Arbeiten mit Git

Ingo Heimbach (i.heimbach@fz-juelich.de)

28. April 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung in Git

1.1 Was ist Git?

- **Verteiltes** Versionsverwaltungssystem
- Zur Verwaltung der Linux-Kernel-Sourcen entwickelt
- Engl. Ausdruck für Depp/Idiot, Grund für die Namensgebung:
  - Linus Torvalds: „I name all my projects after myself. First 'Linux', now 'Git'."
  - Kurz, gut auf einer Tastatur als Kommando zu tippen
  - In der Software-Welt bisher unbenutzt
- Von vielen bekannten Projekten genutzt, wie Android, Gnome, KDE, LibreOffice, Qt, VLC ...

1.2 Wieso Versionsverwaltung?

1.2.1 Vorteile für jeden einzelnen Entwickler

- **Sicherung** des Arbeitsstandes in Form **logischer Schritte**

⇒ **Rückkehr** zu älteren Entwicklungsständen jederzeit möglich

- Code eines Projektes kann einfach auf mehreren Rechnern **synchron** gehalten werden (→ `rsync` mit Extras)
- Rücksprung zur **letzten lauffähigen Version**, indem der aktuelle Arbeitsstand temporär aufgehoben wird (→ *Stashing*)

1.2.2 Vorteile für eine Gruppe von Entwicklern

- Leichter **Code-Austausch** zwischen allen Entwicklern
- **Arbeitsschritte** aller Entwickler **nachvollziehbar** (Logs)
- Verschiedene Entwicklungsweisen möglich:
  - **Isoliertes** Arbeiten an einem Feature (→ *Branching*)
  - **Zusammenarbeit** an einem Feature

- Zu den Entwicklungsweisen: Jeder Mitarbeiter setzt auf der vorhandenen stabilen Code-Basis auf und beginnt somit für jede Entwicklung einen neuen Entwicklungszweig.

⇒ Einzelne Entwicklungen sind daher immer automatisch gegen parallele Entwicklungen isoliert.

- Zusätzlich ist es möglich, seine eigene Entwicklung mit anderen zu teilen, um gemeinsam an einem Feature zu arbeiten.

1.3 Git Konzepte

1.3.1 Verteiltes Versionsverwaltungssystem

- Vollständige **Verteilung**

⇒ Jeder Entwickler hat lokale Kopie des gesamten *Repositories*

- **Repository**: Container mit gesamter Entwicklungsgeschichte

⇒ Arbeiten **ohne Netzwerkanbindung** möglich; Netzwerk nur zur Synchronisierung notwendig

- Es kann dennoch ein **zentrales** *Repository* existieren

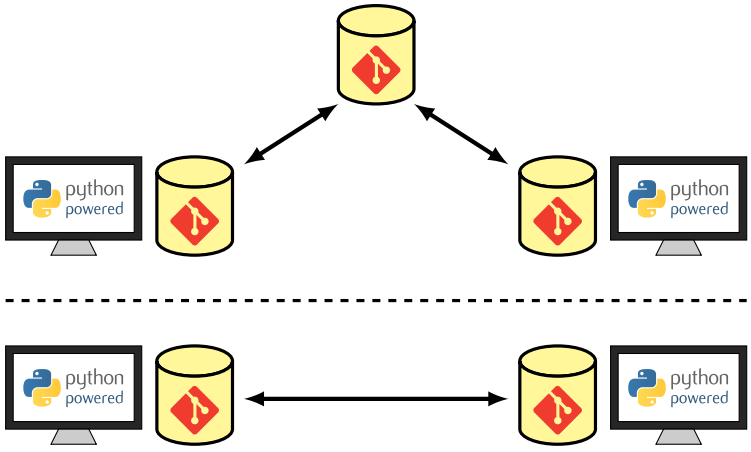


Abbildung 1.3.1: Zentrale vs. dezentrale Arbeitsweise

- Die obige Ansicht zeigt eine Konfiguration mit zentralem *Repository*; Daten werden nur über diesen zentralen Container ausgetauscht.

⇒ Vorteile: Zentraler Anlaufpunkt, ein *Repository* hat sicher immer die aktuellen Daten.

⇒ Nachteil: *Single point of failure*, sofern kein zweiter Backup-Server existiert.

- Alternativ können alle Entwickler direkt Daten austauschen, indem in regelmäßigen Abständen von allen Team-Mitgliedern neue Daten erfragt werden (*fetch*).

⇒ Vorteil: Kein zentraler Server notwendig.

⇒ Nachteil: Jeder Rechner muss für jeden anderen immer direkt erreichbar sein; keine definierte Code-Basis.

1.3.2 Snapshots statt Diffs

Konventioneller Ansatz:

- Konventionelle Versionierungssysteme sind **Datei-basiert**

⇒ **Revisionszähler** für jede Datei

⇒ Intern: Speicherung von **Diffs** zwischen Versionen einer Datei

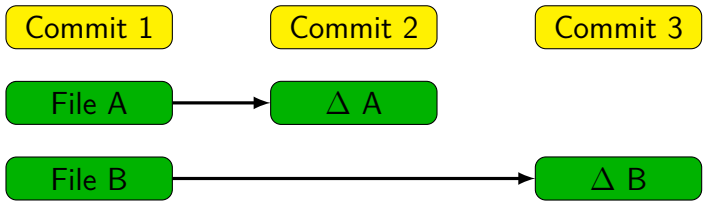


Abbildung 1.3.2: Diff-basierte Speicherung von *Commits*

- CVS, Subversion und ähnliche werden hier als konventionelle Systeme angesehen.
- Die Revisionszähler werden bei jedem *Commit* hochgezählt, sofern die Datei verändert wurde, um jede Dateiversion eindeutig identifizieren zu können.

Ansatz bei Git:

- Gesamtes Projekt als Belegung eines **Dateisystems** ansehen
- Speichern des Arbeitsstandes führt zum Abbild der aktuellen Dateisystembelegung als Ganzes (→ **Commit**)

⇒ Führt zur Vereinfachung anderer Konzepte (s. *Branching*)

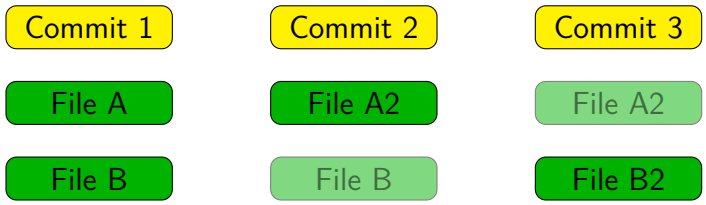


Abbildung 1.3.3: *Commits* als Dateisystem-Snapshots

- Da Git bei jedem *Commit* alle Dateien als Ganzes speichert, entsteht zunächst der Eindruck, das bei kleinen Änderungen viel Speicherplatz verschwendet würde. Dies ist aber nicht der Fall. Git fasst ähnliche Dateien automatisch zusammen und speichert deren Unterschiede über *Diffs* (→ *packfiles*).

1.3.3 Hashes

- Jedes Objekt (Datei, *Commit* usw.) erhält einen **Hashwert**

⇒ Direkte **Referenzierung** einzelner Objekte über Hash möglich

- Jede Projektänderung führt zur Änderung der Hash-Werte

⇒ Git erkennt sofort, dass und **wie** Änderungen vorgenommen wurden (z. B. Dateiverschiebungen)

1.3.4 Nicht-lineare Entwicklung

- Für jedes Feature ein eigener, **isolierter** Zweig möglich

⇒ Features, die parallel entwickelt werden, behindern sich nicht

⇒ Releases und Entwicklungsversionen können voneinander getrennt werden

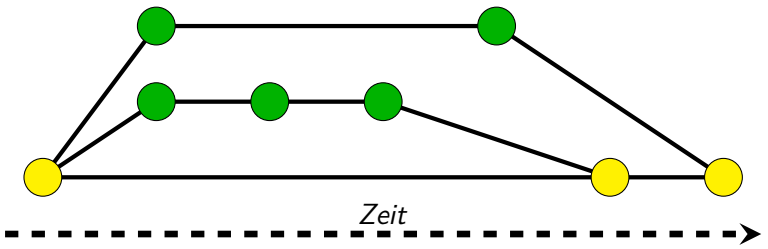


Abbildung 1.3.4: Nicht-lineare Entwicklung mittels parallelen Zweigen

1.3.5 Volle Kontrolle

- Git fügt nur Daten zu einem Repository hinzu; alte Versionen bleiben erhalten
- Über `force`-Parameter kann **Löschen** von Daten **erzwungen** werden
- Gelöschte Objekte **verweilen** noch eine Zeit **im Repository**, bevor sie endgültig entfernt werden

⇒ Können über ihren Hash weiterhin adressiert werden (s. `reflog`)

- Das Löschen von Daten kann sinnvoll sein, um z. B. unvollständige oder fehlerhafte *Commits* durch verbesserte *Commits* zu ersetzen.

1.3.6 *Staging Area*

- **Staging Area**: Bereich mit den Daten, die in den nächsten *Commit* aufgenommen werden

⇒ Geänderte / Neue Dateien müssen zunächst zur *Staging Area* hinzugefügt werden, bevor sie in einen *Commit* wandern

- Änderungen an bereits zur *Staging Area* hinzugefügten Dateien werden **nicht** automatisch übernommen

⇒ Zusätzliche Komplexität, ermöglicht aber, nur **Teiländerungen** zu **committen**

- Komplexität kann fast vollständig eliminiert werden (→ `git commit -a`)

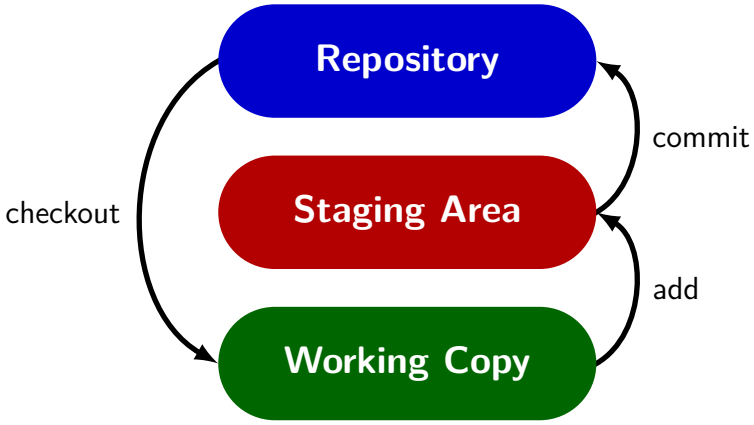


Abbildung 1.3.5: *Stages* der Datenverwaltung in Git

1.4 Git Basics

1.4.1 Neues Git-*Repository* anlegen

- Lokales *Repository* ohne zentralen Server anlegen:

```
cd <zu verwaltendes Verzeichnis>
git init
```

  - Legt `.git`-Unterverzeichnis an

  ⇒ Umwandlung eines bestehenden, u. U. belegten Verzeichnisses in ein Git-*Repository*

- Das `.git`-Verzeichnis enthält alle für Git relevanten Information wie die Datenbank aller erstellten *Commits* und die lokale Konfiguration des *Repositories*.

- Mit `git init` werden lediglich alle Vorbereitungen getroffen, um anschließend mit weiteren Git-Befehlen arbeiten zu können.

⇒ Die enthaltenen Dateien stehen noch nicht automatisch unter Versionskontrolle! (s. `git add`)

- Zentrales *Repository* erstellen:

```
mkdir <Repository-Name>.git
cd <Repository-Name>.git
git init --bare
```

  - Erstellung **immer** mit `--bare`
  - ⇒ *Repository* hat keine eigene **Working Copy**
  - Zwingend erforderlich, um in das *Repository pushen* zu können
  - Anschließend muss eine lokale Kopie geklont werden

- Die Option `--bare` bewirkt, dass das *Repository* ausschließlich aus dem `.git`-Verzeichnis besteht (→ keine *Working Copy*).

- „Pushen“ beschreibt in Git-Terminologie den Vorgang, den lokalen Stand in das zentrale *Repository* zu übertragen.

1.4.2 Zentrales *Repository* klonen

- **Klonen**:
  - **Vollständige**, lokale Kopie eines anderen Repositories erstellen

```
git clone <User>@<Host>:<Repository-Pfad> [Verzeichnis-Name]
```

  - Verbindet sich per Default über **ssh** (*read+write* mit User-ID)
  - Alternative: git-Protokoll über speziellen Dienst (*read only*, keine Authentifizierung)

- Beispiel:

```
git clone heimbach@ifflinux:git_vortrag.git
```

- Um mit Git arbeiten zu können, ist es immer notwendig, entweder ein neues *Repository* zu erstellen oder ein bereits existierendes *Repository* über `git clone` lokal zu kopieren. Reine Arbeitskopien, wie z. B. bei SVN, können nicht angelegt werden.

- Neben dem ssh- und git-Protokoll unterstützt Git noch das lokale file- und das http/s-Protokoll. Das file-Protokoll kann interessant sein, wenn lokal oder über NFS geklont werden soll. ssh sollte http, wenn möglich, aufgrund der Geschwindigkeit immer vorgezogen werden.

- Auf einem Server können mehrere Protokolle parallel angeboten werden, z. B. ssh für Mitarbeiter und git für einen öffentlichen Lesezugriff für Interessenten eines Projektes.
- Für den ssh-Zugriff wird oftmals ein spezieller Benutzer „git“ auf dem zentralen Server angelegt, über den **alle** Mitarbeiter auf das Repository (auch schreibend) zugreifen können. Wer nun tatächlich zugreift, kann über RSA-Schlüssel überprüft werden. Programme wie `Gitosis` oder `gitolite` vereinfachen die Einrichtung und Verwaltung dieses Ansatzes.

1.4.3 Zentrales *Repository* initialisieren

- Einige Git-Kommandos erfordern Vorhandensein min. eines *Commits*

⇒ Ist das geklonte *Repository* noch vollständig unbeschrieben, so sollte es initialisiert werden:

```
touch .gitignore
git add .gitignore
git commit -m "Initial commit"
git push -u origin master
```

- Alle gezeigten Git-Befehle werden an späterer Stelle noch ausführlich eingeführt.
- Für den ersten *Commit* bietet es sich an, eine `.gitignore`-Datei anzulegen, da diese Datei in jedem Projekt gepflegt werden sollte (Liste der von der Versionierung ausgeschlossenen Dateien).
- Wurde der erste *Commit* erstellt, so liegt ein initialisierter Hauptzweig (`master`) vor. Diese Initialisierung sollte daraufhin in das *Repository* übertragen werden, von dem geklont wurde (`origin`). Das `-u` Flag sorgt zusätzlich dafür, dass Git beide Zweige zu *Tracking Branches* verbindet.

1.4.4 Dateien/Verzeichnisse unter Versionsverwaltung stellen

- Dateien müssen explizit unter Git-Verwaltung gestellt werden:

```
git add <Datei/Verzeichnis>
```

⇒ Aktuellen Stand zur **Staging Area** hinzufügen (→ für den nächsten *Commit* vormerken)

- `git add` auf Verzeichnisse wirkt **rekursiv**
- Problem: Git arbeitet Datei-basiert
  - ⇒ Leere Verzeichnisse können nicht verwaltet werden!
  - Workaround: Leere `.gitignore` im Verzeichnis anlegen

1.4.5 Zustand als *Commit* sichern

- Inhalt der *Staging Area* sichern:

```
git commit
```

  - Anschließend Abfrage einer *Commit-Message*
  - Alternativ: Angabe der Option `-m "<Message>"`

⇒ Vor **jedem** *Commit* muss **jede** zu sichernde Datei erneut mit `git add` zur *Staging Area* hinzugefügt werden

- Abgekürztes Verfahren:

```
git commit -a
```

  - Jede jemals mit `git add` hinzugefügte Datei wird in den *Commit* aufgenommen

- Die eingegebene *Commit-Message* darf **nicht leer** sein, sonst wird die Erstellung des *Commits* abgebrochen.

- Der Befehl `git commit -a` ermöglicht einen SVN-ähnlichen Arbeitsfluss:
  1. Dateien mit `git add` einmalig hinzufügen
  2. Änderungen über einen Aufruf von `git commit -a` regelmäßig in das *Repository* übertragen

1.4.6 Verwaltete Dateien löschen/verschieben

- Git-Befehle zum **Löschen**/**Verschieben** sind identisch zu den entsprechenden **Unix-Befehlen**

- Datei/Verzeichnis **löschen**:

```
git rm <Datei>
git rm -r <Verzeichnis>
```

- Datei/Verzeichnis **verschieben**:

```
git mv <Datei/Verzeichnis>
```

- Änderungen werden in der *Staging Area* vermerkt

- Standard `rm` / `mv` auch verwendbar, da Git Änderungen nachträglich meist richtig zuordnen kann (→ *Hashing*)

1.4.7 Aktuellen *Repository*-Status abfragen

```
git status
```

- Gibt Informationen zum Zustand der eigenen Arbeitskopie aus:
  - Dateien, die nicht von Git verwaltet werden
  - Gegenüber dem letzten *Commit* modifizierte Dateien
  - Dateien, die sich in der *Staging Area* befinden
  - Synchronisierungsstand mit *Remote-Repositories*

⇒ Sollte vor **jedem** *Commit* aufgerufen werden, damit kein `git add` vergessen werden kann!

1.4.8 *History* ab aktuellem *Commit* betrachten

- ***History*** ab dem aktuellen *Commit* anzeigen:

```
git log
```

- Listet alle *Commits* in der Form

```
commit 7b248e002e56a1ed23d32f317e6a1eace1917b4e
Author: Ingo Heimbach <i.heimbach@fz-juelich.de>
Date:   Thu Mar 20 10:48:25 2014 +0100

    Hallo Welt hinzugefuegt
```

- `--graph` zeigt zusätzlich Verzweigungen als **ASCII-Art**
- `--oneline` gibt jeden *Commit* kompakt in nur einer Zeile aus
- Beispiel für `git log --graph --oneline`:

```
* b895d43 Merge branch 'develop'
|\
| * 1afabc4 Merge branch 'feature-hallo_welt' into develop
| |\
| | * fc78310 Hallo Welt mit persoenlicher Begruessung!
| | * 99c432f Hallo Welt hinzugefuegt
| |/
| * 89d2759 .gitignore angepasst
|/
* 63cfced Initial commit
```

1.4.9 Dateien/Verzeichnisse ignorieren

- Manche Dateien/Dateitypen sollten **nicht versioniert** werden
  - Compiler-Output (`*.o`, `*.so`, `main`)
  - Temporärer LaTeX-Output (`*.aux`, `*.log`, `*.lof` usw.)
  - ...
- Probleme:
  - `git add` auf ein Verzeichnis würde diese Dateien hinzufügen
  - `git status` listet alle temporären Dateien immer auf

  ⇒ Wird unbrauchbar durch die Masse
- Lösung: Pflege einer **`.gitignore`-Datei**
  - Liste von ignorierten Dateien/Verzeichnissen
- Syntax an regulären Ausdrücken der Bash angelehnt:

```
# Comment
*.so
*.[oa]     # *.o und *.a ignorieren
!libgr.so    # nicht ignorieren
build/     # Verzeichnis muss mit / beendet werden
/manual/*.pdf    # pdf nur in manual/ ignorieren
```

- Nachträgliches Ignorieren von Dateien:

```
git rm --cached <Datei>
```

- Das **Versionieren** ignorierter Dateien kann mit `--force` (Kurzform: `-f`) **erzwungen** werden:

```
git add -f <Datei>
```

- Die Datei `.gitignore` wird typischerweise im obersten Projektverzeichnis (neben `.git`) angelegt.
- Sie gilt rekursiv für alle Verzeichnisse (nur Einträge, die mit `/` starten, sind auf das Verzeichnis beschränkt, in dem die `.gitignore`) liegt.
- Jedes Verzeichnis kann eine eigene `.gitignore` beinhalten, aber Regeln aus Eltern-Verzeichnissen gelten dennoch weiterhin. Es können also ausschließlich Regeln ergänzt bzw. überschrieben werden.
- Leere `.gitignore`-Dateien werden gerne verwendet, um leere Verzeichnisse zu einem Git-*Repository* hinzufügen zu können (Git kann nur Dateien verwalten). Als Alternative hat sich die Verwendung von leeren `.gitkeep`-Dateien durchgesetzt, die keine besondere Bedeutung in Git haben und nur der Erhaltung von Verzeichnissen dienen.

1.4.10 *Branching*

- Behandelt die **Verzweigung des Arbeitsflusses** in parallele Pfade und deren anschließendes Zusammenführen

⇒ **Nicht-lineare Entwicklung** mit isolierten Arbeitsumgebungen

- Kann zu einer unübersichtlichen Entwicklungs-*History* führen
- Die *History* kann jedoch **nachträglich linearisiert** werden (→ `git rebase`)
- Erzeugt zusätzlichen **Overhead** bei der Entwicklung, für die Vorteile aber vollkommen vertretbar
- *Branching* sollte in den normalen **Arbeitsfluss integriert** sein
- Viele Versionierungssysteme sehen *Branching* als Methode, um mögliche Alternativen für eine Programmentwicklung zu erproben. Aufgrund des seltenen Einsatzes ist *Branching* daher mit solchen Systemen meist aufwändig.
- Git ist hingegen so konzipiert worden, dass *Branching* schnell und einfach durchzuführen ist. Somit können die Vorteile von *Branching* (Isolation von anderen Änderungen und Abbilden von parallelen Entwicklungen in der *History*) mit nur wenig Overhead dauerhaft genutzt werden.

1.4.10.1 Einige Interna zum besseren Verständnis

- Git speichert *Commits* als einfach verkettete Liste:

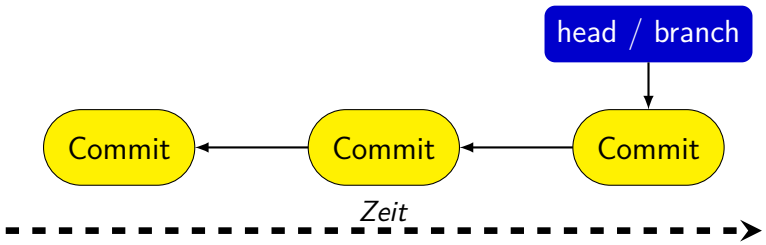


Abbildung 1.4.1: Interne Speicherung von *Commits* als einfach verkettete Liste

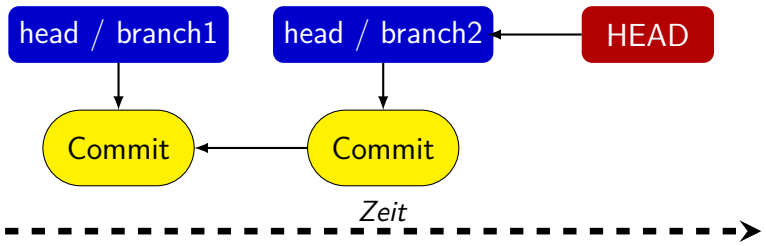
- Zeigerrichtung ist **entgegen** der Zeitrichtung

⇒ *Commits* kennen ihre Vergangenheit, aber nicht ihre Zukunft

- Interpretation des Listenanfangs (`head`-Zeiger) als *Branch*
- Beliebige Anzahl von *Branches* in Git möglich
- Ein Zweig kann als **aktiver** gesetzt werden

- Ein neu erstellter *Commit* zeigt auf den letzten `head`-*Commit* des aktiven Zweiges
- *Branch*-Zeiger wird danach auf neuen *Commit* gesetzt
- Aktiver Zweig wird mit **`HEAD`** referenziert

i)



ii)

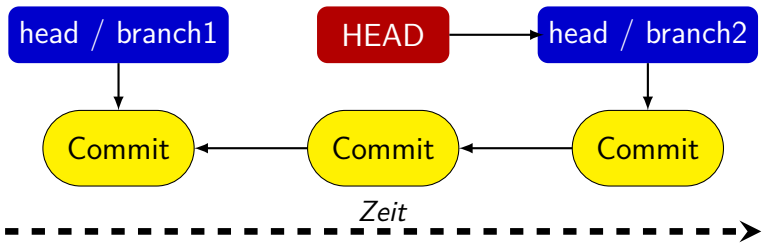
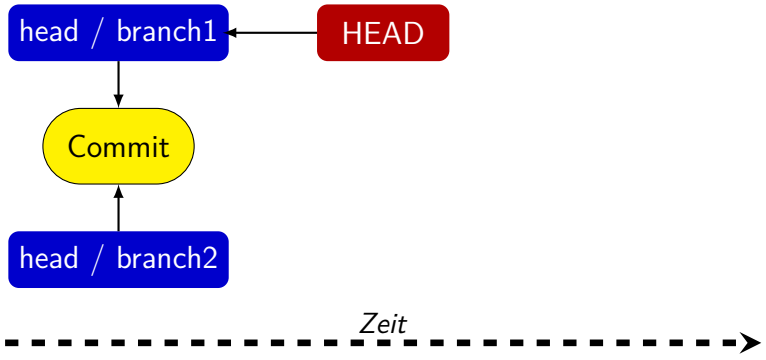


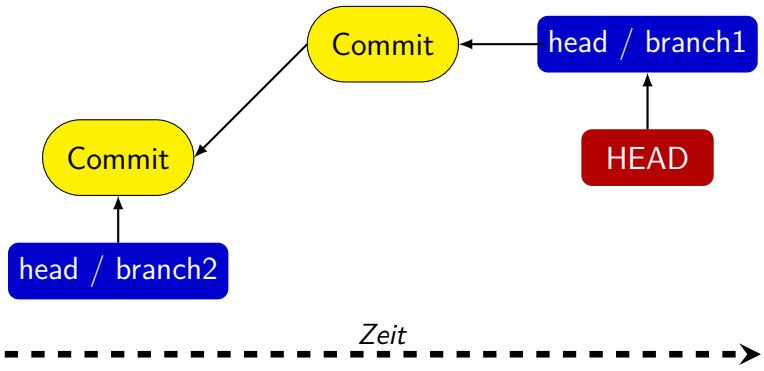
Abbildung 1.4.2: Einfügen eines neuen *Commits* in die verkettete Liste

- Die Nutzung mehrerer *Branches* ermöglicht somit eine **Verzweigung der Liste**

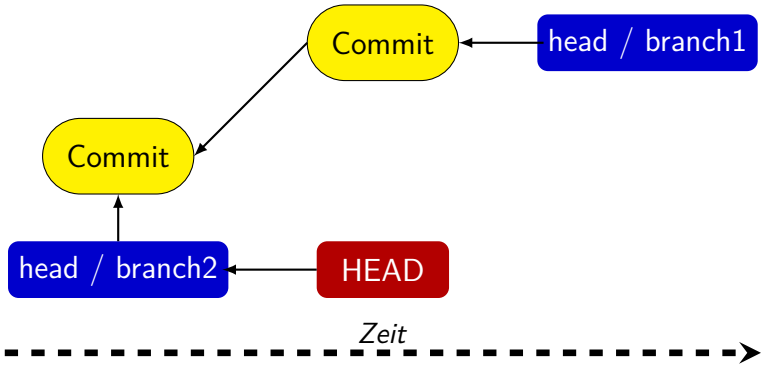
i)



ii)



iii)



iv)

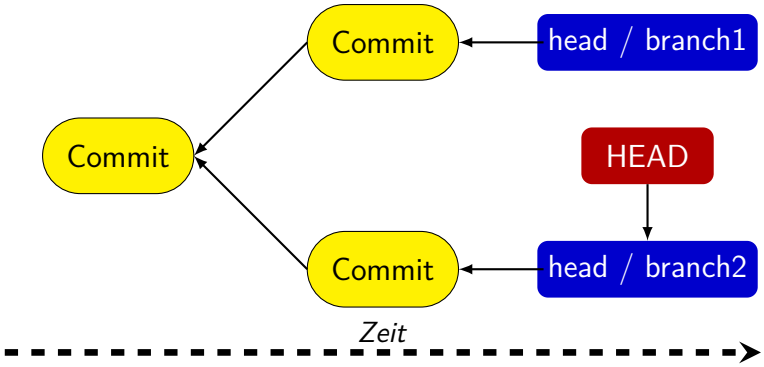
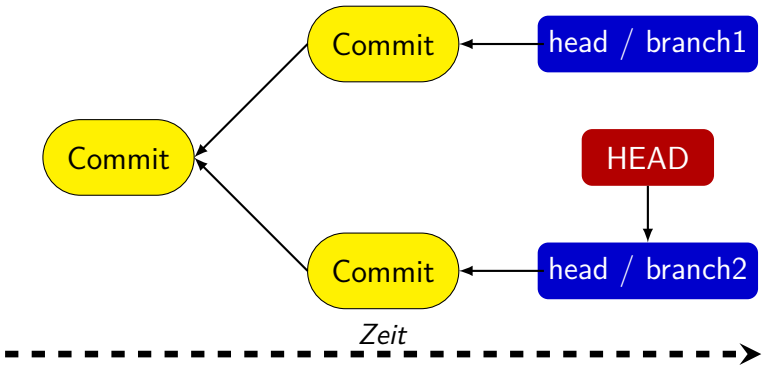


Abbildung 1.4.3: Verwendung mehrerer Zweige über eine nicht-lineare Listenentwicklung

- *Commits* können **mehrere Vorgänger** haben

⇒ Zusammenführen von Verzweigungen möglich

i)



ii)

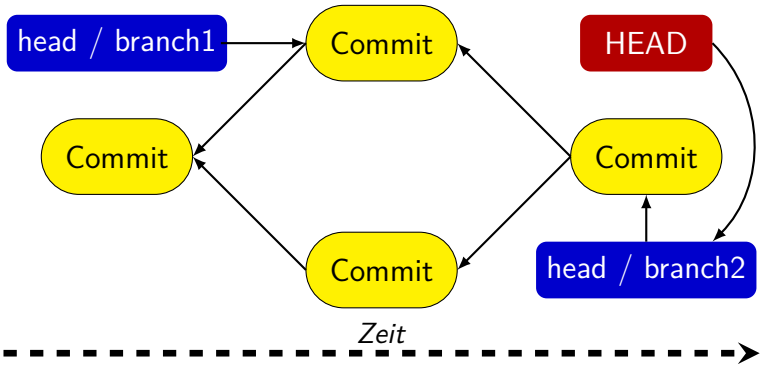
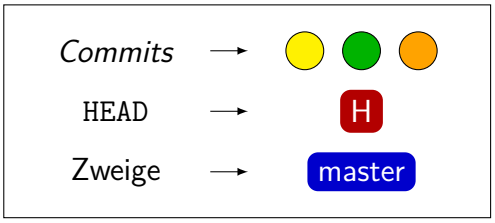


Abbildung 1.4.4: Zusammenführen von Zweigen über *Commits* mit mehreren Vorgängern

- *Commits* mit mehreren Vorgängern werden als **Merge-Commits** bezeichnet.
- Ab jetzt wird eine abgekürzte Darstellungsweise verwendet:



1.4.10.2 Zweig erstellen

- `master`: Default-Zweig, aktiv nach `git init/clone`
- Erstellen eines **neuen Zweiges**:

```
git branch <Zweigname>
```

- Der Zweig zeigt auf den *Commit*, auf den auch `HEAD` zeigt

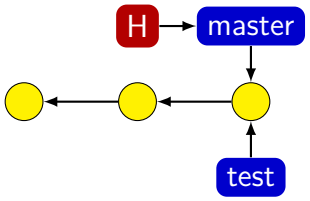


Abbildung 1.4.5: Erstellung von Zweigen

- Auflisten aller schon vorhandenen Zweige über

```
git branch -a
```

- Ohne `-a` werden nur **lokale** Zweige aufgelistet
- Löschen eines Zweiges:

```
git branch -d <Zweigname>
```

  - Aktion nicht möglich bei resultierendem Datenverlust
  - Datenverlust kann mit `-D` erzwungen werden

- Beim Löschen eines Zweiges wird nur der entsprechende `head`-Zeiger entfernt (keine *Commits* werden gelöscht!).

⇒ Datenverlust tritt also nur dann auf, wenn *Commits* durch das Entfernen des Zweiges über keinen anderen Zeiger mehr erreichbar sind.

- Unerreichbare *Commits* werden nicht sofort gelöscht. Stattdessen prüft Git in regelmäßigen Abständen (je nach ausgeführten Befehlen), welche Objekte nicht mehr referenziert werden und löscht diese (**Garbage Collector**). Standardmäßig werden dabei nur Objekte einbezogen, die mindestens seit zwei Wochen nicht mehr referenzierbar sind.
- Die zeitliche Einschränkung von zwei Wochen wird verwendet, da selbst unerreichbare *Commits* nachträglich wiederhergestellt werden können, solange sie nicht durch den *Garbage Collector* gelöscht wurden (s. `git reflog`).

- Der *Garbage Collector* kann mit

```
git gc
```

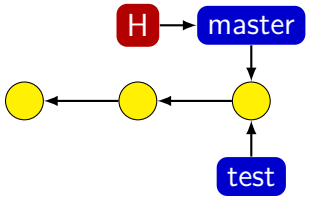
manuell gestartet werden.

1.4.10.3 Aktiven Zweig setzen

- Auf einen Zweig wechseln (→ `HEAD` setzen):

```
git checkout <Zweigname>
```

i)



ii)

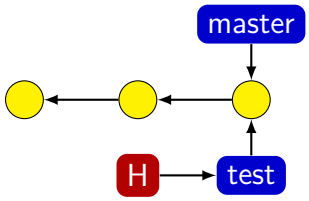


Abbildung 1.4.6: Auschecken eines Zweiges führt zur Umsetzung des `HEAD`-Zeigers

- Zweig erstellen und direkt als aktiv setzen:

```
git checkout -b <Zweigname>
```

- Da man in der Praxis meist sofort auf einem neu erstellten Zweig arbeiten möchte, wird `git checkout -b` üblicherweise wesentlich häufiger als `git branch` verwendet.

- Statt `-b` kann auch `-B` als Option übergeben werden und veranlasst so, dass ein schon existenter Zweig überschrieben wird. Großbuchstaben dienen in Git allgemein dazu, Aktionen zu veranlassen, die mit Datenverlust einhergehen.

1.4.10.4 *Detached HEAD*

- Statt auf einen Zweig kann `HEAD` auch direkt auf einen **beliebigen** ***Commit*** gesetzt werden
- Auswahl eines *Commits* möglich über:
  - Hash-Wert
  - relativ zu Zeigern (z. B. Zweigen)
  - über Logbuch

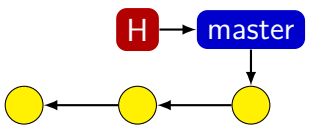
⇒ Entwicklungen auf Basis alter *Commits* möglich

- Beispiele:

```
git checkout c00cfe   # Teil-Hash moeglich
git checkout master~    # Commit vor master
git checkout HEAD~3 # 3 Commits vor HEAD
git checkout master^2 # 2. Eltern-Commit
git checkout master@{5} # master vor 5 Commits
```

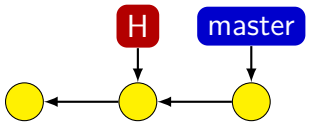
- Der Hash-Wert eines *Commits* kann beispielsweise über die *History*-Ansicht (s. `git log`) ermittelt werden.
- Mögliche relative Angaben:
  - `~`: 1. Eltern *Commit*; eine nachgestellte Zahl gibt an, wie viele *Commits* man zurückverfolgen möchte.
  - `^`: Auswahl des Eltern-Teils über nachgestellte Zahl (für *Merge-Commits*).
  - Relative Angaben können kombiniert werden (z. B. `~^2`: einen *Commit* zurückgehen und dann den zweiten Eltern-*Commit* auswählen).
- Logbuch der `head`-Zeiger über `<Zweig>@{`n`}`:
  - Referenziert den *Commit*, auf den der Zweig vor n Wechseln gestanden hat.
  - Dies entspricht **keiner** relativen Angabe, da Zweige beliebig umgesetzt werden können (s. `git reset`).
- Praktisches Beispiel:

  i)



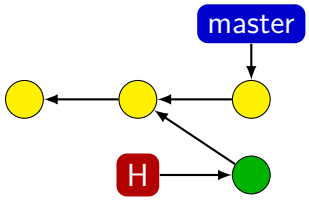
ii)

```
git checkout master~    # Commit vor master
```



iii)

```
git checkout master~    # Commit vor master
git commit
```



iv)

```
git checkout master~    # Commit vor master
git commit
git branch dev    # Commit sichern
```

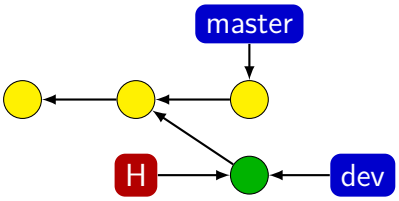


Abbildung 1.4.7: Beispiel für *Detached HEAD*

- Statt nachträglich einen Zweig zu erstellen, hätte man vor `git commit` alternativ ein `git checkout -b dev` ausführen können.
- Wichtig ist nur, dass irgendwann ein Zweig erstellt wird, der die neuen *Commits* referenziert, bevor `HEAD` auf einen anderen *Commit* oder Zweig umgesetzt wird. Ansonsten wären die neu erstellten *Commits* nicht mehr erreichbar.

1.4.11 Zweige zusammenführen

- Git kennt zwei Arten, Zweige zusammenzuführen:
  - **Merging**:
    - Verzweigungen bleiben in der *History* erhalten
    - Erstellung eines *Merge-Commits*, falls notwendig
  - **Rebasing**:
    - Erstellt aus einem Zweig eine Art Diff-Patch und wendet diesen auf einen zweiten Zweig an

    ⇒ Linearisierung der *History*

- *Merging* behält den tatsächlichen Entwicklungsverlauf bei und führt die Änderungen lediglich über einen neuen *Commit* zusammen (falls nötig).
- Durch *Rebasing* erscheint es in der History so, als wären Entwicklungen nacheinander ausgeführt worden, obwohl dies nicht der Fall ist. Aufgrund dieser Linearisierung ist die Entwicklungsgeschichte allerdings übersichtlicher.
- *Rebasing* entspricht dem Vorgang, den SVN bei einem `svn update` ausführt.
- Für Anfänger ist *Merging* meist einfacher zu handhaben als *Rebasing*, da ein `git rebase` zu einer dauerhaften Modifikation der Entwicklungsgeschichte führt, die ungewollte Nebeneffekte haben kann.

1.4.11.1 *Merging*

- Angegebenen Zweig mit dem aktiven Zweig zusammenführen:

```
git merge <Zweig-Name>
```

- Git entscheidet automatisch über die *Merge*-Strategie:
  - **Fast-Forward-Merge**:
    - Wird verwendet, wenn der aktive Zweig in der Vergangenheit des angegebenen Zweigs vollständig enthalten ist

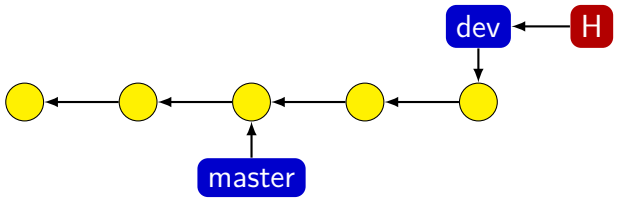
    ⇒ Setze aktiven Zweig auf die Position des angegebenen vor
  - **Three-Way-Merge**:
    - Wird ansonsten eingesetzt

    ⇒ Findet die gemeinsame Wurzel und macht darüber einen Abgleich
- Ist der aktive Zweig im anderen Zweig enthalten, so reicht es logischerweise aus, nur den Zeiger vorzusetzen, um beide Zweige zusammenzuführen (*Fast-Forward-Merge*).

- Der *Three-Way-Merge* wird dann angewendet, wenn tatsächlich parallele Entwicklungen vorliegen:
  - Erzeugt **immer** einen *Merge-Commit*.
  - Gleicht die Änderungen beider Zweige zum letzten gemeinsamen *Commit* ab und wendet die Änderungen beider Zweige im *Merge-Commit* an.

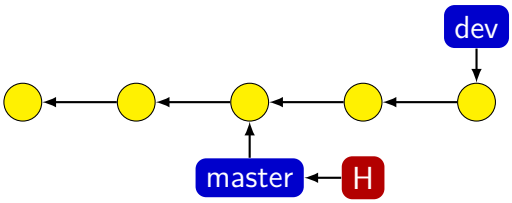
- Beispiel (*Fast-Forward-Merge*):

  i)



  ii)

```
git checkout master
```



  iii)

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

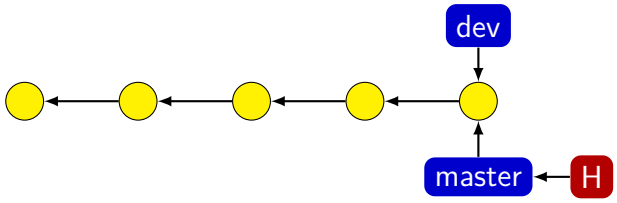
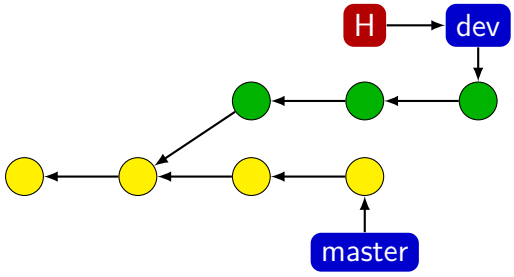


Abbildung 1.4.8: Beispiel eines *Fast-Forward-Merges*

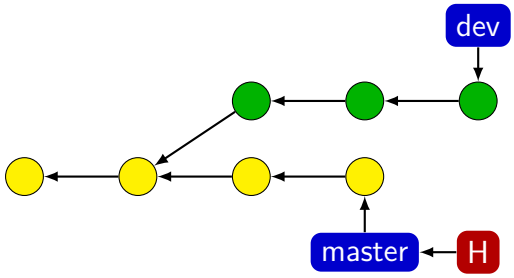
- Beispiel (*Three-Way-Merge*):

i)



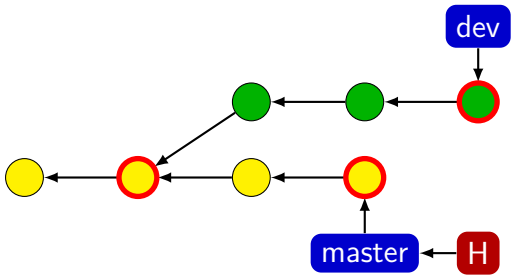
ii)

```
git checkout master
```



iii)

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



iv)

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

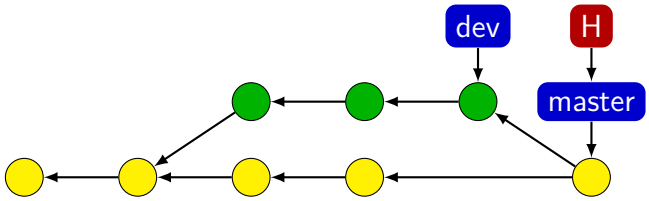


Abbildung 1.4.9: Beispiel eines *Three-Way-Merges*

1.4.11.2 *Rebasing*

- Aktiven Zweig auf einen anderen Zweig neu aufsetzen:

```
git rebase <Zweig-Name>
```

- **Rebasing** ähnelt dem Einspielen von Patches:
  1. Git sucht den **gemeinsamen Vorfahren** beider Zweige
  2. Von dieser Wurzel aus werden die **Änderungen** im aktiven Zweig aus jedem *Commit* **extrahiert** (Diff-Patch)
  3. Die **Patches** werden auf den ausgewählten Zweig **aufgespielt**

- Diff-Patch: Datei, welche nur die Änderungen zwischen zwei Versionen einer Datei oder einer Menge von Dateien enthält (kann mit `diff` manuell erstellt werden).

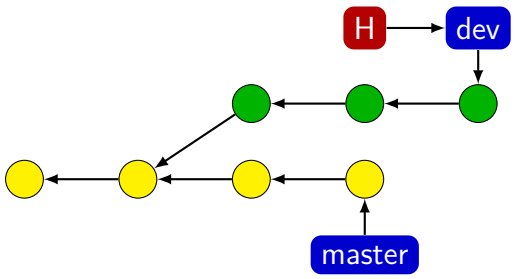
- Sowohl beim *Merging* als auch beim *Rebasing* wird der aktive Zweig verändert. Dennoch müssen die Rollen der verwendeten Zweige umgekehrt werden, um den gewünschten Effekt zu erzielen:
  - Beim *Merging* wird der angegebene Zweig in den aktiven eingebracht. Also muss auf den Zweig gewechselt werden, **in den** die Änderungen eingebracht werden sollen.
  - Beim *Rebasing* wird hingegen der aktive Zweig auf den angegebenen aufgespielt. Demnach muss der Zweig ausgecheckt werden, **dessen** Änderungen eingebracht werden sollen.

- Jeder Patch erzeugt einen neuen *Commit*. Daher erzeugt jeder *Commit* des ursprünglichen Zweiges einen Patch-*Commit* im *Rebase*-Ergebnis.

- Der Zweig, auf den die Änderungen aufgespielt werden, erfährt keine Veränderung. Soll dieser auf den Stand des *Rebase*-Ergebnis gesetzt werden, muss zusätzlich ein *Fast-Forward-Merge* ausgeführt werden (s. nächstes Beispiel).

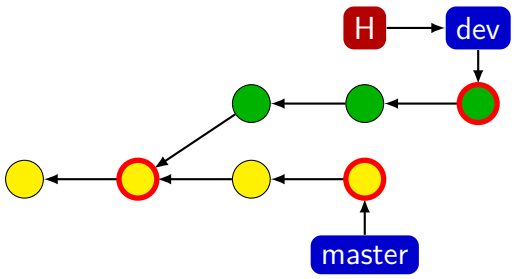
- Beispiel:

  i)



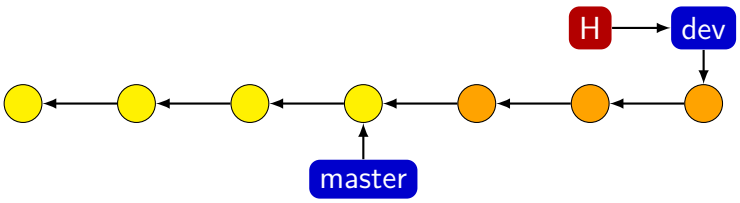
  ii)

```
git rebase master
```



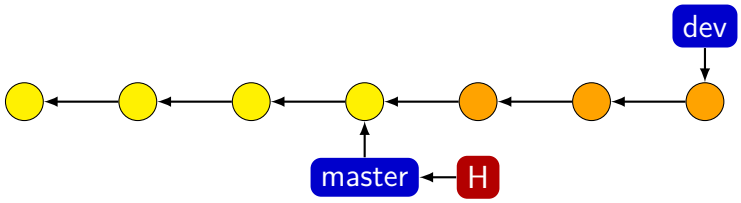
  iii)

```
git rebase master
```



iv)

```
git rebase master
git checkout master
```



v)

```
git rebase master
git checkout master
git merge dev
```

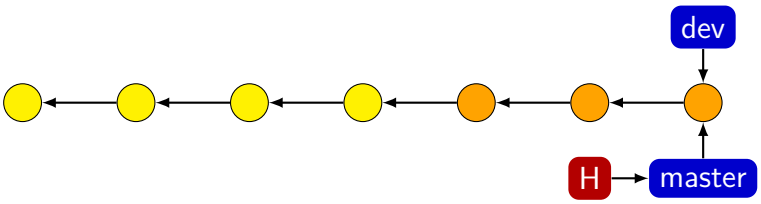


Abbildung 1.4.10: *Rebasing* anhand eines Beispiels

1.4.11.3 *Advanced Rebasing*

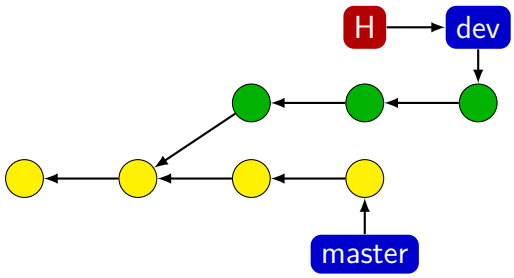
- *Rebasing* kann **präzise gesteuert** werden:

```
git rebase <Zweig-Name> [<Checkout>] [--onto <Basis>]
```

`<Checkout>`:	Wird vor dem *Rebase* ausgecheckt
`<Basis>`:	Zweig, auf dem die Patches aufgespielt werden (Default: `<Zweig-Name>`)

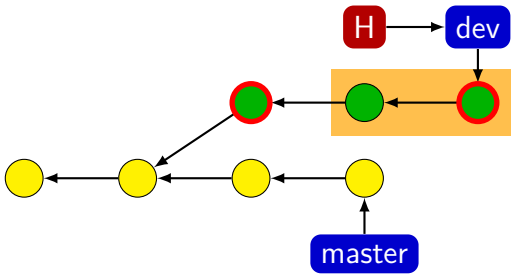
- **Übertragung** eines *Commit*-**Bereichs** auf eine neue Basis
- `-i` startet **interaktive** Shell, mit der die **Reihenfolge** und die **Art** des Aufspielens gesteuert werden kann
- Beispiel:

i)



ii)

```
git rebase dev~2 dev --onto master
```



iii)

```
git rebase dev~2 dev --onto master
```

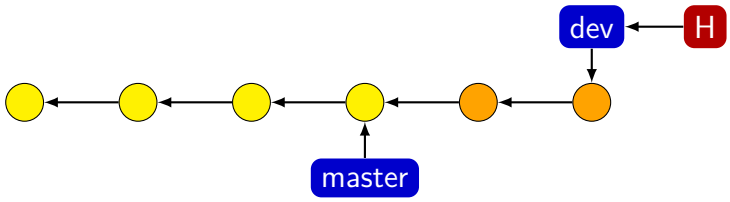


Abbildung 1.4.11: Auswahl eines Bereichs von *Commits* für das *Rebasing*

- Die ersten beiden Argumente von `git rebase` legen **Start** und **Ende** des *Commit*-Bereichs fest:
  - Zunächst wird der *Commit* **vor** dem Start-*Commit* angegeben (`dev~2`).
  - Als zweites folgt der letzte *Commit* des Bereichs (`dev`).

- In diesem Beispiel hätte `dev` ausgelassen werden können, da `dev` bereits der aktive Zweig war.

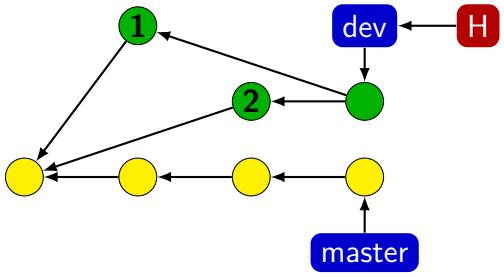
1.4.11.4 *Rebasing* von *Merge-Commits*

Rebasing* von *Merge-Commits
Rebasing erzeugt (per Default) ein **vollständig lineares** Ergebnis!

⇒ *Merge-Commits* werden beim *Rebasing* **verworfen**

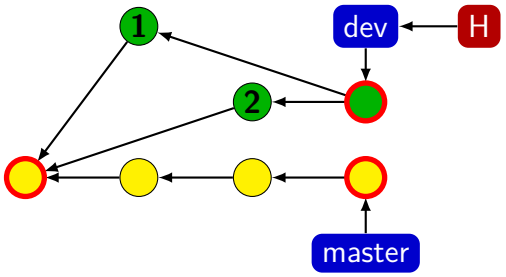
- Aus allen anderen *Commits* werden *Diff-Patches* extrahiert, die **chronologisch** aufgespielt werden
- Option `-p`: *Merge-Commits* werden **nachgebildet** (`-p` ≙ `--preserve-merges`)
- Beispiel:

  i)



  ii)

```
git rebase master
```



iii)

```
git rebase master
```

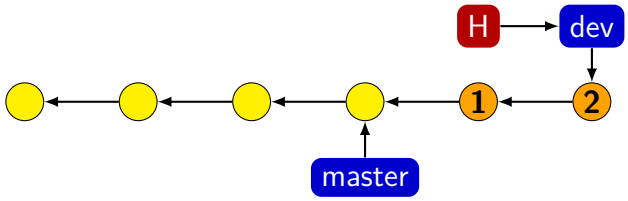
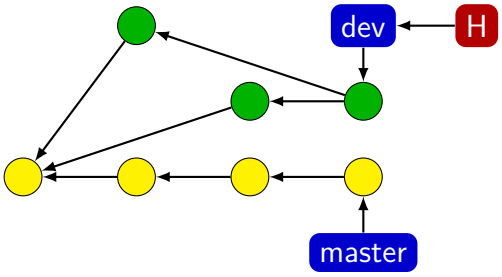


Abbildung 1.4.12: Vollständige Linearisierung durch *Rebasing*

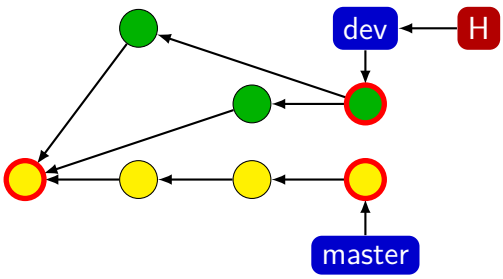
- Beispiel mit Option `-p`:

i)



ii)

```
git rebase -p master
```



iii)

```
git rebase -p master
```

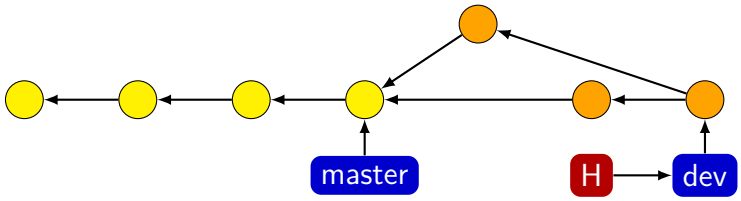


Abbildung 1.4.13: Nachbildung von *Merge-Commits* beim *Rebasing*

1.4.11.5 Konflikte

- Git's *Merging*/*Rebasing*-Algorithmus arbeitet **zeilenbasiert**

⇒ Änderungen auf parallelen Zweigen
  - in derselben Datei und
  - in derselben Zeile

  erzeugen einen **Konflikt** beim Zusammenführen

- Git markiert betroffene Zeilen und trägt beide Versionen ein
- `git status` liefert zu editierende Dateien
- Weiteres Vorgehen von der Art der Zusammenführung abhängig (*Merge* oder *Rebase*)
- Falls für die Lösung des Konfliktes innerhalb einer Datei eine der beiden Alternativ-Versionen vollständig übernommen werden soll, so bietet Git die Möglichkeit, die gewünschte Version zu wählen und so ein manuelles Editieren einzusparen:

```
git checkout --ours -- <Datei>
```

  oder

```
git checkout --theirs -- <Datei>
```

- Beispiel eines eingetragenen Konfliktes:

```
<<<<<<< HEAD
printf("Hallo!");
=======
printf("Hallo Welt!");
>>>>>>> feature-hallo-welt
```

- Git trennt Versionsblöcke durch Zeilen mit `<<<`, `===` und `>>>`
- Hinter `<<<` und `>>>` steht der Zweig, aus dem die Änderung stammt

1.4.11.6 Konflikte bei *Merge*

- Bei einem *Merge* werden alle Änderungen **auf einmal** zusammengeführt

⇒ Git meldet alle Konflikte gemeinsam

- Bei Konflikten befindet sich Git weiterhin im *Merge*-Modus
- Ablauf:
  1. Konfliktbehaftete Dateien **editieren**
  2. **Gelöste Konflikte** durch `git add <Datei>` **anzeigen**
  3. `git commit` aufrufen, um den ***Merge-Commit* abzuschließen**
- Alternativ kann der *Merge*-Modus durch

```
git merge --abort
```

  verlassen und der *Merge* rückgängig gemacht werden

1.4.11.7 Konflikte bei *Rebase*

- *Rebasing* führt Änderungen in Form **einzelner** Patches ein

⇒ Jeder Patch kann neue Konflikte erzeugen

- Ablauf:
  1. Konfliktbehaftete Dateien **editieren**
  2. **Gelöste Konflikte** durch `git add <Datei>` **anzeigen**
  3. **Einspielen** mit `git rebase --continue` **fortsetzen**
  4. **1-3** mit allen konfliktbehafteten Patches **wiederholen**
- Vorzeitiger Abbruch des **Rebasing** mit

```
git rebase --abort
```

  mit Wiederherstellung des Standes **vor** dem *Rebase*

1.4.12 Arbeiten mit *Remote-Repositories*

- **Remote Repository**: Git-*Repository*, welches vom eigenen lokalen *Repository* referenziert wird
- Datenabgleich mit anderen Entwicklern, indem Daten geholt (**fetch**) oder geschoben (**push**) werden
- Geklonte *Repositories* verfügen über eine Standard-Referenz auf die Quelle (**origin**)
- Mit `git remote` können Referenzen beliebig verändert, gelöscht und hinzugefügt werden

1.4.12.1 zentral vs. dezentral

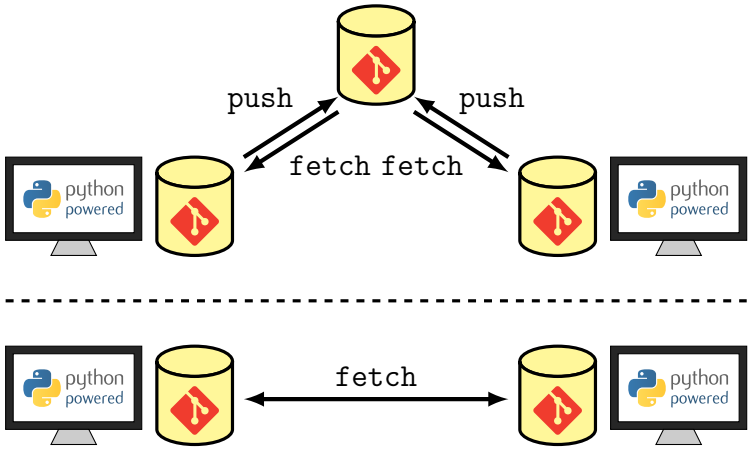


Abbildung 1.4.14: *Fetch* und *Push* bei zentraler und dezentraler Arbeitsweise

- Bei der Arbeit mit einem zentralen Server schieben alle Entwickler ihre Änderungen in Form von *Commits* regelmäßig in das zentrale *Bare-Repository* (`push`) und Holen neue *Commits* der Kollegen von dem Server (`fetch`).
- Die dezentrale Arbeitsweise funktioniert ausschließlich über *Fetching*, da ein `git push` in ein *Repository* mit *Working Copy* nicht erlaubt ist.

1.4.12.2 Entfernte Referenzen anpassen

- Eintragene *Remote Repositories* inkl. URLs ausgeben:

```
git remote -v
```

- Entferntes *Repository* hinzufügen:

```
git remote add <Name> <URL>
```

- Referenz entfernen:

```
git remote remove <Name>
```

- URL korrigieren:

```
git remote set-url <Name> <URL>
```

1.4.12.3 *Fetching*

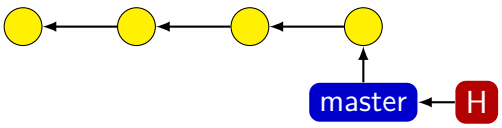
```
git fetch <Name>
```

- Holt die **gesamte *History*** vom angegebenen *Repository*
- Holt nur die Änderungen zum aktuellen Stand
- `git fetch` **ergänzt** nur die lokale *History* – um entfernte *Commits*, Zweige, *Tags* ...

⇒ Der **lokale** Stand bleibt **unverändert** erhalten!

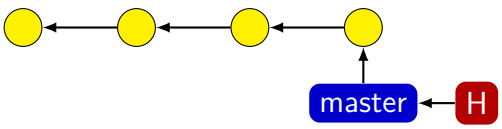
- Mit `--all` werden alle eingetragenen *Repositories* abgefragt
- `git fetch` belässt den lokalen Stand unverändert, da alle geholten Objekte – sofern überhaupt nötig – mit Präfixen eingetragen werden. `master` wird auf diese Weise z. B. zu `origin/master`. *Commits* benötigen keine zusätzliche Kennzeichnung ihrer Quelle, da sie über ihren *Hash* identfiziert werden, der bei inhaltlich unterschiedlichen *Commits* als eindeutig angenommen wird.
- Beispiel:

  i)



  ii)

```
git remote add origin ifflinux:test_project
```



iii)

```
git remote add origin ifflinux:test_project
git fetch origin
```

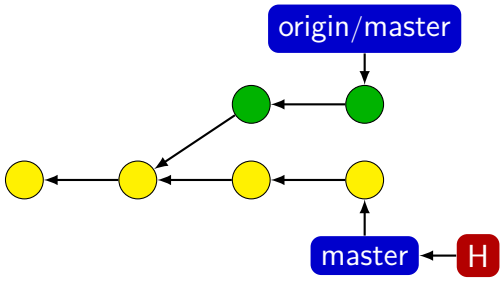


Abbildung 1.4.15: Beispiel für das Holen von Daten entfernter *Repositories*

1.4.12.4 Geholte Daten verwenden

- Zweige entfernter Repositories werden über **Präfixe** konfliktfrei in das lokale *Repository* integriert (z. B. `origin/`)
- Diese können **nicht** wie lokal erstellte Zweige benutzt werden:
  - Nur für den Abgleich mit entfernten Daten
  - Git-Kommandos mit nur lokaler Wirkung haben keinen Effekt

  ⇒ `git checkout` führt zu einem *Detached HEAD*
- Es muss lokale Kopie des entfernten Zweigs erstellt werden
- Lokaler und entfernter Zweig werden meist verknüpft (**Tracking Branch**)
- Eine Kopie eines Zweiges zu erstellen, bedeutet Git-intern lediglich einen Zeiger zu kopieren (**Shallow Copy**). Der Aufwand ist also sehr gering.

1.4.12.5 *Tracking Branches*

- **Tracking Branch** erstellen:

```
git checkout --track <Remote>/<Zweig>
```

- Die lokale Kopie `<Zweig>` kann wie ein gewöhnlicher Zweig bearbeitet werden
- Besonderheiten eines *Tracking Branches*:
  - Git legt eine **Verknüpfung** zwischen lokalem und entferntem Zweig an

- `git status` gibt aus, wie der **Synchronisierungsstand** ist
- `git fetch`, `pull`, `push` können in **verkürzter Form** verwendet werden (später)

1.4.12.6 *Tracking Branches* synchronisieren

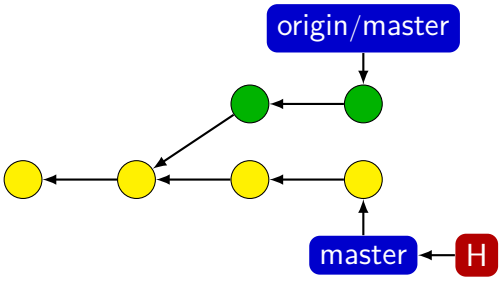
- `git fetch` ändert nur die Position der *Remote Branches*

⇒ Änderungen müssen über **Merging** oder **Rebasing** in die lokalen *Tracking Branches* integriert werden

- Um den Arbeitsauflauf zu vereinfachen, existiert **git pull**:
  - Führt zunächst ein *Fetch* und dann ein *Merge* (Default) bzw. *Rebase* aus

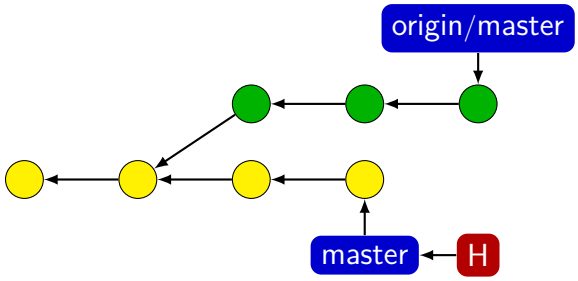
- Beispiel mit Merging:

i)



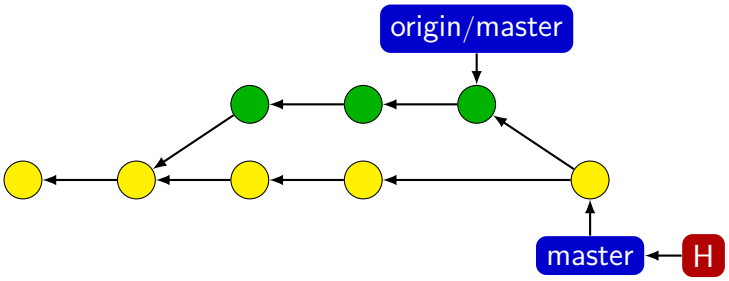
ii)

```
git fetch origin
```



iii)

```
git fetch origin
git merge origin/master
```



iv)

```
# Alternativ:
git pull
```

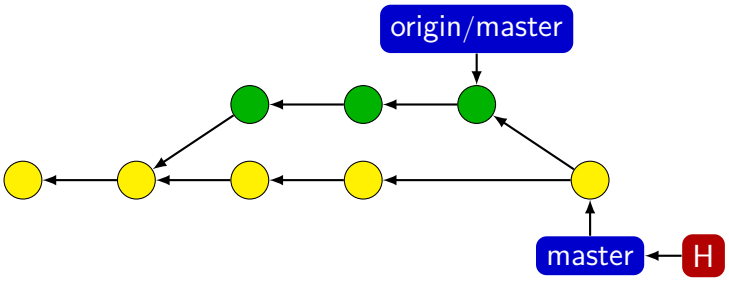
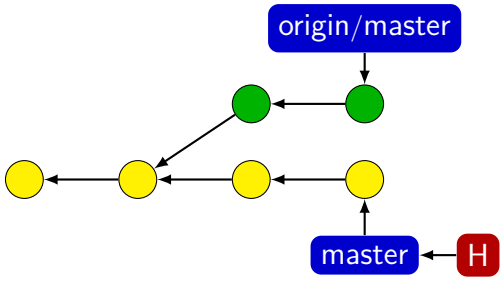


Abbildung 1.4.16: Arbeitsweise von `git pull`

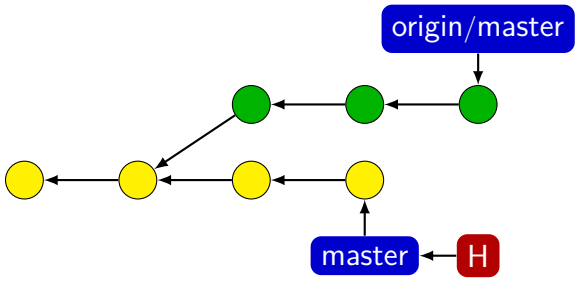
- Beispiel mit Rebasing:

  i)



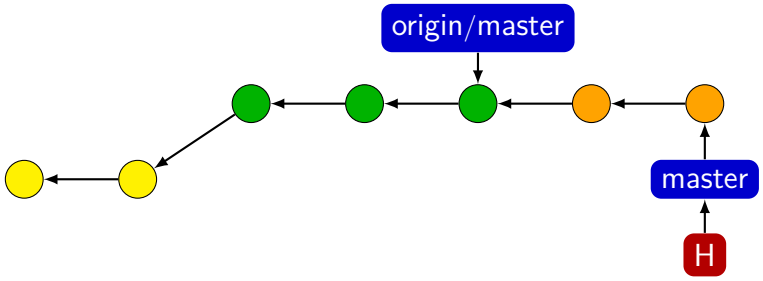
ii)

```
git fetch origin
```



iii)

```
git fetch origin
git rebase origin/master
```



iv)

```
# Alternativ:
git pull --rebase
```

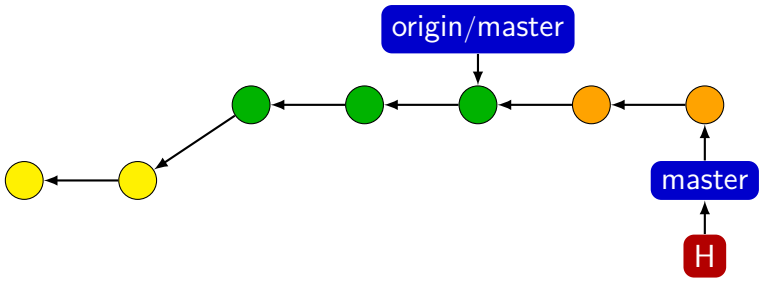


Abbildung 1.4.17: Arbeitsweise von `git pull --rebase`

- Der Befehl `git pull --rebase` führt ein Standard-*Rebase* aus und verwirft daher lokal erstellte *Merge-Commits*. Sollen *Merge-Commits* beibehalten werden, so sollte der Befehl

```
git pull --rebase=preserve
```

verwendet werden. Die Übergabe von `-p` oder `--preserve-merges` funktioniert an dieser Stelle **nicht**.

1.4.12.7 Eigene Änderungen verteilen

- **Dezentrale** Arbeitsweise:
  - Jeder kann auf die lokalen *Repositories* der anderen Team-Mitglieder lesend zugreifen
  - Jeder holt mit `git fetch` die Daten von jedem anderen
  - Integration der geholten Daten über **Merging** oder **Rebasing**
  - Es findet **kein Schieben** von Daten statt (*Push*)
- Vorteile:
  - Kein zentraler Git-Server nötig
  - Kein Single-Point-of-Failure
- Nachteile:
  - Kein Punkt, an dem alle Änderungen zusammengetragen werden

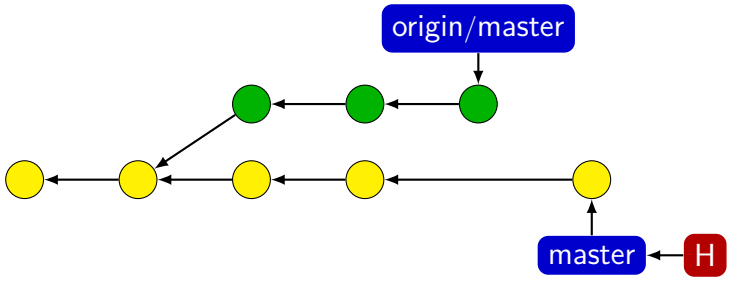
  ⇒ Weniger Struktur, keine direkte Code-Basis
  - Rechner aller Entwickler müssen direkt erreichbar sein
- **Zentrale** Arbeitsweise:
  - Zentraler Server verwaltet Änderungen aller Team-Mitglieder
  - ***Commits*** werden aktiv auf den Server **geschoben** (*Push*)
  - Alle laden regelmäßig mit `git fetch` oder `pull` neue Daten
- Vorteile:
  - Immer erreichbare Anlaufstelle, um die Entwicklung zu verfolgen
- Nachteile:
  - Verwaltung des Servers notwendig
- **Übliche Vorgehensweise** für Projekte mit mehreren Mitarbeitern
- **Daten** auf zentralen Server **schieben**:

```
git push <Remote> <Lok. Zweig>:<Entf. Zweig>
```

- Sendet alle zum lokalen Zweig gehörenden Daten zum Server und erstellt dort einen neuen Zweig `<Entf. Zweig>`
- Existiert der Zweig, so wird er umgesetzt, sofern dadurch keine *Commits* verloren gehen
- Bei identischen Namen kann der Teil ab `:` weggelassen werden
- Option `-u` richtet einen ***Tracking Branch*** ein
- Pushen ist nur zu **Bare-Repositories** erlaubt!
- Beispiel:

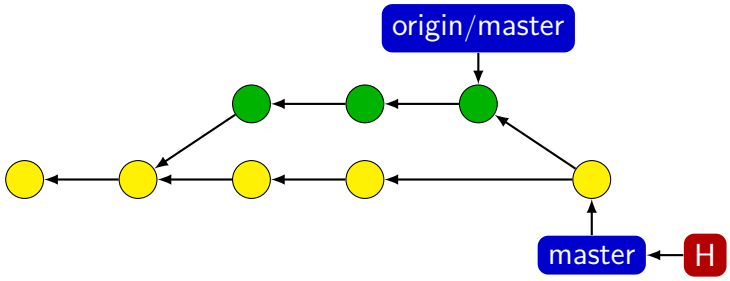
  i)

```
git fetch origin
```



  ii)

```
git fetch origin
git merge origin/master
```



iii)

```
git fetch origin
git merge origin/master
git push -u origin master
```

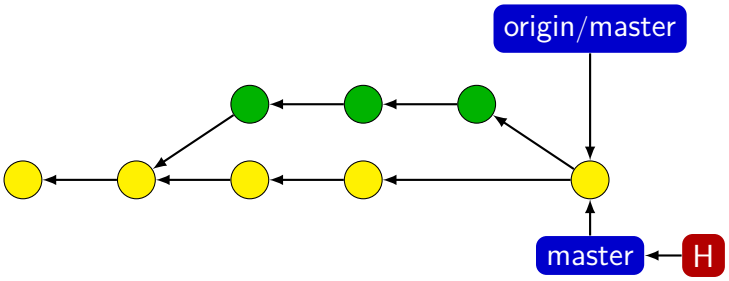
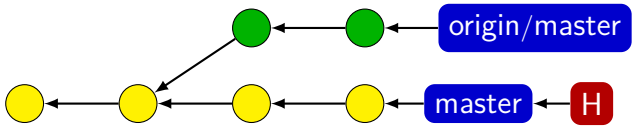


Abbildung 1.4.18: Übertragen von Daten zu *Remote-Repositories*

i)

- Situation, in der ein *Push* **zurückgewiesen** wird:



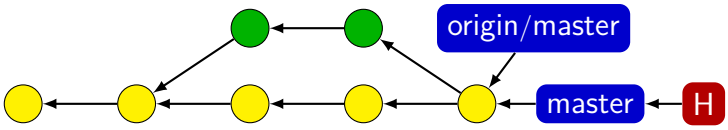
- *Push* würde **einzigen Zeiger** auf grüne *Commits* **entfernen**

⇒ Die grünen *Commits* wären nicht mehr referenzierbar

⇒ Remote-**Änderungen** immer zuerst **integrieren**!

ii)

- Situation, in der ein *Push* **durchgeführt** werden kann:



1.4.12.8 *Push* und *Rebase*

Push* und *Rebase
Auf *Commits*, die bereits gepusht worden sind, darf kein *Rebase* angewendet werden!

- ***Rebasing* verändert die History** eines Zweigs:
  - *Commits* des Zweigs werden als Patches extrahiert
  - Die Patches werden auf einen **anderen** Zweig aufgespielt
- *Push* würde *History* des Servers überschreiben

⇒ Wird vom **Git-Server verweigert**

⇒ Richtige Vorgehensweisen:

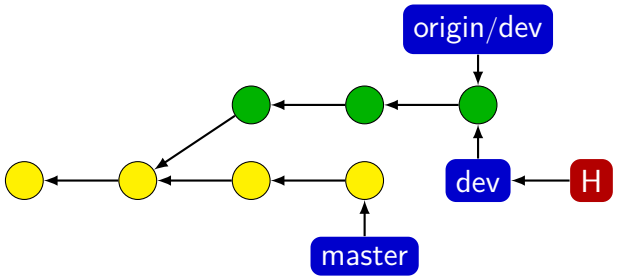
- Erst *Rebase*, dann *Push* oder
- *Merge* statt *Rebase* verwenden

- Bei der Verwendung von *Merge* kann eine analoge Situation nicht auftreten, da *Merging* **nie** die History verändert.

⇒ *Merging* ist daher für Anfänger leichter zu handhaben.

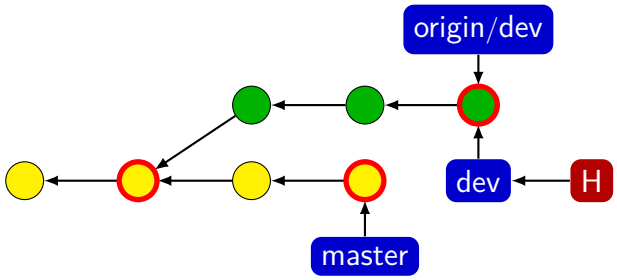
- Demonstration der Problematik:

i)



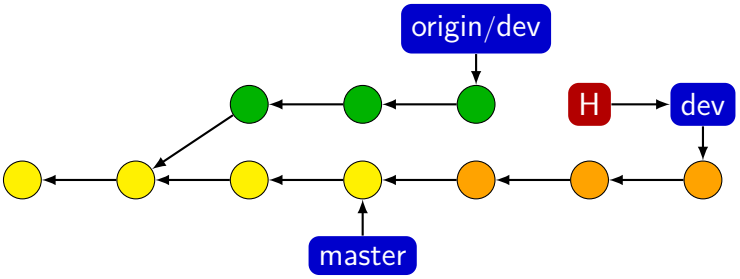
ii)

```
git rebase master
```



iii)

```
git rebase master
```



⇒ `dev` und `origin/dev` **divergieren** durch das *Rebase*, *Push* nicht möglich

Abbildung 1.4.19: *Rebasing* bereits veröffentlichter *Commits*

- Es existiert eine **Ausnahme** der *Rebase*-Regel: Soll `origin/dev` nach dem *Rebase* auf dem Server gelöscht werden, so ist das *Rebasing* in Ordnung, da Git-Server das Löschen eines Zweiges auch dann annehmen, wenn hierdurch Daten verloren gehen (s. nächste Folie).

1.4.12.9 Entfernte Zweige löschen

- Entfernte Zweige mit `git push` löschen:

```
git push <Remote> :<Entf. Zweig>
```

  → Zweignamen ein „`:`“ vorstellen

- **Eselsbrücke** für die Syntax: Pushe „nichts“ an die Stelle des entfernten Zweiges

Vorsicht
Kann **ohne Nachfrage** zur Löschung von *Commits* führen!

⇒ Vorher sicher gehen, dass
- anderer Zweig die ***Commits* beinhaltet** oder
- die ***Commits* nicht mehr benötigt** werden

1.4.12.10 Gelöschte Zweige synchronisieren

- Auf Server gelöschte Zweige werden **nicht** mit `git fetch` auch bei anderen Team-Mitgliedern entfernt

⇒ Synchronisierung gelöschter Zweige erfolgt **manuell**:

```
git remote prune <Remote>
```

- Derjenige, der den Zweig auf dem Server gelöscht hat, muss keine manuelle Synchronisierung ausführen; Git erledigt dies hier automatisch.

1.4.12.11 Kurzformen für *fetch* und *push*

- Auf *Tracking Branches* können `git fetch` und `git pull` **ohne Argument** aufgerufen werden
- Beziehen sich auf das *Remote-Repository* des aktiven Zweiges

→ `git fetch` fragt das *Remote-Repository* des *Tracking Branches* ab

→ `git push` überträgt den Stand **aller** lokalen Zweige, sofern gleich benannte Zweige auf dem Server existieren (***matching***)

- `git push` kann auf den aktiven Zweig begrenzt werden (später)

1.4.13 *Tagging*

- *Commits* können markiert werden (***Tagging***)
- 3 Arten von ***Tags*** möglich:
  - ***Lightweight Tags***:
    - **Konstanter Zeiger** auf einen bestimmten *Commit*

    ⇒ Verhält sich wie ein **unbeweglicher Zweig**
  - ***Annotated Tags***:

    - Zeiger mit eigenem **Tag-Objekt**
    - ⇒ Tags erhalten einen Hash, Ersteller, Datum, Nachricht, ...
    - Sollten **immer bevorzugt** genutzt werden
  - ***Signed Tags***:
    - *Annotated Tag* mit **GPG-Signatur**
    - ⇒ Ersteller kann verifiziert werden
    - Hier nicht weiter behandelt

- *Tags* sind nützlich, um bestimmte *Commits* mit einem Namen zu versehen. Der *Tag*-Name kann anschließend wie ein Zweig verwendet werden, um den *Commit* zu referenzieren.
- *Tags* werden häufig verwendet, um stabile Versionen der Entwicklung mit einer Versionsnummer zu versehen. Auf diese Weise kann in Git leicht zu jeder bisher stabilen Version gesprungen werden:

```
git checkout <Tag-Name>
```

1.4.13.1 *Annotated Tag* erstellen, *Tags* listen und löschen

- *Annotated Tag* **erstellen**:

```
git tag -a <Tag-Name>
```

- Der Tag zeigt auf den aktuell ausgecheckten *Commit*
- **-a** wichtig, sonst wird ein *Lightweight Tag* erstellt!
- Alle erstellten *Tags* **anzeigen**:

```
git tag
```

- *Tag* wieder **löschen**:

```
git tag -d <Tag-Name>
```

1.4.13.2 *Tags* verteilen

- *Tags* zu einem *Remote-Repository* schicken:

```
git push --tags <Remote>
```

- Ohne `--tags` überträgt `git push` keine *Tags*!
- Alle anderen erhalten die *Tags* über das nächste `git fetch`

- Löschen von *Remote-Tags* ähnelt Löschen von Zweigen:

```
git push <Remote> :refs/tags/<Tag-Name>
```

- Remote gelöschte *Tags* werden **nicht** auch lokal entfernt
- Das Löschen von *Tags* bezieht sich auf die interne Verzeichnisstruktur des entfernten Git-Repositories. `refs/` beinhaltet alle gespeicherten Referenzen/Zeiger auf bestehende *Commits* in Form von Dateien, die Zweige (`refs/heads/`) und *Tags* (`refs/tags/`) repräsentieren.
- Mit Hilfe von `git push` können diese Dateien auf dem Server manipuliert (neuen *Commit-Hash* eintragen) oder gelöscht werden.

1.4.13.3 Gelöschte *Tags* synchronisieren

- Synchronisierung gelöschter *Tags* ist **manuell** möglich:
  1. Alle lokalen *Tags* löschen
  2. Alle *Remote-Tags* neu übertragen

```
git tag | xargs git tag -d
git fetch <Remote>
```

- `xargs` nimmt die Ausgabe von `git tag` zeilenweise entgegen und wendet sie als Argument auf `git tag -d` an

1.4.14 Aktuellen Arbeitsstand mit letztem *Commit* vergleichen

- **Inhalt** einer Datei mit dem Stand des letzten *Commits* **vergleichen**:

```
git diff <Datei>
```

- **Alle Änderungen** seit dem letzten *Commit* anzeigen:

```
git diff HEAD
```

- Statt `HEAD` können beliebige *Commits*/Zweige/*Tags* als Vergleichsbasis angegeben werden

1.4.15 Aktuelle Änderungen temporär aufheben

- Alle **Änderungen** seit letztem *Commit* **wegspeichern**:

```
git stash
```

- Erzeugt einen neuen Eintrag im **Stash** (*Stack*)

- Änderungen erneut anwenden:

```
git stash apply
```

- Alle gespeicherten Einträge anzeigen:

```
  git stash list
```

- Inhalt des letzten Eintrages anzeigen:

```
  git stash show
```

- `apply` löscht angewandte Änderungen **nicht** vom *Stack*;
  manuelles Löschen des letzten Eintrages:

```
git stash drop
```

- **Kurzform** für `apply` gefolgt von `drop`:

```
git stash pop
```

- Anzuwendender **Eintrag** kann **selektiert** werden:

```
git stash apply/pop/show stash@{n}
```

  Default ist $n = 0$

- Der Befehl `git stash` eignet sich gut, um die eigene Arbeit kurzzeitig unterbrechen und zu einem anderen Entwicklungsstand springen zu können, wie z. B. der letzten lauffähigen Version.

- Die gespeicherten Änderungen können an beliebiger Stelle wieder angewendet werden, sodass sich `git stash` auch eignet, um Änderungen auf andere Zweige zu übertragen. Evtl. entstehende Konflikte müssen danach manuell behoben werden.

1.4.16 Zweig auf anderen *Commit* setzen

- Aktiven **Zweig** auf einen anderen *Commit* **umsetzen**:

```
git reset <Commit>
```

- `<Commit>` wird in Form eines Hashes, Zweigs, ... referenziert

- Optionen:

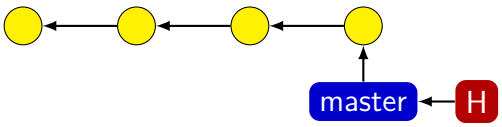
  `--soft`: Behält die *Staging Area* und alle Modifikationen bei

  `--mixed`: Verwirft die *Staging Area*, aber behält Dateiänderungen (**default**)

  `--hard`: Setzt die *Working Copy* vollständig auf den Stand des angegebenen *Commits* zurück

- Beispiel:

  i)



  ii)

```
git reset HEAD~
```

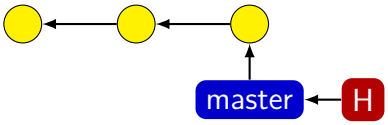


Abbildung 1.4.20: Beispiel für `git reset`

1.4.17 Änderungen rückgängig machen

- **Änderungen** an einer bestimmten Datei **verwerfen**:

```
git checkout -- <Datei>
```

- `--` zur Unterscheidung von Zweig- und Dateinamen
- Datei **aus** der ***Staging Area*** **entfernen** ohne Änderungen zu verwerfen:

```
git reset HEAD <Datei>
```

- Alle Änderungen seit dem letzten *Commit* verwerfen:

```
git reset --hard HEAD
```

1.4.18 Letzten *Commit* rückgängig machen

- 2 Möglichkeiten: **reset** oder **revert**
- `git reset`:
  - Setzt den Zweig um (s. letztes Beispiel)

  ⇒ ***Commit*** **wird** aus der *History* des Zweigs **entfernt** (Datenverlust!)

⇒ Nur verwenden, wenn der *Commit* **nicht gepusht** wurde

- `git revert`:
  - Erstellt einen ***Undo-Commit***

  ⇒ Kein Datenverlust, *Push* stellt kein Problem dar

  ⇒ Bläht die *History* auf

- Sollte `git reset` verwendet worden sein, obwohl der zu löschende *Commit* bereits in ein anderen Git-*Repository* gepusht wurde, so verweigert der Server beim nächsten *Push* die Annahme. Um nachträglich noch ein `revert` auszuführen und das `reset` rückgängig zu machen, können folgende Befehle verwendet werden:

```
git pull
git revert
```

`git pull` stellt den Zustand vor dem `git reset` wieder her, sodass anschließend `git revert` ausgeführt werden kann.

1.4.19 Letzten *Commit* um Änderungen erweitern

- Aktuelle **Änderungen** noch zum letzten *Commit* **hinzufügen**:

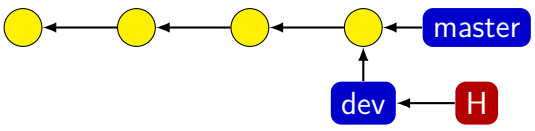
```
git commit --amend
```

- Verhält sich wie:

```
git reset --soft HEAD~
git commit
```

- Alter *Commit* nur dann verloren, wenn in keinem weiteren Zweig mehr enthalten
- Nur verwenden, wenn der *Commit* noch **nicht gepusht** wurde
- Kann mit `-a` kombiniert werden
- Beispiel:

  i)



ii)

```
git commit --amend
```

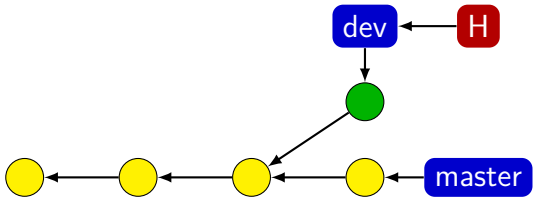


Abbildung 1.4.21: Letzten *Commit* um Änderungen erweitern

1.4.20 Gelöschte *Commits* retten

- Wird in Git ein Zweig/Tag gelöscht/umgesetzt, so werden nicht mehr erreichbare *Commits* **nicht** sofort gelöscht
- Git speichert den Verlauf aller Zweige und des `HEAD`-Zeigers, einsehbar mit

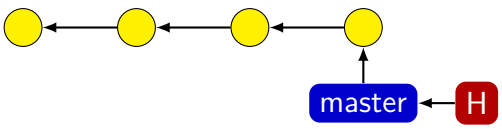
```
git reflog <Zweig-Name>
```

- Per Default wird der Verlauf von `HEAD` ausgegeben
- Liefert eine Ausgabe der Form:

```
c718f8e HEAD@{0}: commit: Neue Version
7b248e0 HEAD@{1}: commit: Tolles Feature eingebaut
c91a17b HEAD@{2}: commit: Readme update
```

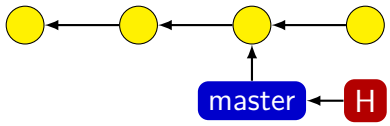
- `HEAD@{0}` ist die gegenwärtige `HEAD`-Position
- Die übrigen `HEAD@{n}` referenzieren die Vergangenheit des `HEAD`-Zeigers
- Beispiel:

i)



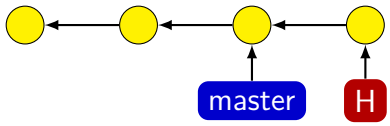
ii)

```
git reset HEAD~
```



iii)

```
git reset HEAD~
git checkout HEAD@{1}
```



iv)

```
git reset HEAD~
git checkout HEAD@{1}
git branch temp
```

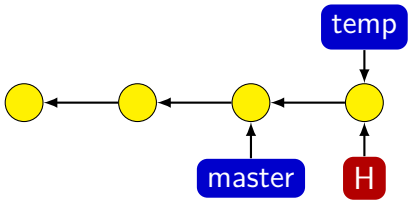


Abbildung 1.4.22: Letzten *Commit* retten

- Analog kann auch die Vergangenheit eines Zweiges referenziert werden, z. B. über `master@{2}`.

1.4.21 *Working Copy* aufräumen

- Alle **Dateien löschen**, die von `git status` als „***untracked***“ markiert werden:

```
git clean -f
```

- Zusätzlich alle von Git ignorierten Dateien entfernen:

```
git clean -fx
```

- `-d` löscht außerdem unversionierte **Verzeichnisse**
- `-n` statt `-f` liefert nur die Dateien, die entfernt würden

1.4.22 Autor eines Code-Bereichs bestimmen

- Nicht immer offensichtlich, wer welchen Code geschrieben hat
- Das Kommando

```
git annotate <Datei>
```

  gibt die gesamte Datei aus, aber ergänzt jede Zeile um
  - ***Commit-Hash*** der letzten Änderung
  - **Autor** der Zeile
  - **Datum und Uhrzeit** der letzten Modifikation

- Statt `git annotate` kann alternativ auch `git blame` verwendet werden. Beide liefern dieselben Informationen, nur die Formatierung der Ausgabe ist etwas unterschiedlich.

1.4.23 Aktuellen Arbeitsstand exportieren

- Ziel: Aktuellen **Entwicklungsstand als Archiv** verschicken
- „Naive“ Lösung:
  - `tar`/`zip` auf den eigenen *Repository*-Klon
- Probleme:
  - Unerwünschte Dateien, wie Build-Files, werden mitgepackt
  - `.git`-Verzeichnis wird eingeschlossen, obwohl unnötig
- Git bietet ein passendes Kommando an:

```
git archive --prefix <Oberverzeichnis>/ --output <Archiv-Name>
    <Commit/Zweig/Tag>
```

- **Exportiert Stand der *Working Copy*** eines *Commits*/Zweigs
- Nur versionierte Dateien werden beachtet
- `--output`:

  - Gibt **Speicherort** des Archivs an
  - Endung gibt den **Archivtypen** vor (`tgz` oder `zip`)

- `--prefix`:
  - Stellt allen Dateien ein Präfix voran
  - Ein abschließendes `/` erzeugt ein **übergeordnetes Verzeichnis**

- Die Option `--prefix` sollte immer verwendet werden, da Git ansonsten **kein** oberstes Projektverzeichnis innerhalb des Archivs erstellt!

1.5 Nützliches

1.5.1 Editor für *Commit-Messages* setzen

- Default-Editor für ***Commit*-Messages**: `${VISUAL}`
  wenn nicht gesetzt, dann meist `vi(m)`

- **Systemweit** den Default-Editor ändern:

```
export VISUAL=<Editor-Name>
```

  zur `.bashrc` (Linux) bzw. `.profile` (OS X) hinzufügen

- Einstellung auf **Git** beschränken:

```
git config --global core.editor <Editor-Name>
```

- `git config` kann auch ohne `--global` ausgeführt werden. Ohne `--global` bezieht sich die Einstellung nur auf das *Repository*, in dem man sich aktuell befindet. Mit `--global` gilt die Einstellung für alle bestehenden und auch zukünftig erstellten *Repositories*, sofern die Einstellung nicht durch eine lokalen `git config`-Aufruf überschrieben wird.

- Statt `--global` kann alternativ `--system` übergeben werden. Im Gegensatz zu `--global` ändert `--system` die globalen Git-Einstellungen des ganzen Systems (also für jeden Benutzer), wohingegen sich `--global` nur auf den aktuellen Benutzer bezieht.

1.5.2 Farbige Ausgaben

- Git bietet die Möglichkeit, Ausgaben farbig zu gestalten:
  - Farben für hinzugefügte und gelöschte Dateien (`git status`)
  - Farbiges Diff
  - Farbiger Log

– ...

- Aktivieren über

```
git config --global color.ui true
```

1.5.3 *Bash Completion*

- Git bietet eine **Autovervollständigung** für die Bash an
- Neben Kommandos werden auch Zweignamen/*Tags*/... vervollständigt
- Installation:
  1. Git-Source klonen:

     ```
     git clone https://github.com/git/git.git
     ```

  2. Passende Git-Version auschecken:

     ```
     git checkout v‘git --version | cut -d " " -f3‘
     ```

  3. Datei `contrib/completion/git-completion.bash` nach `${HOME}` kopieren und in `.bashrc` bzw. `.profile` eintragen:

     ```
     source ~/git-completion.bash
     ```

- Viele Linux-Distributionen installieren die Autovervollständigung automatisch mit, sodass die gezeigten Schritte nicht extra ausgeführt werden müssen.

1.5.4 Aktuellen Zweig im Prompt anzeigen

- Shell-Erweiterung: Aktuellen **Zweig im Prompt** anzeigen
- Die Installation verläuft analog zur *Bash Completion*:
  1. Git-Source klonen und passende Version auschecken
  2. Datei `contrib/completion/git-prompt.sh` nach `${HOME}` kopieren und wie `git-completion.sh` eintragen
  3. Nun steht die Bash-Funktion `__git_ps1` zur Bestimmung des aktuellen Zweiges für den Prompt bereit, Beispiel:

     ```
     export PS1='\u@\h:\W$(__git_ps1) $ '
     ```

- Setzt man zusätzlich

```
export GIT_PS1_SHOWDIRTYSTATE=1
```

so werden Änderungen seit dem letzten *Commit* signalisiert

- Die Funktion `showdirtystate` ist per Default nicht aktiviert, da sie das Wechseln in ein von Git versioniertes Verzeichnis spürbar verlangsamt. Dennoch ist die Aktivierung empfehlenswert, da so immer sofort sichtbar ist, ob die *Working Copy* neue Änderungen beinhaltet und somit `git status`-Aufrufe eingespart werden können.
- Ob Änderungen vorliegen, wird über einen Stern `*` signalisiert.

1.5.5 *Push* auf aktuellen Zweig begrenzen

- `git push` ohne Parameter:
  - Bezug zum *Remote-Repository* des aktiven *Tracking Branches*
  - Überträgt **alle** Zweige, die Remote identisch benannt sind

⇒ Gefahr, versehentlich Zweige zu *pushen*

- **Begrenzung** auf den aktiven Zweig:

```
git config --global push.default upstream
```

- Mit

```
git config --global push.default matching
```

  werden wieder alle Zweige gepusht (**Default**)

- Neben `matching` und `upstream` existieren weitere mögliche Optionen:
  - `nothing`: `git push` darf nicht ohne Parameter aufgerufen werden
  - `current`: Überträgt den aktiven Zweig, sofern im *Remote-Repository* ein identisch benannter Zweig existiert. Funktioniert auch für Zweige, die keine *Tracking Branches* sind, wenn das *Remote-Repository* als Parameter übergeben wird.
  - `simple`: Funktioniert wie `upstream`, aber überträgt einen Zweig nur, wenn er auf dem Server gleich benannt ist. Wird ein *Remote-Repository* übergeben, so verhält sich `git push` so, als wäre `current` eingestellt worden.
- Momentan (Git v1.x) ist `matching` das Standardverhalten, welches ab Version 2.0 von `simple` abgelöst werden wird.

1.5.6 Log in grafischer Oberfläche

- Das Programm **GitX** (Mac OS X) bzw. **Gitg** (Linux) bietet eine grafische Oberfläche, um
  - die **History** eines *Repositories* zu **betrachten**
  - **Änderungen** zwischen *Commits* **anzuzeigen**
  - **Zweige** zu **verwalten**

– ...

- Erhältlich unter

  `http://gitx.frim.nl/`

  bzw.

  `http://git.gnome.org/browse/gitg/`

- Viele Entwicklungsumgebungen bringen außerdem Plugins mit, um grafisch *Commits* abzusetzen oder Diffs erstellen zu lassen. Diese Werkzeuge sind praktisch, aber um Git zu erlernen, ist es dennoch empfehlenswert, zunächst einige Zeit mit Git auf der Konsole zu arbeiten.

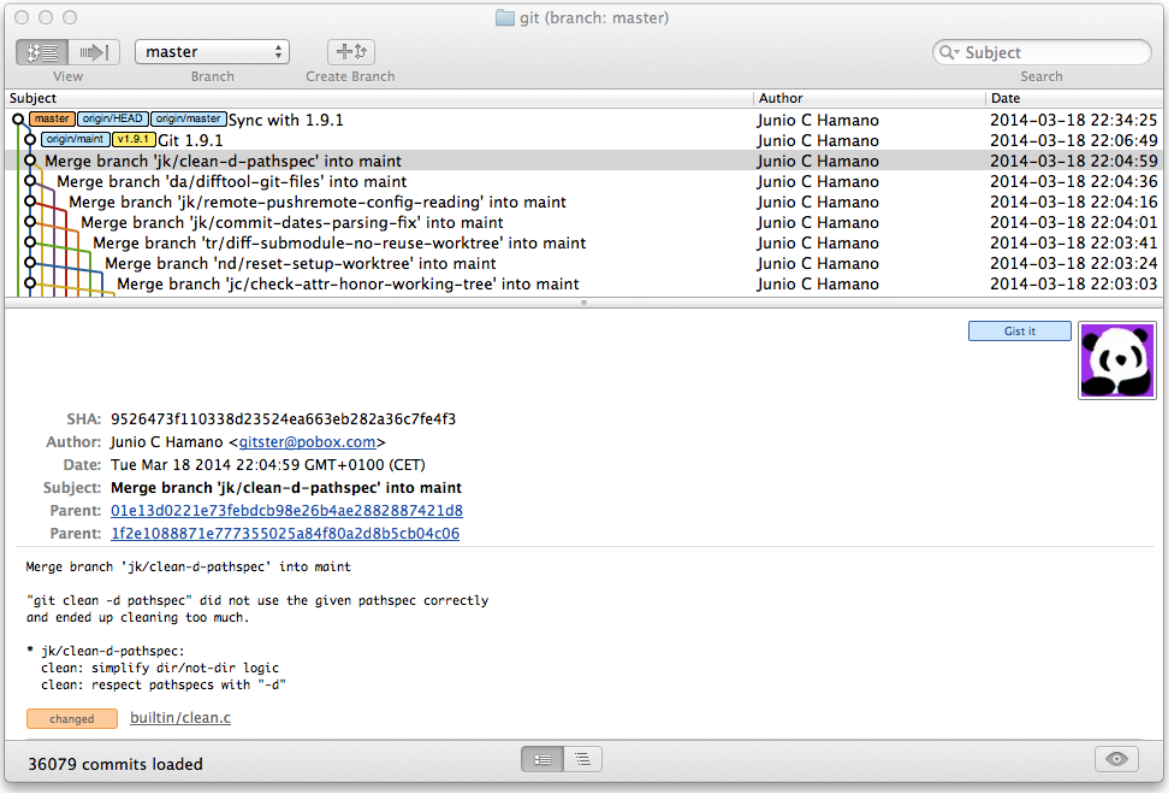


Abbildung 1.5.1: *History* in der grafischen Oberfläche von GitX