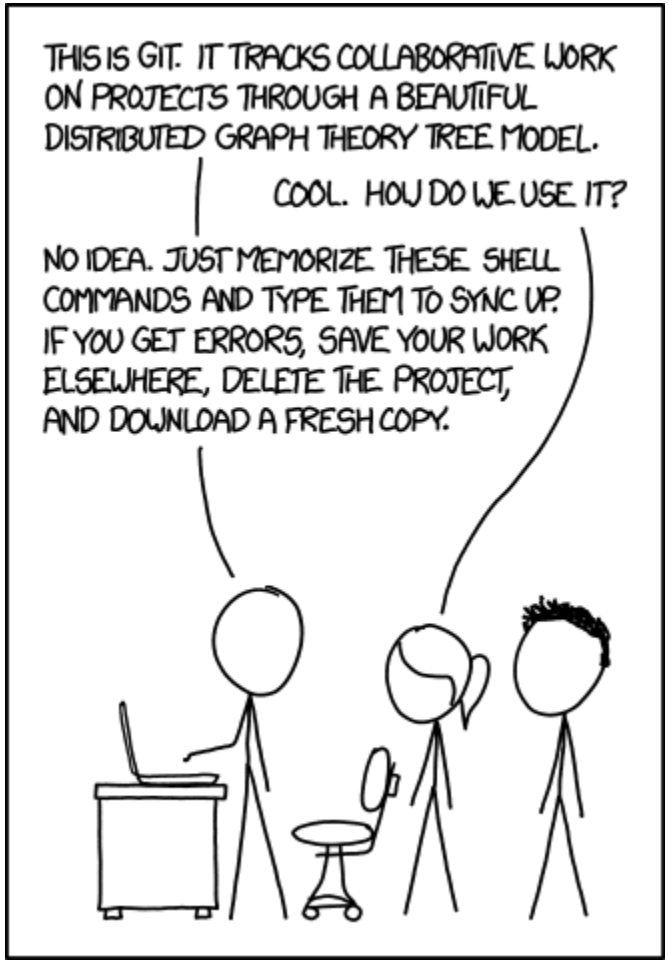
WSE1

Werkzeuge im Software Engineering

Git

Stefan Wagner

Git



http://xkcd.com/1597/

Git vs. Subversion

Git

- dezentrales VCS
- Versionen werden mit Hashes referenziert
- Verzeichnisse werden nicht versioniert
- Arbeitskopie umfasst immer alle Dateien einer Version
- Branches/Tags sind Zeiger auf Commits
- Branch/Tag umfasst immer alle Dateien einer Version
- Berechtigungsmanagement nur für ganze Repositories möglich

Subversion

- zentrales VCS
- Versionen werden mit Revisionsnummer referenziert
- auch Verzeichnisse werden versioniert
- partielle Arbeitskopie eines Teils einer Version möglich
- Branches/Tags sind Kopien von Verzeichnissen
- Branch/Tag eines Teils einer Version möglich
- Berechtigungsmanagement innerhalb eines Repositories möglich

Git Workspaces

- Git Workspace bezeichnet ein Verzeichnis, in dem Dateien mit Git versioniert und verwaltet werden
- Git Workspace umfasst:
  - ein Repository mit der gesamten Historie aller Versionen inkl. aller Branches/Tags/etc.
  - einen Stand der versionierten Dateien, mit denen gearbeitet werden kann
- jeder Workspace enthält auf oberster Ebene ein Verzeichnis *.git*, in dem Git die gesamten Daten des Repositories verwaltet
- Dateien im Workspace können beliebig strukturiert werden:

- *bare* Repositories haben keinen umschließenden Workspace
  - dienen zur Ablage auf Servern, auf denen nicht direkt gearbeitet wird
  - enthalten nur die Daten des Repositories, also den Inhalt des *.git* Verzeichnisses
  - Namenskonvention: bare Repositories enden auf *.git*

Anlegen eines Git Workspace

- Git Workspace kann entweder neu angelegt oder von einem bestehenden Repository kopiert werden

- *git init*
  - erstellt einen neuen Git Workspace
  - mit der Option *-b* (*--initial-branch*) kann der Name des initialen Branch festgelegt werden (z.B. main statt master)
  - Option *--bare* legt nur ein bare Repository an
  - Beispiel: `git init -b main`

- *git clone*
  - kopiert ein bestehendes Git Repository und erstellt dafür einen neuen Workspace
  - mit der Option *-o* (*--origin*) kann der Name des Ursprungs festgelegt werden
  - Option *-b* (*--branch*) gibt den Namen des Branch an, der ausgecheckt werden soll
  - Beispiel: `git clone https://github.com/microsoft/vscode`

Einstellungen

- Git speichert Einstellungen in Konfigurationsdateien
- *git config*
  - Option *-l* (*--list*) zeigt alle vorhandenen Einstellungen an
  - Option *--unset* entfernt eine Einstellung
  - 3 Ebenen:
    - System (*--system*)
      - systemweite Einstellungen für alle Benutzer
      - unter /etc/gitconfig (Linux) oder *<GitInstallDir>*\etc\gitconfig (Windows)
    - Benutzer (*--global*)
      - benutzerspezifische Einstellungen für alle Repositories dieses Benutzers
      - unter *<UserHomeDir>*/.gitconfig
    - Repository (*--local* = Standard)
      - individuelle Einstellungen für ein einzelnes Repository
      - unter *<WorkspaceDir>*/.git/config
  - spezifischere Ebenen überlagern allgemeinere Ebenen: Repository → Benutzer → System
  - Option *--show-origin* zeigt an, wo eine Einstellung definiert wurde
- initial sollten zumindest Name und E-Mail-Adresse eingestellt werden:

```
git config --global user.name "Dave Developer"
git config --global user.email dave@developers.com
```

- mit *alias.<command>* können individuelle Kurzformen für Befehle festgelegt werden:

```
git config --global alias.unstage 'reset HEAD --'
git config --global alias.graph 'log --all --decorate --oneline --graph'
git config --global alias.guiclient '!gitk'
```

Git Repositories

- Git Repositories enthalten:
  - Dateiinhalte (*Blobs*)
    - textuell oder binär
    - werden unabhängig von Dateinamen gespeichert
  - Verzeichnisse (*Trees*)
    - verknüpfen Dateinamen mit Dateiinhalten
    - können weitere Verzeichnisse enthalten
  - Versionen (*Commits*)
    - speichern einen Stand des Wurzelverzeichnisses, d.h. des gesamten Arbeitsverzeichnisses
    - enthalten zusätzlich noch Metadaten (Autor, Committer, Zeitstempel, Beschreibung, Verweis auf Vorgängerversion)
- für jeden Blob/Tree/Commit wird ein Hashwert gespeichert, über den das jeweilige Element referenziert werden kann
- Hashwerte von Commits entsprechen den Revisionsnummern in Subversion
- identische Inhalte werden nur einmal gespeichert
- Daten können nachträglich nicht verändert werden, da sich durch eine Änderung des Inhalts auch der Hashwert ändern würde

Datenstruktur von Commits

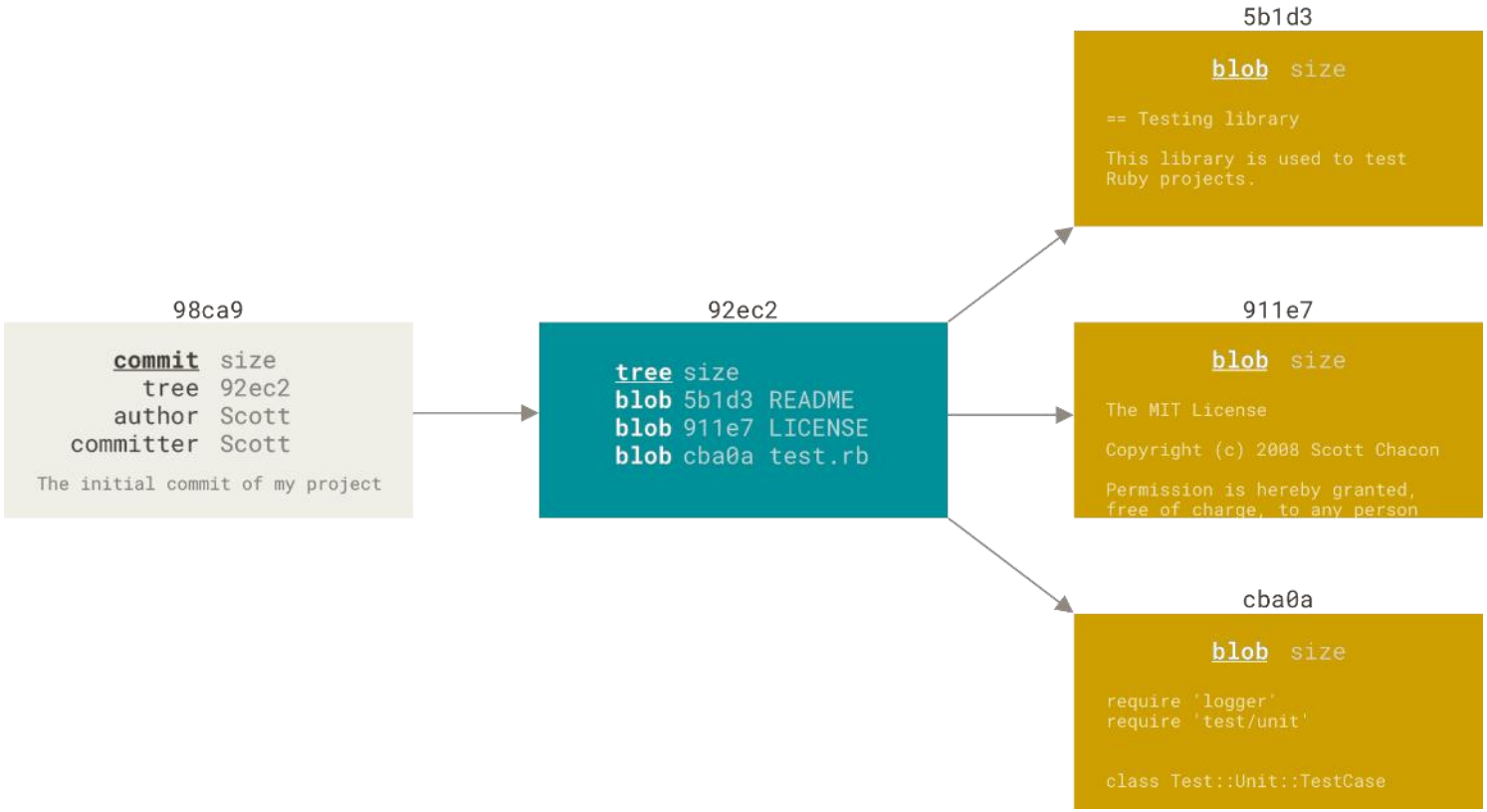


Abbildung aus S. Chacon, B. Straub: Pro Git

Datenstruktur von Commits

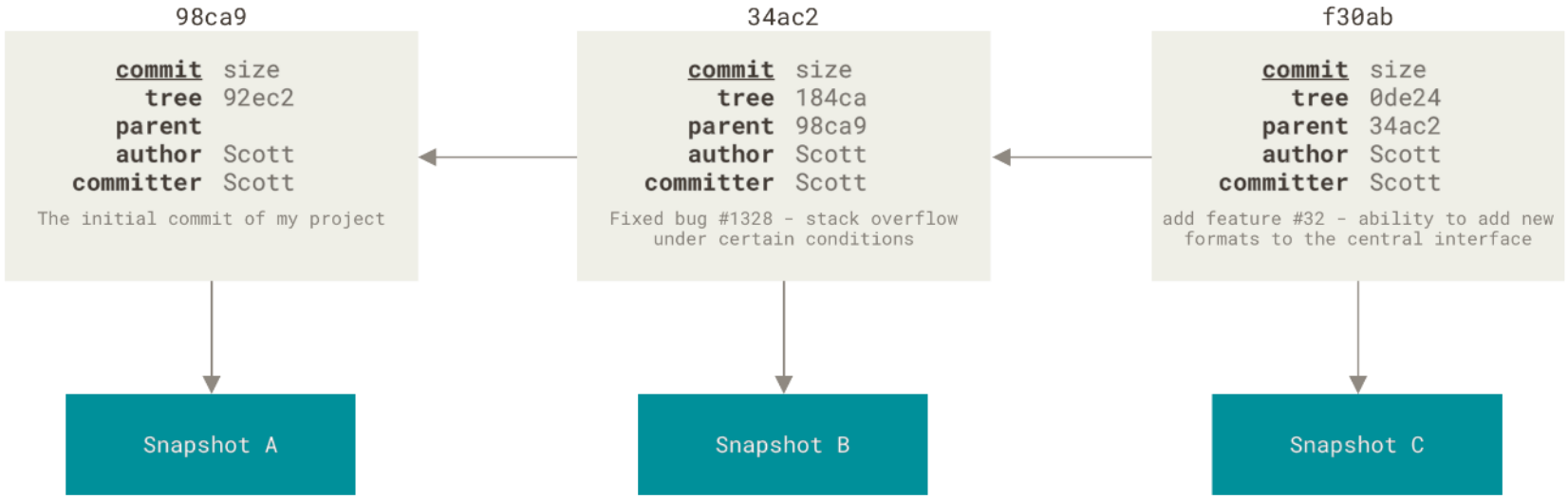


Abbildung aus S. Chacon, B. Straub: Pro Git

Plumbing vs. Porcelain

- Git hat sehr viele verschiedene Befehle
- manche dieser Befehle sind Standardbefehle (porcelain), mit denen laufend gearbeitet wird
- andere sind Basisbefehle (plumbing), die selten oder nie direkt benötigt werden
- ob ein Befehl zu den Standardbefehlen oder zu den Basisbefehlen gehört, ist leider oft nicht sofort zu erkennen

- Beispiel für einen Basisbefehl: *git cat-file*
  - gibt den Typ und den Inhalt von Objekten im Repository an
  - Option *-t* gibt den Typ des Objekts aus
  - Option *-p* gibt den Inhalt des Objects aus
  - Beispiel: `git cat-file -p HEAD`

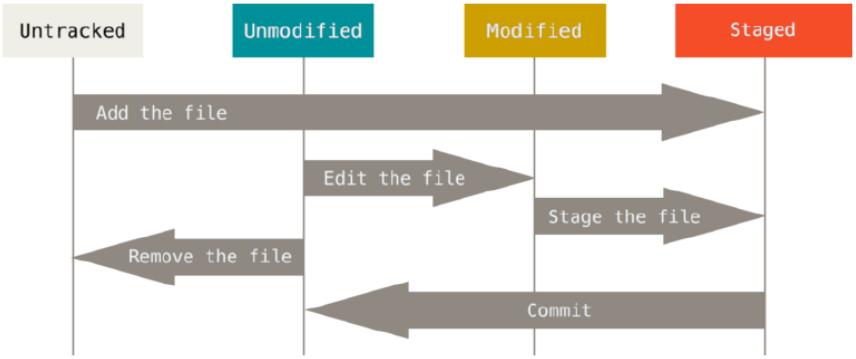
- kurze Erklärung einiger Plumbing Commands: A Plumber's Guide to Git

Typischer Arbeitsablauf mit Git

- lokales Repository aktualisieren:
  git pull
- Änderungen durchführen und überprüfen:
  git status, *git diff*
- Änderungen zur Staging Area hinzufügen:
  git add
- Änderungen in das lokale Repository schreiben:
  git commit
- Änderungen an das entfernte Repository übertragen:
  git push

Zustände von Dateien

- Dateien in einem Git Workspace können folgende Zustände haben:
  - **untracked**
    - nicht unter Versionskontrolle
  - **unmodified**
    - versioniert und nicht verändert
  - **modified**
    - versioniert und verändert, aber nicht für den nächsten Commit vorgesehen
  - **staged**
    - versioniert, verändert und für den nächsten Commit vorgesehen



- alle Änderungen, die vom nächsten Commit erfasst werden sollen, müssen mit *git add* in die Staging Area (Index) übertragen werden
  - Staging Area wird unter .git/index gespeichert

- nachdem eine Änderung in die Staging Area übertragen wurde, werden nachträgliche Änderungen nicht automatisch übernommen
  - falls eine Datei nach dem Übertragen in die Staging Area nochmals verändert wird und diese neue Änderungen ebenfalls vom nächsten Commit erfasst werden sollen, muss git add nochmals aufgerufen werden

Abbildung aus S. Chacon, B. Straub: Pro Git

Verschieben und Löschen von Dateien

- Verschiebungen, Umbenennungen oder Löschungen von Dateien müssen ebenfalls in die Staging Area übertragen werden, um vom nächsten Commit erfasst zu werden

- *git rm*
  - entfernt eine Datei aus dem Arbeitsverzeichnis und trägt die Löschung in die Staging Area ein
  - Option *--cached* trägt Löschung nur in die Staging Area ein und behält aber die Datei im Arbeitsverzeichnis
  - anstatt *git rm* kann auch ein normales *rm* mit anschließendem *git add* verwendet werden

```
git rm code.c          entspricht      rm code.c
                                       git add code.c
```

- *git mv*
  - verschiebt eine Datei im Arbeitsverzeichnis und trägt Verschiebung in die Staging Area ein
  - Verschiebungen/Umbenennungen von Dateien werden von Git auch automatisch aufgrund des gleichen Dateiinhalts (d.h. Hashwerts) erkannt
  - anstatt *git mv* kann auch ein normales *mv* mit anschließendem *git add* verwendet werden

```
git mv old.c new.c     entspricht      mv old.c new.c
                                       git add old.c
                                       git add new.c
```

Überprüfen von Änderungen

- *git status*
  - zeigt den Zustand der Dateien im Arbeitsverzeichnis an
  - Option *-s* (*--short*) zeigt Kurzform (ähnlich zu svn status)

- *git diff*
  - zeigt die inhaltlichen Änderungen in einer Datei im *unified diff* Format an
  - ohne weitere Option wird das Arbeitsverzeichnis mit der Staging Area verglichen
  - Option *--cached* vergleicht die Staging Area mit dem letzten Commit

Ignorieren von Dateien

- Dateien, die nicht versioniert werden sollen, können in *.gitignore* Dateien eingetragen werden
  - jedes Verzeichnis kann eine eigene *.gitignore* Datei enthalten
  - üblich ist jedoch nur eine *.gitignore* Datei im Wurzelverzeichnis des Workspace
- eine zu ignorierende Datei bzw. ein zu ignorierendes Verzeichnis pro Zeile
- Einträge ohne "/" zu Beginn werden rekursiv in allen Unterverzeichnissen angewendet
- Pfad ist relativ zu dem Verzeichnis, in dem die .gitignore Datei gespeichert ist
- Ausnahmen können mit "!" angegeben werden
- Platzhalter können verwendet werden, um mehrere Dateien auszunehmen:
  - * entspricht einer beliebigen Zeichenkette (inkl. der leeren Zeichenkette)
  - ** entspricht beliebigen verschachtelten Unterverzeichnissen
  - ? entspricht einem einzelnen Zeichen
  - [abc] entspricht einem Zeichen aus der angegebenen Menge (z.B. "a", "b" oder "c")

Ignorieren von Dateien

- Beispiel für eine *.gitignore* Datei:

.gitignore

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

- weitere Beispiele für viele Entwicklungsumgebungen und Programmiersprachen auf GitHub: https://github.com/github/gitignore

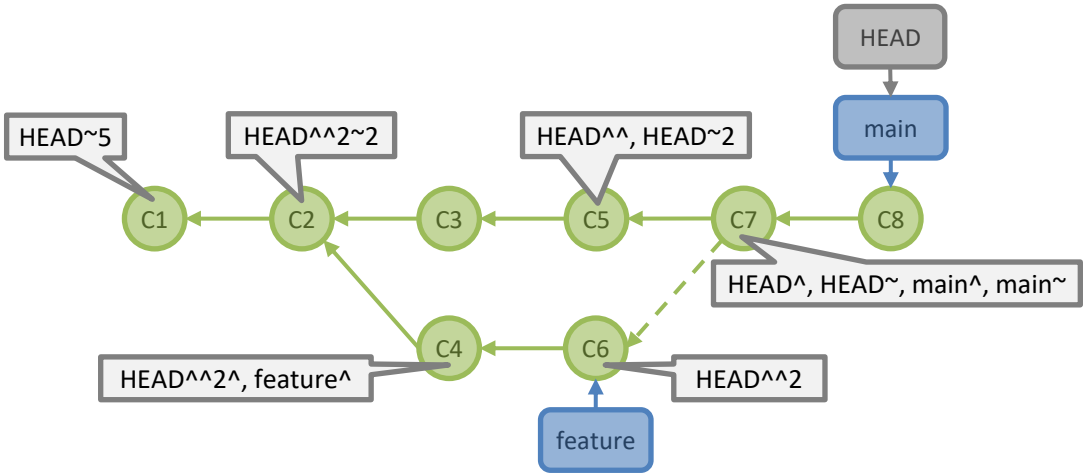
Beispiel aus S. Chacon, B. Straub: Pro Git

Speichern von Versionen im lokalen Repository

- *git commit*
  - speichert Änderungen in der Staging Area als neuen Commit im lokalen Repository
  - Option *-m* (*--message*) zur Angabe einer Beschreibung, ansonsten wird automatisch ein Editor gestartet
  - Option *-a* (*--all*) überträgt alle aktuellen Änderungen automatisch in die Staging Area und führt anschließend einen Commit durch
  - Option *--amend* ergänzt letzten Commit mit den aktuellen Änderungen in der Staging Area

Referenzieren von Commits

- Git verwendet Namen als Referenzen, um auf bestimmte Commits zu verweisen
- jeder Branch bzw. Tag ist eine solche Referenz
- HEAD referenziert den aktuellen Branch oder einen Commit (*detached head*)
- Zirkumflex (^) und Tilde (~) bezeichnen den Vorgänger
  - z.B. HEAD^ oder HEAD~ steht für den vorletzten Commit des aktuellen Branch
  - beide Zeichen können auch wiederholt werden
    z.B. HEAD^^ oder HEAD~~ steht für den vorvorletzten Commit
  - beide Zeichen können mit einer Zahl kombiniert werden
    - bei ^ steht die Zahl für das jeweilige Elternteil
      z.B. HEAD^2 = 2. Elternteil von HEAD
    - bei ~ steht die Zahl für Anzahl der Vorgänger
      z.B. HEAD~3 = HEAD~~~



Anzeigen der Historie

- *git log*
  - listet die Historie vergangener Commits auf
  - Option *--oneline* zeigt eine kompaktere Darstellung mit nur einer Zeile pro Commit
  - Option *--graph* zeigt zusätzlich eine textuelle Visualisierung des Commitgraphen
  - Option *--decorate* fügt zusätzlich Referenzen auf Commits hinzu (d.h. Namen von Branches bzw. Tags)
  - neben einigen vordefinierten Standardformaten kann das Format der Ausgabe auch individuell angegeben werden (Option *--format=<format>*)
  - zahlreiche weitere Optionen um Historie zu filtern
  - Option *-p* (*--patch*) zeigt zusätzlich die konkreten Änderungen als unified diff an (vgl. *git diff*)

Anzeigen der Historie

- git show
  - zeigt Informationen zu einem Commit an (Autor, Datum, Commit Message, Änderungen, etc.)
  - Option *--oneline* zeigt die Metainformationen zum Commit in einer kompakteren Darstellung in nur einer Zeile an
  - Option *--name-status* zeigt nur die geänderten Dateinamen und den Status (hinzugefügt, geändert, gelöscht) ohne detaillierte Änderungen an

- git diff
  - zeigt die Änderungen zwischen zwei Commits bzw. einem Commit und der Arbeitskopie oder der Staging Area an
  - bei der Angabe von zwei Commits werden die Änderungen vom ersten zum zweiten Commit angezeigt
  - Option *--name-status* zeigt nur die geänderten Dateinamen und den Status (hinzugefügt, geändert, gelöscht) ohne detaillierte Änderungen an
  - Option *--cached* vergleicht die Staging Area mit dem letzten Commit

Befehl	**Änderungen von** →	**auf**
`git diff`	Staging Area	Arbeitskopie
`git diff --cached`	HEAD	Staging Area
`git diff HEAD`	HEAD	Arbeitskopie
`git diff HEAD^ HEAD`	Vorgänger von HEAD	HEAD
`git diff branch1 main`	branch1	main
`git diff branch1...main`	Ursprung von branch1	main

Zurücknehmen von Änderungen

- *git checkout* bewegt HEAD und aktualisiert das Arbeitsverzeichnis
  - kann auch verwendet werden, um einzelne Dateien im Arbeitsverzeichnis wiederherzustellen
- *git reset* bewegt den Branch auf den HEAD zeigt
  - mehrere Funktionsarten:
    - Option --*soft* bewegt nur den Branch/HEAD
    - Option --*mixed* ändert zusätzlich die Staging Area (Standardmodus)
    - Option --*hard* ändert zusätzlich das Arbeitsverzeichnis
- *git restore* stellt Dateien wieder her und verwirft Änderungen
  - Ziel der Wiederherstellung kann angegeben werden
    - Option -*W* (--*worktree*) stellt Datei im Arbeitsverzeichnis wieder her
    - Option -*S* (--*staged*) stellt Datei in der Staging Area wieder her
    - Optionen können auch gemeinsam verwendet werden
- *git revert* erzeugt einen neuen Commit der inhaltlich einem alten Commit entspricht
  - wenn der zurückzunehmende Commit bereits gepusht wurde, muss git revert verwendet werden, da die Historie nicht verändert werden darf

Zurücknehmen von Änderungen

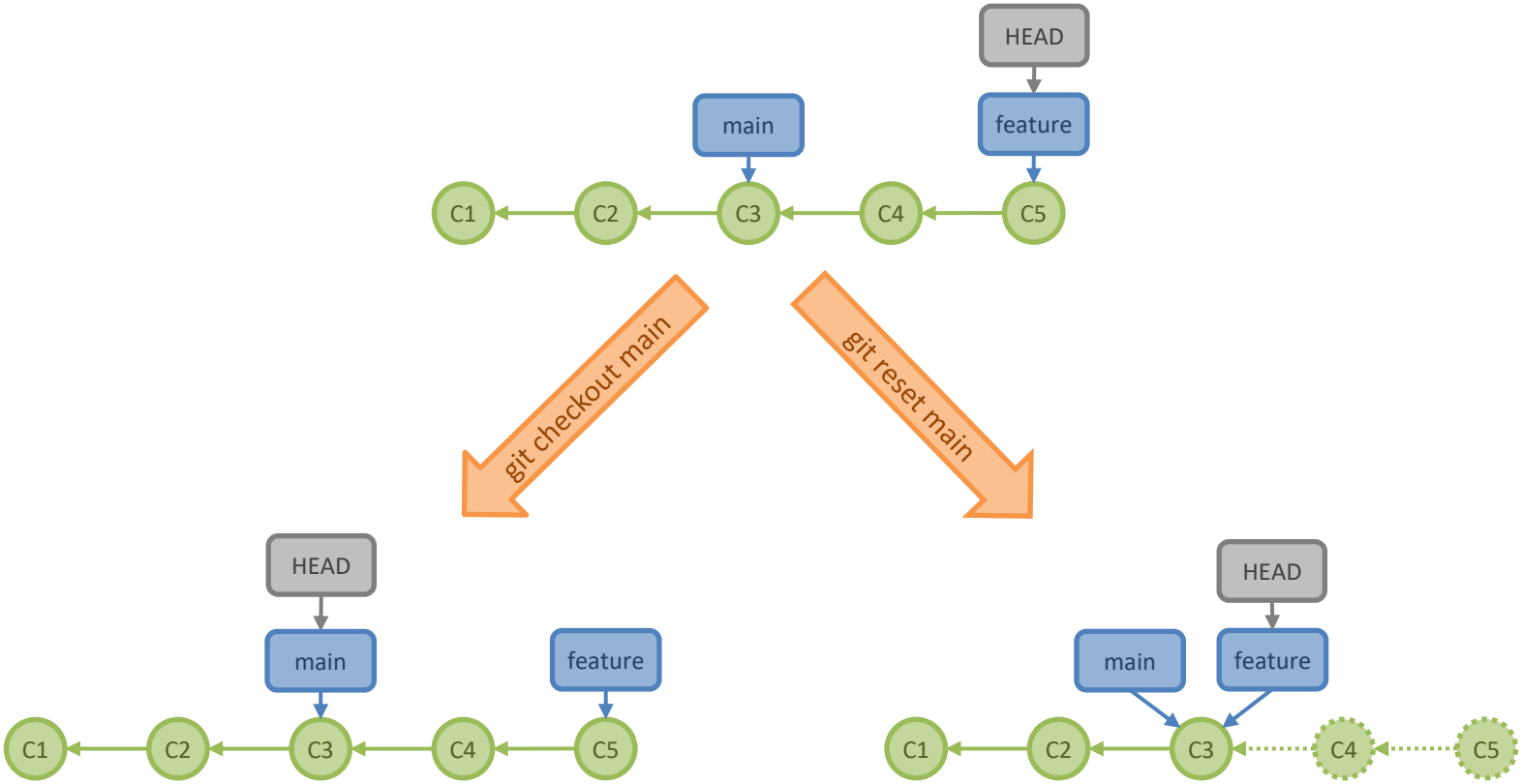
- Änderungen im Arbeitsverzeichnis rückgängig machen:
  `git restore <file>`
  `git checkout -- <file>`
  `git reset --hard` (Achtung: alle Änderungen)

- Staging einer Datei rückgängig machen:
  `git restore --staged <file>`
  `git reset <file>`

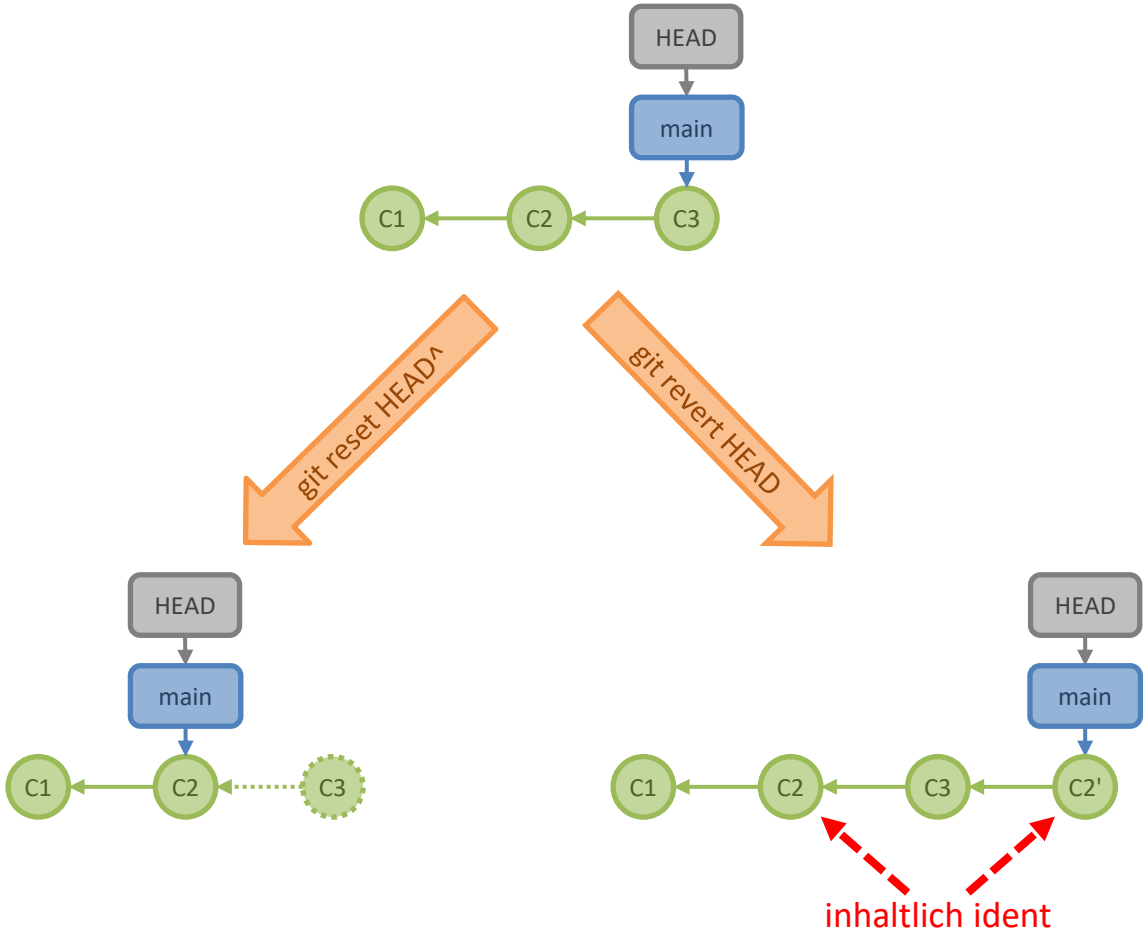
- Commit rückgängig machen:
  `git commit --amend`
  `git reset --soft HEAD^` gefolgt von `git commit`

- alten Commit wiederherstellen:
  `git revert HEAD`

Reset vs. Checkout



Reset vs. Revert



Zurücknehmen von Änderungen

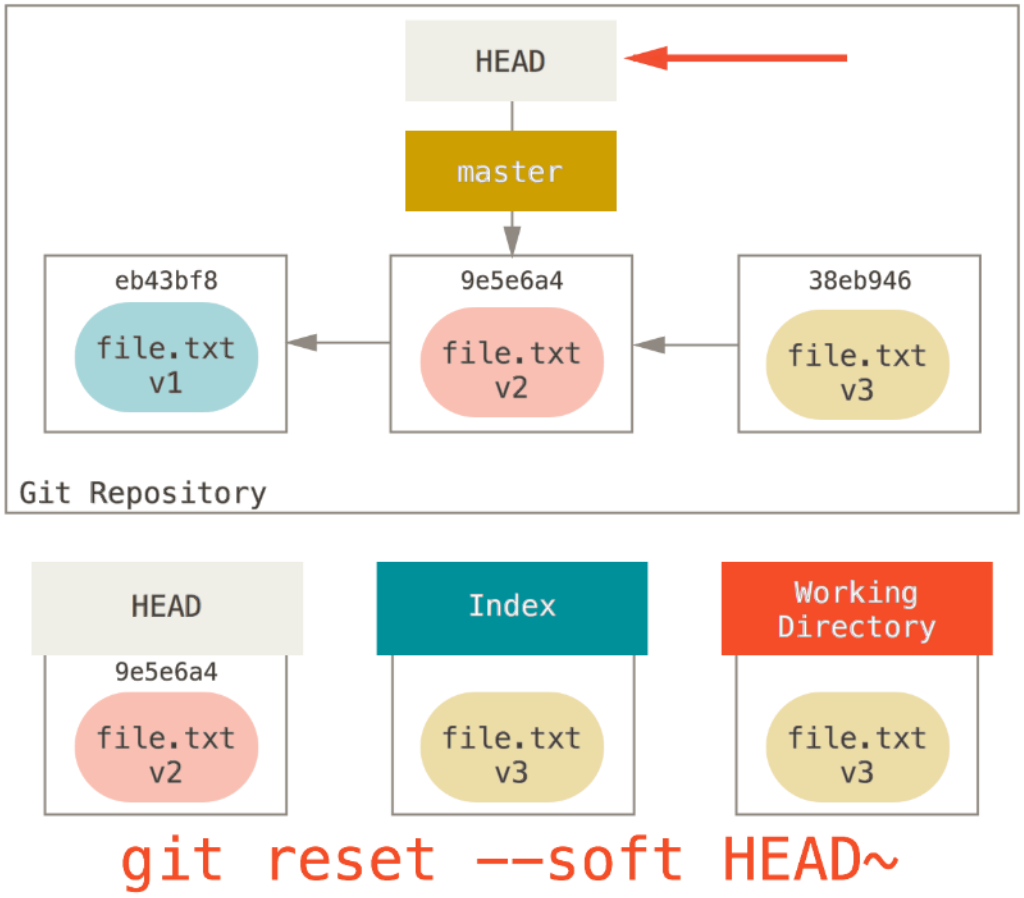


Abbildung aus S. Chacon, B. Straub: Pro Git

Zurücknehmen von Änderungen

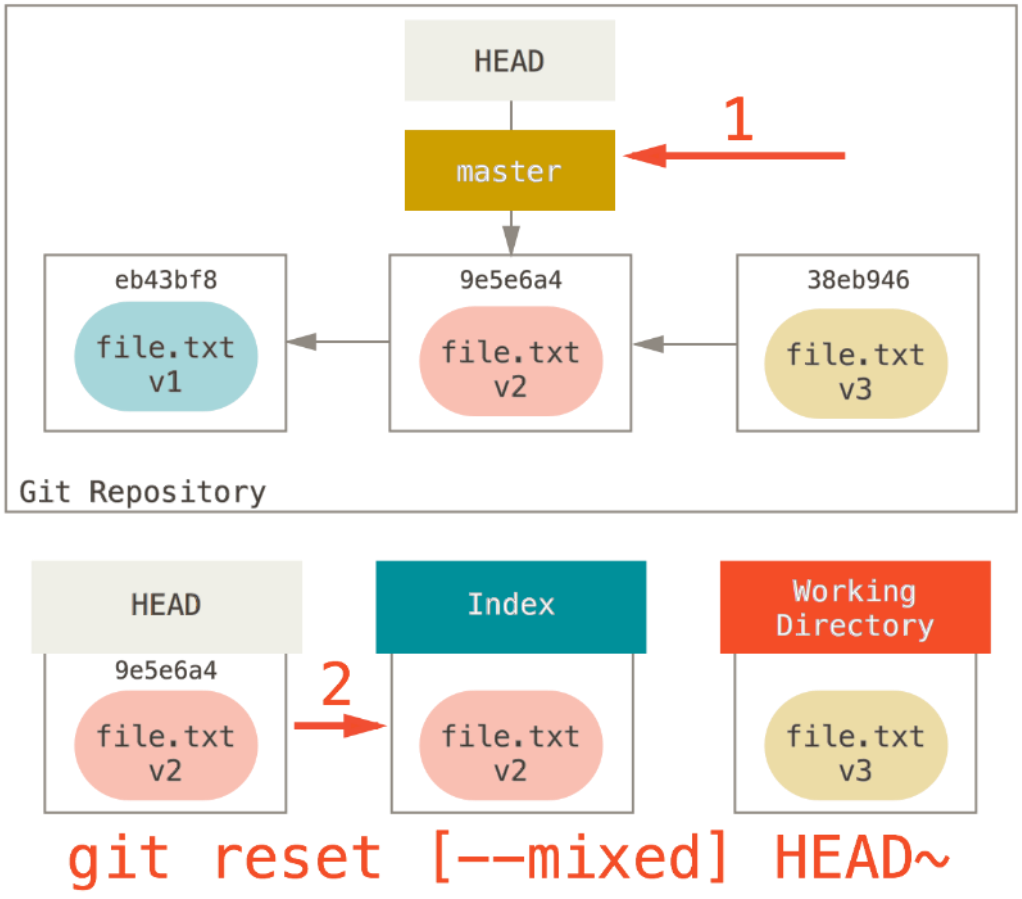


Abbildung aus S. Chacon, B. Straub: Pro Git

Zurücknehmen von Änderungen

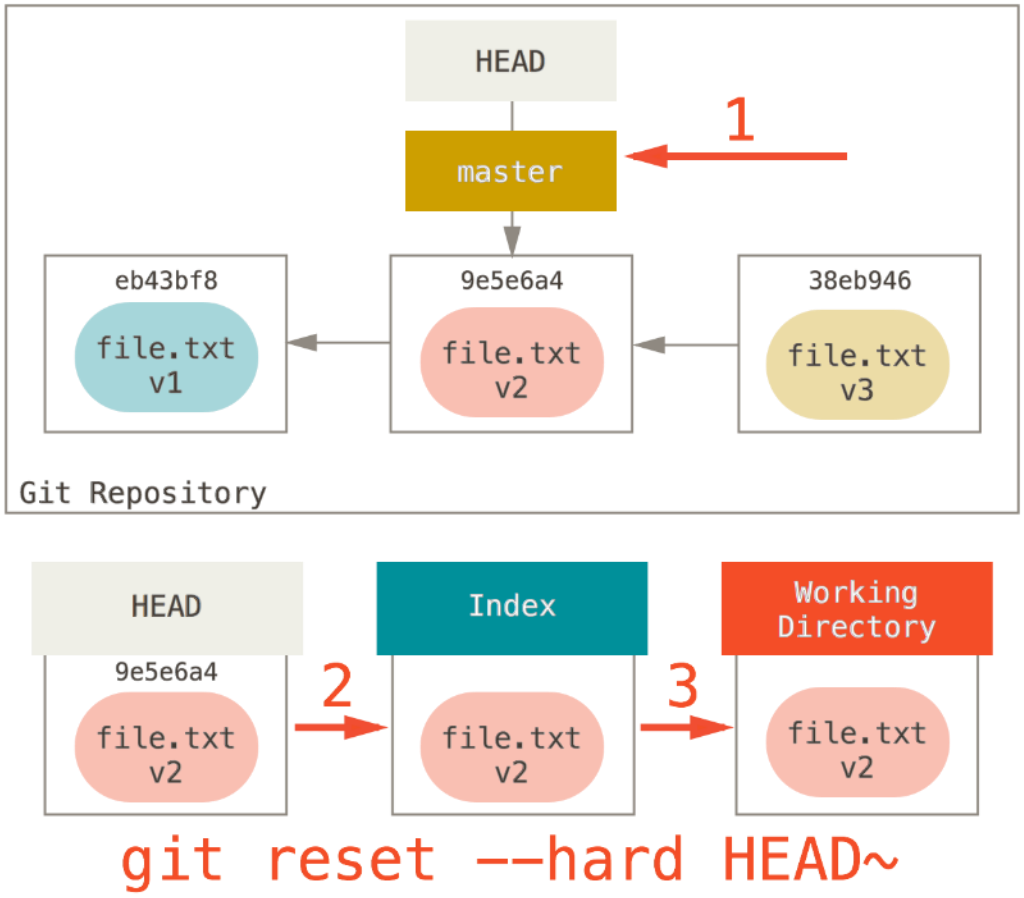
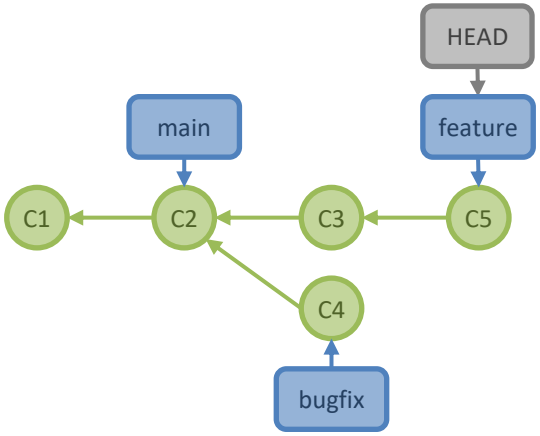


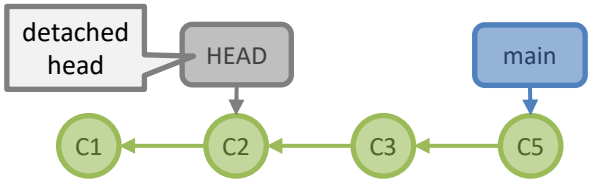
Abbildung aus S. Chacon, B. Straub: Pro Git

Branches

- Branches sind Zeiger auf Commits
- HEAD Zeiger zeigt auf aktuellen Branch, an dem gerade entwickelt wird
- bei einem Commit wird der Branch, auf den HEAD zeigt, automatisch auf den neuen Commit gesetzt

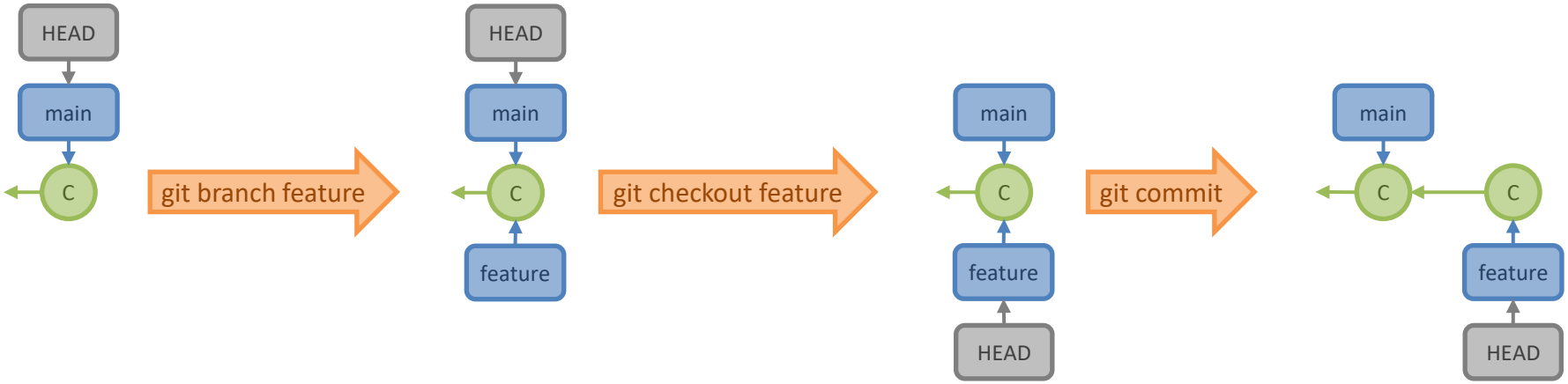


- HEAD kann auch direkt auf einen Commit gesetzt werden (*detached head*)
- bei einem Commit wird dann HEAD direkt auf den neuen Commit gesetzt
- kann z.B. verwendet werden, um Branches auf alten Commits zu erzeugen



Branches

- *git branch*
  - legt einen neuen Branch an
- *git checkout*
  - setzt HEAD auf einen bestimmten Branch und aktualisiert das Arbeitsverzeichnis entsprechend
  - Option *-b* legt einen neuen Branch an, falls der angegebene Branch nicht existiert
  - kann auch verwendet werden, um eine einzelne Datei zurückzusetzen
- *git switch*
  - Alternative zu *git checkout* (seit Version v2.23)
  - gleiche Funktionalität wie *git checkout*, einzelne Dateien können aber nicht zurückgesetzt werden
  - Option *-c* (*--create*) legt einen neuen Branch an, falls der angegebene Branch nicht existiert
  - Option -d (*--detach*) setzt HEAD auf einen beliebigen Commit (*detached head*)

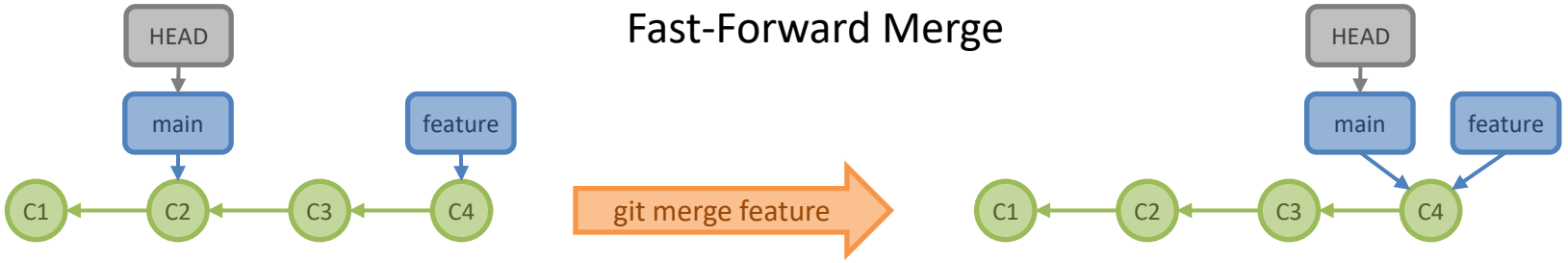


Merge

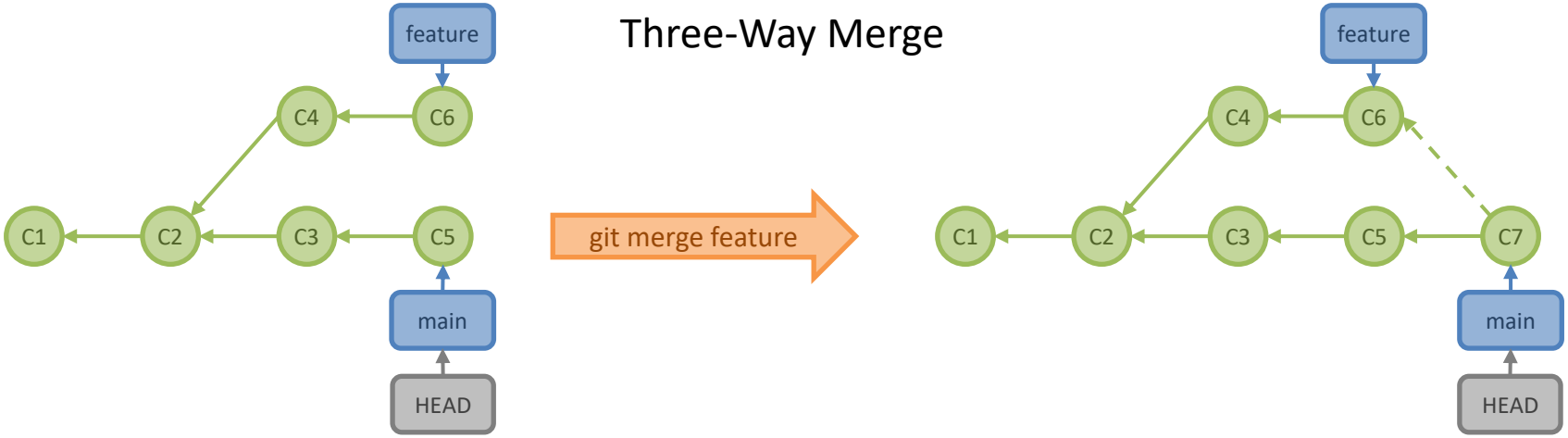
- *git merge*
  - integriert einen anderen Branch (Quellbranch) in den aktuellen Branch (Zielbranch)
- 2 mögliche Arten:
  - **Fast-Forward Merge:**
    - Quellbranch ist direkter Nachfolger des Zielbranch
    - Zielbranch muss nur auf den Commit gesetzt werden, auf den Quellbranch zeigt
  - **Three-Way Merge:**
    - Quellbranch und Zielbranch haben sich auseinanderentwicklet
    - Merge-Commit erforderlich, der beide Branches wieder zusammenführt
- bei einem Three-Way Merge entsteht ein neuer Merge-Commit mit zwei Vorgängern
- bei einem Three-Way Merge können Konflikte austreten, die manuell aufgelöst werden müssen
- Option *--no-ff* verhindert Fast-Forward Merges, wodurch immer ein Merge-Commit erstellt wird

Merge

Fast-Forward Merge



Three-Way Merge

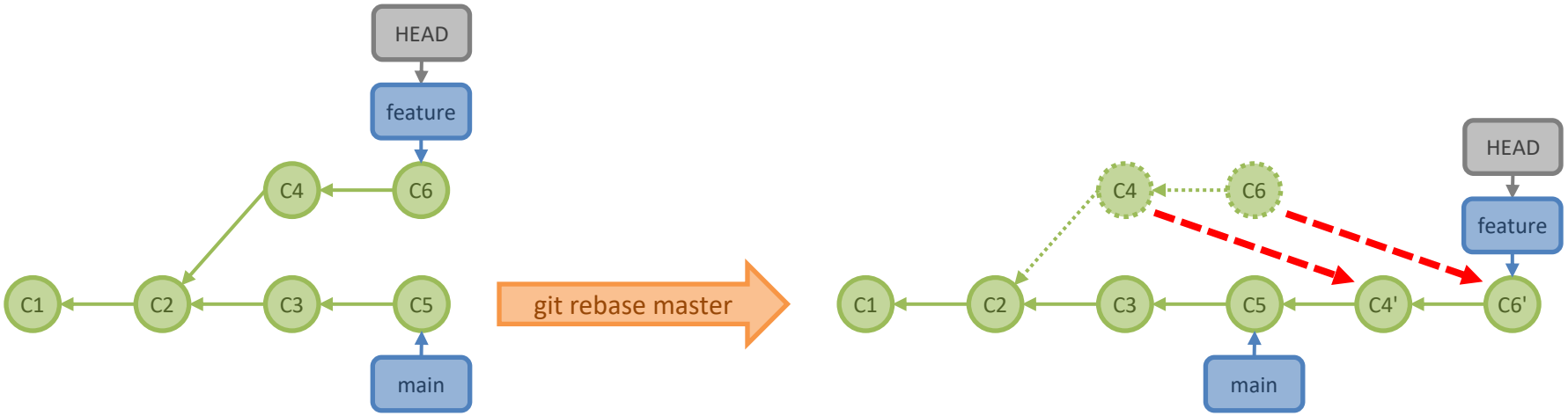


Rebase

- *git rebase* zieht Änderungen eines anderen Branches auf den Zielbranch (Replay)
- Zeiger des Branches liegt damit vor dem Zeiger des Zielbranches
- anschließender Merge auf dem Zielbranch ist damit nur ein Fast-Forward Merge

- Vorteil: hält die Historie des Zielbranch linear und somit einfach und übersichtlich
- Achtung: Rebase sollte immer nur auf lokalen Commits angewendet werden, die noch nicht in ein gemeinsames Repository übertragen wurden, da durch Rebase die Historie verändert wird

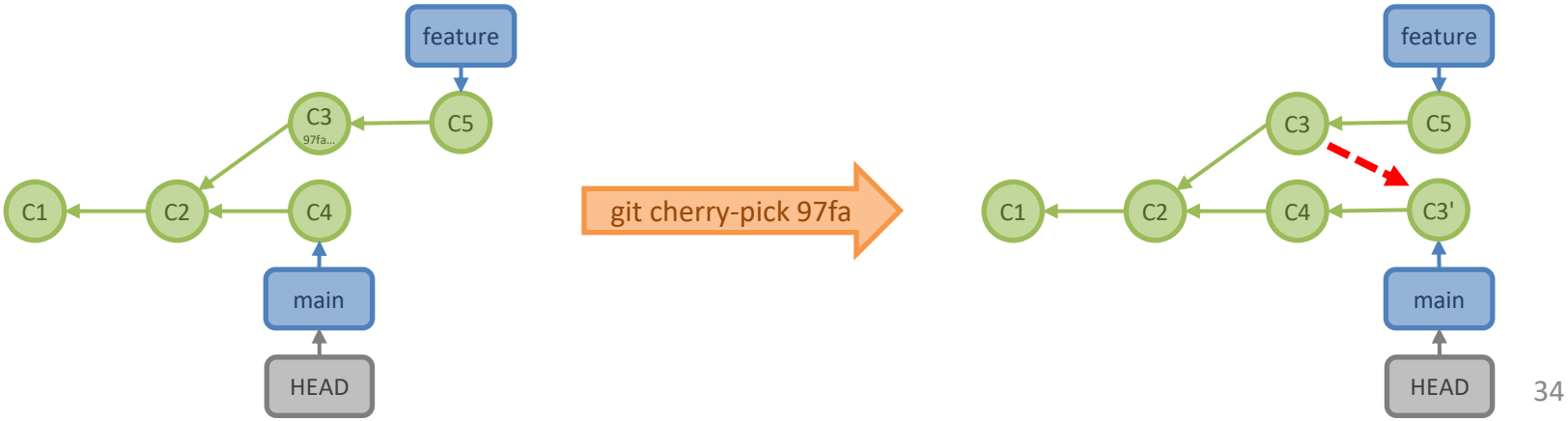
Rebase


HEAD
feature
C4
C6
C1
C2
C3
C5
main
git rebase master
HEAD
C4
C6
feature
C1
C2
C3
C5
C4'
C6'
main

Cherry Picking

- manchmal müssen einzelne Commits bereits vorzeitig in einen anderen Branch übernommen werden, noch bevor ein Merge/Rebase des gesamten Branch durchgeführt werden kann
  - z.B. wenn in einem Feature Branch ein dringender Bugfix implementiert wurde, bevor die Entwicklung des Features abgeschlossen ist
- dieses selektive Übernehmen und Anwenden von Commits wird als *cherry picking* bezeichnet

- *git cherry-pick*
  - überträgt einen oder mehrere Commits auf den aktuellen HEAD
  - Option *-n* (*--no-commit*) überträgt Änderungen nur in das aktuelle Arbeitsverzeichnis und in die Staging Area, führt aber keinen Commit durch
  - bei einem späteren Rebase werden bereits übernommene Commits automatisch ausgelassen



Stashing

- Wechsel auf einen anderen Branch schlägt fehl, wenn Änderungen im aktuellen Arbeitsverzeichnis nicht kompatibel sind
- falls die Änderungen im Arbeitsverzeichnis noch nicht mit einem Commit gespeichert werden sollen, müssen sie zwischengelagert werden

- *git stash*
  - speichert aktuelle Änderungen im Arbeitsverzeichnis und in der Staging Area in einem Stack ab
  - stellt einen sauberen Stand gemäß des aktuellen HEAD wieder her
  - die abgespeicherten Änderungen können später wiederhergestellt werden (auch auf einem anderen Branch)
  - *git stash list* listet gespeicherte Einträge auf
  - *git stash show* zeigt Details zu einem gespeicherten Eintrag
  - *git stash apply* überträgt einen Eintrag auf das aktuelle Arbeitsverzeichnis
  - *git stash drop* entfernt einen Eintrag
  - *git stash clear* löscht alle gespeicherten Einträge

Veränderungen der Historie

- Git erlaubt es, nachträglich Veränderungen an der Historie vorzunehmen (vgl. z.B. *git reset*, *git commit --amend*)
  - Bereinigung und Vereinfachung der Historie ist somit auch post factum noch möglich

- *git rebase* mit der Option *-i* (*--interactive*) ermöglicht ein Rebase, bei dem interaktiv festgelegt werden kann, was genau mit den einzelnen Commits geschehen soll
  - Optionen für jeden Commit sind:
    - *pick*: Commit wird übernommen
    - *reword*: Commit wird übernommen, aber die Commit Message wird verändert
    - *squash*: Commit wird mit dem vorherigen Commit verschmolzen
    - *drop*: Commit wird ausgelassen

- **ACHTUNG: eine Änderung der Historie sollte nur dann gemacht werden, wenn noch nicht gepusht wurde**

Reflog als Sicherheitsnetz

- Git protokolliert im Reflog alle Aktionen, die am HEAD durchgeführt werden
- kein Commit wird tatsächlich verändert oder geht verloren
  - z.B. *git commit --amend* verändert nicht den letzten Commit sondern erzeugt einen neuen Commit, der alte Commit kann im Reflog immer noch gefunden werden
- mit Hilfe des Reflog kann jede Aktion auch wieder rückgängig gemacht werden

- *git reflog* zeigt die Einträge im Reflog an
  - mit HEAD@{1}, HEAD@{2}, etc. kann auf die alten HEADs zurückgegriffen werden

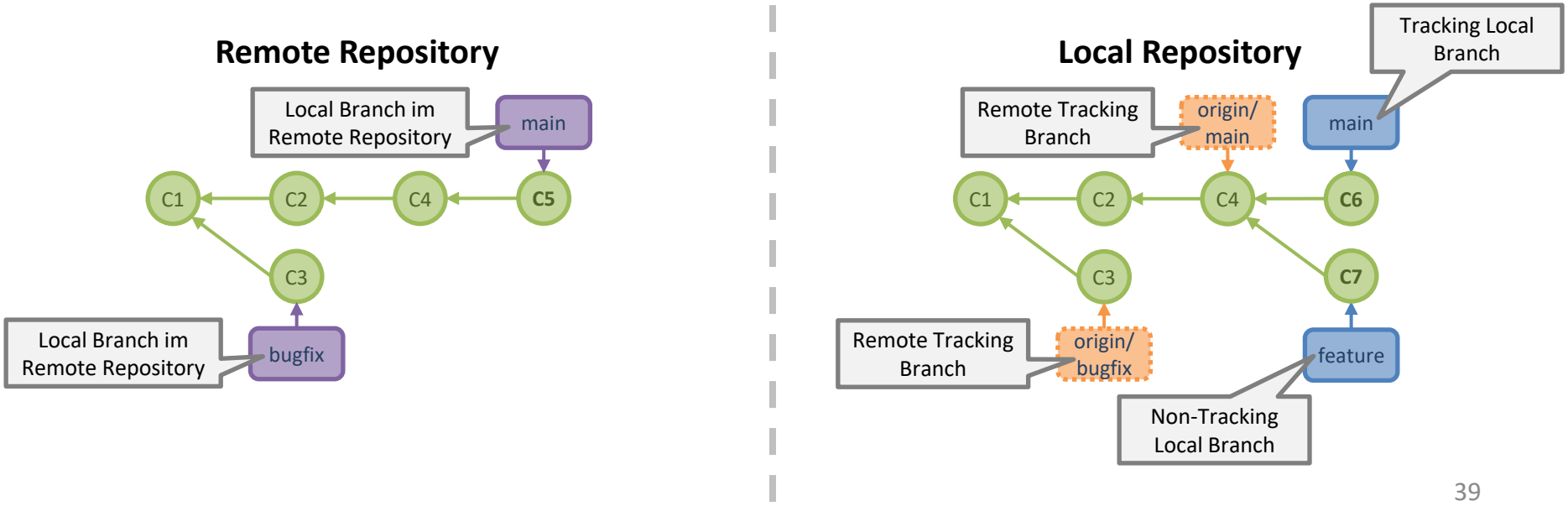
Austausch mit einem entfernten Repository

- *git clone* erstellt eine lokale Kopie eines entfernten Repositories
- entferntes Repository wird meist als *origin* bezeichnet
- lokales Repository muss regelmäßig mit dem entfernten Repository abgeglichen werden
- Branches im lokalen und im entfernten Repository können sich unterschiedlich entwickelt haben und müssen wieder integriert werden

- ein lokales Repository kann mit mehreren entfernten Repositories interagieren
- *git remote* dient zur Verwaltung der entfernten Repositories
  - *git remote -v* listet alle entfernten Repositories auf
  - *git remote show* zeigt Details zu einem entfernten Repository an
  - *git remote add* fügt ein entferntes Repository hinzu
  - *git remote remove* entfernt ein entferntes Repository

Remote Tracking Branches

- Branches im lokalen und im entfernten Repository müssen abgeglichen werden
- Remote Tracking Branches verbinden einen lokalen Branch mit einem Branch im entfernten Repository
  - z.B. *origin/main* entspricht dem Branch *main* im entfernten Repository
- Remote Tracking Branches repräsentieren den Stand eines Branches im entfernten Repository zum Zeitpunkt des letzten Abgleichs
- Remote Tracking Branches werden beim Abgleich automatisch aktualisiert
- Commits können nicht direkt auf Remote Tracking Branches geschrieben werden



Austausch mit einem entfernten Repository

- *git fetch* holt Änderungen von einem entfernten Repository
  - alle Remote Tracking Branches werden aktualisiert
  - Änderungen werden aber nicht automatisch mit lokalem Branch zusammengeführt
  - Integration muss anschließend manuell mit einem Merge gemacht werden
- *git pull* holt Änderungen von einem entfernten Repository wie *git fetch* und integriert diese sofort mit dem lokalen Branch
- *git push* überträgt lokale Änderungen in ein entferntes Repository
  - push ist nur erlaubt, wenn lokale Änderungen direkte Nachfolger des Branch im Remote Repository sind
  - Option *-u* (*--set-upstream*) erzeugt einen Branch im entfernten Repository
  - Option *-d* (*--delete*) löscht einen Branch im entfernten Repository
  - Option *-f* (*--force*) erzwingt einen Push, selbst wenn lokale Änderungen keine direkten Nachfolger des Branch im Remote Repository sind
  - **ACHTUNG: force push ist sehr problematisch, da dadurch Commits ausgehängt werden können, die von andere Personen noch verwendet werden**

Fetch vs. Pull