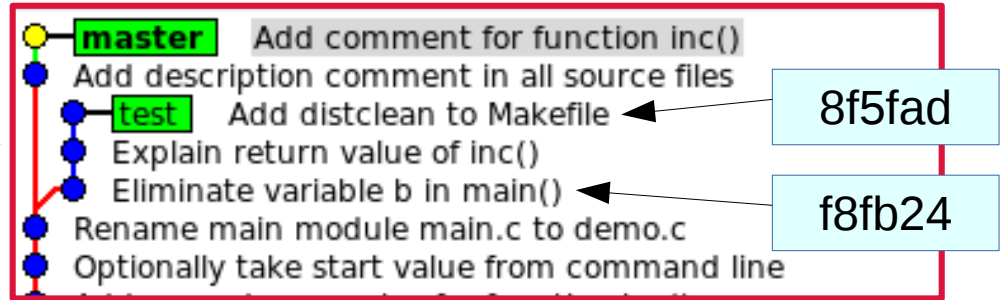
- Branch `test` auf den Zustand vor dem Rebase zurücksetzen
 - `git checkout test`
 - `git reset --hard 8f5fad`
- Branch `master` auf den Zustand vor dem Rebase zurücksetzen
 - `git checkout master`
 - `git reset --hard 1c641e`
 - `gitk --all`



Beispiel: Variante 3: Cherry-Picking

- Nur zwei Commits übernehmen

- gitk --all
- git cherry-pick 8f5fad
- git cherry-pick f8fb24
- gitk --all



Zweige zusammenführen

- `git merge <commit>`
 - Führt den genannten Commit (i.a. ein Branch) mit der aktuellen Position (HEAD) zusammen
 - Der einfachste Merge ist ein Vorspulen (Fast Forward), dann wird HEAD nur verschoben
 - Sonst wird ein neuer Commit erzeugt, der zwei Parents hat
- `git rebase [--interactive] <commit>`
 - “Verschiebt” alles ab HEAD (i.a. ein Branch) an den angegebenen Commit
 - Verschoben wird alles bis zum ersten gemeinsamen Parent von HEAD und <commit>, d.h. alles was von HEAD aus, aber nicht von <commit> aus zu sehen ist.
 - Die Änderungen werden dabei neu an der angegebenen Position eingefügt
 - Bei `--interactive` können die Commits des Zweigs noch umsortiert, gesplittet, zusammengefasst und verändert werden → sehr mächtig!
- `git cherry-pick <commit>`
 - Fügt den angegebenen Commit nochmal erneut an der aktuellen Position (HEAD) ein

Branches zurücksetzen

- `git reset --hard <commit>`
 - Der aktuelle Branch und das Working Directory werden auf `<commit>` gesetzt, der Index wird geleert
 - Man ist wieder komplett auf dem Stand von `<commit>`
- `git reset --mixed <commit>`
 - Der aktuelle Branch wird auf `<commit>` gesetzt, der Index wird geleert; aber das Working Directory bleibt unverändert
 - Man hat ggf. gleich veränderte Dateien, da diese von HEAD abweichen
- `git reset --soft <commit>`
 - Der aktuelle Branch wird auf `<commit>` gesetzt, aber Index und Working Directory bleiben unverändert
 - Man kann Daten rüberretten und neu einchecken

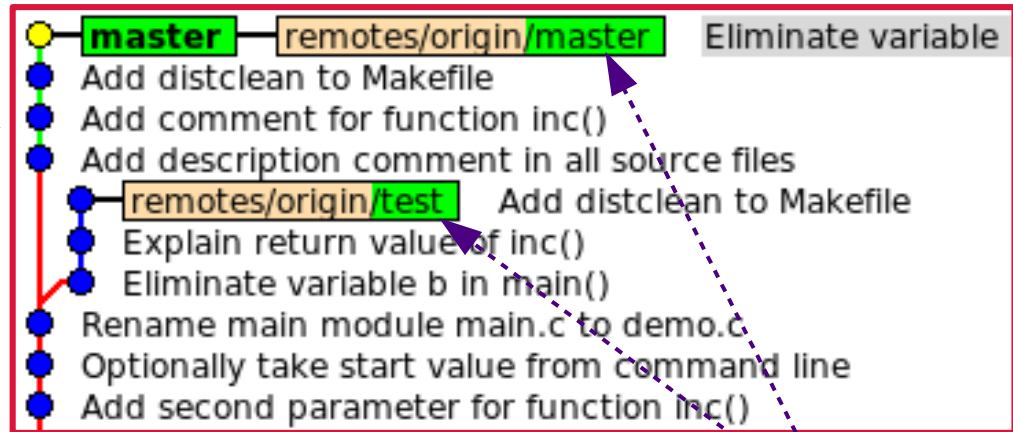
Repository klonen

- `git clone [--bare] [--origin <name>] [--branch <branch>] <source> [<dir>]`
 - Erzeugt eine Kopie des Repositories bei <source> (lokaler Pfad oder URL)
 - Bei lokalen Klonen werden die Objekte per Hardlink kopiert, brauchen also nicht nochmal Speicher; nur die ausgecheckten Dateien brauchen Platz
 - Wenn <dir> nicht angegeben ist, wird ein Verzeichnis mit dem Namen von <source> angelegt
 - Es wird ein Remote-Zugriff unter dem Namen <name> angelegt (default: origin)
 - Falls nicht --bare angegeben ist, wird der Branch <branch> ausgecheckt (default: branch, wo HEAD steht)
 - Ein Bare-Repository besteht sozusagen nur aus dem .git-Verzeichnis

Beispiel: Repository klonen

- `cd ..`
- `git clone demo copy`
- `cd copy`
- `git remote -v`
- `gitk --all`
- `git branch`
- `git branch -a`

```
origin /home/keller/git-kurs/demo (fetch)
origin /home/keller/git-kurs/demo (push)
```



Remote Tracking
Branches

```
* master
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/test
```

```
* master
```

Beispiel: Lokale Änderung

- emacs inc.c
- emacs Makefile
- git add inc.c Makefile
- git commit

```
/* Small demo program to show GIT features */  
#include "inc.h"  
  
int inc(int x, int i)  
{  
    return x + i;  
}
```

```
CC = gcc  
CFLAGS += -Wall  
  
OBJS = demo.o inc.o  
  
demo: $(OBJS)  
    $(CC) $(CFLAGS) $^ -o $@  
  
$(OBJS): %.o: %.c  
    $(CC) -c $(CFLAGS) $< -o $@  
  
.PHONY: clean distclean  
clean:  
    rm -f demo $(OBJS)  
  
distclean: clean  
    rm -f *~  
  
demo.o: demo.c inc.h  
inc.o: inc.c inc.h
```

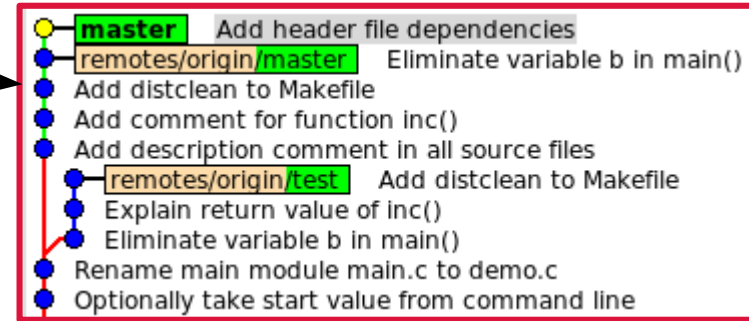
Add header file dependencies

The Makefile needs dependencies to the header files, or otherwise changes on the header do not trigger recompilation. Also include inc.h in inc.c to make sure the function signatures do not differ.

Beispiel: Änderung pushen (1)

- `gitk --all`
- `git push origin`

```
Delta compression using up to 6 threads.
Komprimiere Objekte: 100% (4/4), Fertig.
Schreibe Objekte: 100% (4/4), 560 bytes | 560.00 KiB/s, Fertig.
Total 4 (delta 2), reused 0 (delta 0)
remote: error: refusing to update checked out branch: refs/heads/master
remote: error: Standardmäßig wird die Aktualisierung des aktuellen Branches in einem
remote: Nicht-Bare-Repository zurückgewiesen, da dies den Index und das Arbeits-
remote: verzeichnis inkonsistent zu dem machen würde, was Sie gepusht haben, und
remote: 'git reset --hard' erforderlich wäre, damit das Arbeitsverzeichnis HEAD
remote: entspricht.
remote:
remote: Sie könnten die Konfigurationsvariable 'receive.denyCurrentBranch' im
remote: Remote-Repository auf 'ignore' oder 'warn' setzen, um den Push in den
remote: aktuellen Branch zu erlauben; dies wird jedoch nicht empfohlen außer
remote: Sie stellen durch andere Wege die Aktualität des Arbeitsverzeichnisses
remote: gegenüber dem gepushten Stand sicher.
remote:
remote: Um diese Meldung zu unterdrücken und das Standardverhalten zu behalten,
remote: setzen Sie die Konfigurationsvariable 'receive.denyCurrentBranch' auf
remote: 'refuse'.
To /home/keller/git-kurs/demo
! [remote rejected] master -> master (branch is currently checked out)
error: Fehler beim Versenden einiger Referenzen nach '/home/keller/git-kurs/demo'
```



- Remote-Repository (demo) kann Daten nicht annehmen, da diese dort ausgecheckt sind
- Für den dortigen Benutzer würde sich sonst das Repository relativ zu seinem Working-Directory ändern, ohne dass er es selbst veranlasst hat

Beispiel: Änderungen pushen (2)

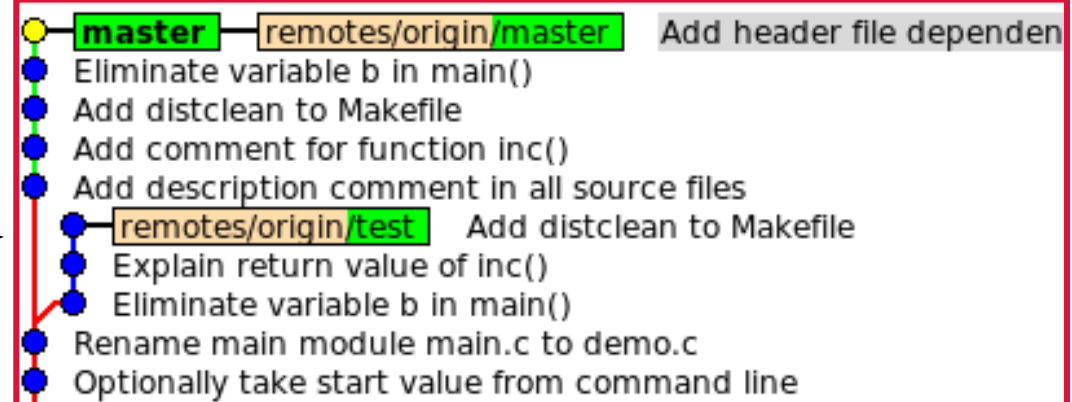
- Testweise in “demo” den Branch `test` auschecken

- `cd ../demo`
- `git checkout test`

```
Zähle Objekte: 4, Fertig.  
Delta compression using up to 6 threads.  
Komprimiere Objekte: 100% (4/4), Fertig.  
Schreibe Objekte: 100% (4/4), 560 bytes | 560.00 KiB/s, Fertig.  
Total 4 (delta 2), reused 0 (delta 0)  
To /home/keller/git-kurs/demo  
31a4cf8..a290db0 master -> master
```

- Erneut probieren

- `cd ../copy`
- `git push origin`
- `gitk --all`

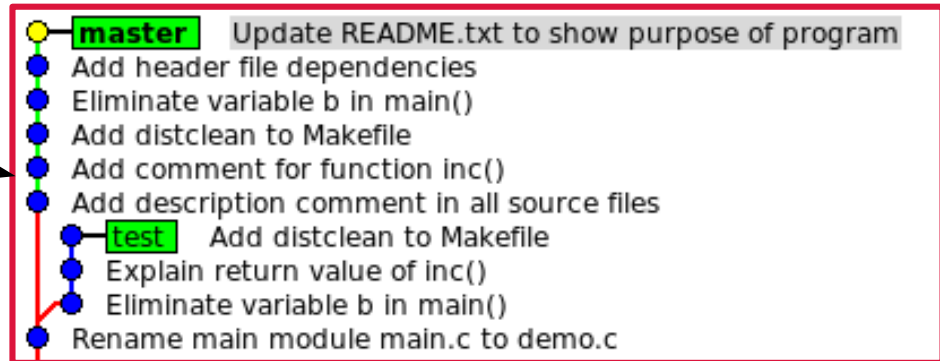


Beispiel: Änderung im Remote-Repository

- `cd ../demo`
- `git checkout master`
- `emacs README.txt`
- `git add README.txt`
- `git commit`
- `gitk --all`

This is a program to demonstrate how GIT works. It accepts an optional number and shows some more numbers relative to it.

Update README.txt to show purpose of program



Beispiel: Änderungen vom Remote-Repository holen

- `cd ../copy`
- `git fetch origin`
- `gitk -all`
- `git merge origin/master`
- `gitk --all`

```
remote: Zähle Objekte: 3, Fertig.  
remote: Komprimiere Objekte: 100% (3/3), Fertig.  
remote: Total 3 (delta 1), reused 0 (delta 0)  
Entpacke Objekte: 100% (3/3), Fertig.  
Von /home/keller/git-kurs/demo  
a290db0..16d7d0a master -> origin/master
```

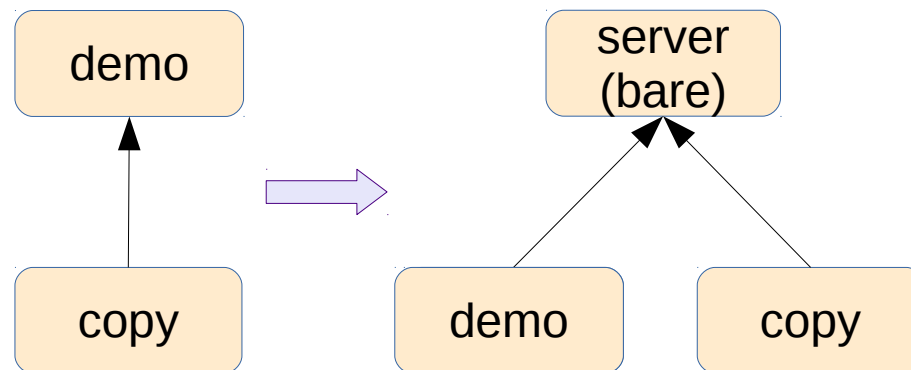
remotes/origin/master Update README.txt to show purpose
master Add header file dependencies
Eliminate variable b in main()
Add distclean to Makefile
Add comment for function inc()
Add description comment in all source files
remotes/origin/test Add distclean to Makefile
Explain return value of inc()
Eliminate variable b in main()
Rename main module main.c to demo.c

```
Aktualisiere a290db0..16d7d0a  
Fast-forward  
 README.txt | 3 ++-  
 1 file changed, 2 insertions(+), 1 deletion(-)
```

master remotes/origin/master Update README.txt to sh
Add header file dependencies
Eliminate variable b in main()
Add distclean to Makefile
Add comment for function inc()
Add description comment in all source files
remotes/origin/test Add distclean to Makefile
Explain return value of inc()
Eliminate variable b in main()
Rename main module main.c to demo.c

Beispiel: Bare-Repository

- Repositories, auf die gepushed werden soll, sind üblicherweise Bare-Repositories (ohne ausgecheckte Files)
- Erzeuge ein Bare-Repository “server”
 - `cd ..`
 - `git clone --bare demo server`
 - Erzeugt keine Remote Tracking Branches
- Clone davon “demo”
 - `rm -rf demo`
 - `git clone server demo`
- Ändere remote-Eintrag für “copy” ab
 - `cd copy`
 - `git remote set-url origin ~/git-kurs/server`
 - `git remote -v`



```
origin /home/keller/git-kurs/server (fetch)
origin /home/keller/git-kurs/server (push)
```

Beispiel: User "demo" fügt Commit hinzu

- `cd ../demo`
- `emacs demo.c`
- `git add demo.c`
- `git commit`
- `git push origin`

```
/* Small demo program to show GIT features */  
  
#include <stdio.h>  
#include <stdlib.h>          /* strtol() */  
#include "inc.h"  
  
int main(int argc, char *argv[])  
{  
    int a;  
  
    if (argc > 2) {  
        printf("Error, only one argument allowed\n");  
        return 1;  
    }  
  
    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;  
    printf("a=%d, a+1=%d\n", a, inc(a, 1));  
  
    return 0;  
}
```

Add error check if more than argument is given

If the user specifies more than one argument on the command line, show an error and return value 1.

Beispiel: User “copy” fügt Änderungen hinzu (1)

- `cd ../copy`
- `emacs demo.c` →
- `git add demo.c`
- `git commit`

Add option --help
Show usage if argument is --help.

```
/* Small demo program to show GIT features */  
  
#include <stdio.h>  
#include <stdlib.h>          /* strtol() */  
#include <string.h>          /* strcmp() */  
#include "inc.h"  
  
int main(int argc, char *argv[])  
{  
    int a;  
  
    if ((argc > 1) && !strcmp(argv[1], "--help")) {  
        printf("\nUsage: %s [--help] [n]\n", argv[0]);  
        printf("Show some meaningless numbers relative to n.\n");  
        printf("If n is not given, use 5 as default.\n\n");  
  
        return 0;  
    }  
  
    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;  
  
    printf("a=%d, a+1=%d\n", a, inc(a, 1));  
  
    return 0;  
}
```

Beispiel: User “copy” fügt Änderungen hinzu (2)

- emacs inc.c
- emacs demo.c
- git add inc.c demo.c
- git commit

```
/* Small demo program to show GIT features */  
#include "inc.h" /* Own interface */  
  
int inc(int x, int i)  
{  
    return x + i;  
}
```

```
/* Small demo program to show GIT features */  
#include <stdio.h> /* printf() */  
#include <stdlib.h> /* strtol() */  
#include <string.h> /* strcmp() */  
#include "inc.h" /* inc() */  
  
int main(int argc, char *argv[])  
{  
    int a;
```

Explain why includes are needed

Add comments for each include line with the reason why this include is needed.

Beispiel: User “copy” fügt Änderungen hinzu (3)

- emacs demo.c
 - Fehler nachbessern
printk → printf
- git add demo.c
- git commit --amend
 - Message kann auch geändert werden
 - Ergänzt den letzten Commit, also weiterhin ein Commit

```
/* Small demo program to show GIT features */  
  
#include <stdio.h>           /* printf() */  
#include <stdlib.h>          /* strtol() */  
#include <string.h>          /* strcmp() */  
#include "inc.h"             /* inc() */  
  
int main(int argc, char *argv[])  
{  
    int a;
```

Explain why includes are needed

Add comments for each include line with the reason why this include is needed.

Beispiel: User “copy” fügt Änderungen hinzu (4)

- emacs demo.c
- git add demo.c
- git commit

```
/* Small demo program to show GIT features */  
  
#include <stdio.h>           /* printf() */  
#include <stdlib.h>          /* strtol() */  
#include <string.h>          /* strcmp() */  
#include "inc.h"             /* inc() */  
  
int main(int argc, char *argv[])  
{  
    int a;  
  
    if ((argc > 1) &&  
        (!strcmp(argv[1], "--help") || !strcmp(argv[1], "-h"))) {  
        printf("\nUsage: %s [--help | -h] [n]\n", argv[0]);  
        printf("Show some meaningless numbers relative to n.\n");  
        printf("If n is not given, use 5 as default.\n\n");  
  
        return 0;  
    }  
  
    a = (argc > 1) ? strtol(argv[1], NULL, 0) : 5;  
    printf("a=%d, a+1=%d\n", a, inc(a, 1));  
  
    return 0;  
}
```

Also add option -h for help

Not only have --help, but also -h as option to show the usage.

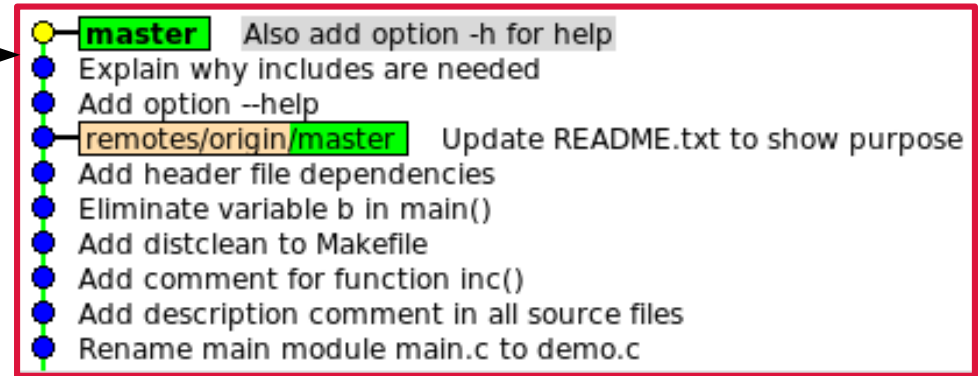
Beispiel: Commits neu ordnen (1)

- gitk

- 1. und 3. Commit sollten eigentlich gemeinsam in einem Commit sein

```
pick 3899998 Add option --help
pick 6814416 Explain why includes are needed
pick 1c63c78 Also add option -h for help

# Rebase von 16d7d0a..1c63c78 auf 16d7d0a (3 Kommandos)
#
# Befehle:
# p, pick = Commit verwenden
# r, reword = Commit verwenden, aber Commit-Beschreibung bearbeiten
# e, edit = Commit verwenden, aber zum Nachbessern anhalten
# s, squash = Commit verwenden, aber mit vorherigem Commit vereinen
# f, fixup = wie "squash", aber diese Commit-Beschreibung verwerfen
# x, exec = Befehl (Rest der Zeile) mittels Shell ausführen
# d, drop = Commit entfernen
#
# Diese Zeilen können umsortiert werden; Sie werden von oben nach unten
# ausgeführt.
#
# Wenn Sie hier eine Zeile entfernen, wird DIESER COMMIT VERLOREN GEHEN.
#
# Wenn Sie jedoch alles löschen, wird der Rebase abgebrochen.
#
# Leere Commits sind auskommentiert.
```



- git rebase --interactive origin/master

- Bietet Liste der Commits an
 - Achtung: Reihenfolge von alt nach neu!
- Kann umsortiert werden

Beispiel: Commits neu ordnen (2)

```
pick 3899998 Add option --help
pick 6814416 Explain why includes are needed
pick 1c63c78 Also add option -h for help
```

```
# Das ist eine Kombination aus 2 Commits.
# Das ist die erste Commit-Beschreibung:
```

```
Add option --help
```

```
Show usage if argument is --help.
```

```
# Das ist Commit-Beschreibung #2:
```

```
Also add option -h for help
```

```
Not only have --help, but also -h as option to show the usage.
```

```
# Bitte geben Sie eine Commit-Beschreibung für Ihre Änderungen ein. Zeilen,
# die mit '#' beginnen, werden ignoriert, und eine leere Beschreibung
# bricht den Commit ab.
```

```
# Datum: Fri Apr 12 16:53:49 2019 +0200
```

```
# interaktives Rebase im Gange; auf 16d7d0a
```

```
# Zuletzt ausgeführte Befehle (2 Befehle ausgeführt):
```

```
# pick 3899998 Add option --help
```

```
# squash 1c63c78 Also add option -h for help
```

```
# Nächster auszuführender Befehl (1 Befehle verbleibend):
```

```
# pick 6814416 Explain why includes are needed
```

```
# Sie sind gerade beim Rebase von Branch 'master' auf '16d7d0a'.
```

```
# zum Commit vorgemerkte Änderungen:
```

```
# geändert: demo.c
```

```
pick 3899998 Add option --help
squash 1c63c78 Also add option -h for help
pick 6814416 Explain why includes are needed
```

```
Add options --help and -h to show usage
Show usage if argument is --help or -h.
```

```
[losgelöster HEAD bd46c01] Add options --help and -h to show usage
Date: Fri Apr 12 16:53:49 2019 +0200
1 file changed, 10 insertions(+)
Successfully rebased and updated refs/heads/master.
```

```
● master Explain why includes are needed
● Add options --help and -h to show usage
● remotes/origin/master Update README.txt to show purpose
● Add header file dependencies
● Eliminate variable b in main()
```


Beispiel: User “copy” will pushen

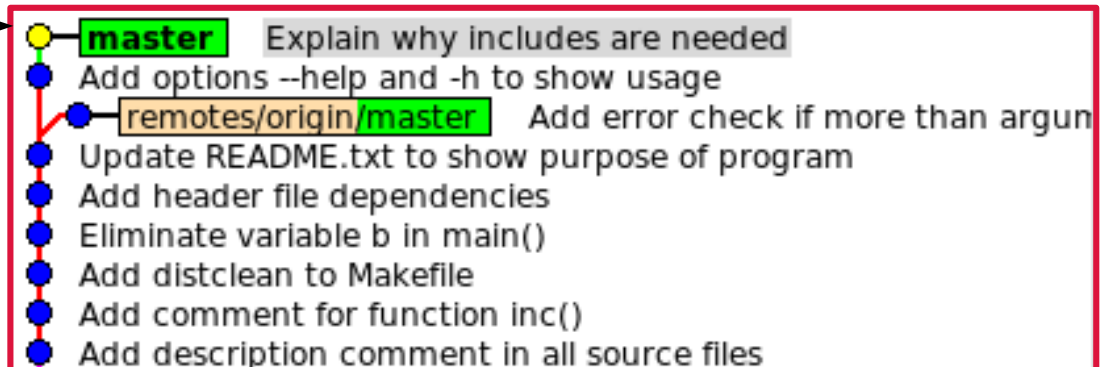
- `git push origin` →

- Schägt fehl, weil im Remote-Repository neue Commits sind

```
To /home/keller/git-kurs/server
! [rejected]        master -> master (fetch first)
error: Fehler beim Versenden einiger Referenzen nach '/home/keller/git-kurs/server'
Hinweis: Aktualisierungen wurden zurückgewiesen, weil das Remote-Repository Commits enthält,
Hinweis: die lokal nicht vorhanden sind. Das wird üblicherweise durch einen "push" von
Hinweis: Commits auf dieselbe Referenz von einem anderen Repository aus verursacht.
Hinweis: Vielleicht müssen Sie die externen Änderungen zusammenführen (z. B. 'git pull ...')
Hinweis: bevor Sie erneut "push" ausführen.
Hinweis: Siehe auch die Sektion 'Note about fast-forwards' in 'git push --help'
Hinweis: für weitere Details.
```

- `git fetch origin` →

- Lokaler Branch und Remote-Branch sind auseinander gelaufen
- Merge oder Rebase notwendig



Beispiel: Versuch mit Rebase

- `git rebase origin/master`
 - Eigentlich schöner, da keine Verzweigungen



- Konfliktauflösung wäre:
 - `git mergetool`
 - Alle Konfliktfiles durchgehen
 - `git rebase -continue`
- Wir machen:
 - `git rebase --abort`

```
Zunächst wird der Branch zurückgespult, um Ihre Änderungen
darauf neu anzuwenden ...
Wende an: Add options --help and -h to show usage
Verwende Informationen aus der Staging-Area, um ein Basisverzeichnis nachzustellen ...
M      demo.c
.git/rebase-apply/patch:24: trailing whitespace.

warning: 1 Zeile fügt Whitespace-Fehler hinzu.
Falle zurück zum Patchen der Basis und zum 3-Wege-Merge ...
automatischer Merge von demo.c
KONFLIKT (Inhalt): Merge-Konflikt in demo.c
error: Merge der Änderungen fehlgeschlagen.
Anwendung des Patches fehlgeschlagen bei 0001 Add options --help and -h to show usage
Die Kopie des fehlgeschlagenen Patches befindet sich in: .git/rebase-apply/patch

Wenn Sie das Problem aufgelöst haben, führen Sie "git rebase --continue" aus.
Falls Sie diesen Patch auslassen möchten, führen Sie stattdessen "git rebase --skip" aus.
Um den ursprünglichen Branch wiederherzustellen und den Rebase abzubrechen,
führen Sie "git rebase --abort" aus.
```

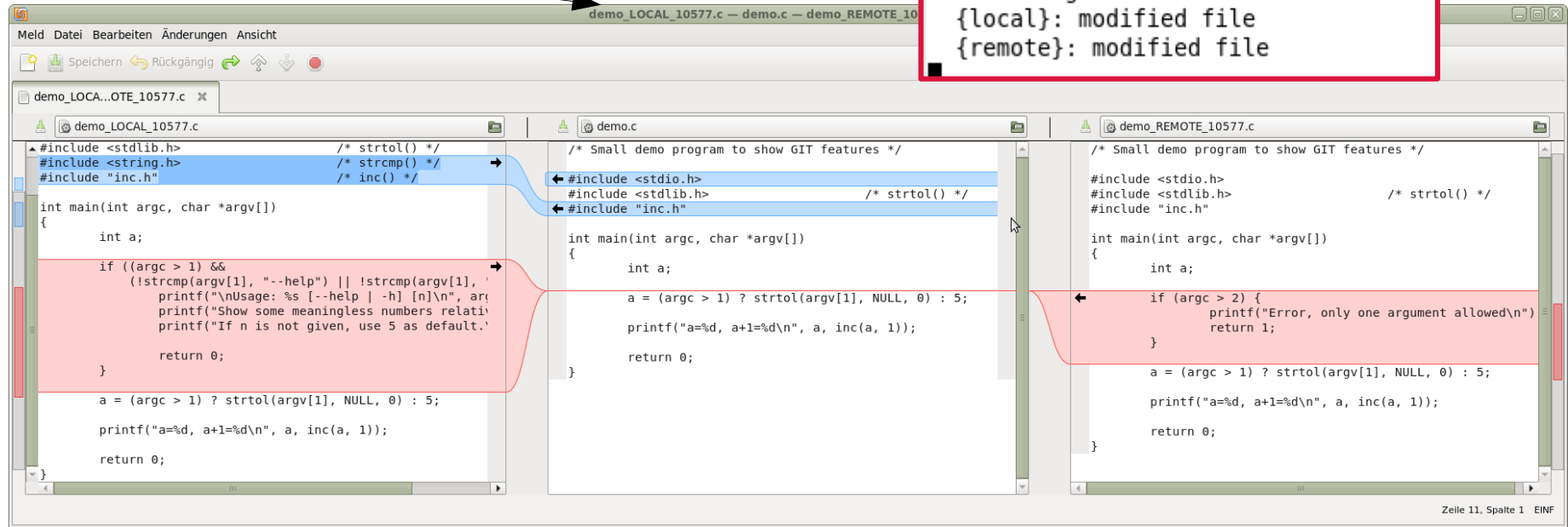
Beispiel: Versuch mit Merge

automatischer Merge von demo.c
KONFLIKT (Inhalt): Merge-Konflikt in demo.c
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie dann das Ergebnis.

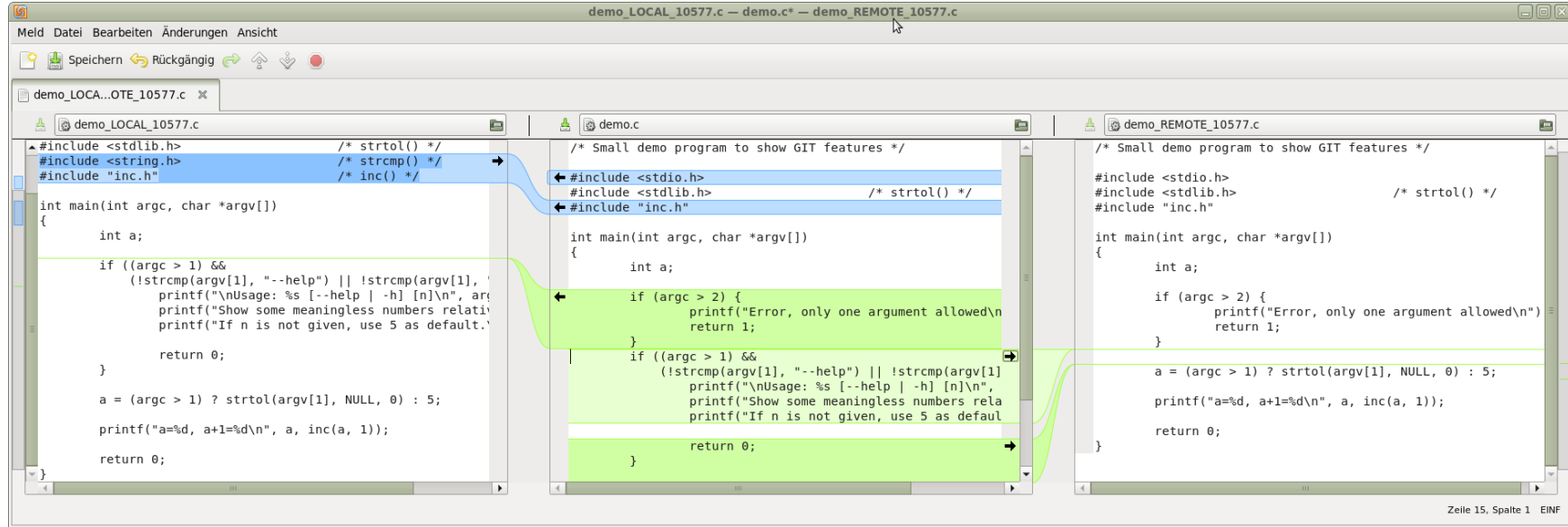
- git merge origin/master
- git mergetool

Merging:
demo.c

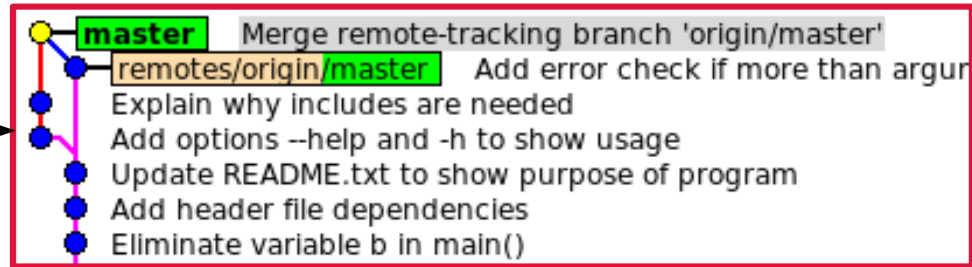
Normal merge conflict for 'demo.c':
{local}: modified file
{remote}: modified file



Beispiel: Konflikt auflösen



- git commit
- gitk --all
- git push origin



Zusammenfassung Remote Repository (1)

- `git fetch [--tags | --no-tags] <name>`
 - Holt alle neuen Commits vom Remote-Repository <name>
 - Wird `--tags` angegeben, werden alle Tags geholt, sonst nur die, die an den übertragenen Commits hängen
 - Mit `--no-tags` werden garantiert keine Tags geholt
- `git pull [--ff-only] <name>`
 - Entspricht `git fetch` gefolgt von `git merge`
 - Mit `--ff-only` kann man sagen, dass nur dann gemerged wird, wenn es ein Fast-Forward wäre, also keine Verzweigung entsteht
 - Guter Weg um in einem Rutsch den lokalen Zweig zu aktualisieren und dennoch die Möglichkeit für einen Rebase zu haben, wenn es nicht trivial ist
- `git push [--tags] <name>`
 - Schickt die lokalen Commits des aktuellen Branches zum zugehörigen Tracking-Branch auf dem Remote-Repository <name>
 - Nur mit `--tags` werden auch neue Tags übermittelt

Zusammenfassung Remote Repository (2)

- `git remote [-v]`
 - Listet alle Remote-Repositories (mit -v ausführlich)
- `git remote add [--no-tags] <name> <url>`
 - Fügt <url> als zusätzliches Remote-Repository unter dem Namen <name>
 - Beispiel: `git remote add second <url>`
- `git remote set-url <name> <url>`
 - Ändert die URL für das Remote-Repository <name>
- `git remote rename <old> <new>`
 - Ändert den Namen des Remote-Repositories von <old> auf <new>
 - Beispiel: `git remote rename origin myname`
- `git remote remove <name>`
 - Entfernt das Remote-Repository <name>
 - **Achtung!** Falls noch Referenzen auf das Repository bestehen (z.B. Tags), bleiben die Referenzen aktiv und werden ggf. gepusht. Wird mit mehreren Repositories gearbeitet um Cherry-Picks zu machen, immer `git remote add --no-tags` verwenden und genau aufpassen, dass nach `git remote remove` auch alle Referenzen wieder weg sind.

Änderungen vorübergehend verstecken

- Manche Befehle (z.B. `git rebase`) funktionieren nur, wenn das Working Directory sauber ist, also keine veränderten Dateien da sind. Solche Änderungen temporär zu sichern ist grundsätzlich gar nicht so einfach:
 - Branch `tmp` erstellen, auf den Branch wechseln
 - Dort alle Dateien einchecken
 - Auf den originalen Branch zurückwechseln
 - Gewünschte Änderungen durchführen
 - Anschließend auf Branch `tmp` wechseln
 - Branch `tmp` mit `git rebase` oder `git cherry-pick` auf den neuen Commit aufspielen
 - Branch `tmp` löschen
- Aber: das kann GIT schon alleine
 - `git stash save`
 - Gewünschte Änderungen durchführen
 - `git stash pop`
 - Der Stash ist als Stack angelegt, man kann also beliebig viele Stashes generieren und wieder abrufen

Mit Patches arbeiten

- `git format-patch [--numbered] <revision-range>`
 - Gibt für jeden Commit, der im Bereich `<revision-range>` angegeben ist, einen Patch aus
 - Mit `--numbered` werden die Patches durchnummeriert
 - Beispiel: gebe einen Patch für die drei neuesten Commits aus:
 - `git format-patch --numbered HEAD~3..HEAD`
- `git apply <patch>`
 - Wendet Patch an, aber erzeugt keinen Commit
 - Vergleichbar zu `patch -p1 < <patch>`
- `git am <dir> | <patch> ...`
 - Wendet alle genannten Patches oder alle Patches im Verzeichnis `<dir>` an und erzeugt für jeden Patch einen Commit (`am` = apply multiple)

Speziellen Commit finden

- Falls irgendwann in der Vergangenheit ein Fehler eingebracht wurde, man aber nicht weiß wo, kann man diese Stelle suchen
 - Aktuelle Version ist “schlecht”: `git bisect bad`
 - Eine bekannte alte Version ist “gut”: `git bisect good <commit>`
 - GIT checkt nun eine Version in der Mitte aus
 - Man testet, ob diese Version gut oder schlecht ist und sagt entsprechend: `git bisect good` oder `git bisect bad`
 - GIT checkt eine neue Version aus, wieder in der Mitte der entsprechenden Hälfte
 - Das setzt man fort, bis die Stelle identifiziert ist, wo der Fehler reinkam
 - Anschließend bisect beenden: `git bisect reset`
 - Git checkt wieder die aktuelle Version von vor dem bisect aus
- Das Konzept ist auch allgemeiner anwendbar um beliebige Stellen zu finden
 - Statt good und bad kann man auch old und new sagen.

Arbeiten mit mehreren Worktrees

- `git worktree add <dir> [<branch>]`
 - Checkt alle Dateien (ggf. des angegebenen branches <branch>) erneut in das neue Verzeichnis <dir> aus
 - Es entsteht eine neue Sicht auf das Repository, ohne dass es geklont werden muss
 - Wird wie ein Branch gehandhabt
 - Dort kann beliebig eine andere Version ausgecheckt werden
- `git worktree list`
 - Listet alle aktiven Worktrees auf
- Wenn fertig, Directory <dir> einfach löschen
 - Referenzen werden mit der Zeit automatisch gelöscht
 - Von Hand: `git worktree prune`

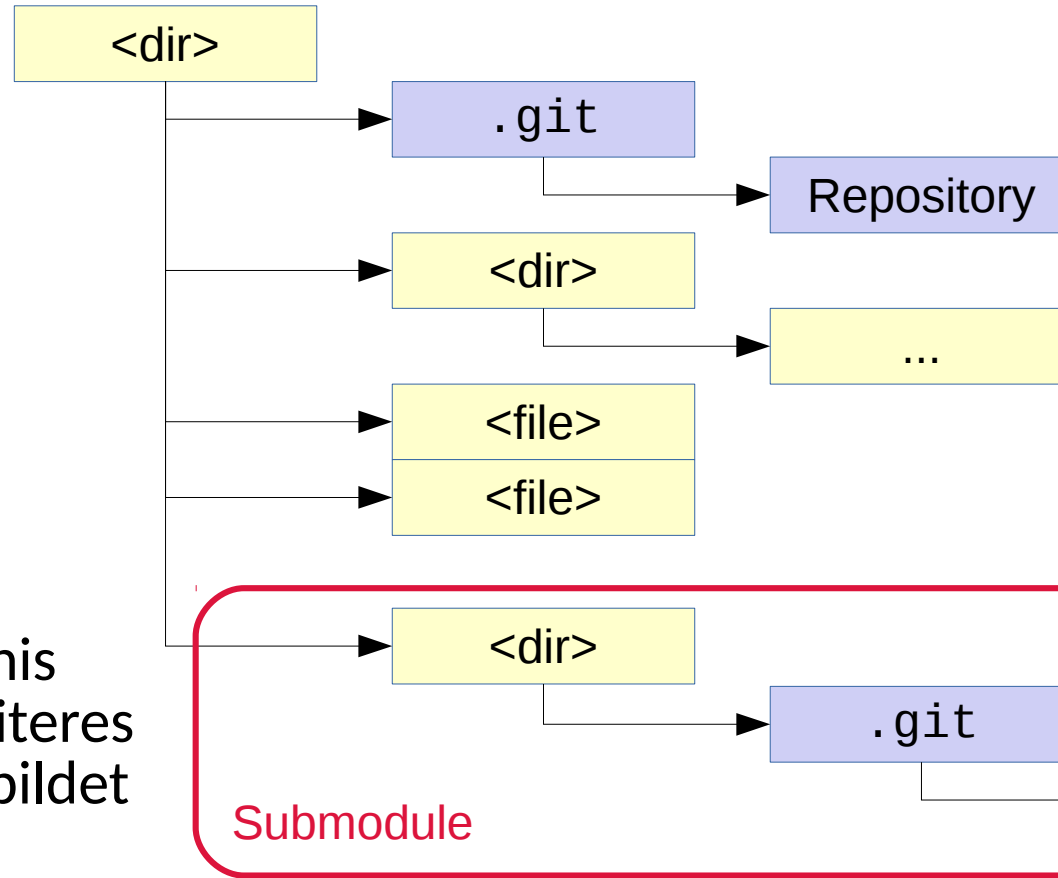
Weitere GIT-Funktionen (1)

- `git archive [--format=<fmt>] [--prefix=<prefix>/] [-o <file>]`
 - Erzeuge ein Archiv vom Typ <fmt> und schreibe es auf stdout (oder in die Datei <file>)
 - Allen Dateien wird <prefix>/ vorangestellt. Da dies i.a. ein Verzeichnis sein wird, muss man dem Präfix ein / anhängen
 - Als Formate sind tar (default) und zip möglich
 - Beispiele:
 - `git archive --prefix=linux-fsimx6/ | bzip2 -9 >fsimx6-B2019.03.tar.bz2`
 - `git archive -prefix=test/ master | tar -x -v -C../test`
- `git revert <commit>`
 - Es wird ein neuer Commit erzeugt, der die Änderungen des angegeben Commits rückgängig macht, also sozusagen ein inverser Commit
- `git blame <path>`
 - Zeigt für jede Zeile einer Datei, aus welchem Commit diese stammt und wer den Commit gemacht hat, also wer verantwortlich ist
- `git clean`
 - Lösche alle Dateien, die nicht von GIT erfasst sind.

Weitere GIT-Funktionen (2)

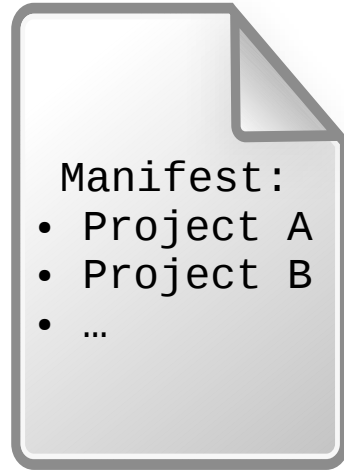
- `git grep <pattern>`
 - Suche <pattern> in allen Dateien des Worktrees, die GIT bekannt sind
- `git ls-files [--cached][--modified][--unmerged][--ignored][--deleted][--others] [<files>]`
 - Listet die angegebenen Files auf
- `git fsck`
 - Konsistenz des GIT-Repositories überprüfen
- `git gc`
 - Garbage Collector aufrufen, entfernt Dangling Commits, packt Objekte
- `git notes`
 - Anmerkungen zu Commits hinzufügen, ohne den Commit anzutasten
- `git count-objects`
 - Gibt Informationen zur Anzahl Objekte und deren Größe aus

Submodules

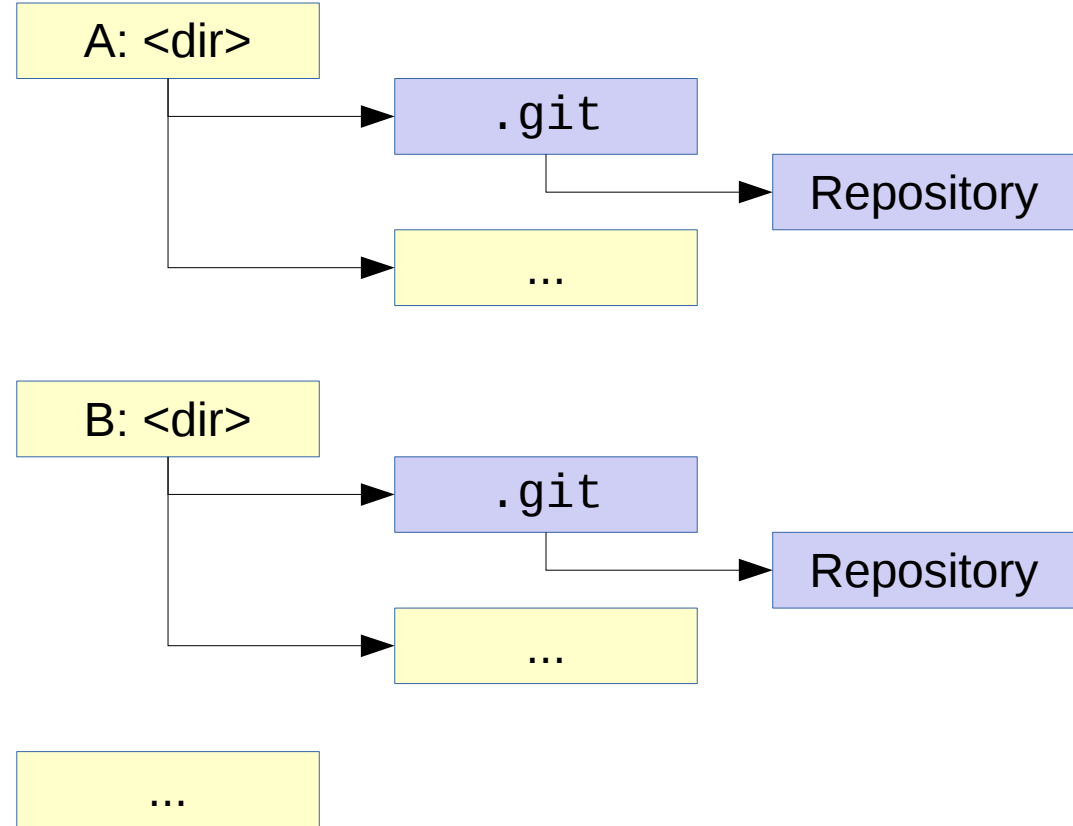


- Ein Unterverzeichnis wird durch ein weiteres GIT-Repository gebildet

Meta-GIT: repo



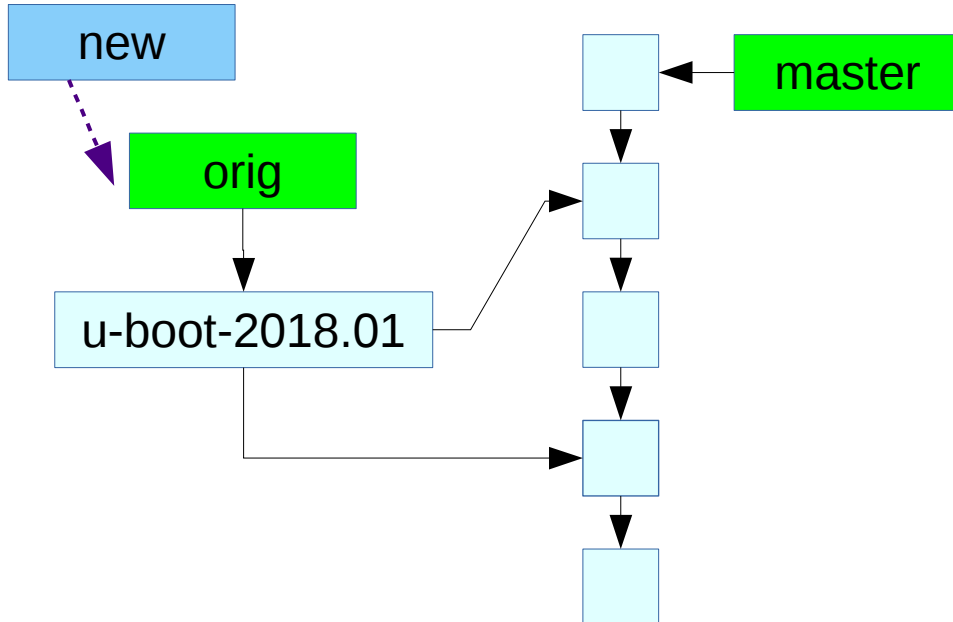
- Komplexe Systeme (z.B. Android) bestehen aus vielen Einzelprojekten; jedes Einzelprojekt ist ein GIT
- Mit repo werden viele solche GITs gemeinsam geklont
 - Ein Manifest definiert, welche Repositories enthalten sind
 - Manifest laden: `repo init <url>`
 - Alle GITs synchronisieren: `repo sync`



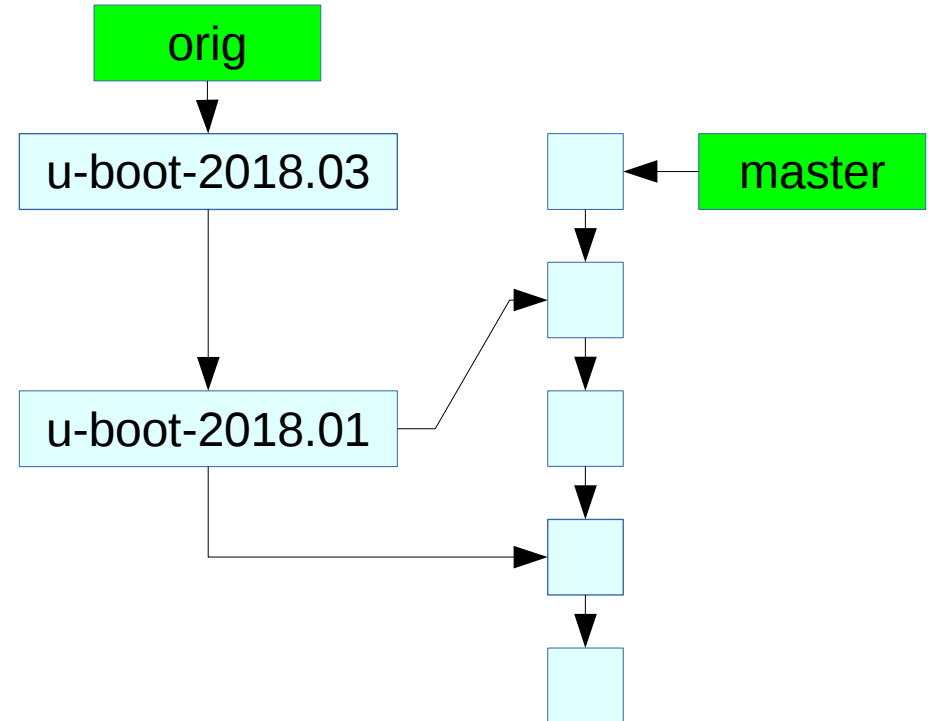
Komplexe Versionen einchecken

- Ausgangssituation

- u-boot-2018.03.tar.bz2
ausgepackt nach Verzeichnis new



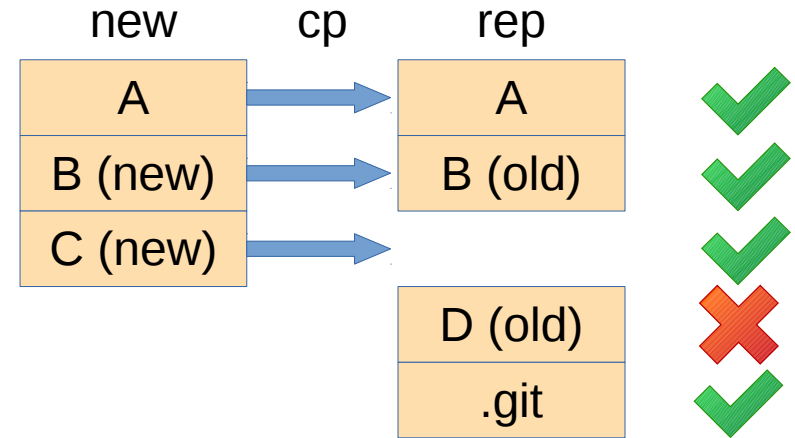
- Ziel



Problem

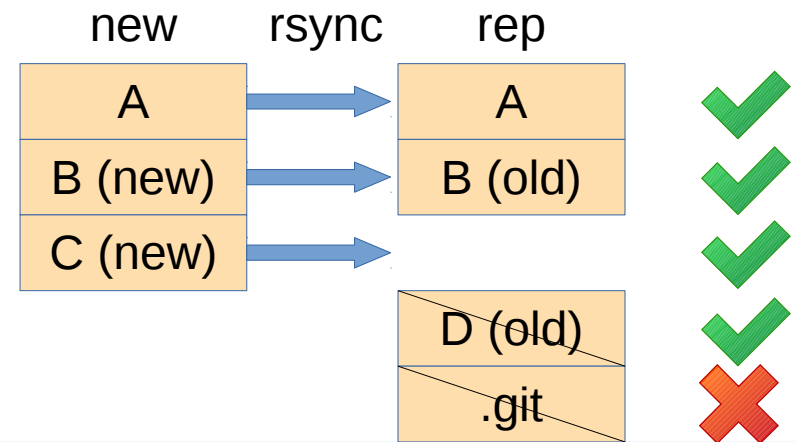
- Mit cp kopieren

- `git checkout orig`
- `cp -r ../new .`
- Files, die es in der neuen Version nicht mehr gibt, werden nicht gelöscht



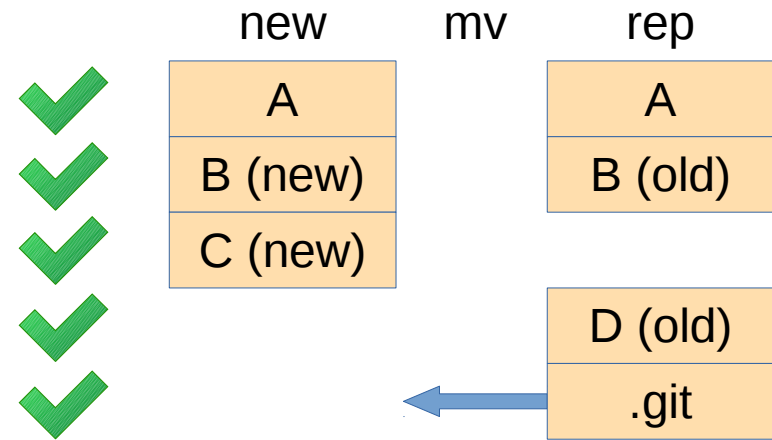
- Mit `rsync --delete` kopieren

- `git checkout orig`
- `rsync -a --delete ../new/ ./`
- Gefahr, dass `.git` gelöscht wird
 - Repository wäre dann weg → Fatal



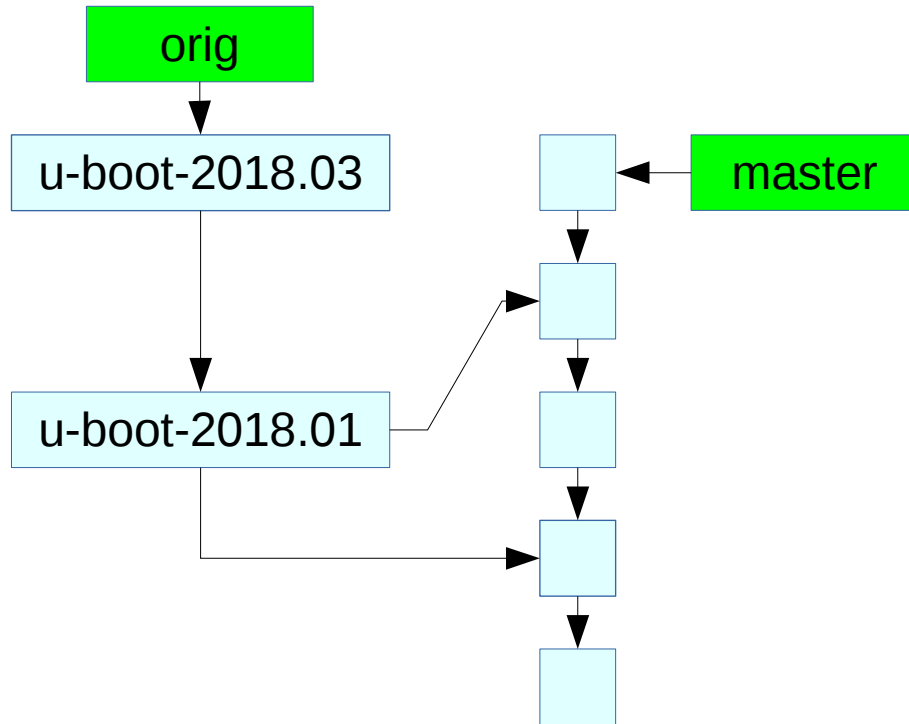
Idee

- Nicht die Dateien zum Repository bringen, sondern das Repository zu den Dateien
 - `git checkout orig`
 - `mv .git ../new`
 - `cd ../new`
 - `git add --all --force`
 - Alle Dateien einbinden, auch welche die ggf. in `.gitignore` stehen
 - `git commit`

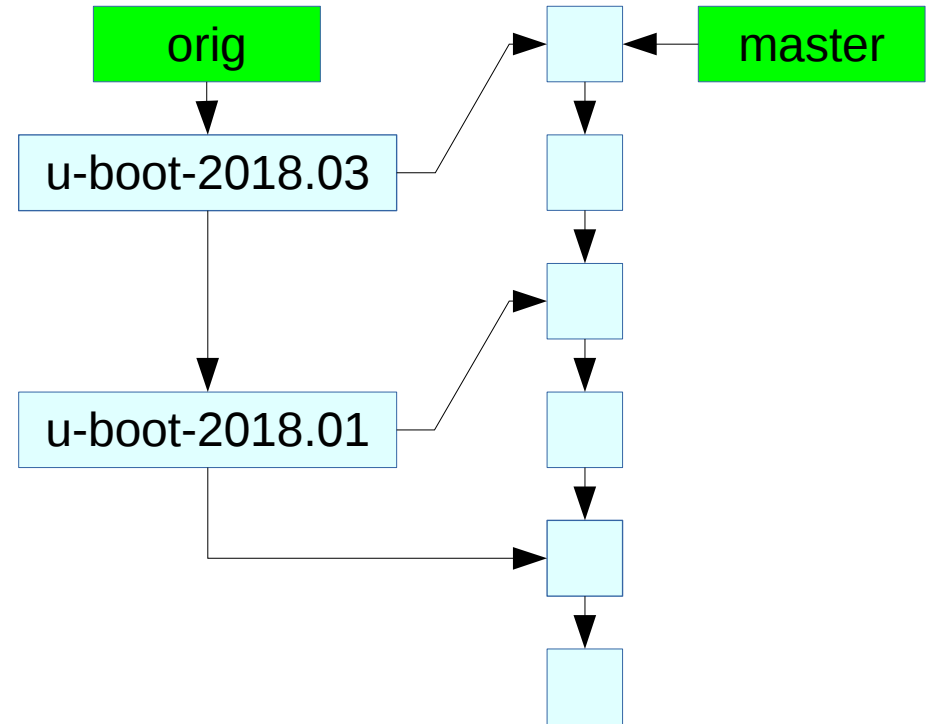


Komplexe Konflikte auflösen

- Ausgangssituation

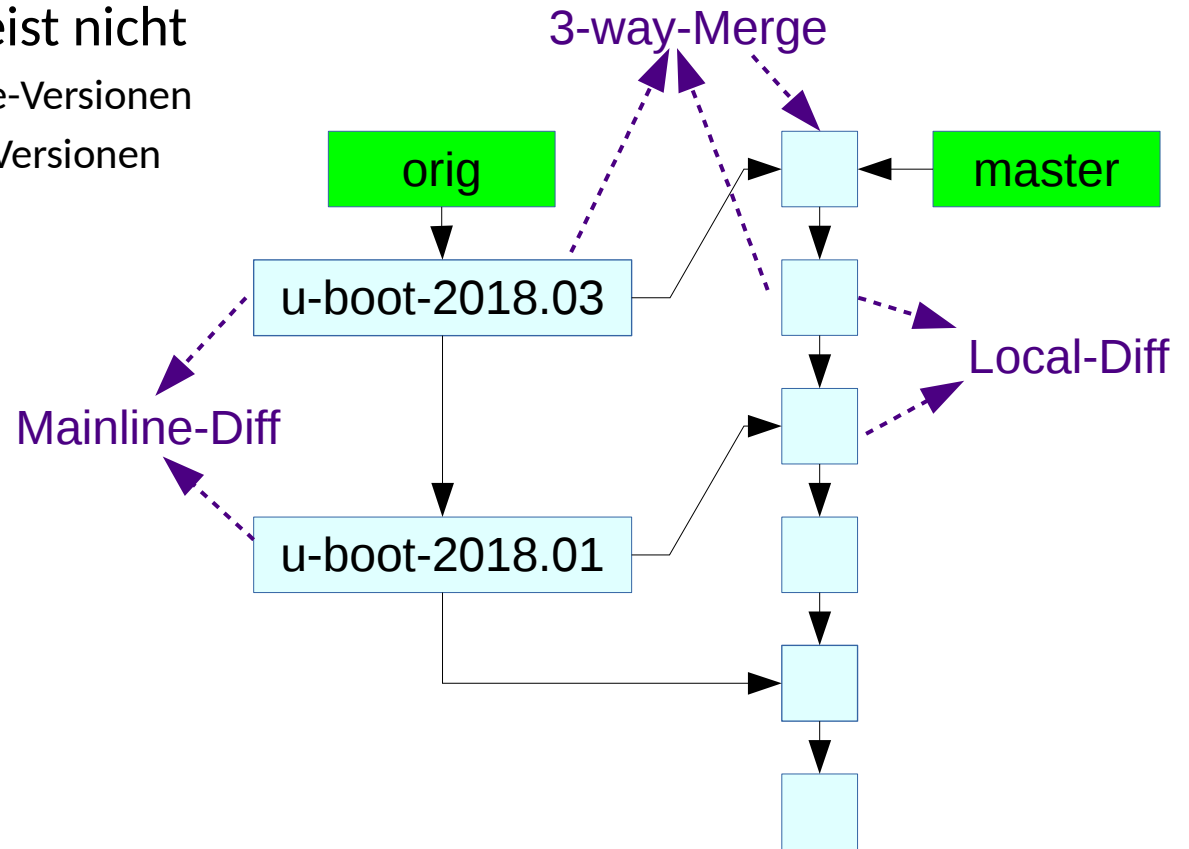


- Ziel



Mehrere Diffs vergleichen

- Der 3-fach-Merge reicht meist nicht
 - Zusätzlich Vergleich der Mainline-Versionen
 - Zusätzlich Vergleich der lokalen Versionen



Zum Abschluss ein paar Tipps

- Alles, was gepushed ist, ist tabu!
 - Kein umsordieren, kein Ändern der Commits auf dem Server
 - Falls Fehler enthalten → neuer Commit
 - Darum nicht zu schnell pushen, da sonst keine Korrekturen mehr möglich
 - Commits sammeln, erst dann pushen
- Eingeecheckte Commits sollten compilierbar sein!
 - Keine halben Commits
- Kein `git rebase --interactive` über Merge-Commits hinweg!
 - Sonst wird die komplette Historie des Merge-Commits erneut angebracht
- Keine Patch-Serien über Merge-Commits hinweg erstellen!
 - Merges funktionieren nicht als Patch
- Wer unsicher ist: vor komplexen Änderungen ein Backup des Repositories anlegen!
 - `tar jcvf <repo>.tar.bz2 <repo>`
- Miteinander reden!
 - Spart oft doppelte Arbeit



Hartmut Keller

keller@fs-net.de | +49 (0)711 / 123722-0