Arbeiten mit Git

28. April 2015 | Ingo Heimbach (i.heimbach@fz-juelich.de) | PGI/JCNS

Arbeiten mit Git

Vortragsteile

- Teil I:
  - Einführung in die **Grundlagen**
- Teil II:
  - Grundlagen sinnvoll in der Praxis **anwenden**
  - Mögliche **Workflows**
  - Vorstellung von Tools, die auf Git aufbauen

Arbeiten mit Git

Teil I: Einführung in Git

28. April 2015 | Ingo Heimbach (i.heimbach@fz-juelich.de)

Inhaltsverzeichnis

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 ...

Wieso Versionsverwaltung?

Vorteile für jeden einzelnen Entwickler

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

$\Rightarrow$ **Rückkehr** zu älteren Entwicklungsständen jederzeit möglich

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

Wieso Versionsverwaltung?

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 ($\rightarrow$ *Branching*)
  - **Zusammenarbeit** an einem Feature

Git Konzepte

Verteiltes Versionsverwaltungssystem

- Vollständige **Verteilung**

$\Rightarrow$ Jeder Entwickler hat lokale Kopie des gesamten *Repositories*

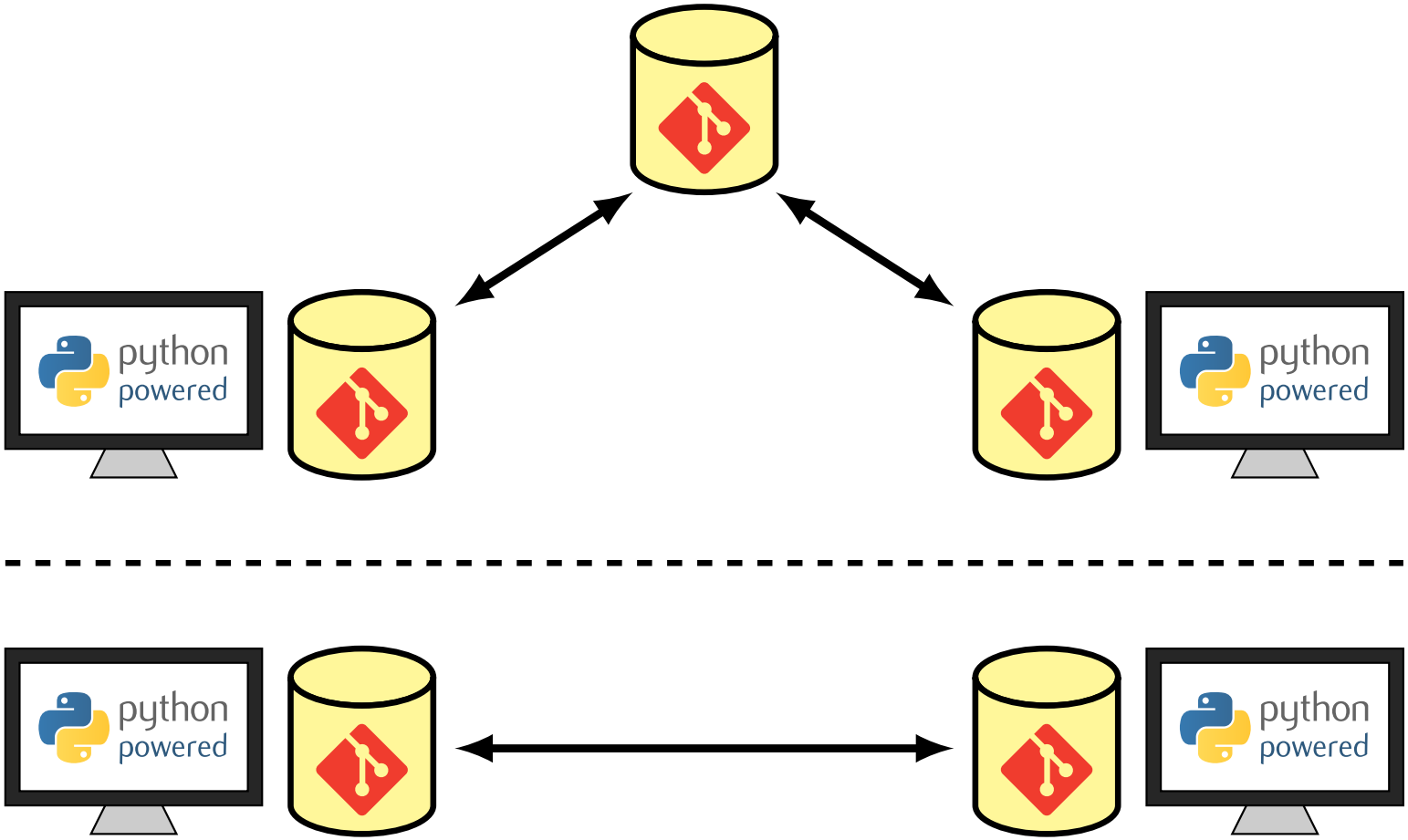
- **Repository**: Container mit gesamter Entwicklungsgeschichte

$\Rightarrow$ Arbeiten **ohne Netzwerkanbindung** möglich;
Netzwerk nur zur Synchronisierung notwendig

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

Git Konzepte

Verteiltes Versionsverwaltungssystem



Git Konzepte

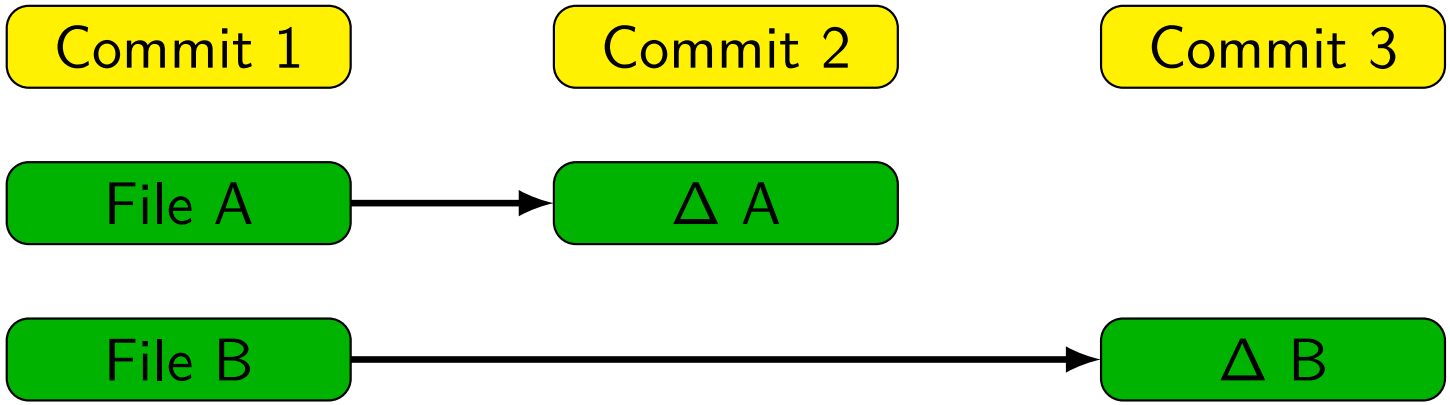
Snapshots statt Diffs

Konventioneller Ansatz:

- Konventionelle Versionierungssysteme sind **Datei-basiert**

$\Rightarrow$ **Revisionszähler** für jede Datei

$\Rightarrow$ Intern: Speicherung von **Diffs** zwischen Versionen einer Datei



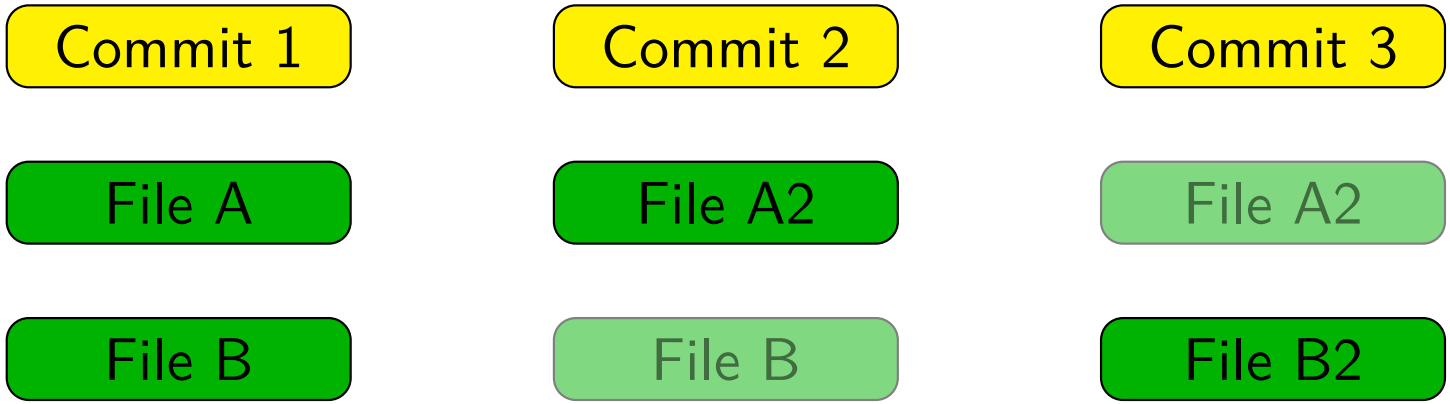
Git Konzepte

Snapshots statt Diffs

Ansatz bei Git:

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

$\Rightarrow$ Führt zur Vereinfachung anderer Konzepte (s. *Branching*)



Git Konzepte

Hashes

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

$\Rightarrow$ Direkte **Referenzierung** einzelner Objekte über Hash möglich

- Jede Projektänderung führt zur Änderung der Hash-Werte

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

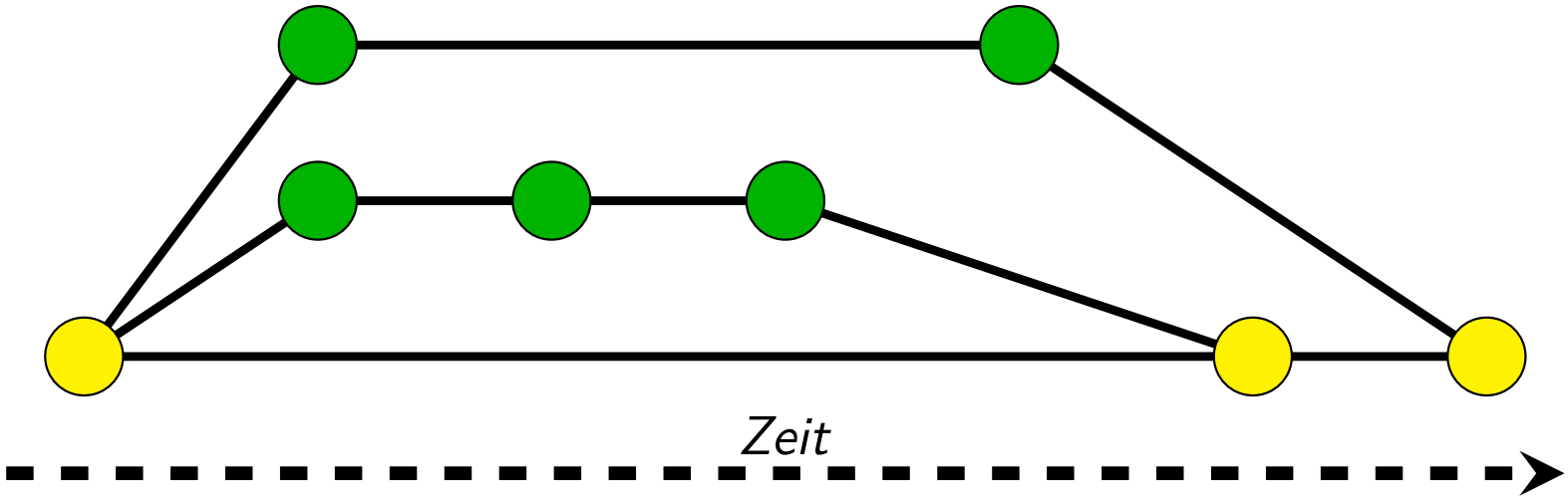
Git Konzepte

Nicht-lineare Entwicklung

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

$\Rightarrow$ Features, die parallel entwickelt werden, behindern sich nicht

$\Rightarrow$ Releases und Entwicklungsversionen können voneinander getrennt werden



Git Konzepte

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

$\Rightarrow$ Können über ihren Hash weiterhin adressiert werden (s. `reflog`)

Git Konzepte

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

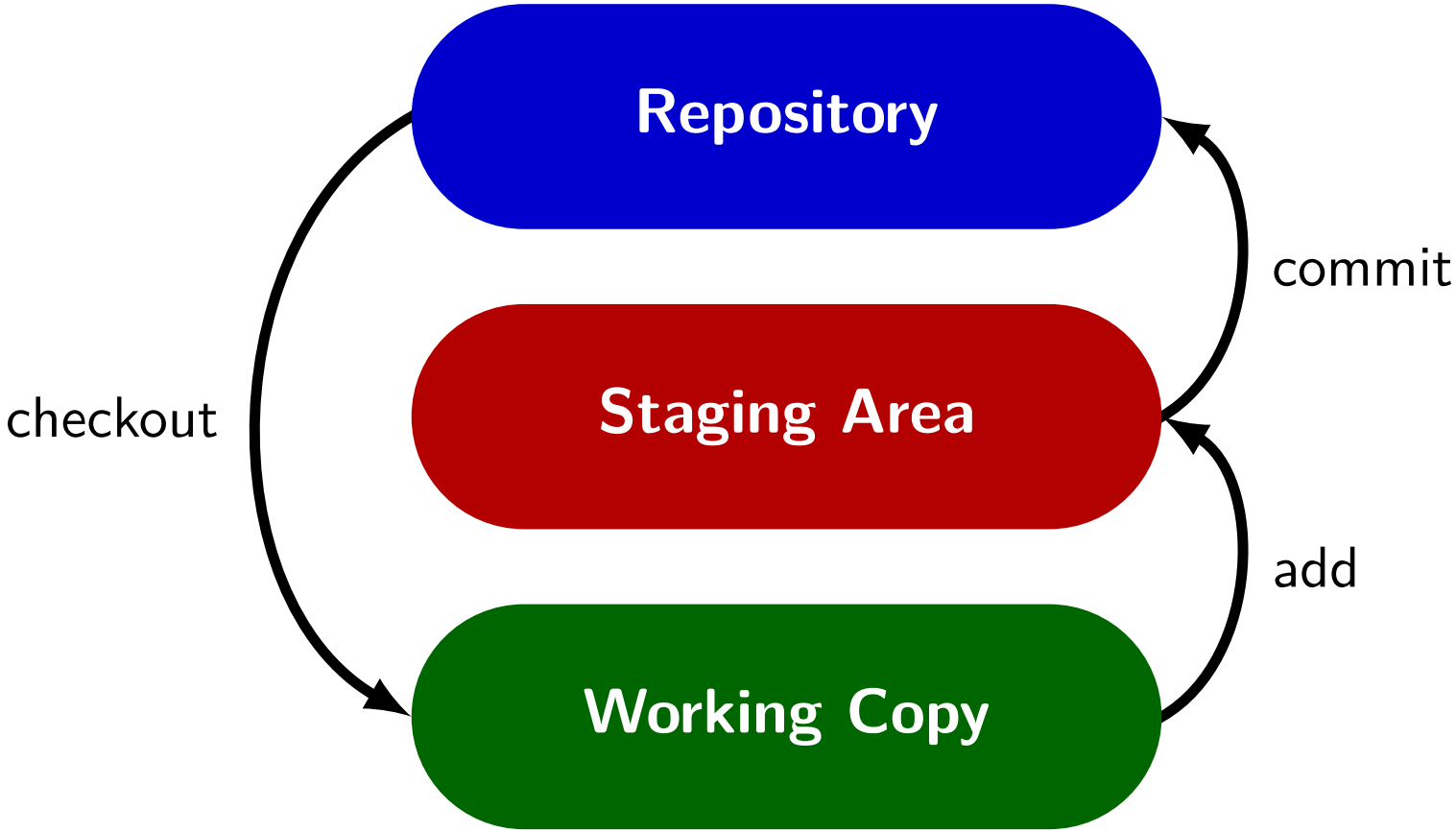
- Änderungen an bereits zur *Staging Area* hinzugefügten Dateien werden **nicht** automatisch übernommen

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

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

Git Konzepte

Staging Area



Git Basics

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*

Git Basics

Neues Git-*Repository* anlegen

- 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

Git Basics

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
```

Git Basics

Zentrales *Repository* initialisieren

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

$\Rightarrow$ 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
```

Git Basics

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

Git Basics

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

Git Basics

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 ($\rightarrow$ *Hashing*)

Git Basics

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!

Git Basics

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

Git Basics

History ab aktuellem *Commit* betrachten

- 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
```

Git Basics

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

  $\Rightarrow$ Wird unbrauchbar durch die Masse
- Lösung: Pflege einer **`.gitignore`-Datei**
  - Liste von ignorierten Dateien/Verzeichnissen

Git Basics

Dateien/Verzeichnisse ignorieren

- 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>
```

Git Basics

Branching

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

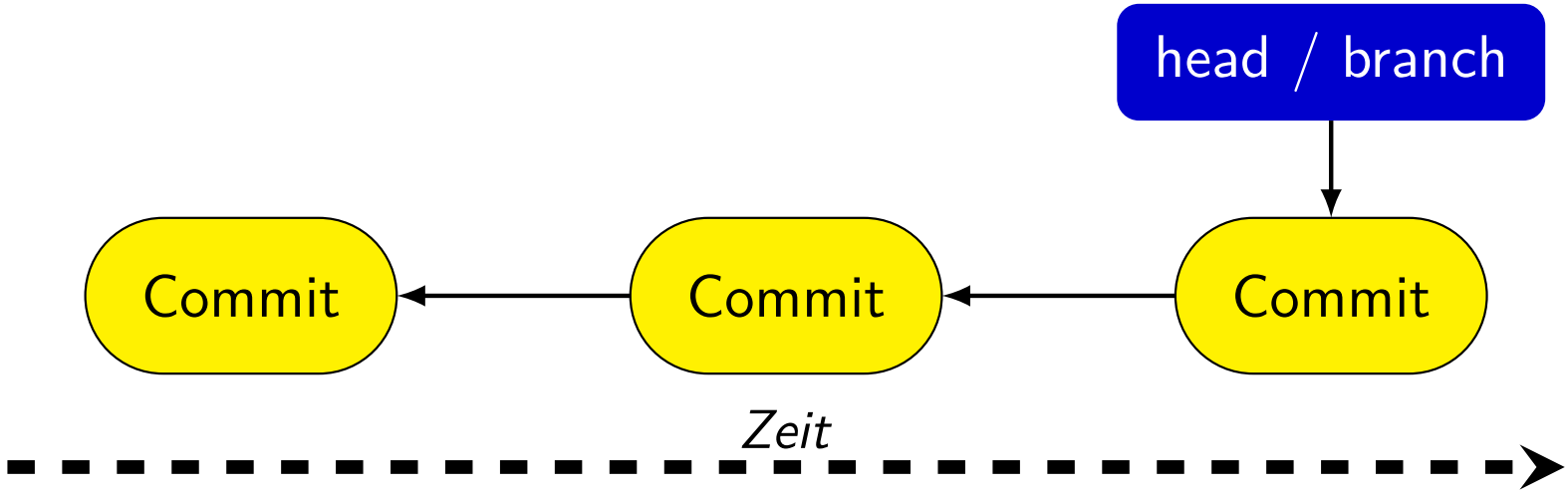
$\Rightarrow$ **Nicht-lineare Entwicklung** mit isolierten Arbeitsumgebungen

- Kann zu einer unübersichtlichen Entwicklungs-*History* führen
- Die *History* kann jedoch **nachträglich linearisiert** werden ($\rightarrow$ `git rebase`)
- Erzeugt zusätzlichen **Overhead** bei der Entwicklung, für die Vorteile aber vollkommen vertretbar
- *Branching* sollte in den normalen **Arbeitsfluss integriert** sein

Git Basics

Branching – Einige Interna zum besseren Verständnis

- Git speichert *Commits* als einfach verkettete Liste:



- Zeigerrichtung ist **entgegen** der Zeitrichtung

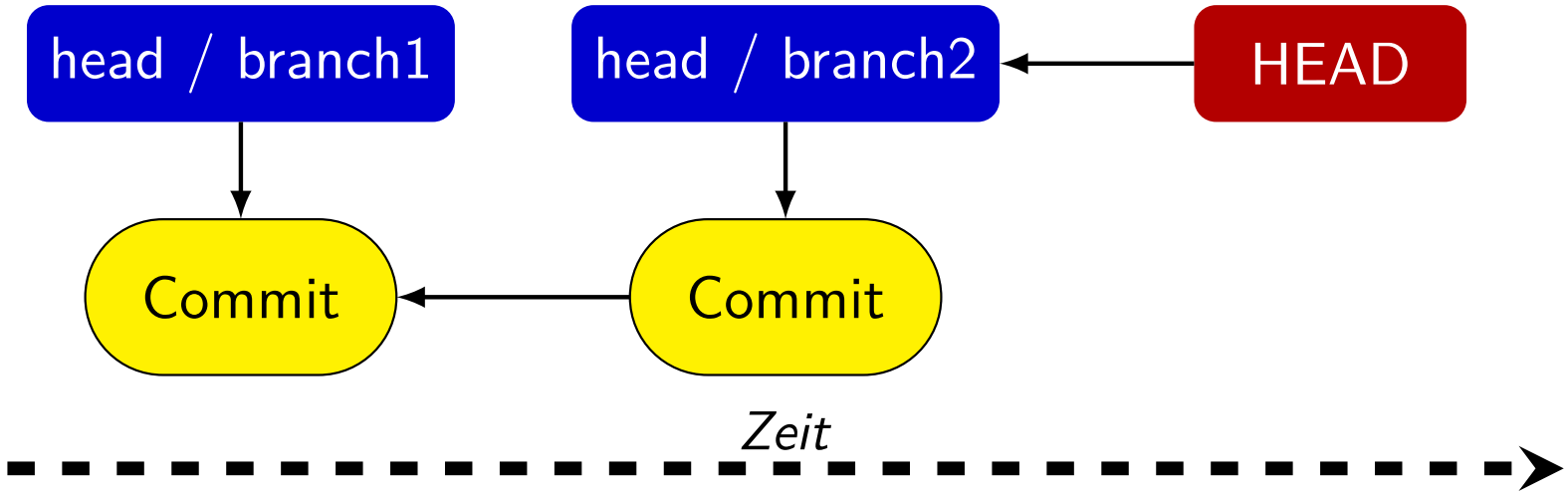
$\Rightarrow$ *Commits* kennen ihre Vergangenheit, aber nicht ihre Zukunft

- Interpretation des Listenanfangs (`head`-Zeiger) als *Branch*

Git Basics

Branching – Einige Interna zum besseren Verständnis

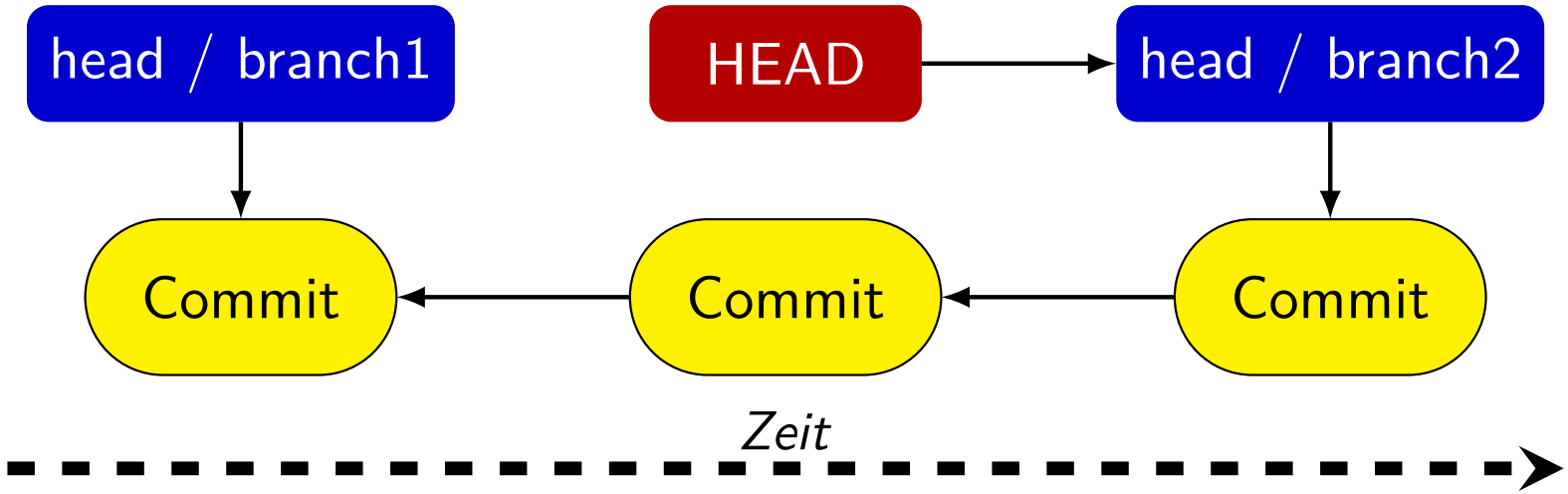
- Beliebige Anzahl von *Branches* in Git möglich
- Ein Zweig kann als **aktiver** gesetzt werden
- Ein neu erstellter *Commit* zeigt auf den letzten `head`-*Commit* des aktiven Zweiges
- *Branch*-Zeiger wird danach auf neuen *Commit* gesetzt
- Aktiver Zweig wird mit `HEAD` referenziert



Git Basics

Branching – Einige Interna zum besseren Verständnis

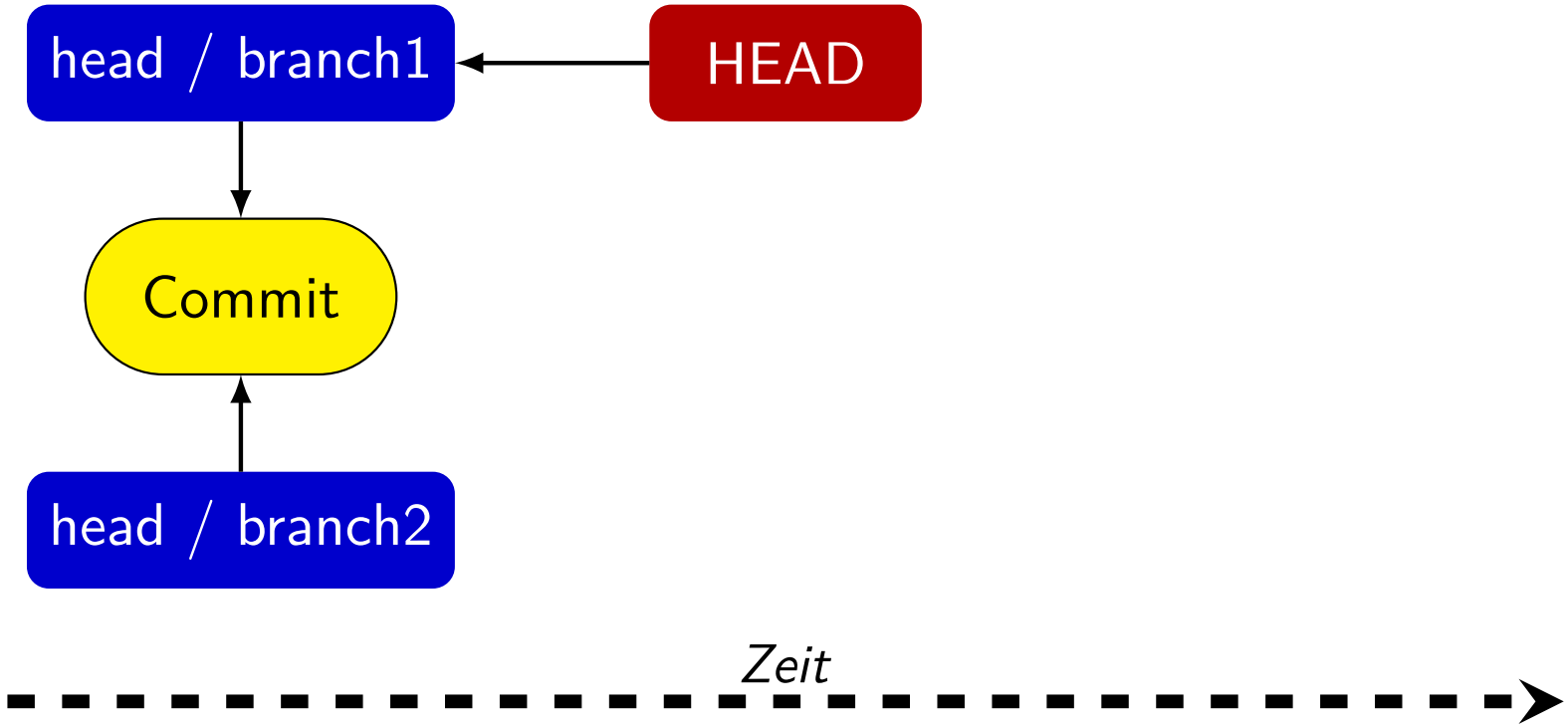
- Beliebige Anzahl von *Branches* in Git möglich
- Ein Zweig kann als **aktiver** gesetzt werden
- Ein neu erstellter *Commit* zeigt auf den letzten `head`-*Commit* des aktiven Zweiges
- *Branch*-Zeiger wird danach auf neuen *Commit* gesetzt
- Aktiver Zweig wird mit **`HEAD`** referenziert



Git Basics

Branching – Einige Interna zum besseren Verständnis

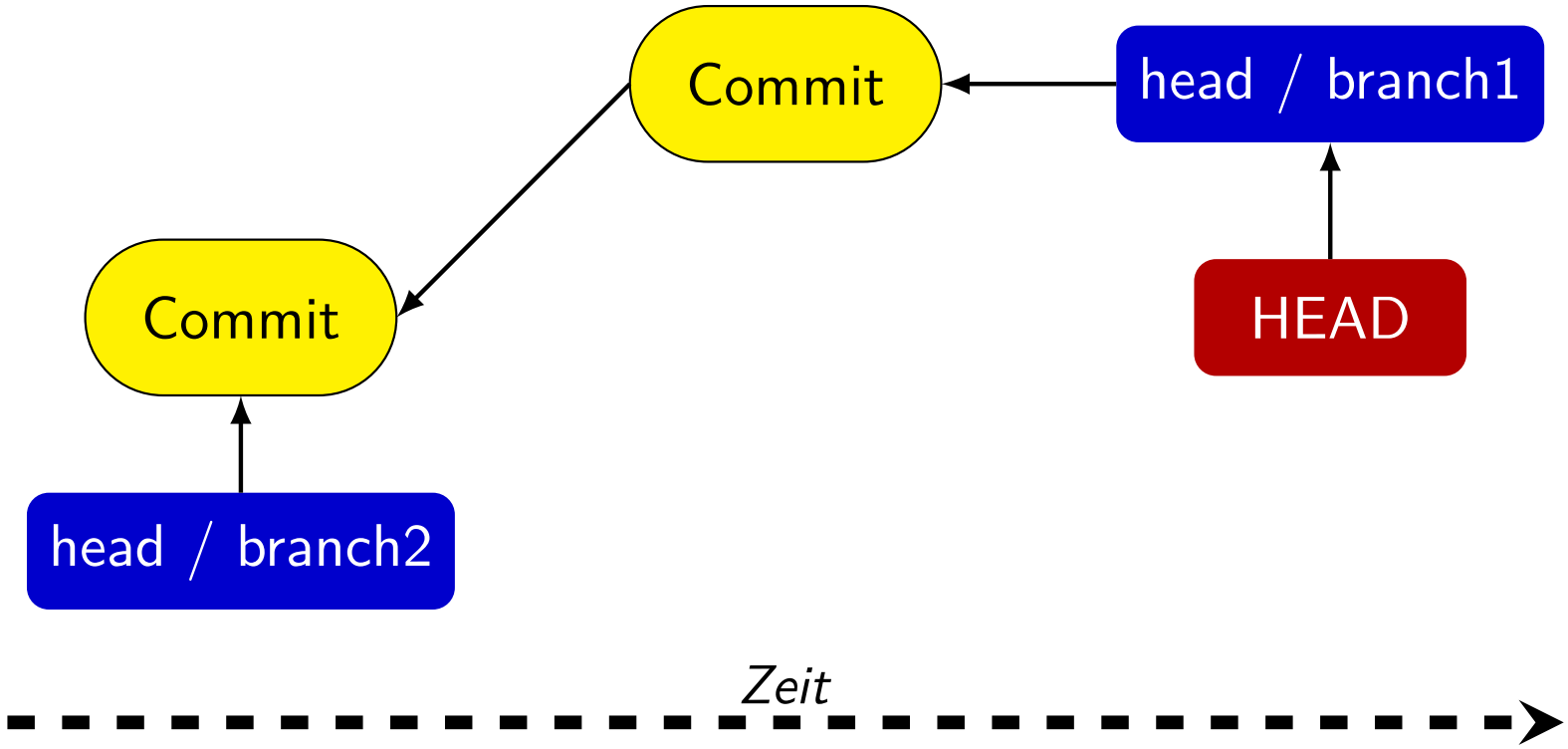
- Die Nutzung mehrerer *Branches* ermöglicht somit eine **Verzweigung der Liste**



Git Basics

Branching – Einige Interna zum besseren Verständnis

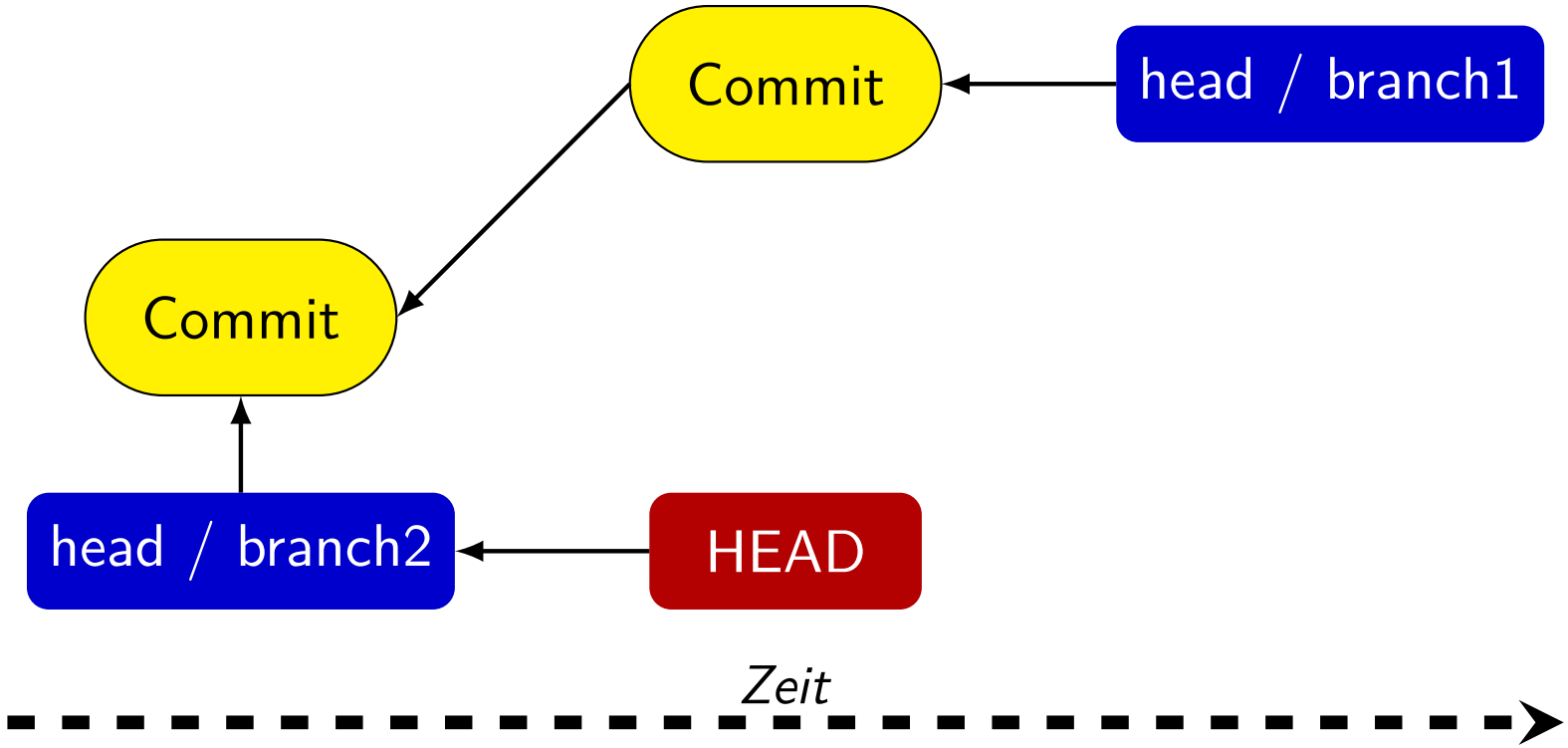
- Die Nutzung mehrerer *Branches* ermöglicht somit eine **Verzweigung der Liste**



Git Basics

Branching – Einige Interna zum besseren Verständnis

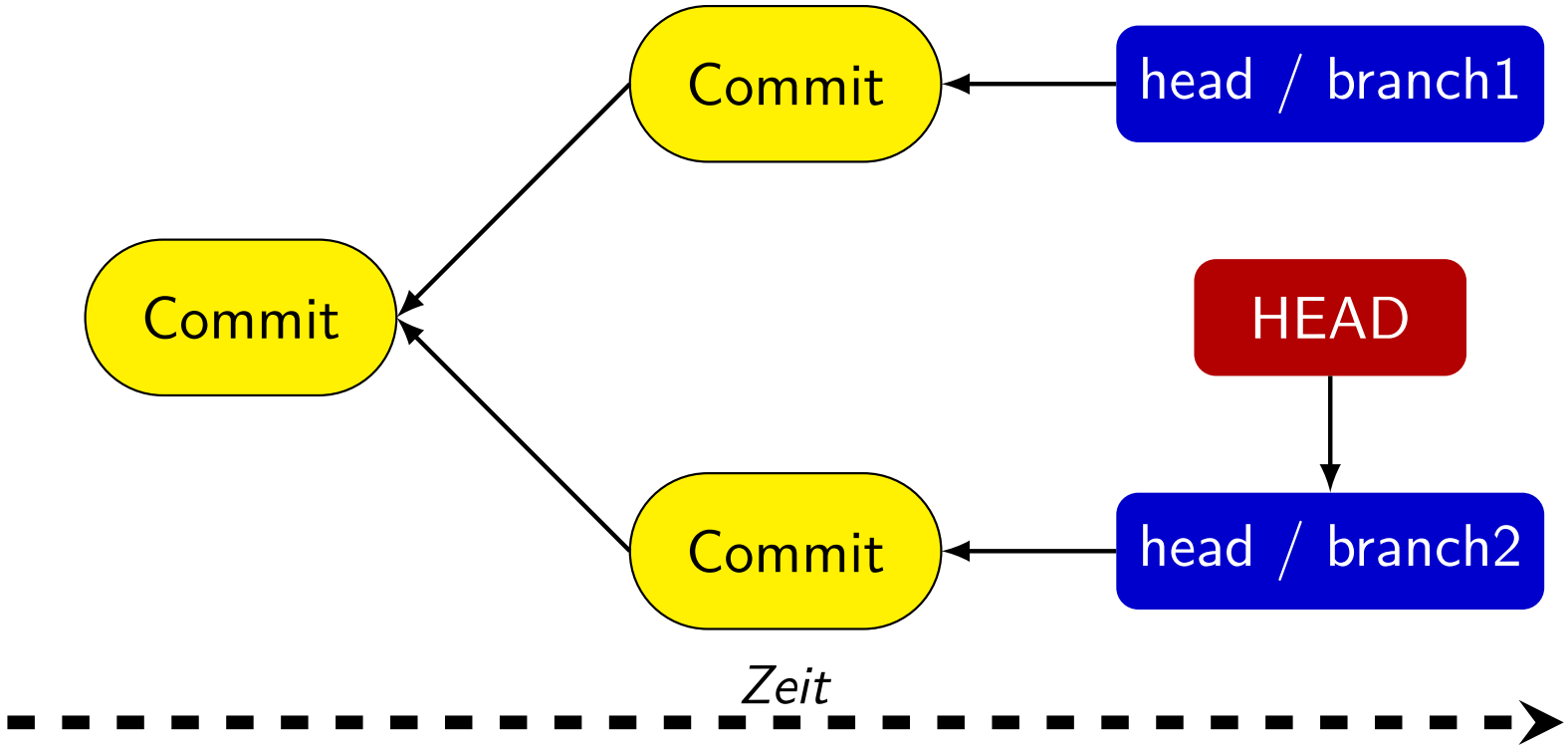
- Die Nutzung mehrerer *Branches* ermöglicht somit eine **Verzweigung der Liste**



Git Basics

Branching – Einige Interna zum besseren Verständnis

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

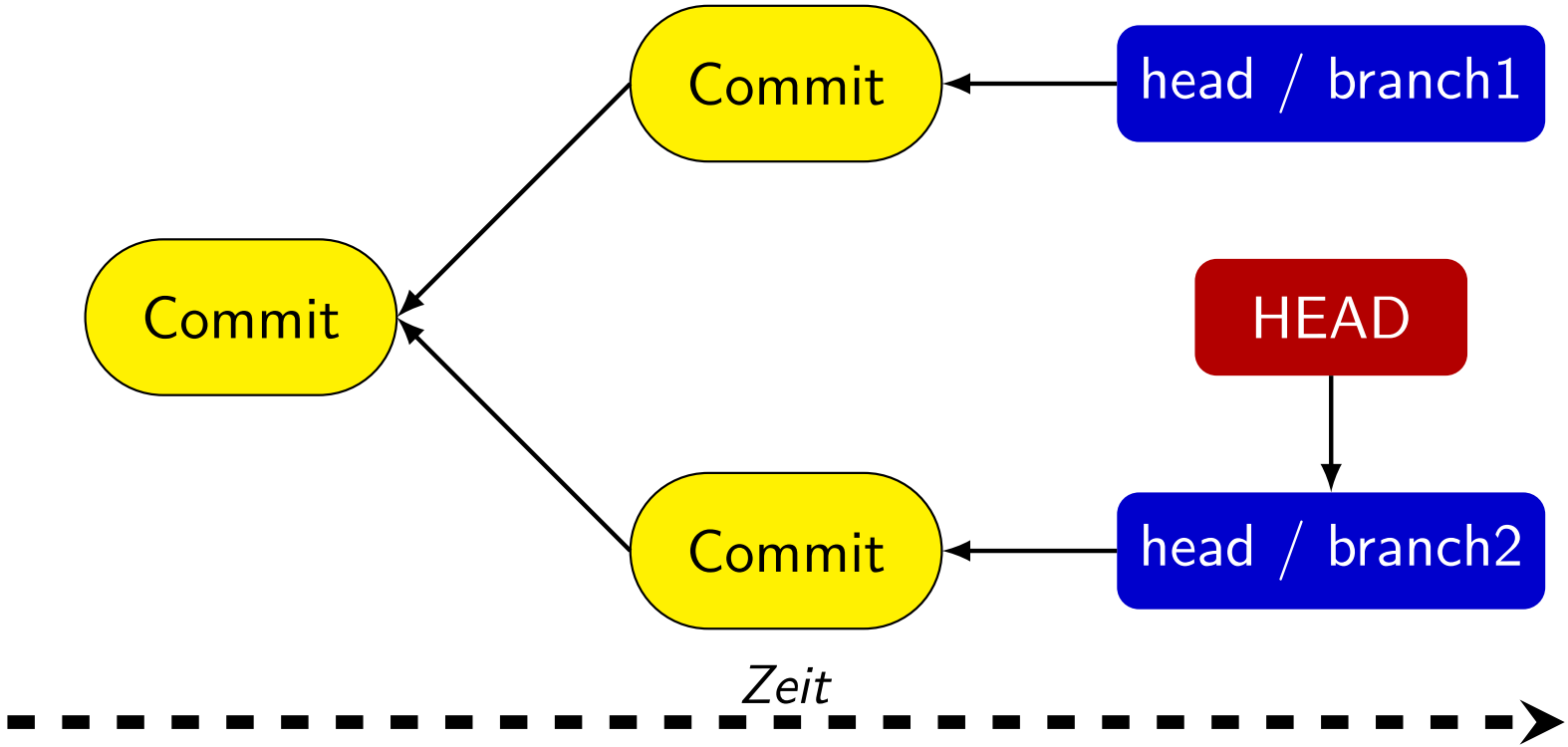


Git Basics

Branching – Einige Interna zum besseren Verständnis

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

$\Rightarrow$ Zusammenführen von Verzweigungen möglich

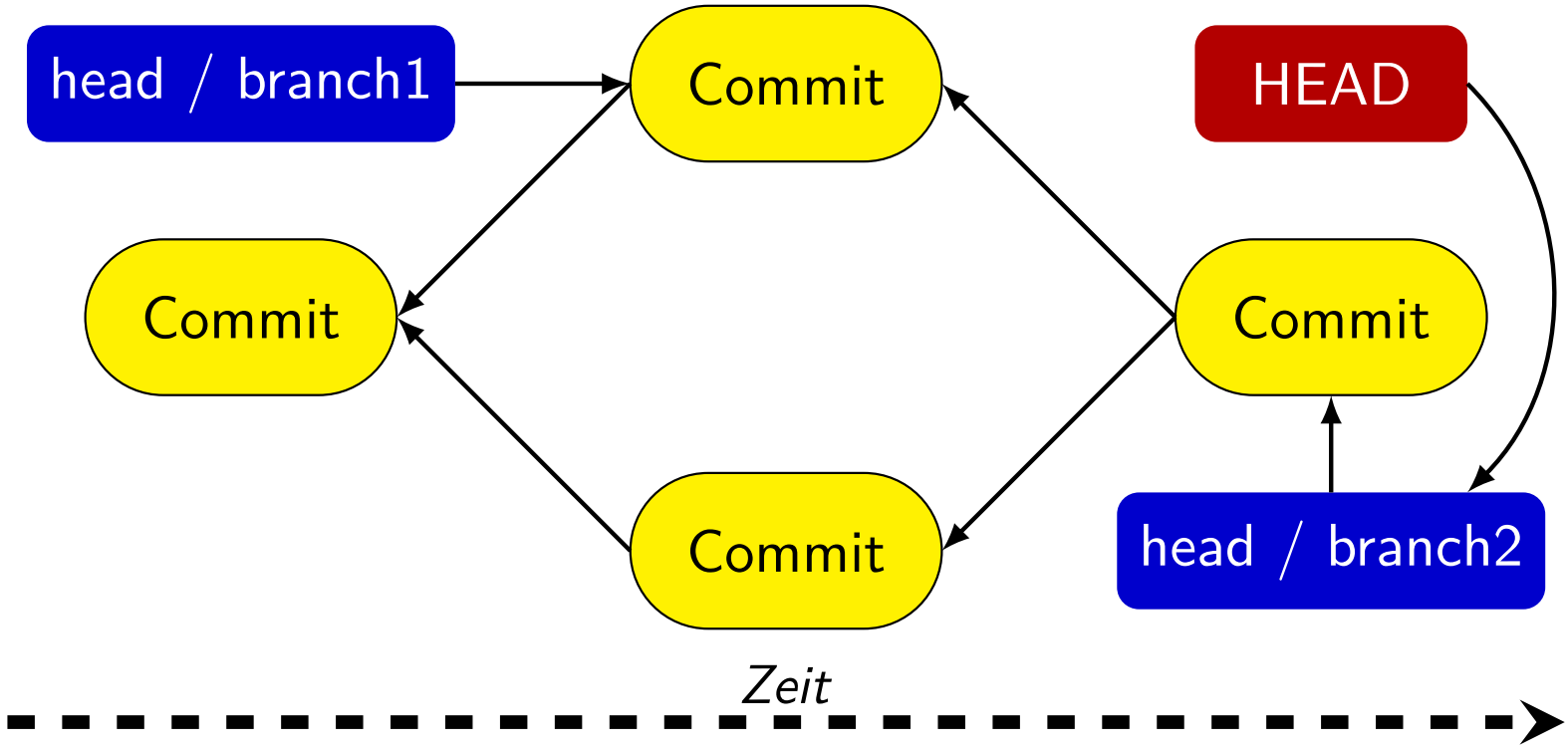


Git Basics

Branching – Einige Interna zum besseren Verständnis

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

$\Rightarrow$ Zusammenführen von Verzweigungen möglich



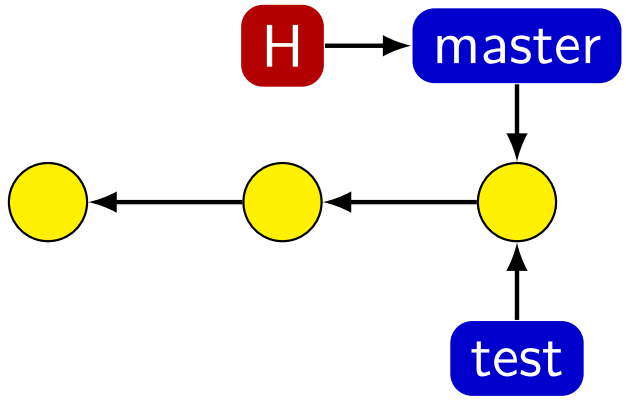
Git Basics

Branching – 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



Git Basics

Branching – Zweig erstellen

- Auflisten aller schon vorhandenen Zweige über

```
git branch -a
```

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

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

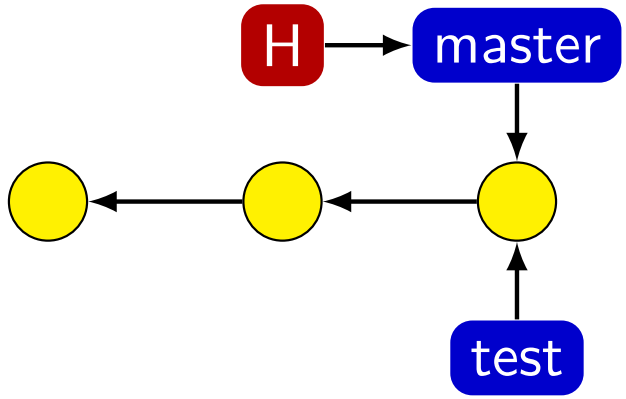
  - Aktion nicht möglich bei resultierendem Datenverlust
  - Datenverlust kann mit -D erzwungen werden

Git Basics

Branching – Aktiven Zweig setzen

- Auf einen Zweig wechseln ($\rightarrow$ `HEAD` setzen):

```
git checkout <Zweigname>
```



- Zweig erstellen und direkt als aktiv setzen:

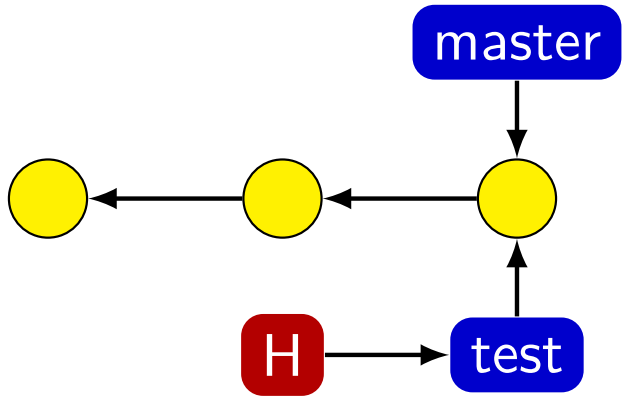
```
git checkout -b <Zweigname>
```

Git Basics

Branching – Aktiven Zweig setzen

- Auf einen Zweig wechseln ($\rightarrow$ `HEAD` setzen):

```
git checkout <Zweigname>
```



- Zweig erstellen und direkt als aktiv setzen:

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

Git Basics

Branching – 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

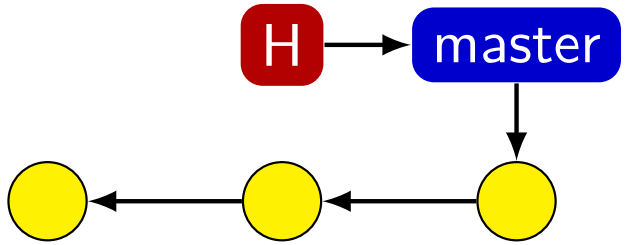
- Beispiele:

```
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
```

Git Basics

Branching – Detached HEAD

- Praktisches Beispiel:

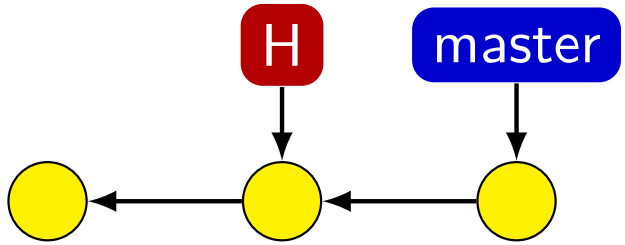


Git Basics

Branching – Detached HEAD

- Praktisches Beispiel:

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

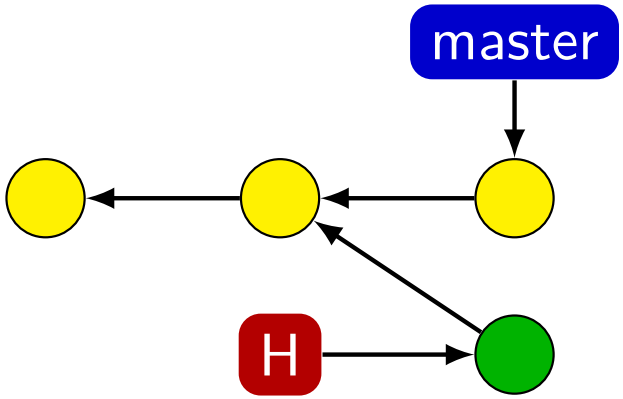


Git Basics

Branching – Detached HEAD

- Praktisches Beispiel:

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

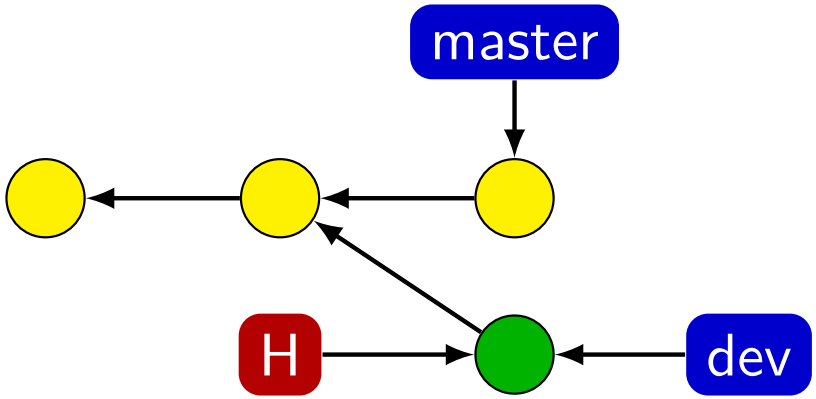


Git Basics

Branching – Detached HEAD

- Praktisches Beispiel:

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



Git Basics

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

    $\Rightarrow$ Linearisierung der *History*

Git Basics

Zweige zusammenführen – *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

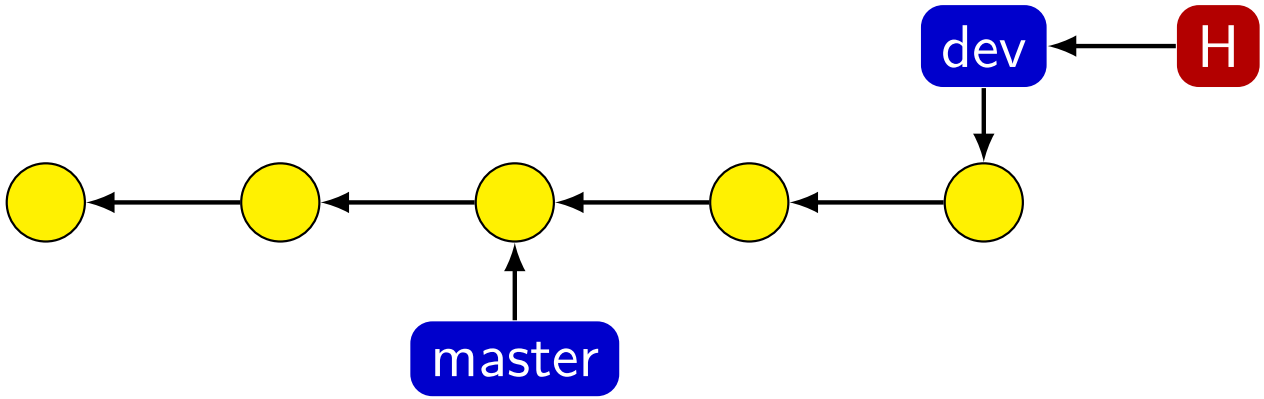
    ⇒ Setze aktiven Zweig auf die Position des angegebenen vor
  - **Three-Way-Merge**:
    - Wird ansonsten eingesetzt

    ⇒ Findet die gemeinsame Wurzel und macht darüber einen Abgleich

Git Basics

Zweige zusammenführen – *Merging*

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

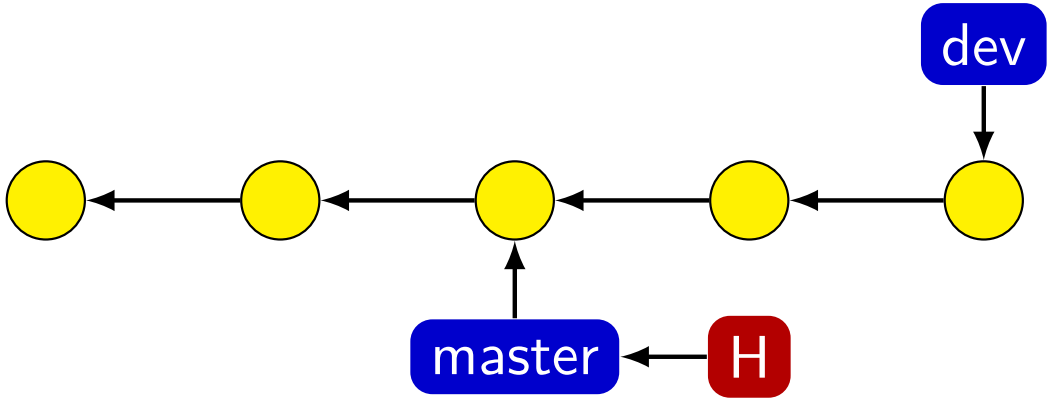


Git Basics

Zweige zusammenführen – *Merging*

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

```
git checkout master
```

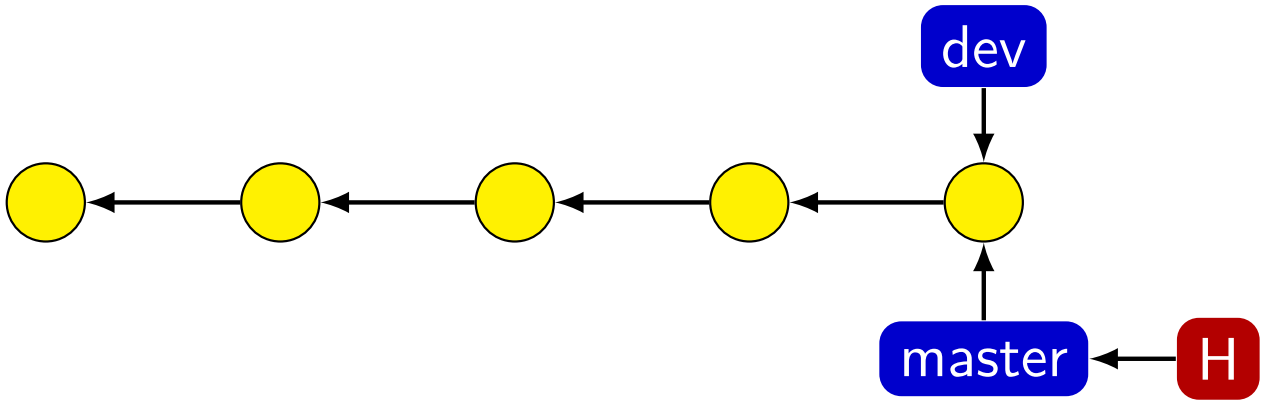


Git Basics

Zweige zusammenführen – *Merging*

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

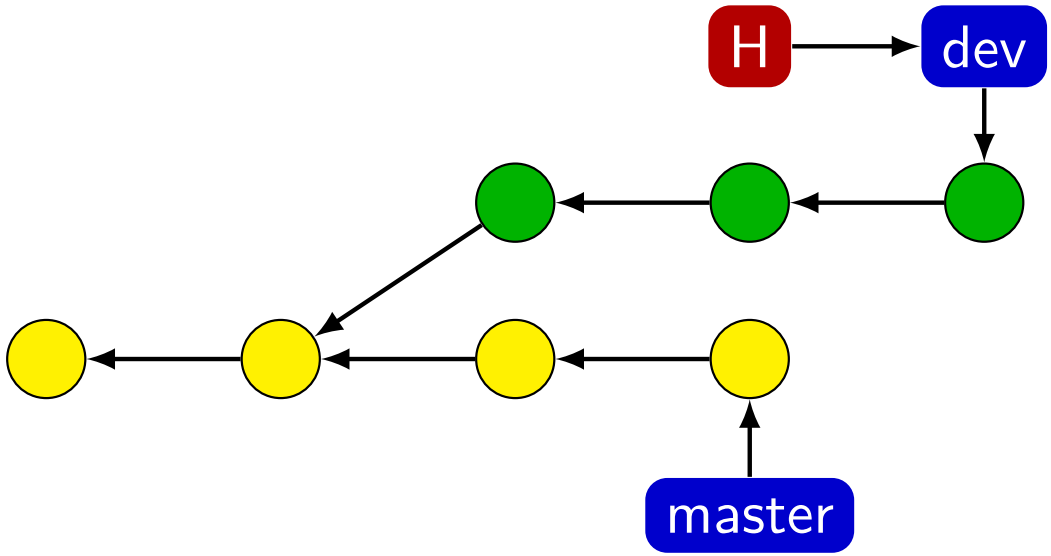
```
git checkout master
git merge dev
```



Git Basics

Zweige zusammenführen – *Merging*

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

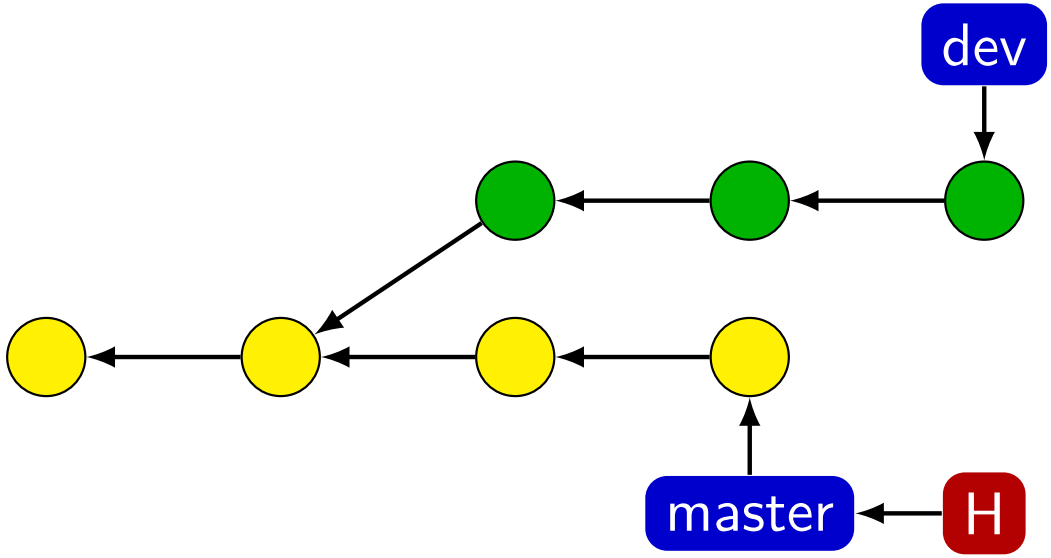


Git Basics

Zweige zusammenführen – *Merging*

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

```
git checkout master
```

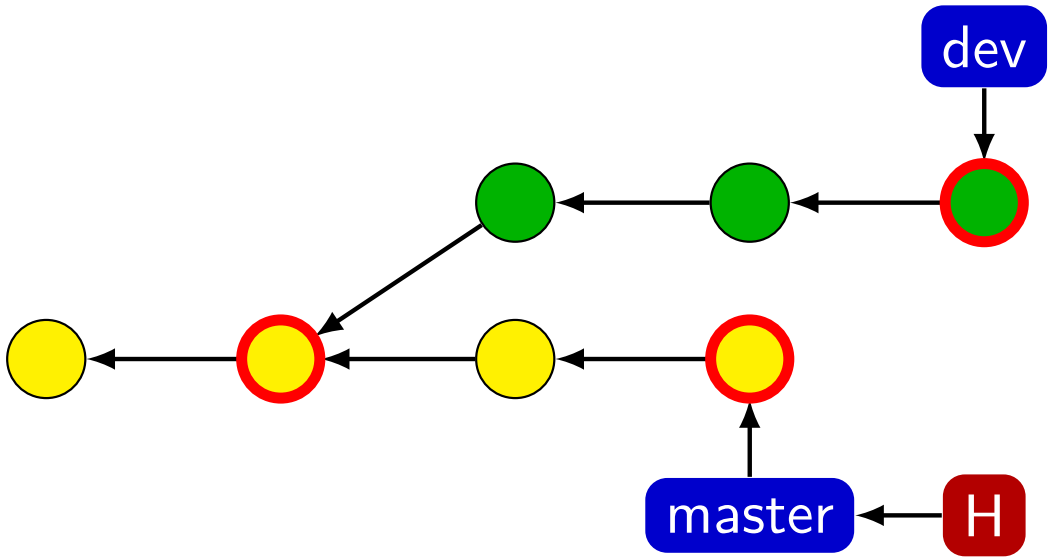


Git Basics

Zweige zusammenführen – *Merging*

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

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

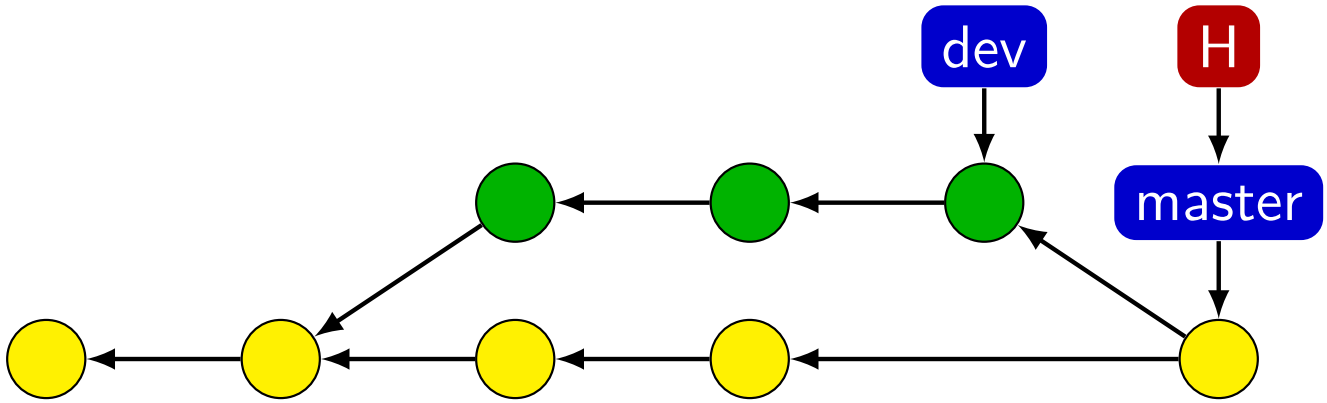


Git Basics

Zweige zusammenführen – *Merging*

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

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



Git Basics

Zweige zusammenführen – *Rebasing*

- Aktiven Zweig auf einen anderen Zweig neu aufsetzen:

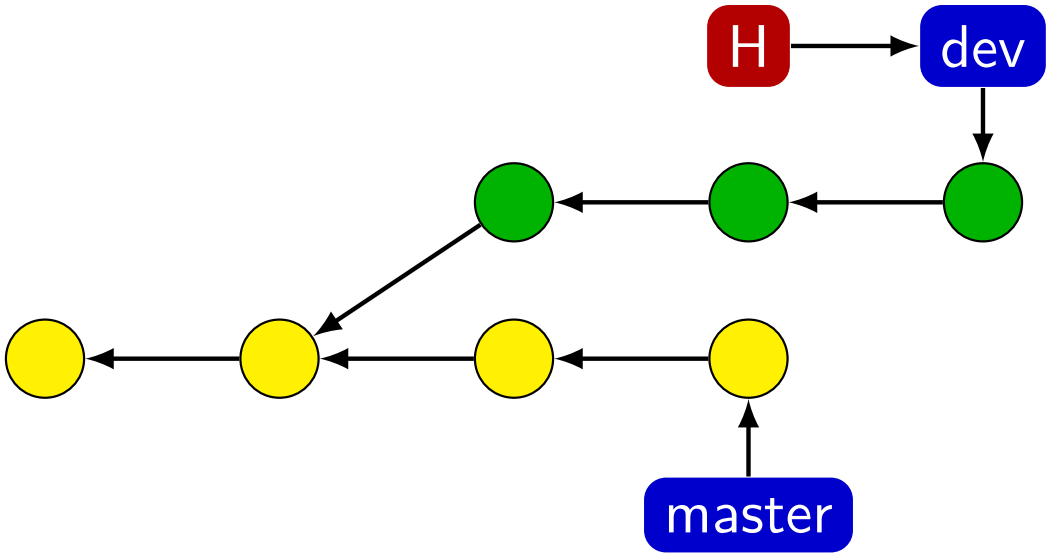
```
git rebase <Zweig-Name>
```

- **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**

Git Basics

Zweige zusammenführen – *Rebasing*

- Beispiel:

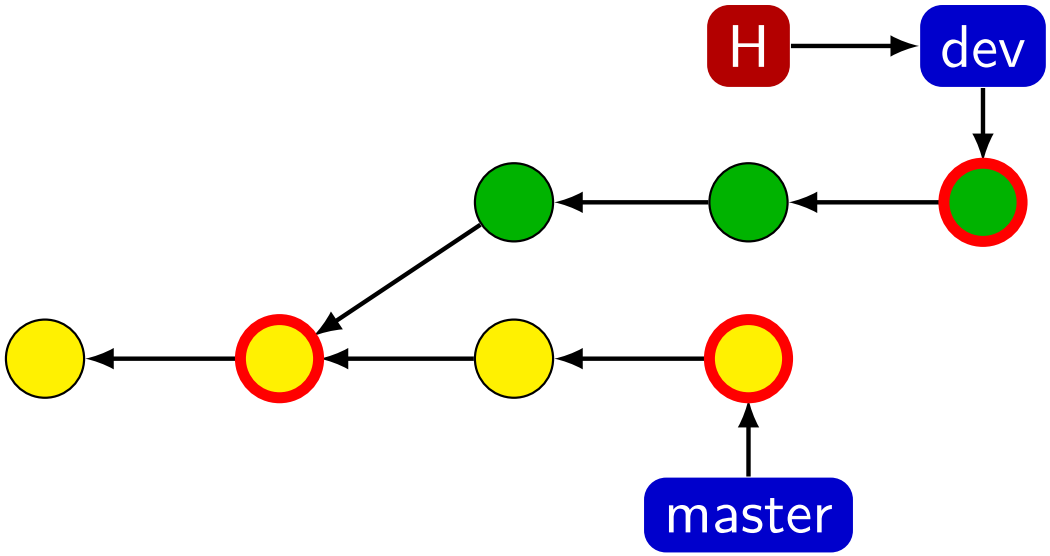


Git Basics

Zweige zusammenführen – *Rebasing*

- Beispiel:

```
git rebase master
```

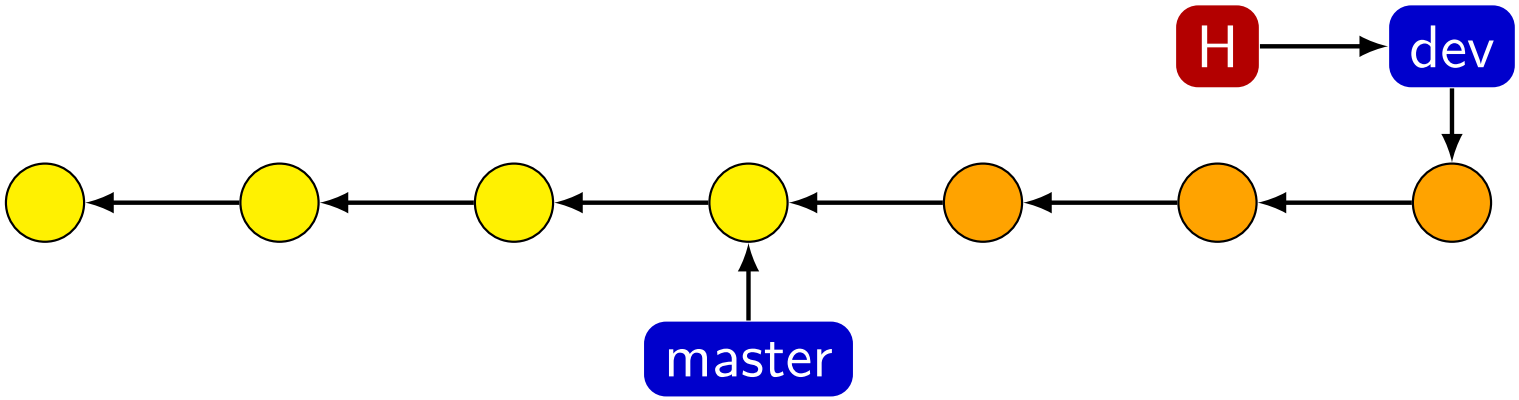


Git Basics

Zweige zusammenführen – *Rebasing*

- Beispiel:

```
git rebase master
```

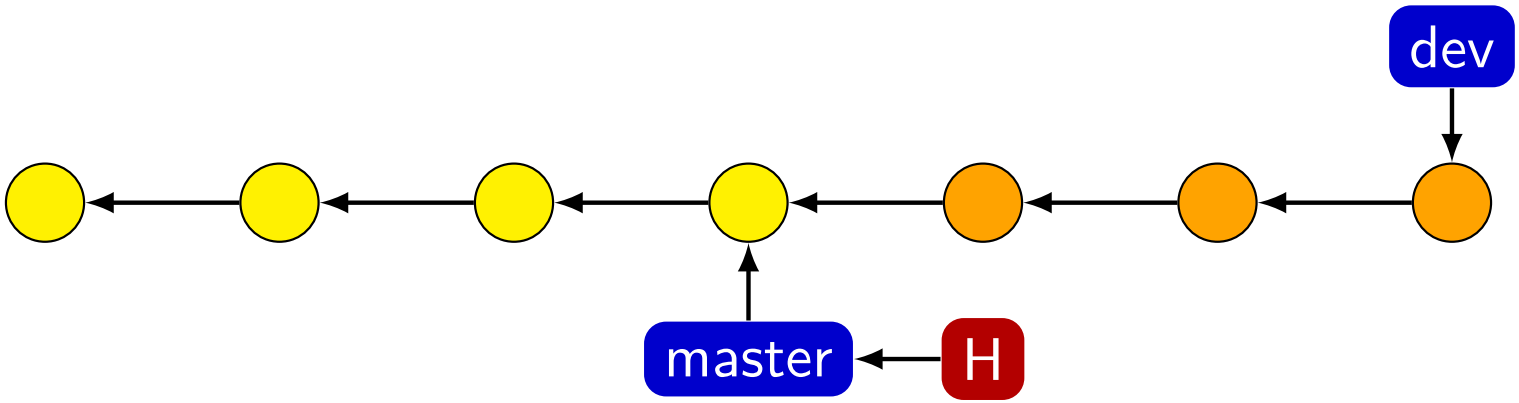


Git Basics

Zweige zusammenführen – *Rebasing*

- Beispiel:

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

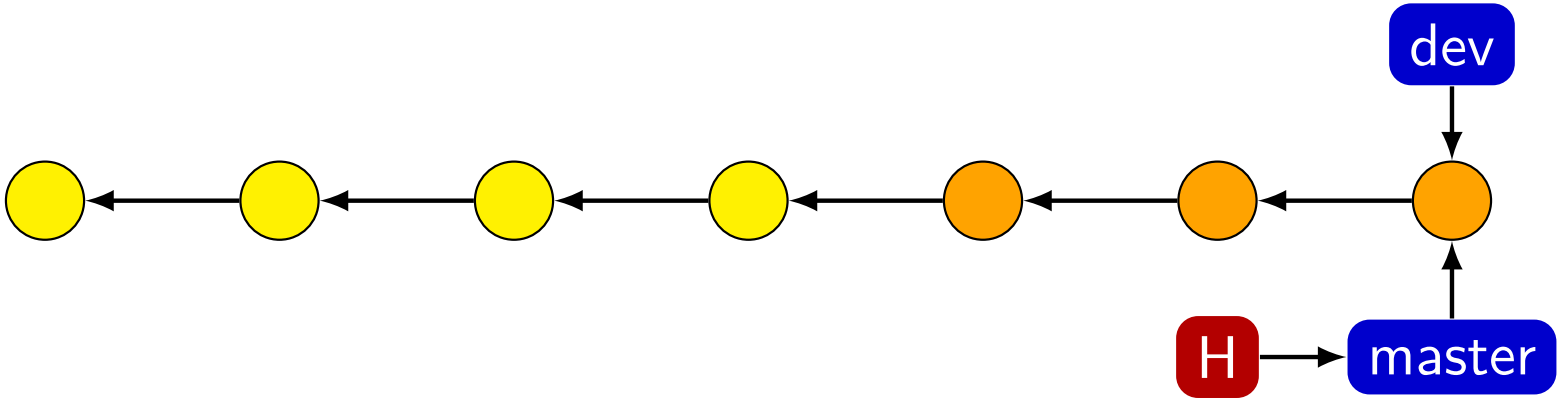


Git Basics

Zweige zusammenführen – *Rebasing*

- Beispiel:

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



Git Basics

Zweige zusammenführen – *Advanced Rebasing*

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

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

`<Checkout>`: Wird vor dem *Rebase* ausgecheckt

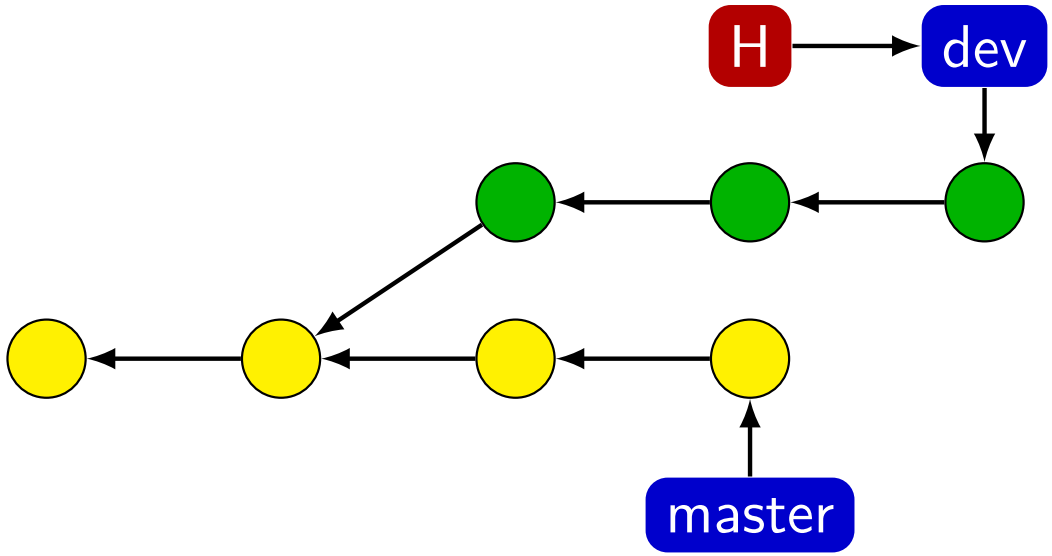
`<Basis>`: Zweig, auf dem die Patches aufgespielt werden (Default: `<Zweig-Name>`)

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

Git Basics

Zweige zusammenführen – *Advanced Rebasing*

- Beispiel:

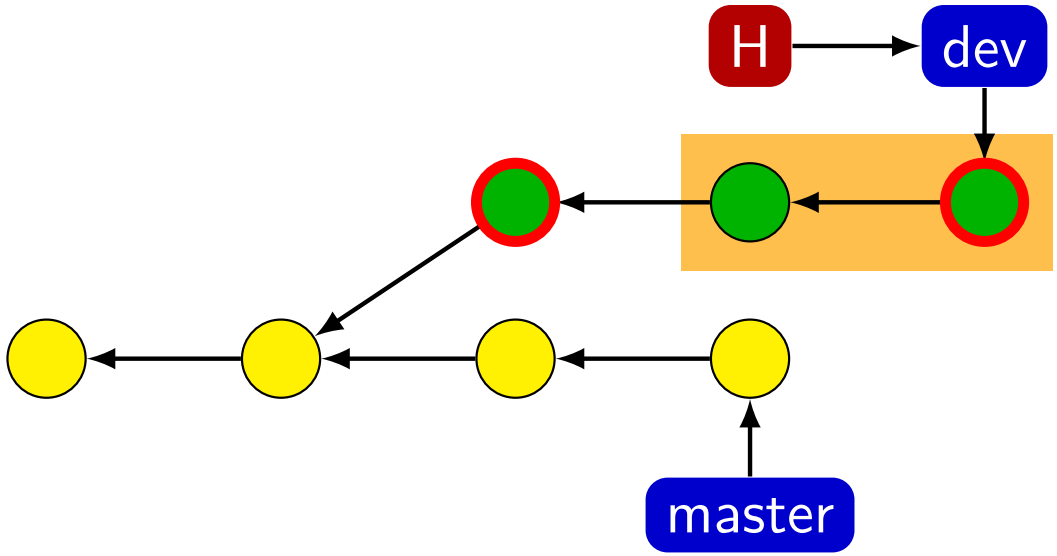


Git Basics

Zweige zusammenführen – *Advanced Rebasing*

- Beispiel:

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

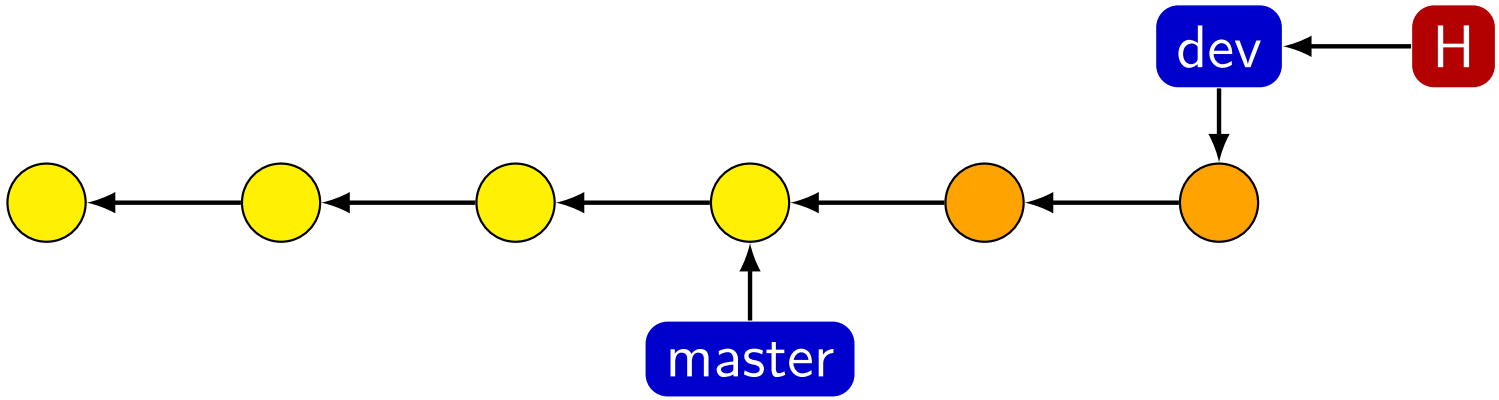


Git Basics

Zweige zusammenführen – *Advanced Rebasing*

- Beispiel:

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



Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

Rebasing von *Merge-Commits*

Rebasing erzeugt (per Default) ein **vollständig lineares** Ergebnis!

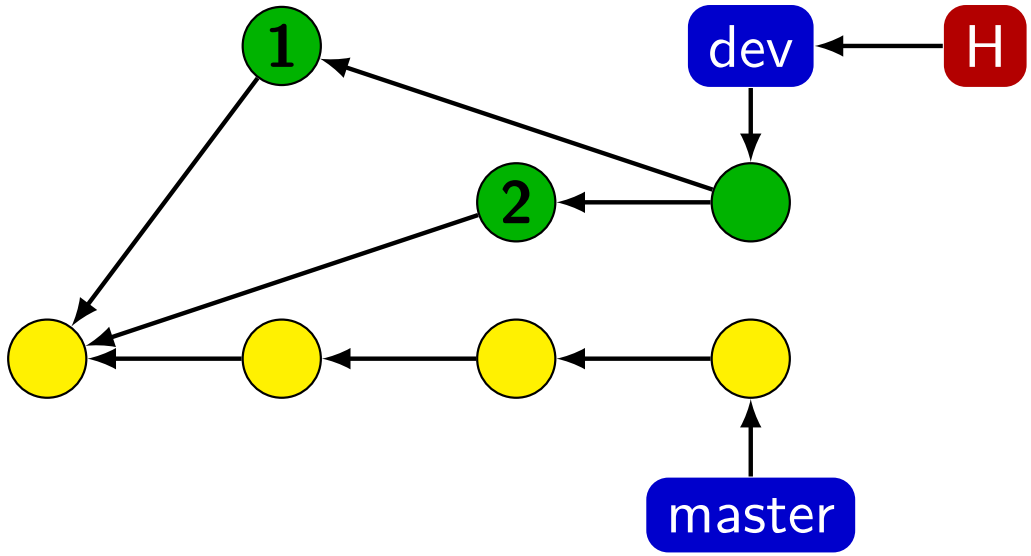
$\Rightarrow$ *Merge-Commits* werden beim *Rebasing* **verworfen**

- Aus allen anderen *Commits* werden *Diff-Patches* extrahiert, die **chronologisch** aufgespielt werden
- Option `-p`: *Merge-Commits* werden **nachgebildet** (`-p` $\widehat{=}$ `--preserve-merges`)

Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel:

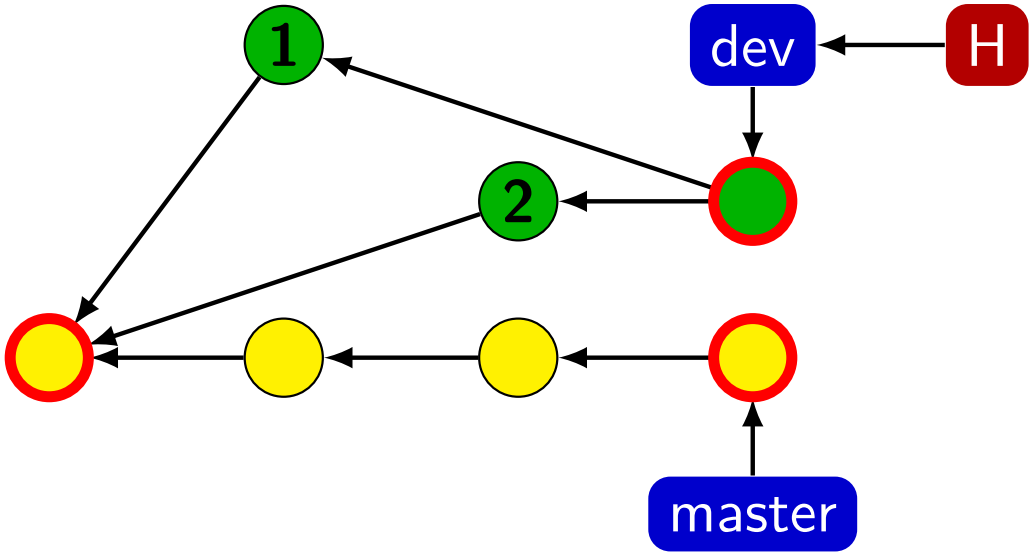


Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel:

```
git rebase master
```

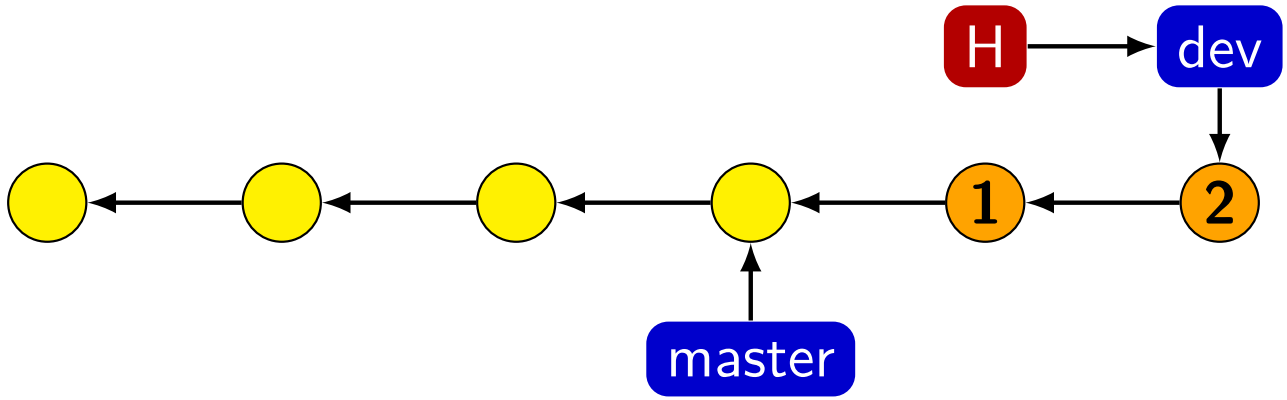


Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel:

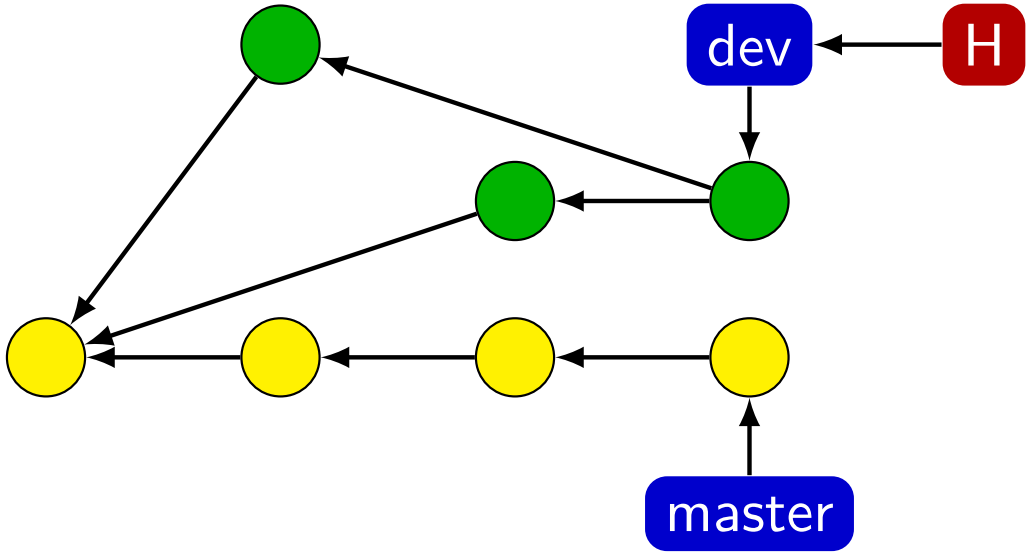
```
git rebase master
```



Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel mit Option `-p`:

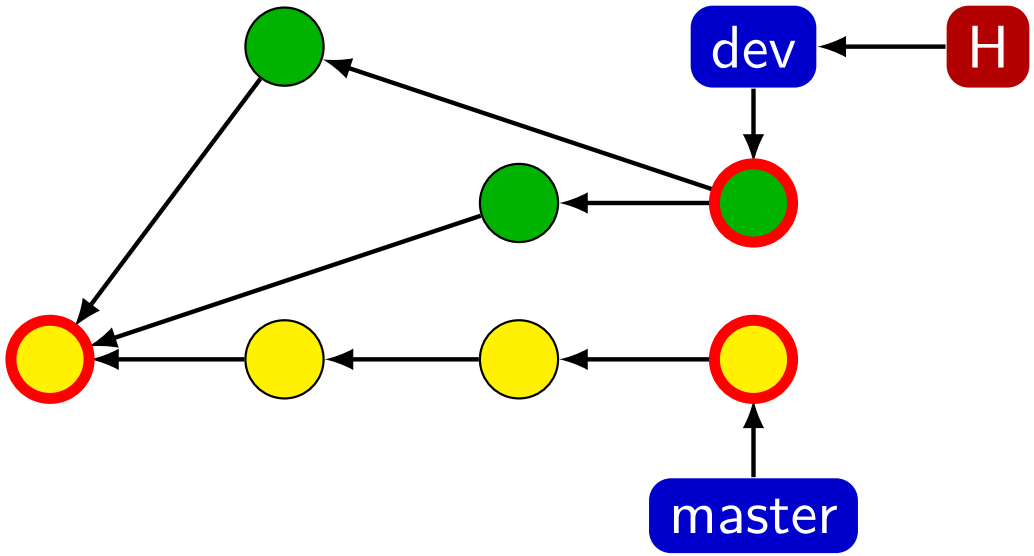


Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel mit Option `-p`:

```
git rebase -p master
```

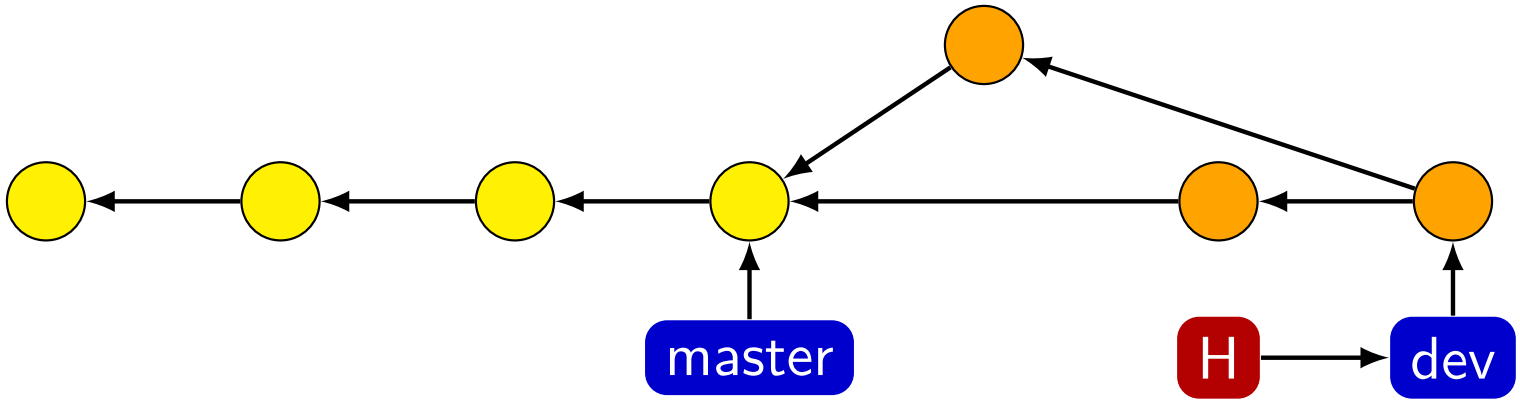


Git Basics

Zweige zusammenführen – *Rebasing* von *Merge-Commits*

- Beispiel mit Option `-p`:

```
git rebase -p master
```



Git Basics

Zweige zusammenführen – Konflikte

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

$\Rightarrow$ 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*)

Git Basics

Zweige zusammenführen – Konflikte

- 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

Git Basics

Zweige zusammenführen – Konflikte bei *Merge*

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

$\Rightarrow$ 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

Git Basics

Zweige zusammenführen – Konflikte bei *Rebase*

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

$\Rightarrow$ 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*

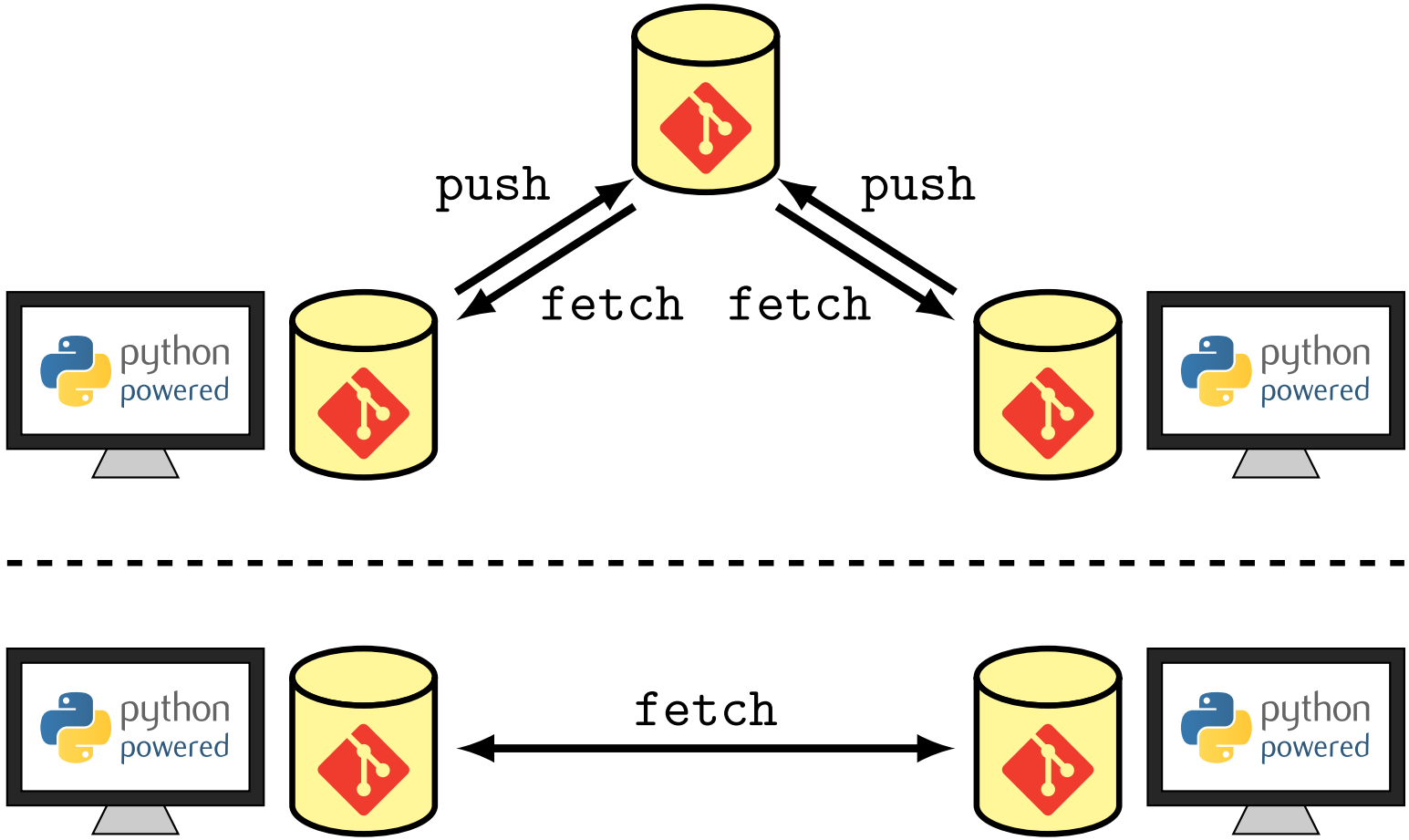
Git Basics

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

Git Basics

Arbeiten mit *Remote-Repositories* – zentral vs. dezentral



Git Basics

Arbeiten mit *Remote-Repositories* – 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>
```

Git Basics

Arbeiten mit *Remote-Repositories – 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* ...

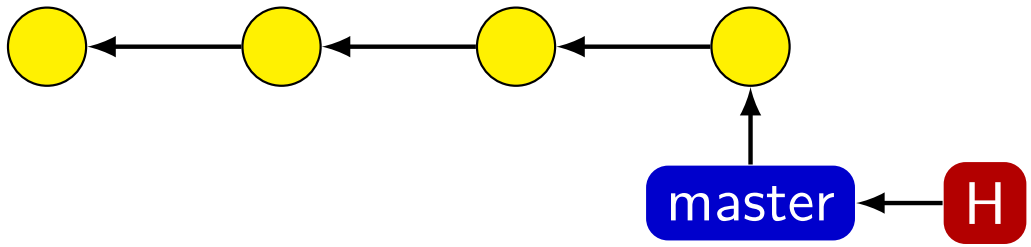
⇒ Der **lokale** Stand bleibt **unverändert** erhalten!

- Mit `--all` werden alle eingetragenen *Repositories* abgefragt

Git Basics

Arbeiten mit *Remote-Repositories – Fetching*

- Beispiel:

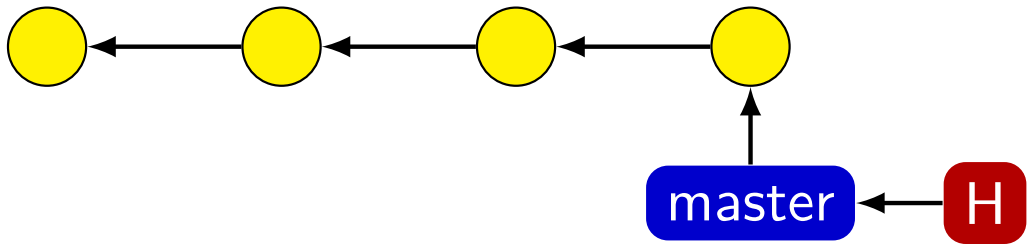


Git Basics

Arbeiten mit *Remote-Repositories – Fetching*

- Beispiel:

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

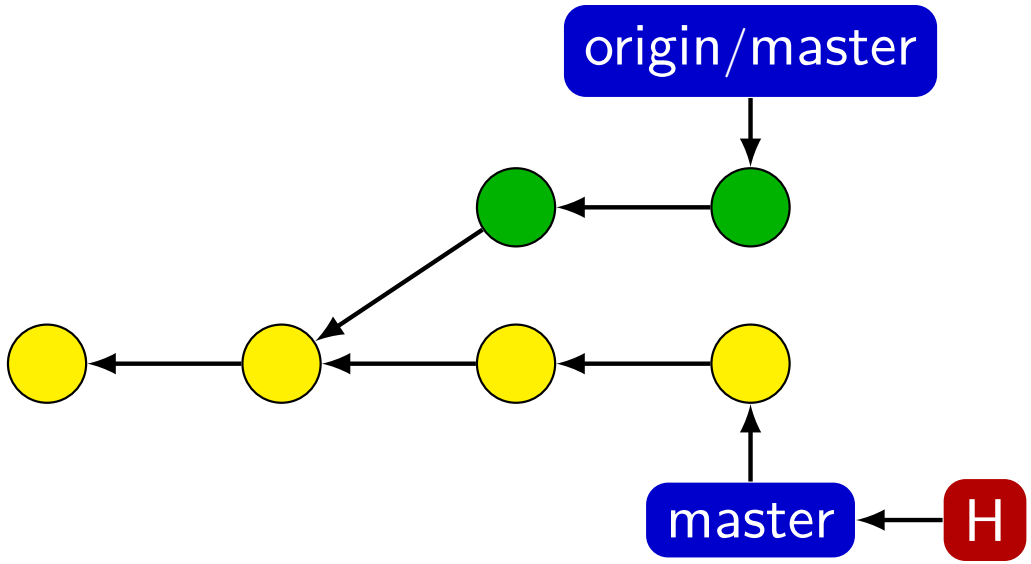


Git Basics

Arbeiten mit *Remote-Repositories – Fetching*

- Beispiel:

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



Git Basics

Arbeiten mit *Remote-Repositories* – 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

  $\Rightarrow$ `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**)

Git Basics

Arbeiten mit *Remote-Repositories – 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)

Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

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

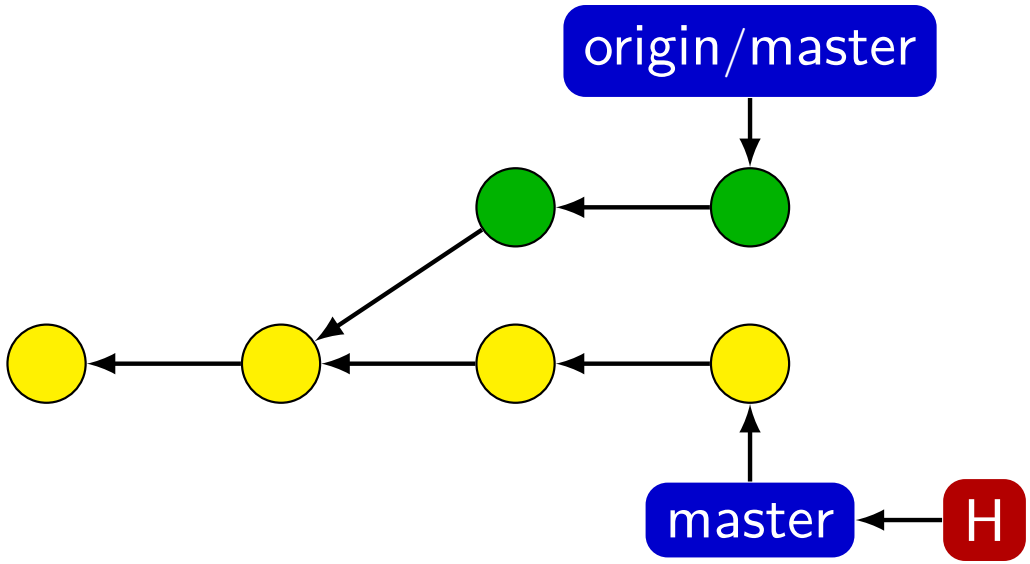
$\Rightarrow$ Änderungen müssen über **Merging** oder **Rebasing** in die lokalen *Tracking Branches* integriert werden

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

Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Merging:

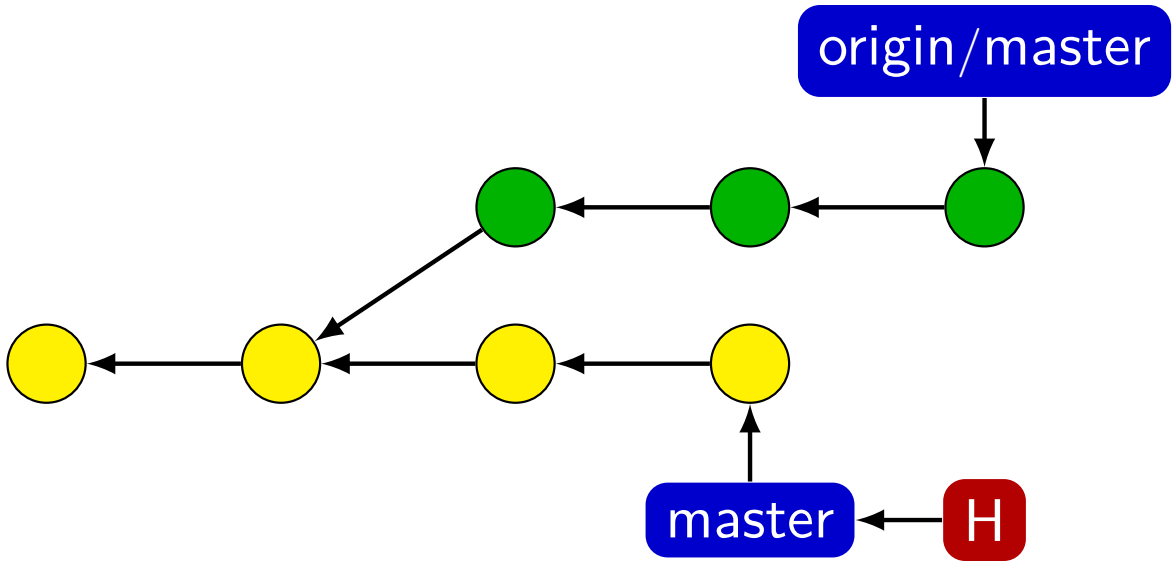


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Merging:

```
git fetch origin
```

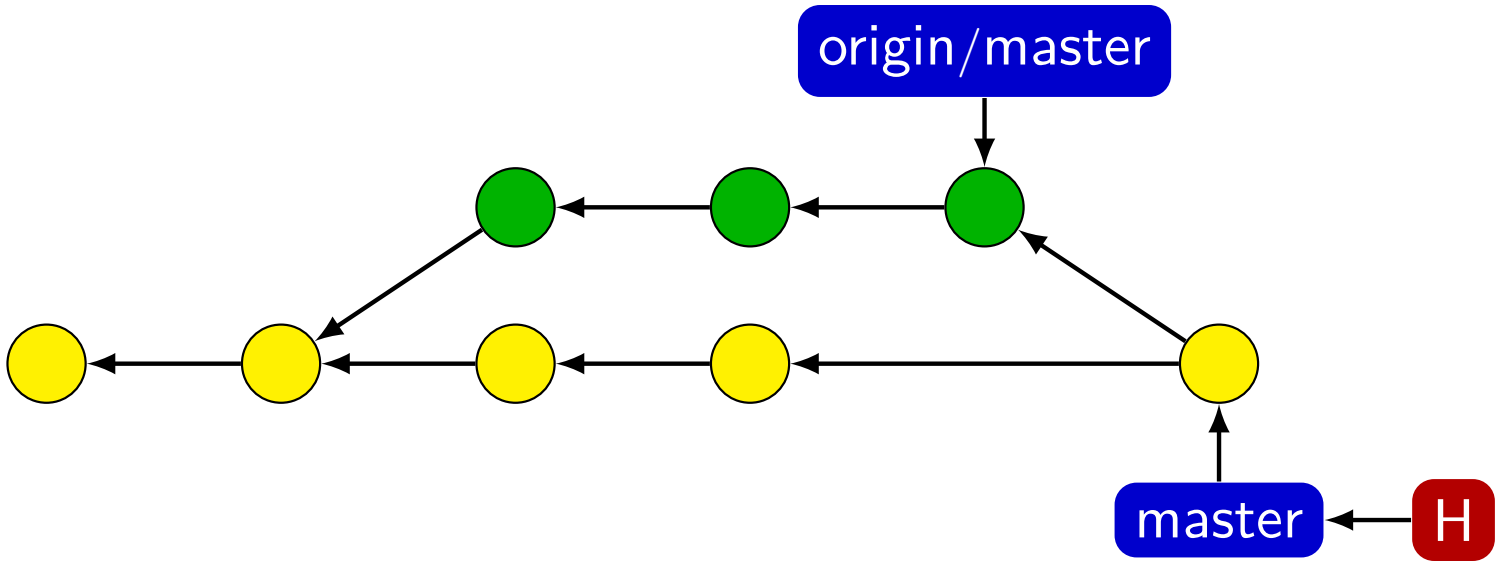


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Merging:

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

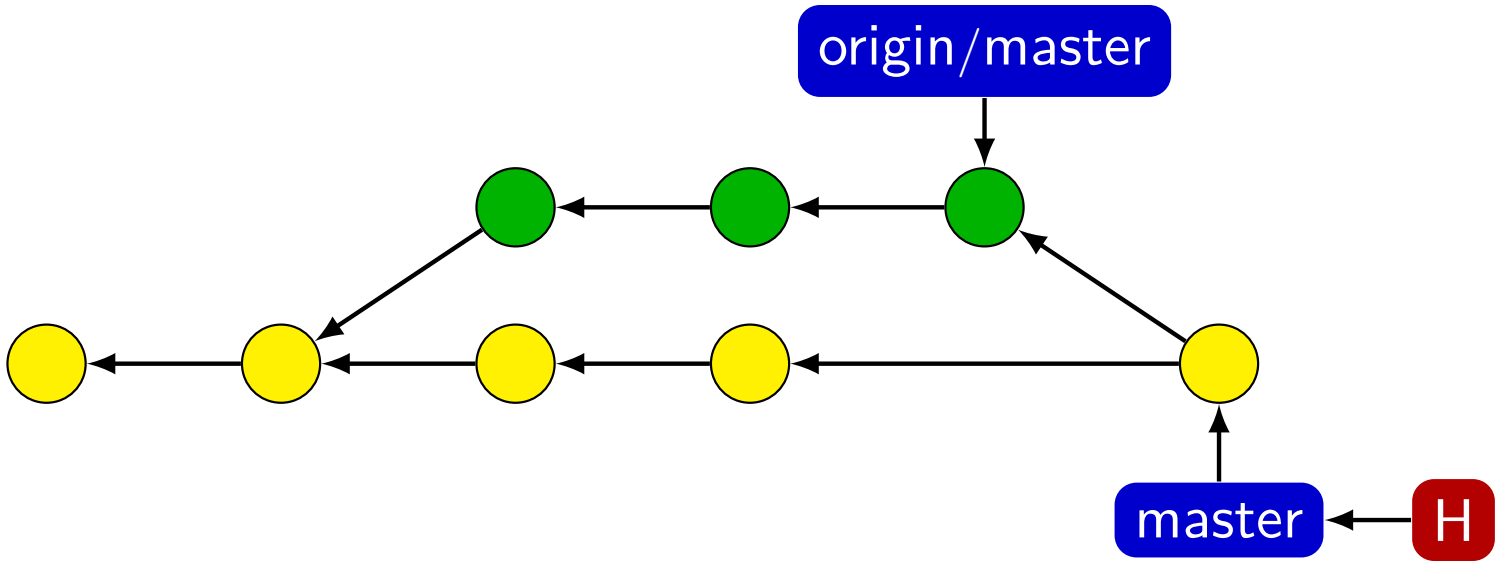


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Merging:

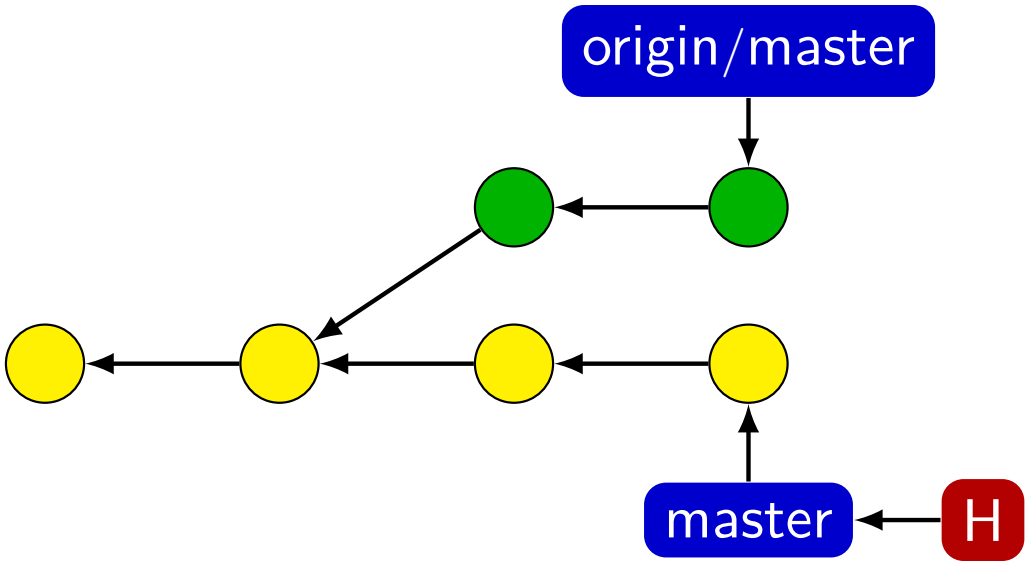
```
# Alternativ:
git pull
```



Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Rebasing:

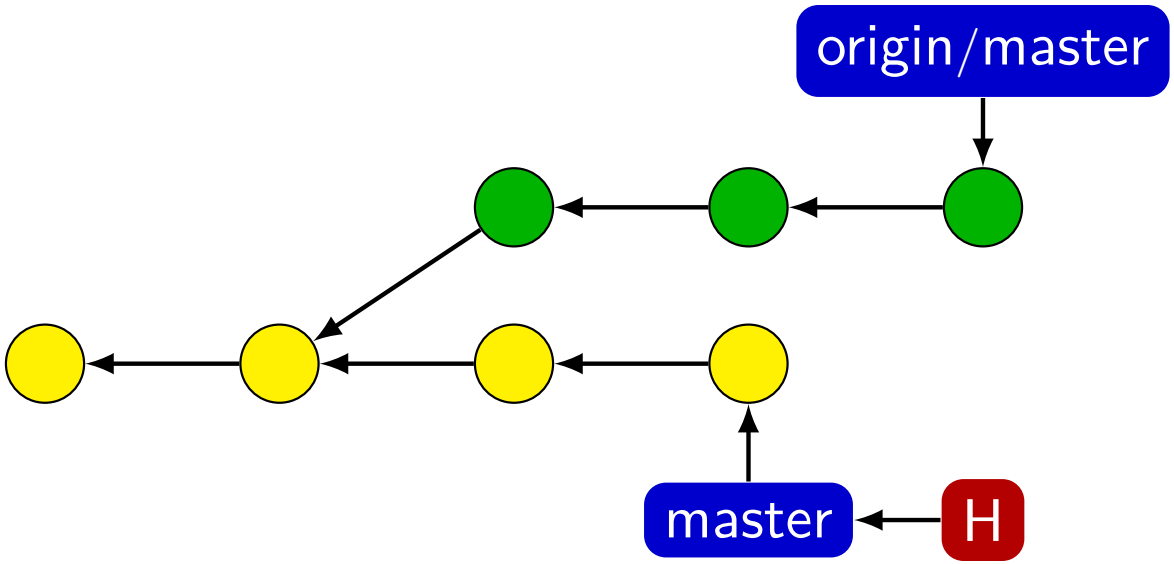


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Rebasing:

```
git fetch origin
```

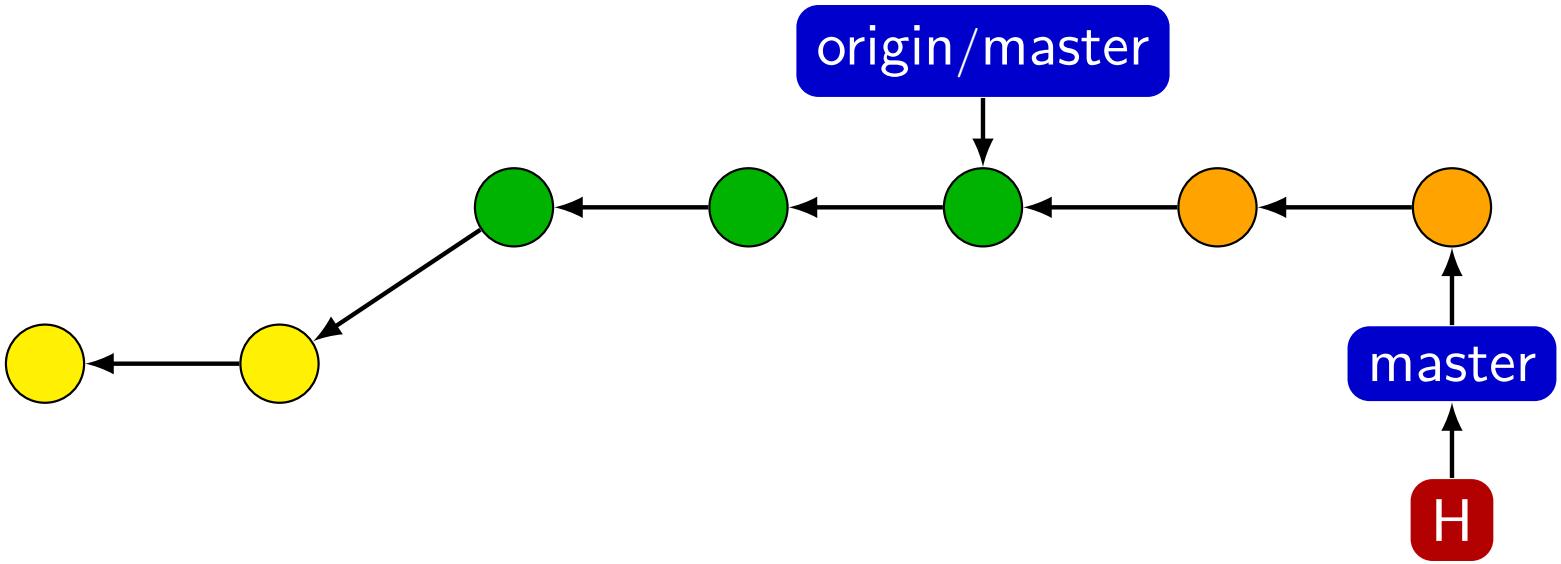


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Rebasing:

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

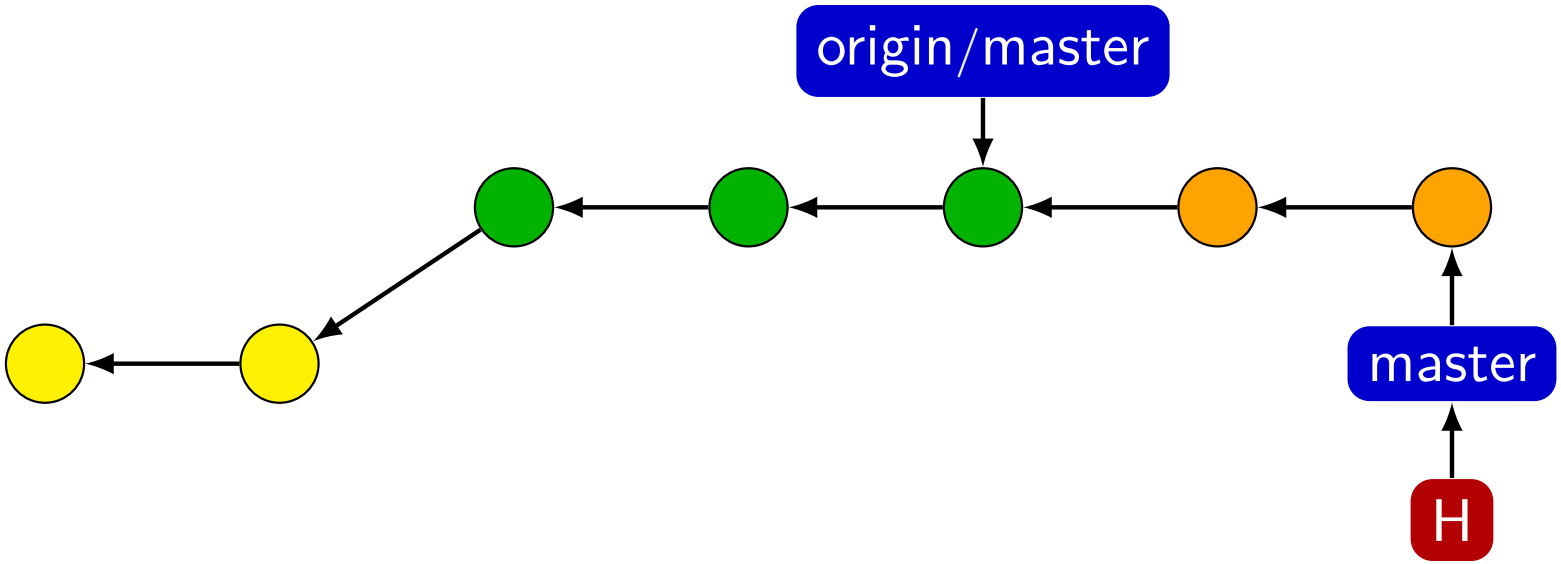


Git Basics

Arbeiten mit *Remote-Repositories* – *Tracking Branches* synchronisieren

- Beispiel mit Rebasing:

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



Git Basics

Arbeiten mit *Remote-Repositories* – 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

  $\Rightarrow$ Weniger Struktur, keine direkte Code-Basis
  - Rechner aller Entwickler müssen direkt erreichbar sein

Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

- **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

Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

- **Daten** auf zentralen Server **schieben**:

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

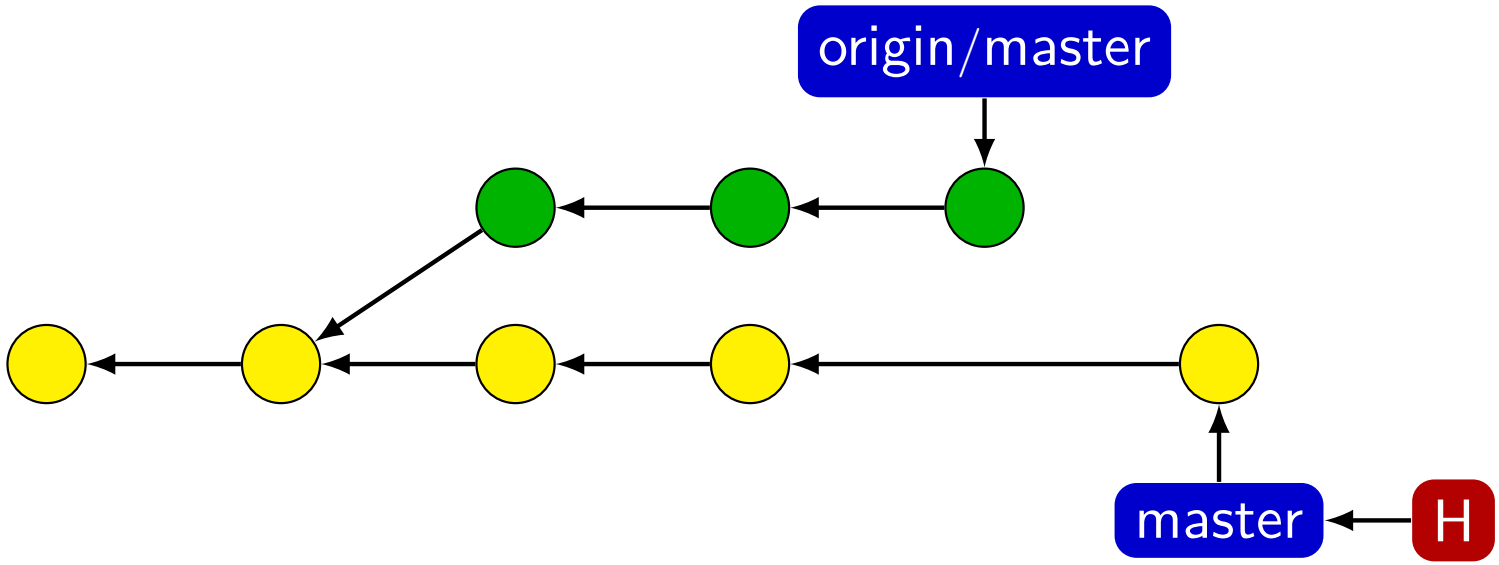
- Sendet alle zum lokalen Zweig gehörenden Daten zum Server und erstellt dort einen neuen Zweig `<Entf. Zweig>`
- Existiert der Zweig, so wird er umgesetzt, sofern dadurch keine *Commits* verloren gehen
- Bei identischen Namen kann der Teil ab `:` weggelassen werden
- Option `-u` richtet einen ***Tracking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

Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

- Beispiel:

```
git fetch origin
```

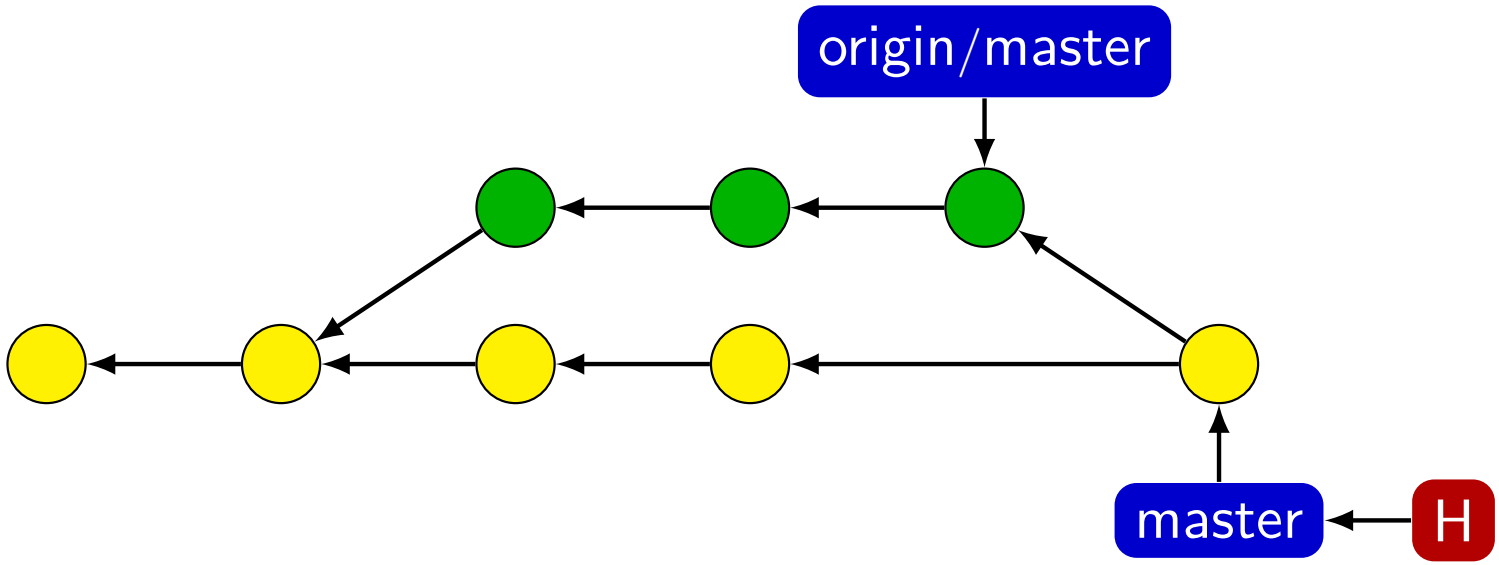


Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

- Beispiel:

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

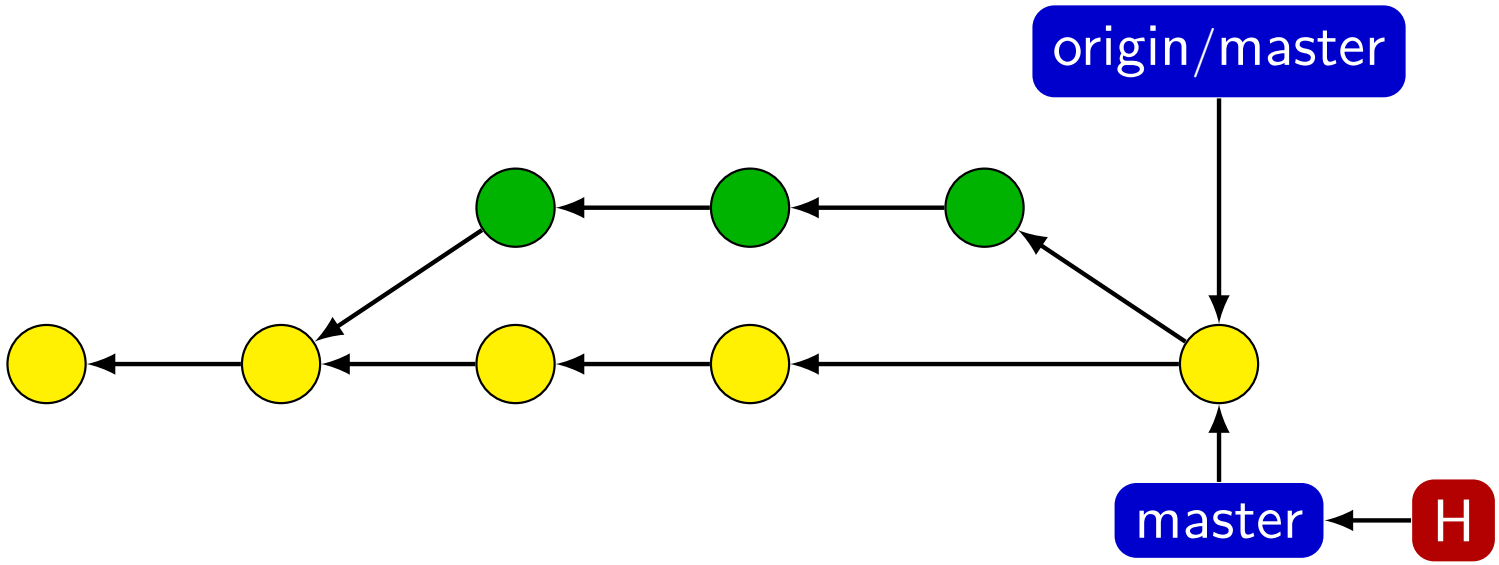


Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

- Beispiel:

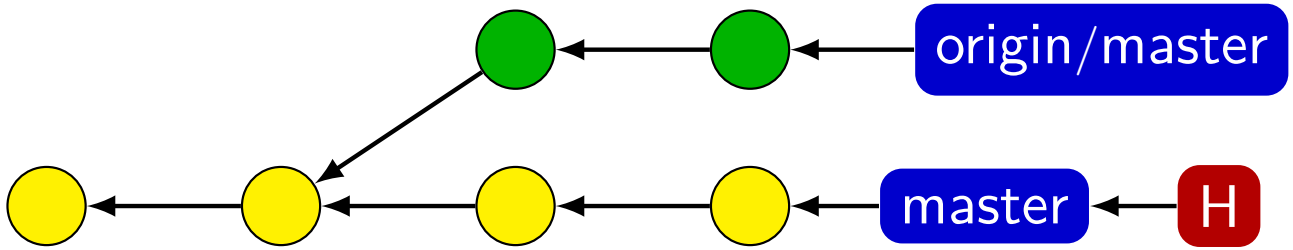
```
git fetch origin
git merge origin/master
git push -u origin master
```



Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

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



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

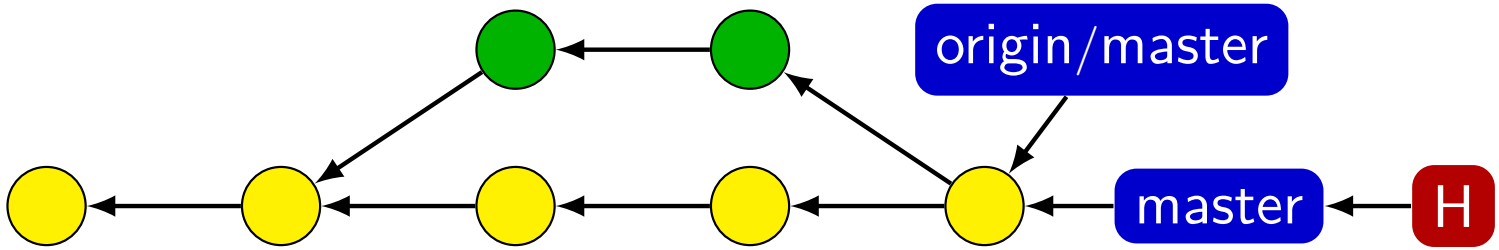
$\Rightarrow$ Die grünen *Commits* wären nicht mehr referenzierbar

$\Rightarrow$ Remote-**Änderungen** immer zuerst **integrieren**!

Git Basics

Arbeiten mit *Remote-Repositories* – Eigene Änderungen verteilen

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



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

$\Rightarrow$ Die grünen *Commits* wären nicht mehr referenzierbar

$\Rightarrow$ Remote-**Änderungen** immer zuerst **integrieren**!

Git Basics

Arbeiten mit *Remote-Repositories* – *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

$\Rightarrow$ Wird vom **Git-Server verweigert**

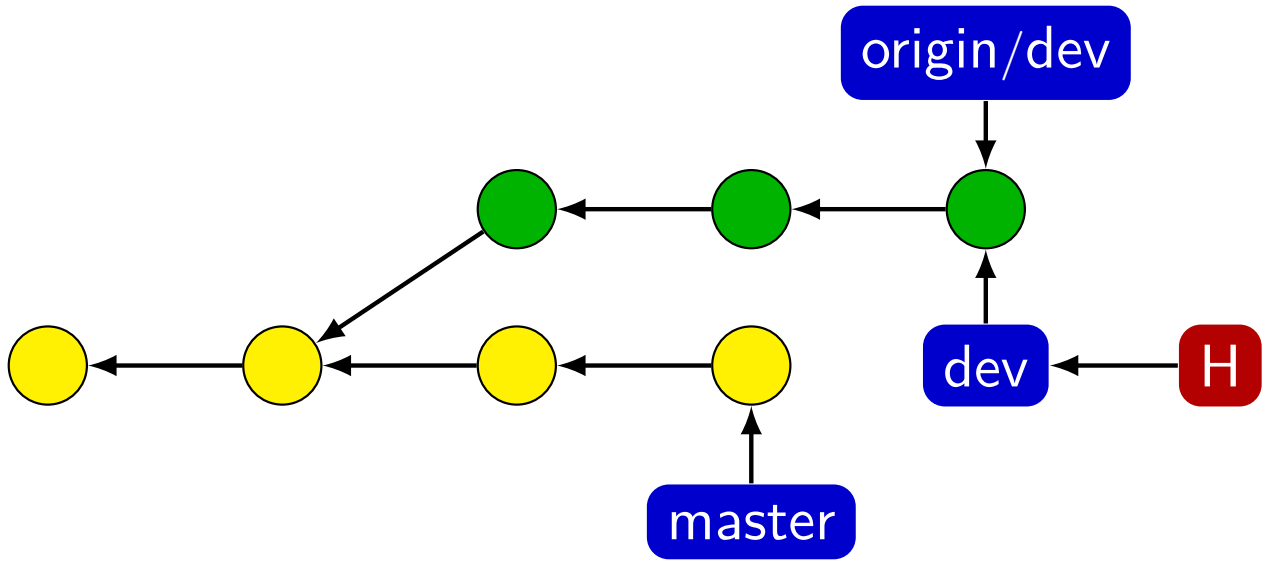
$\Rightarrow$ Richtige Vorgehensweisen:

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

Git Basics

Arbeiten mit *Remote-Repositories* – *Push* und *Rebase*

- Demonstration der Problematik:

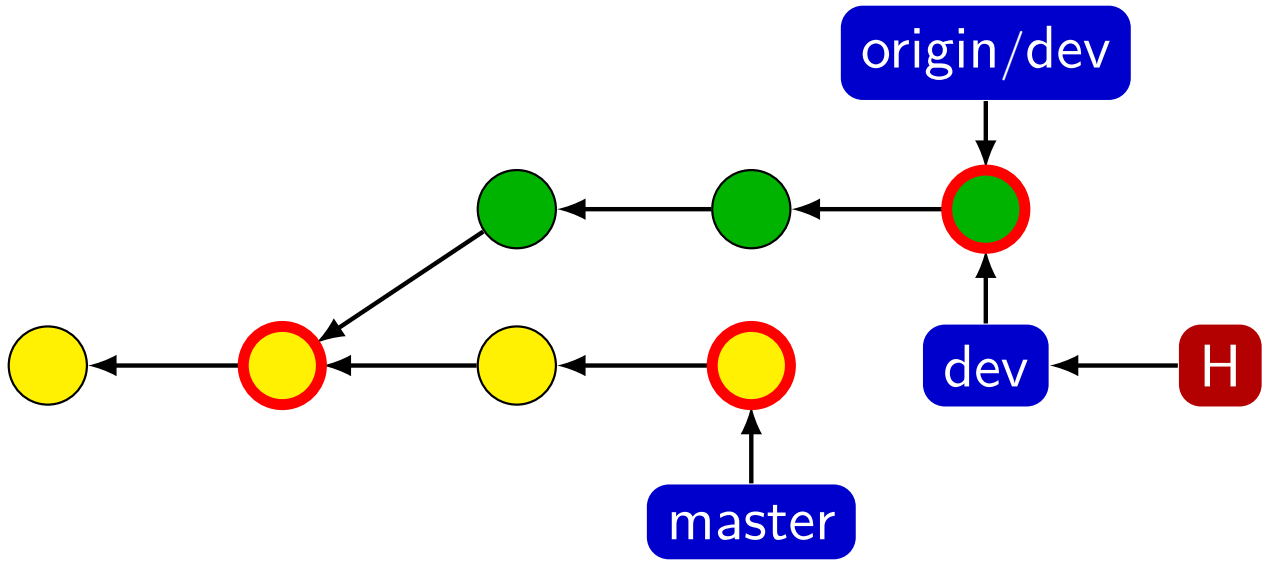


Git Basics

Arbeiten mit *Remote-Repositories* – *Push* und *Rebase*

- Demonstration der Problematik:

```
git rebase master
```

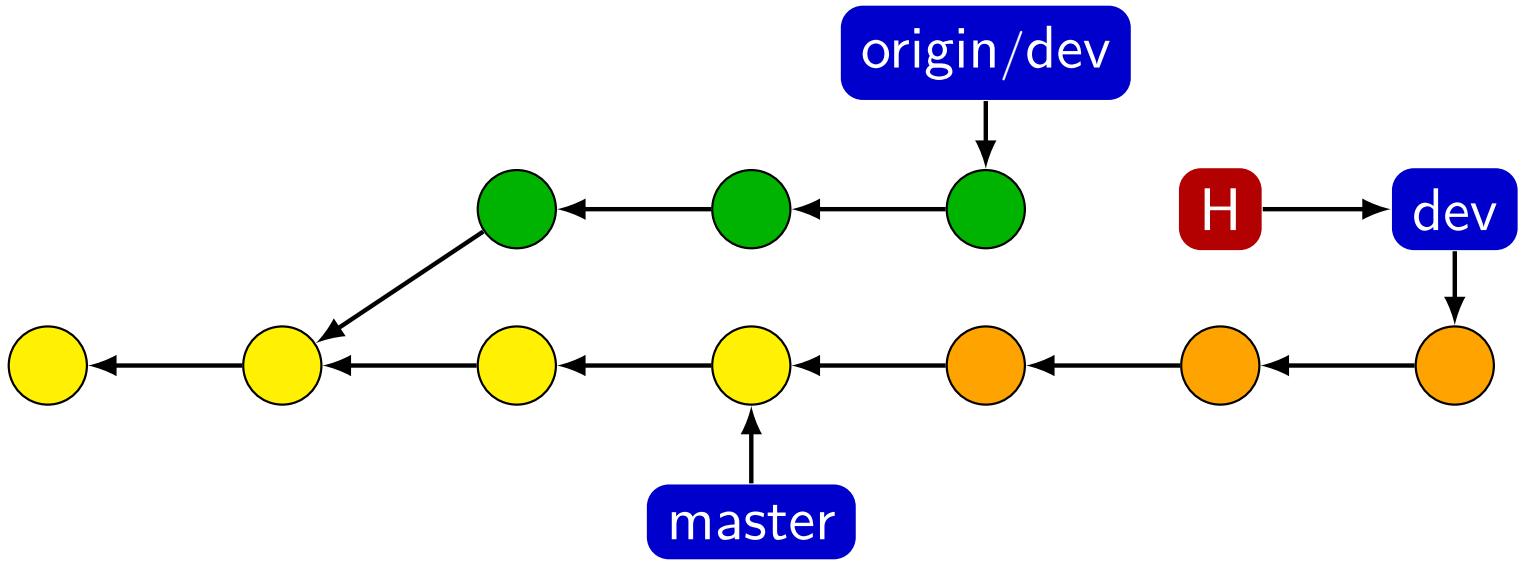


Git Basics

Arbeiten mit *Remote-Repositories* – *Push* und *Rebase*

- Demonstration der Problematik:

```
git rebase master
```



Git Basics

Arbeiten mit *Remote-Repositories* – 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

Git Basics

Arbeiten mit *Remote-Repositories* – Gelöschte Zweige synchronisieren

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

$\Rightarrow$ Synchronisierung gelöschter Zweige erfolgt **manuell**:

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

Git Basics

Arbeiten mit *Remote-Repositories* – 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)

Git Basics

Tagging

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

    $\Rightarrow$ Verhält sich wie ein **unbeweglicher Zweig**
  - ***Annotated Tags***:
    - Zeiger mit eigenem **Tag-Objekt**

    $\Rightarrow$ Tags erhalten einen Hash, Ersteller, Datum, Nachricht, ...
    - Sollten **immer bevorzugt** genutzt werden
  - ***Signed Tags***:
    - *Annotated Tag* mit **GPG-Signatur**

    $\Rightarrow$ Ersteller kann verifiziert werden
    - Hier nicht weiter behandelt

Git Basics

Tagging – 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>
```

Git Basics

Tagging – *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

Git Basics

Tagging – 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

Git Basics

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

Git Basics

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
```

Git Basics

Aktuelle Änderungen temporär aufheben

- `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 $n = 0$

Git Basics

Zweig auf anderen *Commit* setzen

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

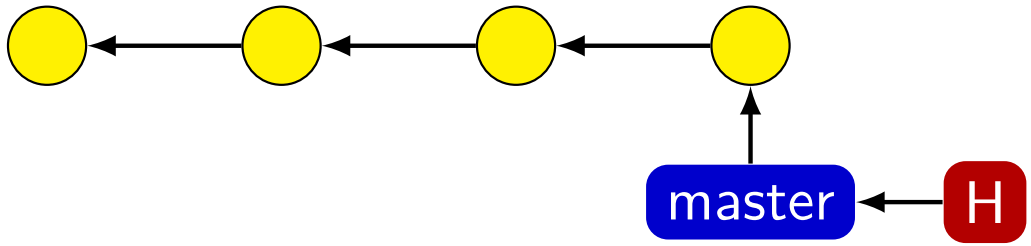
```
git reset <Commit>
```

- <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

Git Basics

Zweig auf anderen *Commit* setzen

- Beispiel:

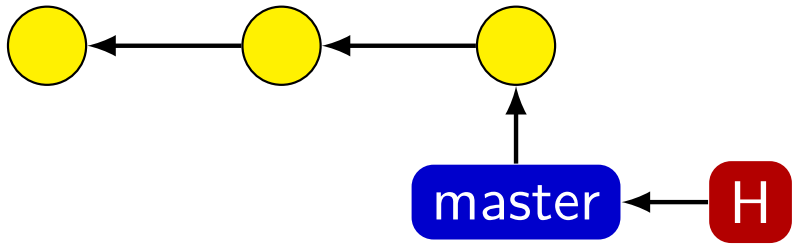


Git Basics

Zweig auf anderen *Commit* setzen

- Beispiel:

```
git reset HEAD~
```



Git Basics

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
```

Git Basics

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

Git Basics

Letzten *Commit* um Änderungen erweitern

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

```
git commit --amend
```

- Verhält sich wie:

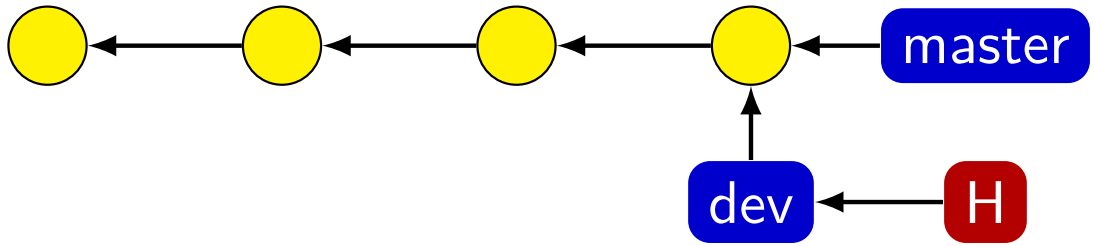
```
git reset --soft HEAD~
git commit
```

- 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

Git Basics

Letzten *Commit* um Änderungen erweitern

- Beispiel:

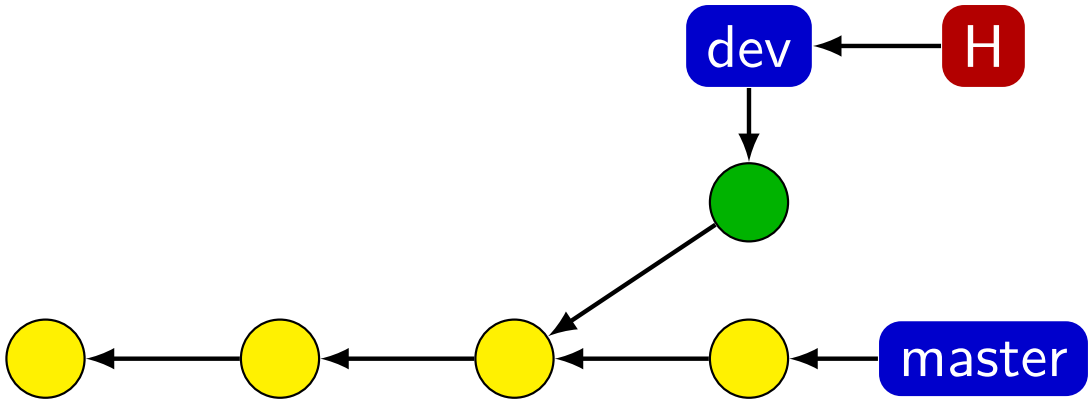


Git Basics

Letzten *Commit* um Änderungen erweitern

- Beispiel:

```
git commit --amend
```



Git Basics

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>
```

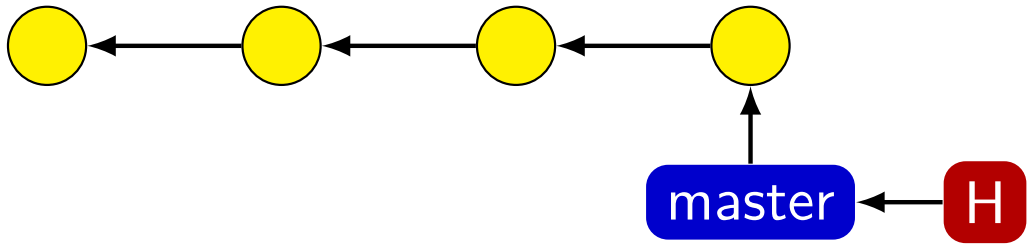
- Per Default wird der Verlauf von `HEAD` ausgegeben
- Liefert eine Ausgabe der Form:

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

Git Basics

Gelöschte *Commits* retten

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

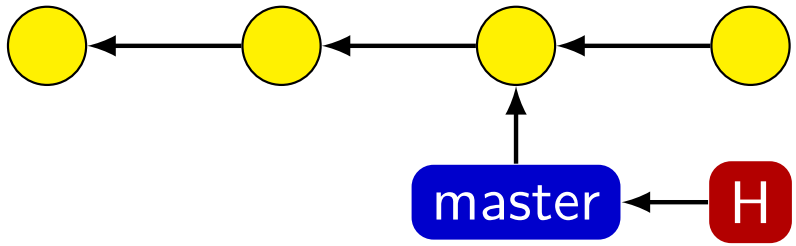


Git Basics

Gelöschte *Commits* retten

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

```
git reset HEAD~
```

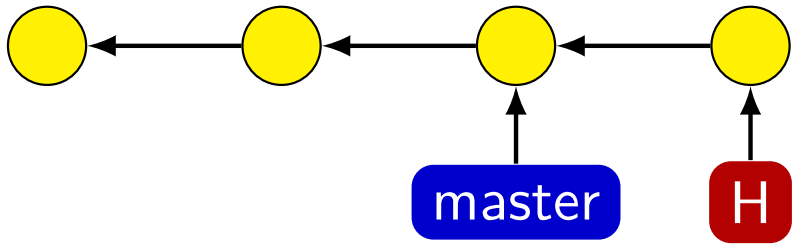


Git Basics

Gelöschte *Commits* retten

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

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

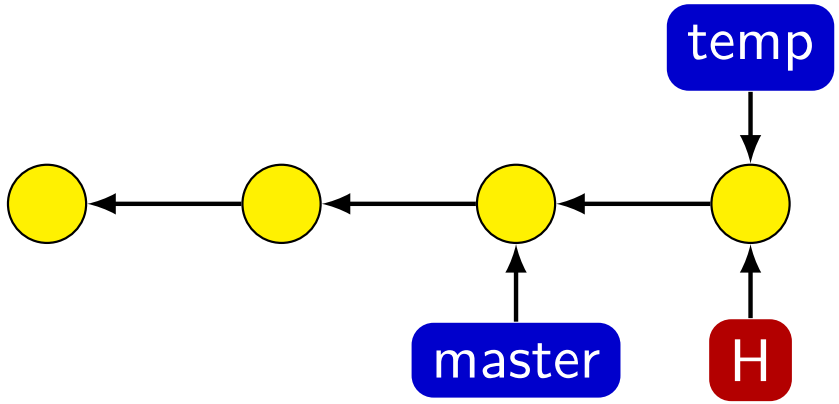


Git Basics

Gelöschte *Commits* retten

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

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



Git Basics

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

Git Basics

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

Git Basics

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 Basics

Aktuellen Arbeitsstand exportieren

- 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**

Nützliches

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>
```

Nützliches

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
```

Nützliches

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
```

Nützliches

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

Nützliches

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

$\Rightarrow$ 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**)

Nützliches

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/`

Nützliches

Log in grafischer Oberfläche