



Git

Ein dezentrales Versionsverwaltungssystem



Cegos Group

inspire
qualify
change



Inhalts- verzeichnis



Einführung



Workflows



Erste Schritte



Git-Clients



Arbeiten mit Git



Distributed Repositories



Git auf dem Server



Einführung



- Übersicht Versionsverwaltung
- Installation von Git
- Einrichtung
- Dokumentation und Referenz



Übersicht

Versionsverwaltung



Zentrale Versionsverwaltungs- systeme

- Zentrale Ablage auf dem Server des Version Control Systems (VCS)
- Beispiel
 - Subversion, CVS, Clearcase



Arbeitsweise: Zentrales Repository

- Alle Daten liegen auf dem Server
- Eine Kommunikation erfolgt ausschließlich über das zentrale Repository
- Sperrmechanismen sind möglich
 - aber nicht unbedingt notwendig und gewünscht
- Grundlegende Funktionen werden auf dem Server ausgeführt
- Datenhaltung häufig durch Erstellen einer Delta-Historie
- Authentifizierung und Autorisierung



Verteilte Versionsverwaltungs- systeme

- Dateiablage in gleichberechtigten Repositories
- Beispiel
 - Git, Mercurial




Arbeitsweise: Dezentrales Repository

- Jedes Repository ist prinzipiell gleichberechtigt
- Alle Funktionen können lokal ausgeführt werden
- Synchronisation mit anderen Repositories nur bei Bedarf
- Keine Sperrmechanismen
- Authentifizierung und Autorisierung nur bei Kommunikation mit anderen Repositories notwendig
- Zentrale Server-Lösungen sind möglich, aber nicht verpflichtend
 - Produkt-Lösungen
 - Atlassian BitBucket
 - GitHub
 - GitLab




Installation von Git


Download:
<https://git-scm.com/downloads>


 **git** --distributed-even-if-your-workflow-isnt


About
Documentation
Blog
Downloads
GUI Clients
Logos
Community

Downloads

 **Mac OS X**

 **Windows**

 **Linux**


 **Solaris**

Older releases are available and the [Git source repository](#) is on GitHub.

GUI Clients
Git comes with built-in GUI tools (**git-gui**, **gitk**), but there are several third-party tools

Logos
Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in

Latest source Release
2.6.3
[Release Notes \(2015-11-05\)](#)
[Downloads for Windows](#)



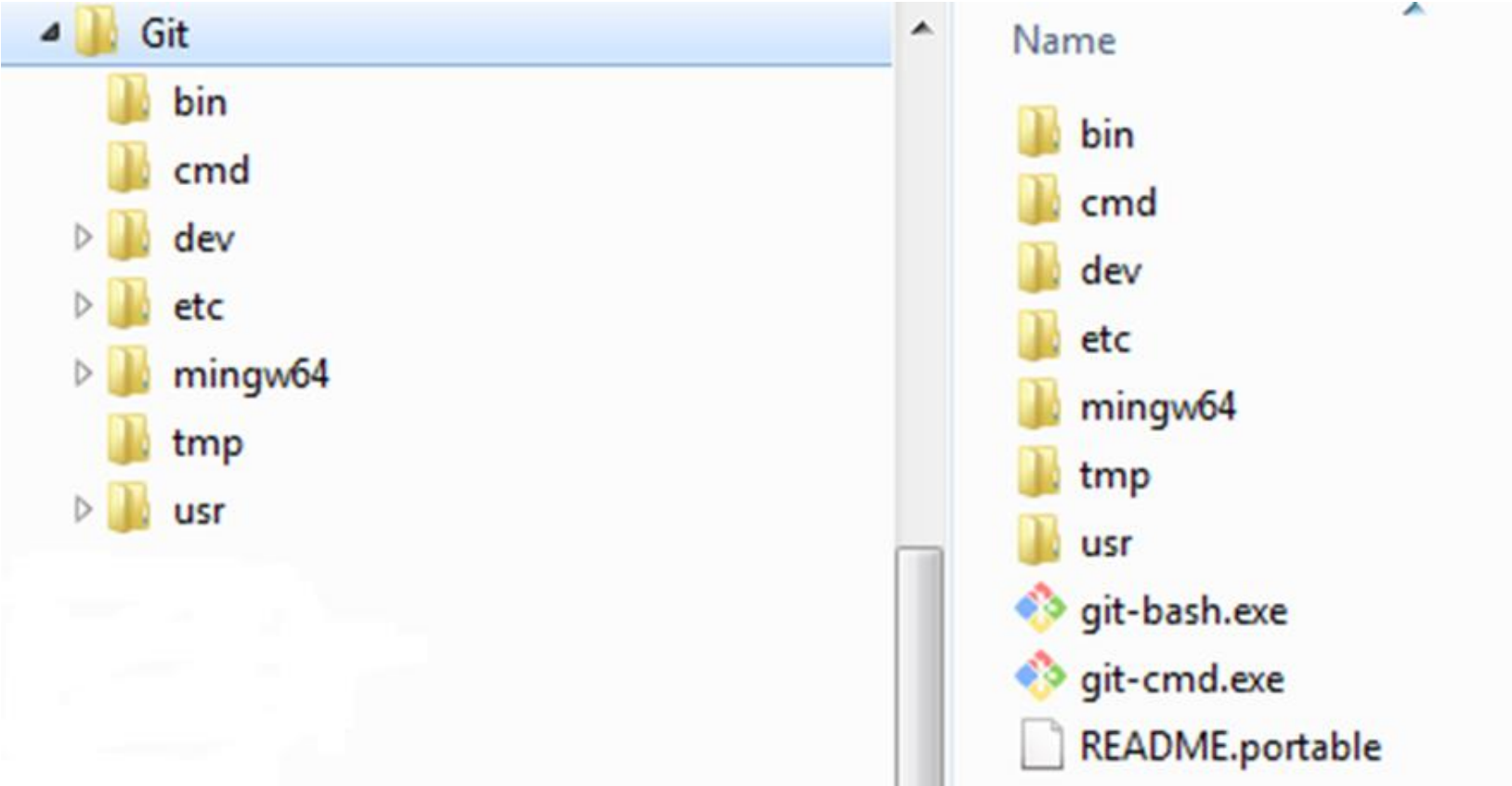


Git-Distributionen

- Verfügbar für alle gängigen Betriebssysteme
 - Linux
 - Windows
 - iOS
- Portable Installation verfügbar
 - Keine Änderung der System-Einstellungen notwendig
 - Verteilung als ZIP
 - Allerdings keine native Unterstützung von Git-Funktionen im Kontext-Menü der installierten Anwendungen
- `git-Executable` im `bin`-Verzeichnis



Verzeichnisse und Dateien





Einrichtung



Zentrale Konfiguration

- System-weit pro Rechner
- User-spezifisch
 - `.gitconfig` im User-Home
- Repository-spezifisch
 - `config` in `.git`
- Minimale Konfiguration enthält Benutzer und Mail-Adresse
 - Kommandozeilen-Tools
 - `git config -global user.name GitUser`
 - `git config -global user.email User@Git.org`
- Die Einstellungen werden in einer Textdatei abgelegt
 - `[user]`
 - `name = GitUser`
 - `email = User@Git.org`



Lokales Repository

- Der Befehl `git` ist prinzipiell nichts anderes als ein FileController
 - operiert auf einem Verzeichnis
 - `.git`
 - stellt in diesem eigenes, spezielles File-System zur Verfügung, das Repository
 - Sperrmechanismen und Daten-Integrität sind implementiert
- Bestandteile
 - Workspace mit beliebigem Inhalt
 - Stashing-Area mit beliebig vielen lokalen Kopien eines Workspaces
 - Weitere Meta-Informationen
 - Das eigentliche Repository
- Bare Repositories
 - Bestehen nur aus Meta-Informationen und dem Repository
 - Einsatz vorwiegend als gemeinsam genutztes Repository auf einem Git-Server

Verzeichnisstruktur eines Git-Projekts





Kommunikation zwischen Repositories

- Lokale Kommunikation über `file`-Protokoll
- Remote Kommunikation über Netzwerk
 - `http` und `https`
 - "Smart" mit speziellen Git –Kommandos
 - "Dump" mit Standard-http-Verben
 - SSH
 - jeweils mit Authentifizierung



Dokumentation und Referenz



Hilfefunktion

- Bestandteil der Distribution
 - Aber auch Online verfügbar
 - <https://git-scm.com/docs>
- Aufruf lokal
 - `git help <command>`



Beispiel:

Hilfefunktion für log

git-log(1) Manual Page

NAME

git-log - Show commit logs

SYNOPSIS

```
git log [<options>] [<revision range>] [[--] <path>...]
```

DESCRIPTION

Shows the commit logs.

The command takes options applicable to the `git rev-list` command to control what is shown and how, and options applicable to the `git diff` commands to control how the changes each commit introduces are shown.

OPTIONS

`--follow`

Continue listing the history of a file beyond renames (works only for a single file).

`--no-decorate`

`--decorate[=short|full|no]`

Print out the ref names of any commits that are shown. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. The default option is *short*.



Cheat Sheet von github.com

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
<https://windows.github.com>

GitHub for Mac
<https://mac.github.com>

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
<http://git-scm.com>

CONFIGURE TOOLING

Configure user information for all local repositories

<code>\$ git config --global user.name "[name]"</code>
Sets the name you want attached to your commit transactions
<code>\$ git config --global user.email "[email address]"</code>
Sets the email you want attached to your commit transactions
<code>\$ git config --global color.ui auto</code>
Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

<code>\$ git init [project-name]</code>
Creates a new local repository with the specified name
<code>\$ git clone [url]</code>

MAKE CHANGES

Review edits and craft a commit transaction

<code>\$ git status</code>
Lists all new or modified files to be committed
<code>\$ git diff</code>
Shows file differences not yet staged
<code>\$ git add [file]</code>
Snapshots the file in preparation for versioning
<code>\$ git diff --staged</code>
Shows file differences between staging and the last file version
<code>\$ git reset [file]</code>
Unstages the file, but preserve its contents
<code>\$ git commit -m "[descriptive message]"</code>
Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

<code>\$ git branch</code>
Lists all local branches in the current repository
<code>\$ git branch [branch-name]</code>
Creates a new branch
<code>\$ git checkout [branch-name]</code>
Switches to the specified branch and updates the working directory
<code>\$ git merge [branch]</code>
Combines the specified branch's history into the current branch
<code>\$ git branch -d [branch-name]</code>



Cheat Sheet von github.com, Seite 2

REFACTOR FILENAMES

Relocate and remove versioned files

\$ git rm [file]

Deletes the file from the working directory and stages the deletion

\$ git rm --cached [file]

Removes the file from version control but preserves the file locally

\$ git mv [file-original] [file-renamed]

Changes the file name and prepares it for commit

SUPPRESS TRACKING

Exclude temporary files and paths

***.log
build/
temp-***

A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

\$ git ls-files --other --ignored --exclude-standard

Lists all ignored files in this project

REVIEW HISTORY

Browse and inspect the evolution of project files

\$ git log

Lists version history for the current branch

\$ git log --follow [file]

Lists version history for a file, including renames

\$ git diff [first-branch]...[second-branch]

Shows content differences between two branches

\$ git show [commit]

Outputs metadata and content changes of the specified commit

REDO COMMITS

Erase mistakes and craft replacement history

\$ git reset [commit]

Undoes all commits after [commit], preserving changes locally

\$ git reset --hard [commit]

Discards all history and changes back to the specified commit



Online Referenz

- <https://git-scm.com/>
- <http://gitref.org/>



Erste Schritte



Das Repository



Git Kommandos



Abläufe im Detail



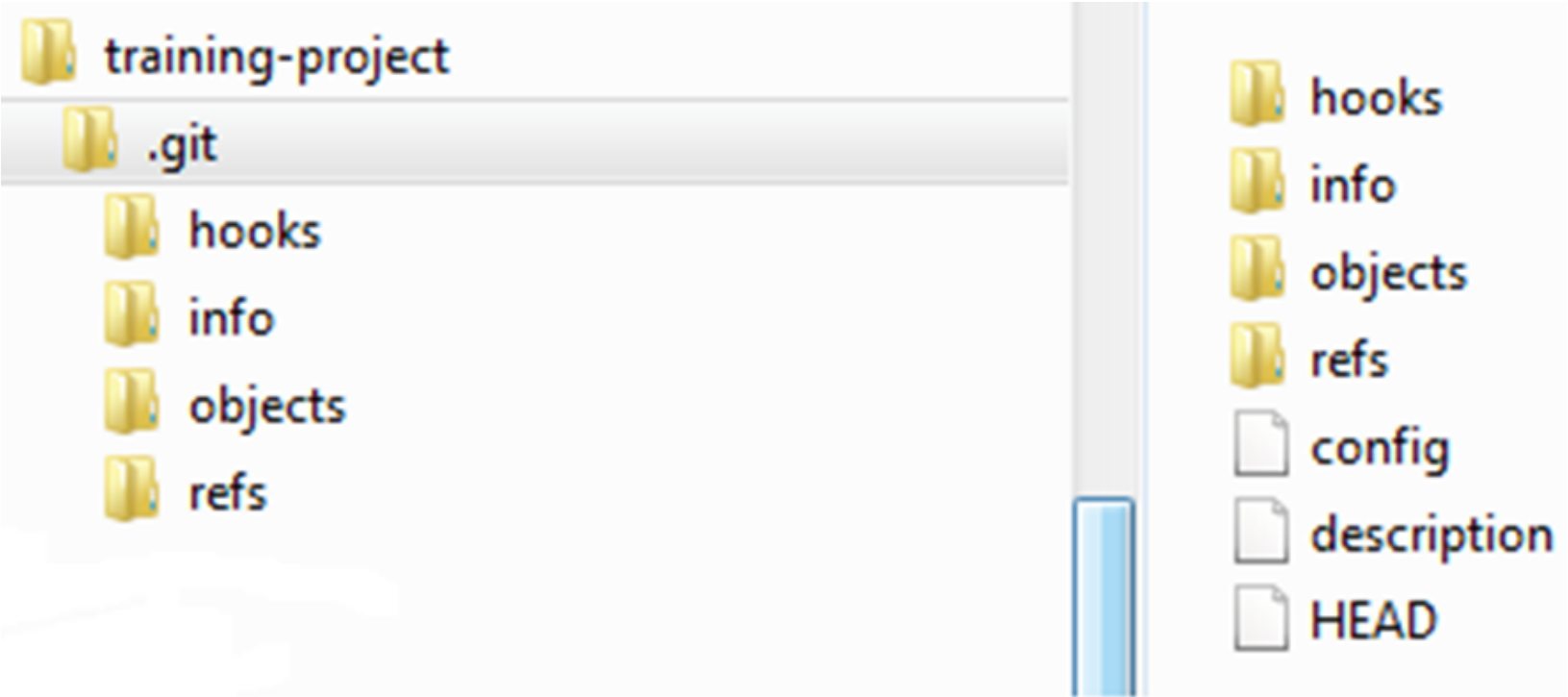
Das Repository



Anlegen eines neuen Git-Repositories

- Verzeichnis anlegen
- Darin aufrufen
 - `git init`
- Damit ist bereits ein komplett funktionierendes Repository eingerichtet
 - und kann sofort benutzt werden!
- Alternativ: Clone eines entfernten Repositories
 - `git clone <URL>`
 - Nun wird das Repository von der angegebenen URL kopiert
 - Eine weitere Verbindung zu dem Original-Repository ist nach dem clone nicht mehr notwendig
 - Der Clone "weiß", von wem er stammt
 - `push` und `pull` benutzen diese Information
 - Details später

Verzeichnisstruktur eines Git- Repositories





Bestandteile des Git-Projekts

- Das Arbeitsverzeichnis, der "workspace"
 - Ein normales Verzeichnis, das die Benutzer-Dateien enthält
- `.git` enthält das eigentliche Repository
 - Dieses Verzeichnis wird von `git` gepflegt
 - Benutzer sollten die Existenz dieses Verzeichnisses ignorieren
 - Insbesondere ist eine Manipulation dieses Verzeichnisses zu unterlassen
- Logische Unterteilung des Repositories
 - Stash-Verzeichnis
 - Staging- oder Index-Bereich
 - Weitere Meta-Informationen



Bereiche des Repositories

- Stashes
 - Ein Stash ist nichts anderes als ein Backup des aktuellen Arbeitsverzeichnisses
 - Hat nichts mit Versionierung zu tun!
 - Damit kann der Workspace bei Bedarf komplett weggesichert werden
- Stage oder Index
 - Die Stage-Area enthält
 - Kompaktierte Dateien
 - Diese werden über einen Hashwert identifiziert
 - Dieser wird weltweit eindeutig generiert
 - Der Hash ist Analog zu "Referenzen" einer Objekt-orientierten Sprache
- Weitere Meta-Informationen
 - Commit-Objekte
 - Tags
 - Branches
 - Remote Branches
 - Upstream
 - Downstream



Git Kommandos



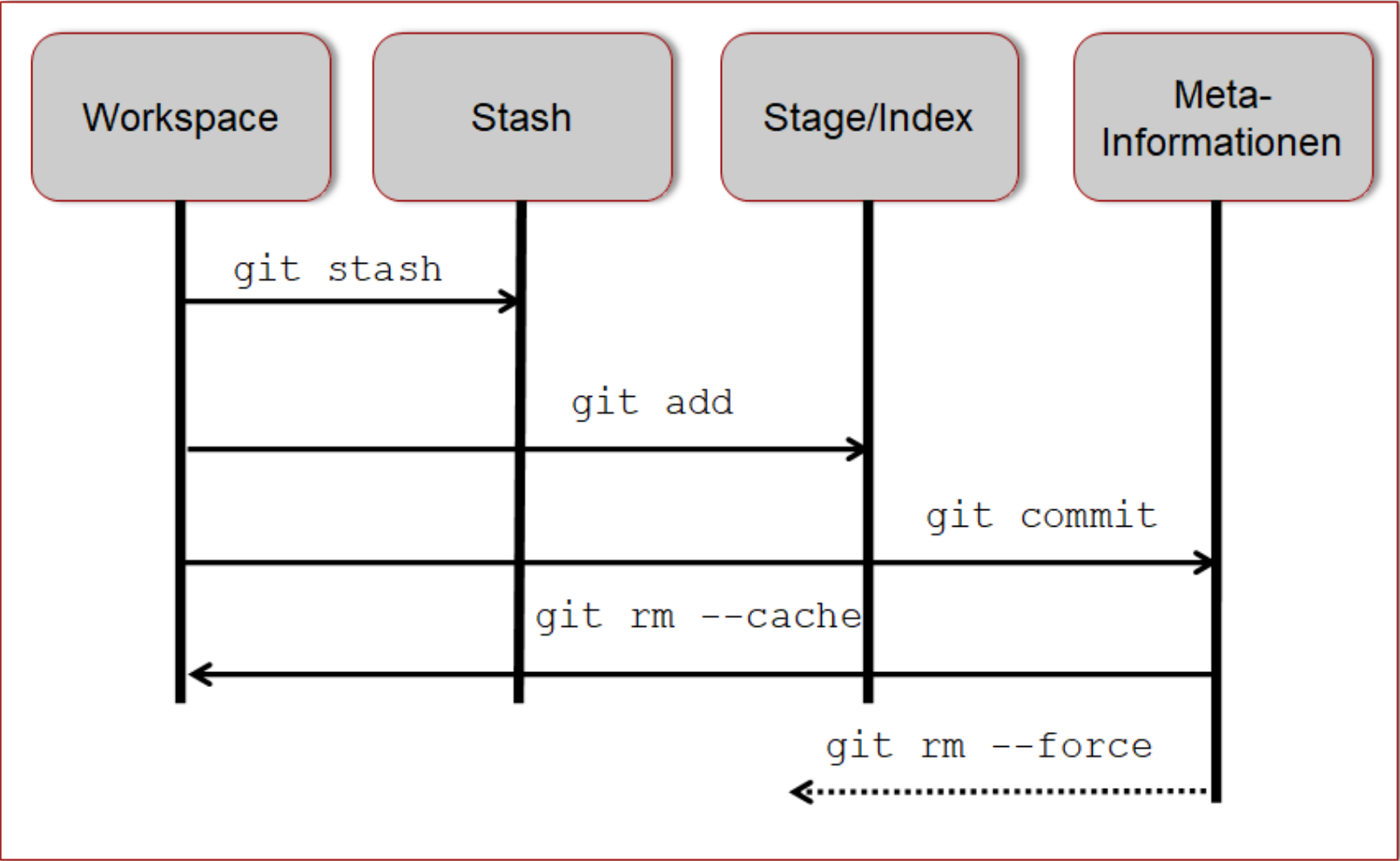
Erste wichtige git-Kommandos

- `status`
- `add`
- `commit`
- `checkout`
- `log`
- `stash`
- `rm`



Abläufe im Detail

Kommandos im Git-Projekt





Ablauf: Stashing

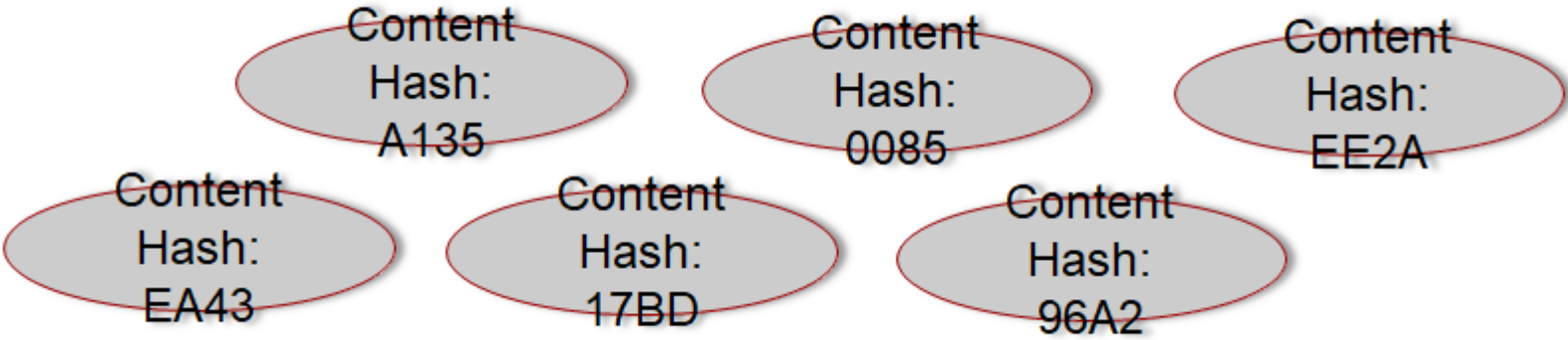
- `git stash`
 - Der gesamte Workspace wird unter einem Stash-Namen im Stash-Directory abgelegt
 - Ein simpler Backup



Ablauf: add

- `git add <file>`
 - Für `<file>` wird ein Hashcode erzeugt
 - Die Datei wird komprimiert und im Index abgelegt
- Nach dem `add` ist die Datei bereits im Repository bekannt, aber noch nicht bestätigt
- Damit ist dieser Vorgang nur ein Zwischenschritt, nicht ein stabiler End-Zustand
 - Die hinzugefügten Dateien sind überwacht
- Jede hinzugefügte Datei wird vollständig verarbeitet
 - Keine Delta-Historien!
 - Es ist wichtig, Dateien einfach wiederherstellen zu können
 - Der verschwendete Platz auf der Festplatte wird dabei akzeptiert

Content-Objekte

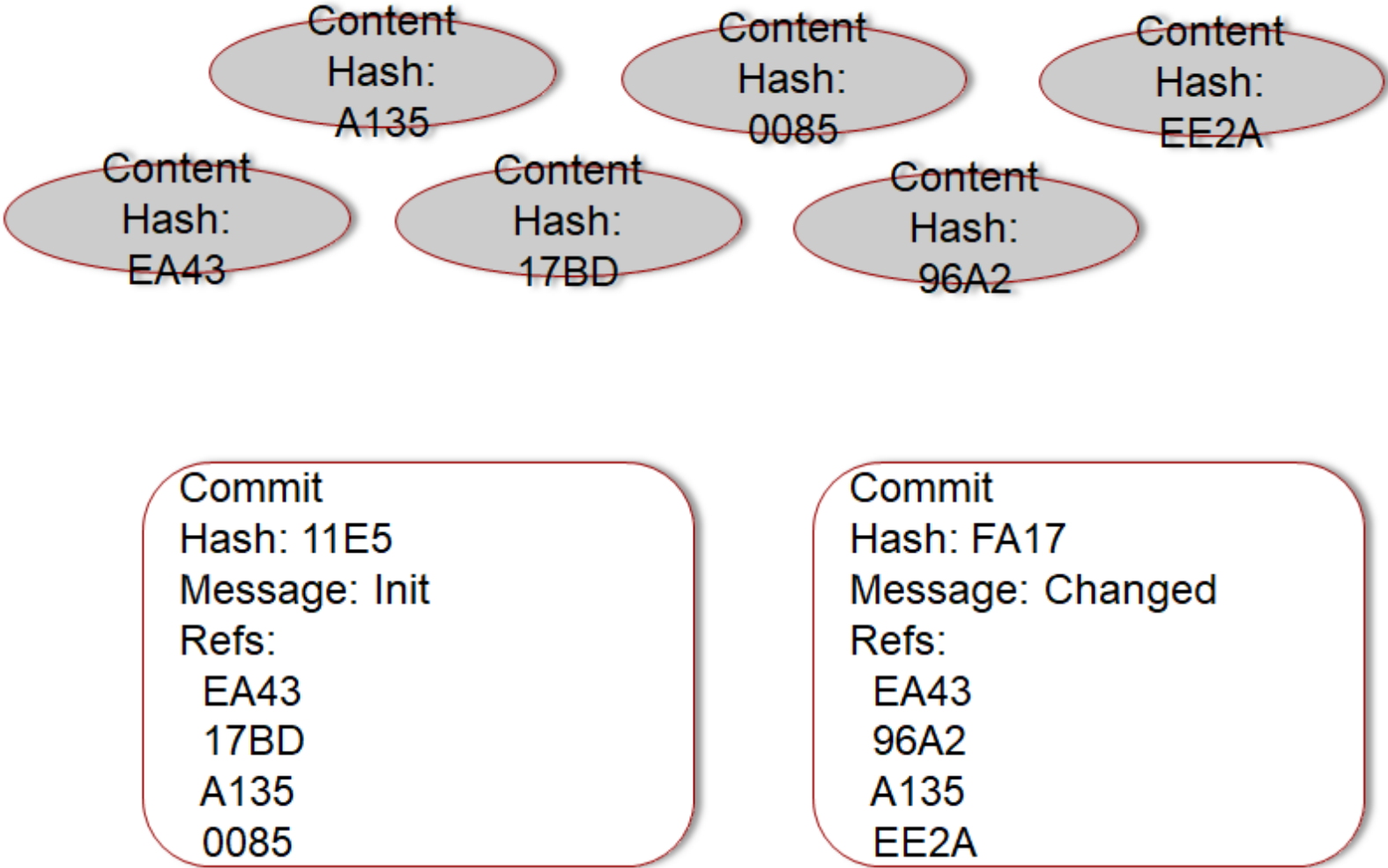




Ablauf: commit

- `git commit`
 - Es wird ein Commit-Objekt erzeugt
 - Committer
 - Timestamp
 - Commit Message
 - Einer Liste aller Hashwerte aller Objekte, die aktuell im Index vorhanden sind
- Damit wird durch den Commit Struktur in den "Brei" der Content-Objekte gebracht
- Commit-Objekte sind stets vollständig
 - Selbst wenn nur eine einzige Datei geändert wurde enthält das Commit-Objekt eine vollständige Liste
 - Auch hier gilt: Einfachheit geht vor Plattenbelegung

Commit-Objekte





add und commit im Detail

- `add` ist ein Transfer in den Staging-Bereich
- Änderungen nach dem `add` sind lokale Änderungen im Arbeitsverzeichnis
 - und müssen deshalb gegebenenfalls nochmals hinzugefügt werden
- `git commit -a`
 - Alle getrackten Dateien werden mit ihren Änderungen committed
 - Damit müssen bereits im Index vorhandene Dateien vor dem `commit` nicht nochmals hinzugefügt werden



Commits: Klassische Versionsverwaltung

- Delta-Informationen



Commits: Git Snapshots

- Jeder Commit ist ein vollständiger Snapshot



Aktualisierung des Arbeitsverzeichnisses

- `git stash apply`
- `git checkout <hash>`
 - Der über den Hash identifizierte Commit wird in den Arbeitsbereich kopiert
- Hinweis
 - Benutzer verwenden aber selten direkt Hashes
 - Tags und Branches ermöglichen ein komfortables Arbeiten

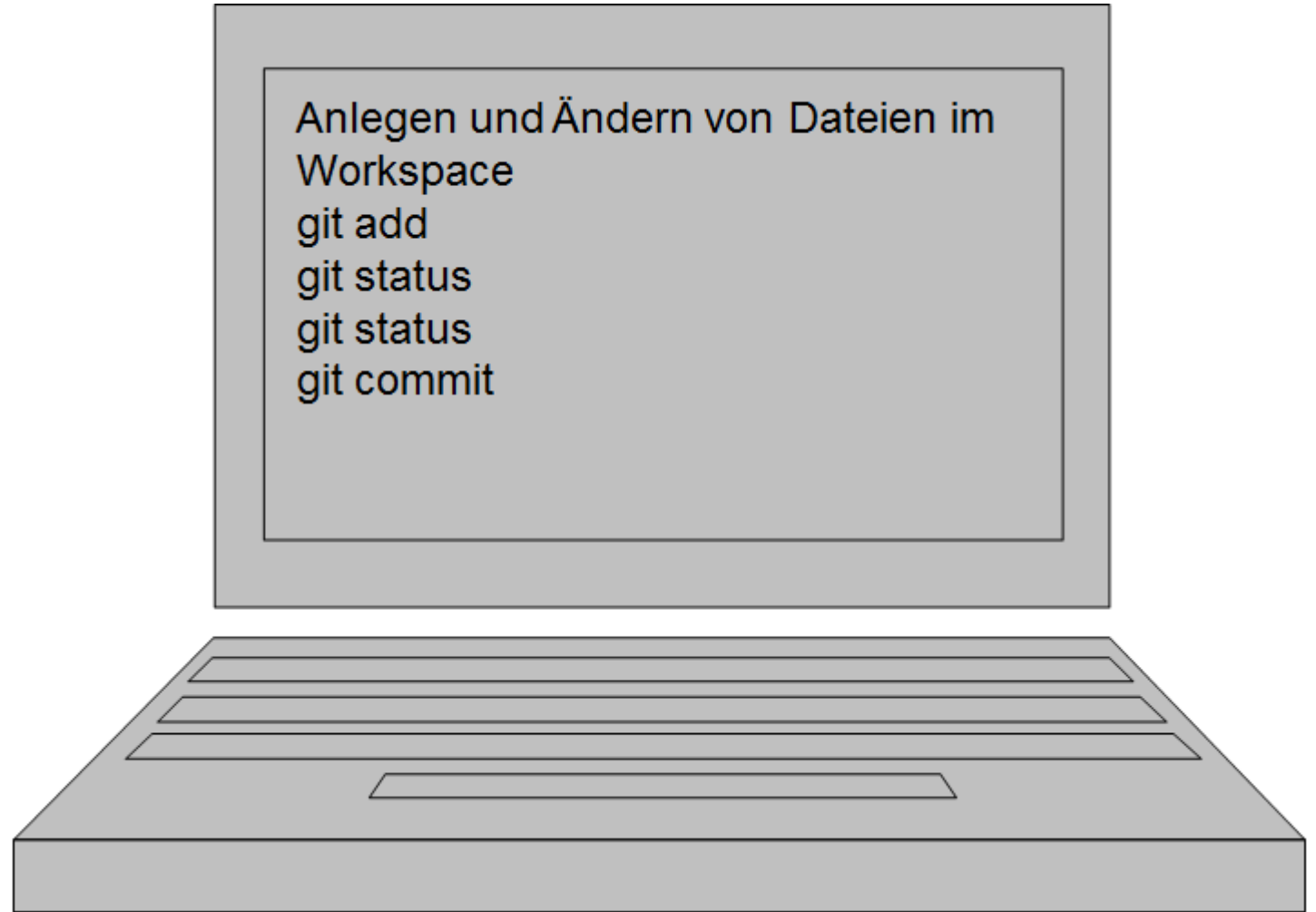


Abgreifen von Informationen

- `git status`
 - Zeigt an, welche Dateien sich aus der Sicht von Git heraus in einem unsynchronisierten Zustand befinden
- `git log`
 - Logging-Ausgaben der bekannten Commits
 - unter anderem der Hash des Commit-Objekts
 - Die Ausgabe kann durch eine Vielzahl von Optionen kontrolliert werden
 - `git log --decorate`
 - `git log --branches`



Hands on!





Arbeiten mit Git



Tags



Branches



Merging



Rebasing



Customizing



Tags



Tags in Git

- Ein Tag ist eine interne, unveränderbare Referenz auf einen Commit
 - Auf Grund der Snapshot-Technik ist ein Tag in Git damit extrem einfach
- Zwei Kategorien
 - Lightweight
 - Nur die Referenz
 - Annotated
 - Zusätzliche Meta-Informationen
 - Message
 - Committer
 - Timestamp
 - Optionale Signatur des Committers
 - Damit können Tags verifiziert werden



Tag-Verwaltung

- `git tag <new-lightweight-tagname>`
- `git tag -a -m "message" <new-annotated-tagname>`
- **Standard-Optionen**
 - `-l`
 - Liste
 - `-d`
 - Löschen



Branches



Commit und Parents

- Jeder Commit kennt seinen Parent



HEAD

- Ein Commit wird identifiziert über
 - Einen SHA-Hash
 - ein Tag
- Ein Branch referenziert ein Commit-Objekt
- HEAD referenziert den aktuell ausgecheckten Branch
- Falls ein Commit-Objekt über einen Hash-Wert oder ein Tag ausgecheckt worden ist, befindet sich Git in einem Ausnahmestand
 - "Detached HEAD"
 - In diesem Zustand sollte nur ein Tag oder ein neuer Branch angelegt werden
 - Keine Commits durchführen!
 - Der HEAD wird durch den checkout eines Branches wieder attached



Anlegen eines neuen Branches

- `git branch <new-branch-name>`
 - z.B. `testing`

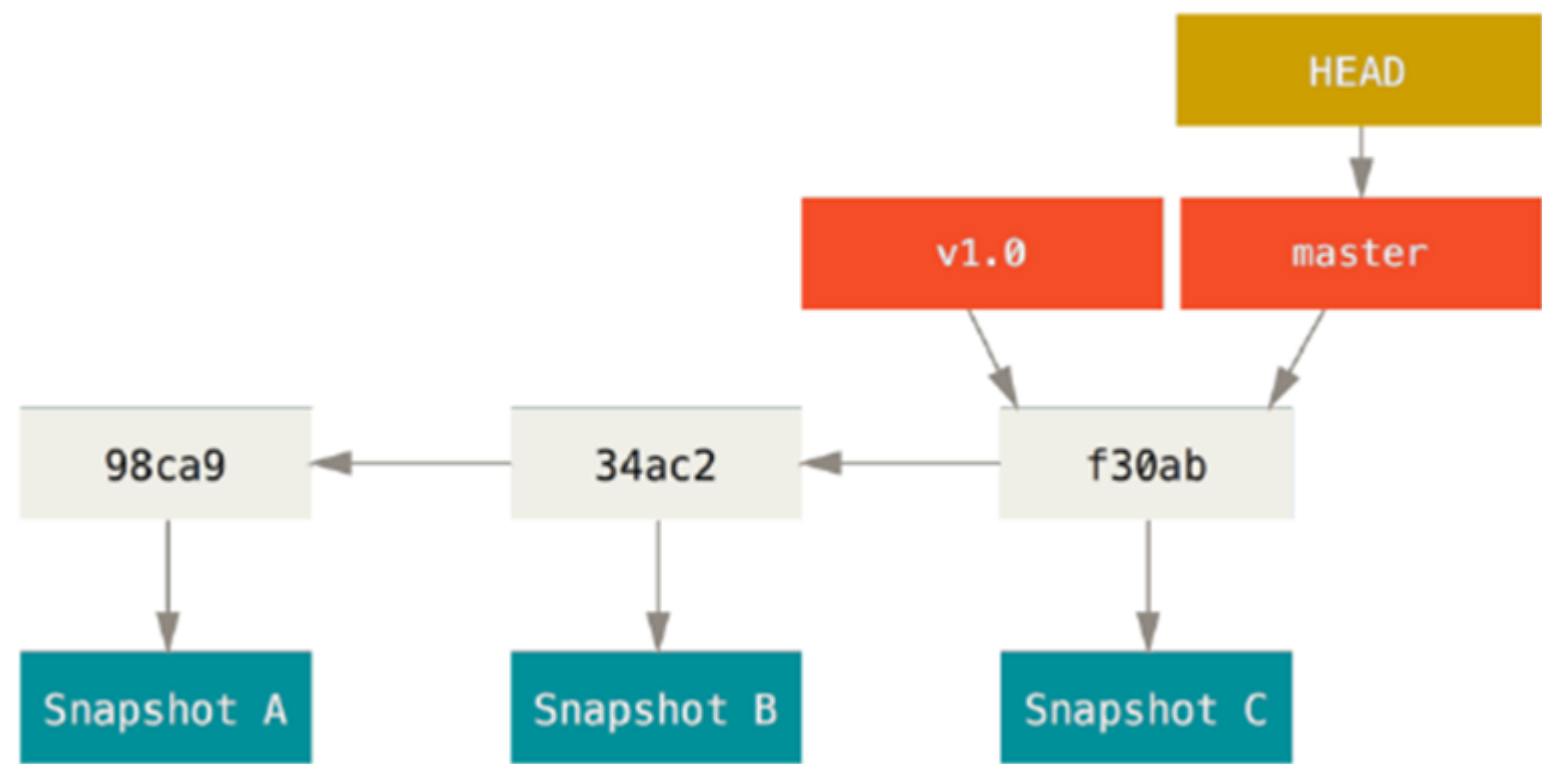


Wechsel des Branches

- `git checkout master`



Tag, Branch und HEAD





Wechsel eines Branches

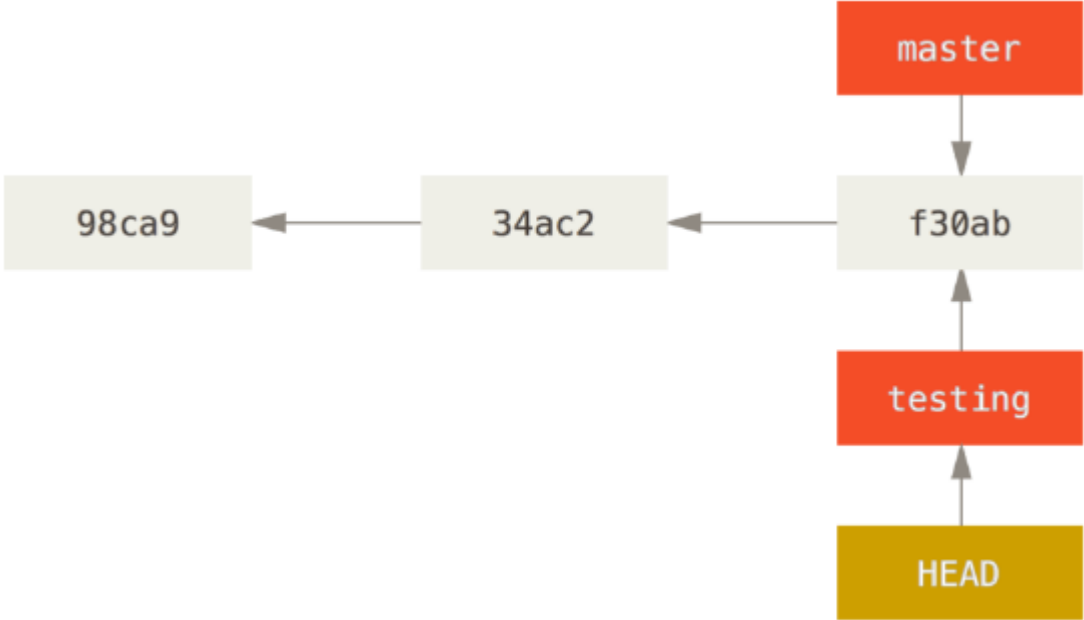
- `git checkout <branch-name>`
 - z.B. `testing`



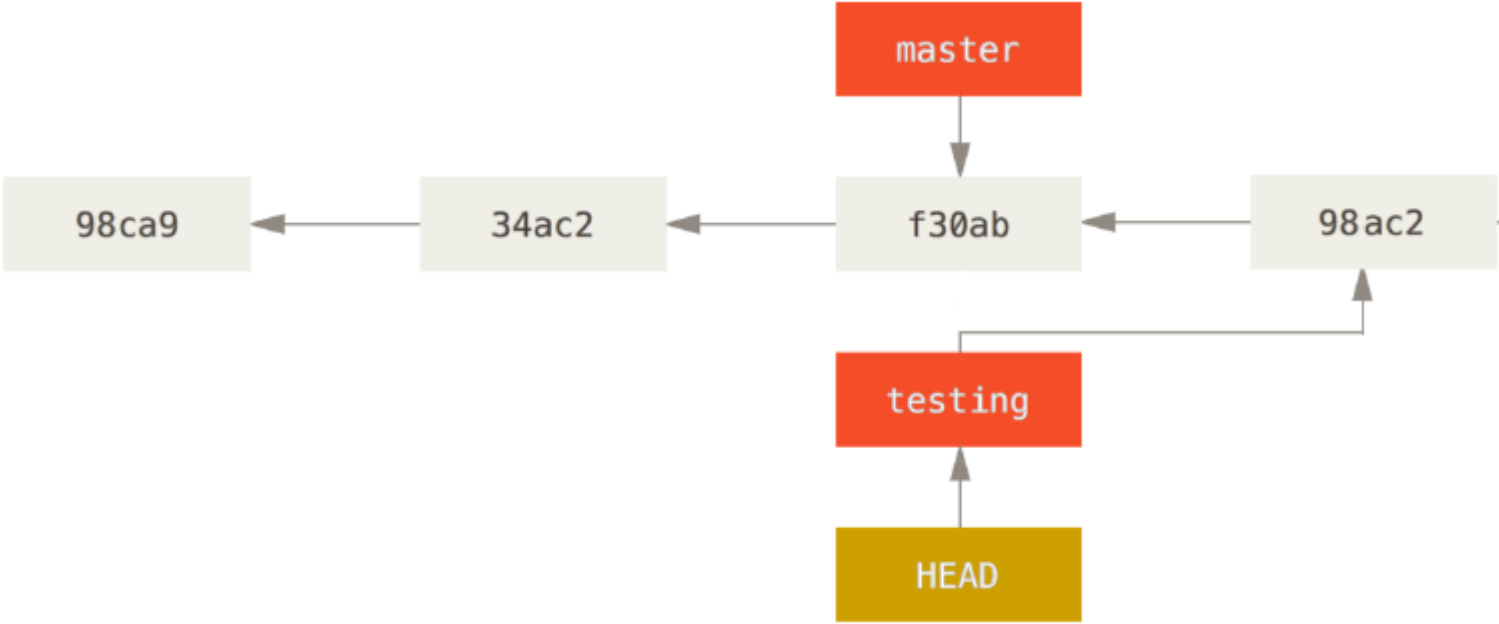
Branches, HEAD und Commit

- Was passiert bei einem Commit mit dem HEAD?
 - Nichts!
 - Der HEAD referenziert weiter den attached Branch
- Was passiert bei einem Commit mit dem ausgecheckten Branch?
 - Dieser referenziert das neu erzeugte Commit-Objekt
 - Damit bewegt sich der Branch
 - Der HEAD wird nur indirekt mitgezogen

Branch und Commit: Ausgangssituation



Branch und Commit: Nach Commit

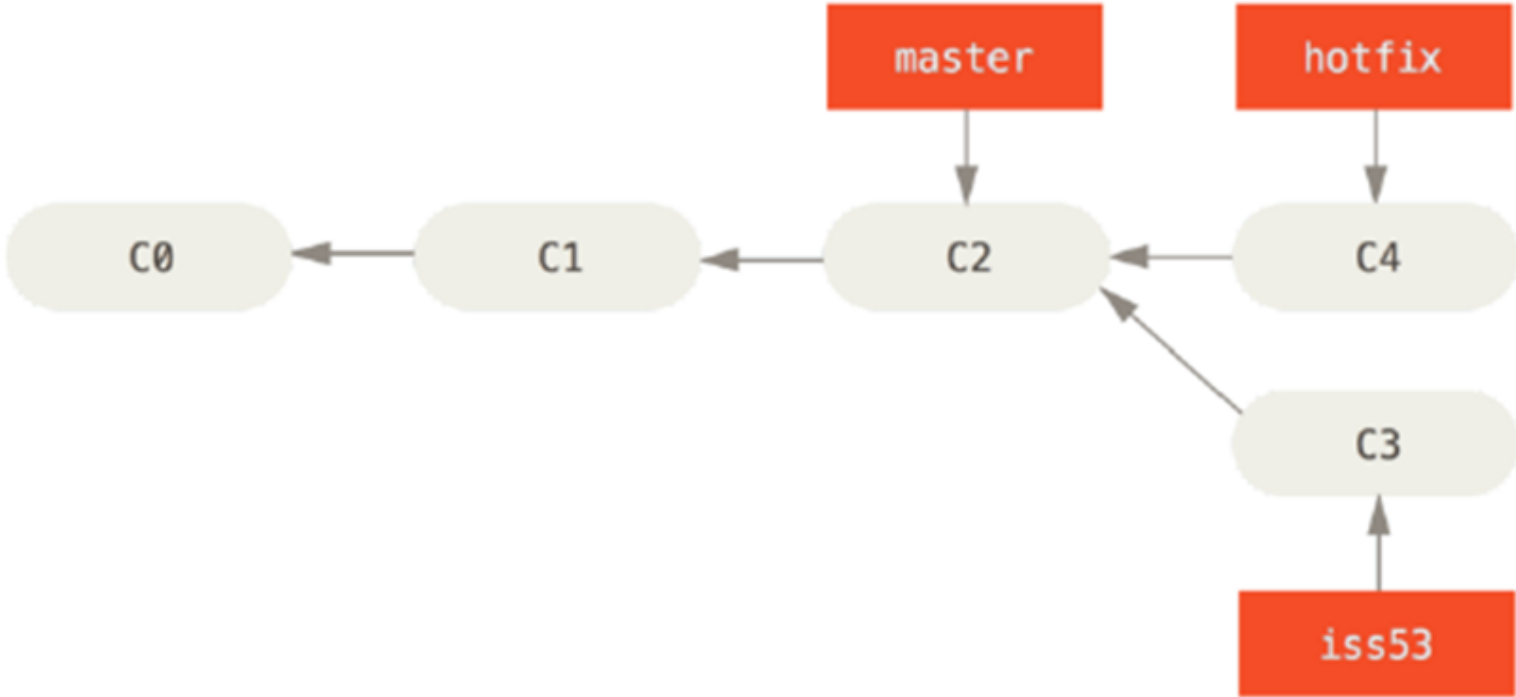




Merging



Fast Forward Merge: Ausgangssituation





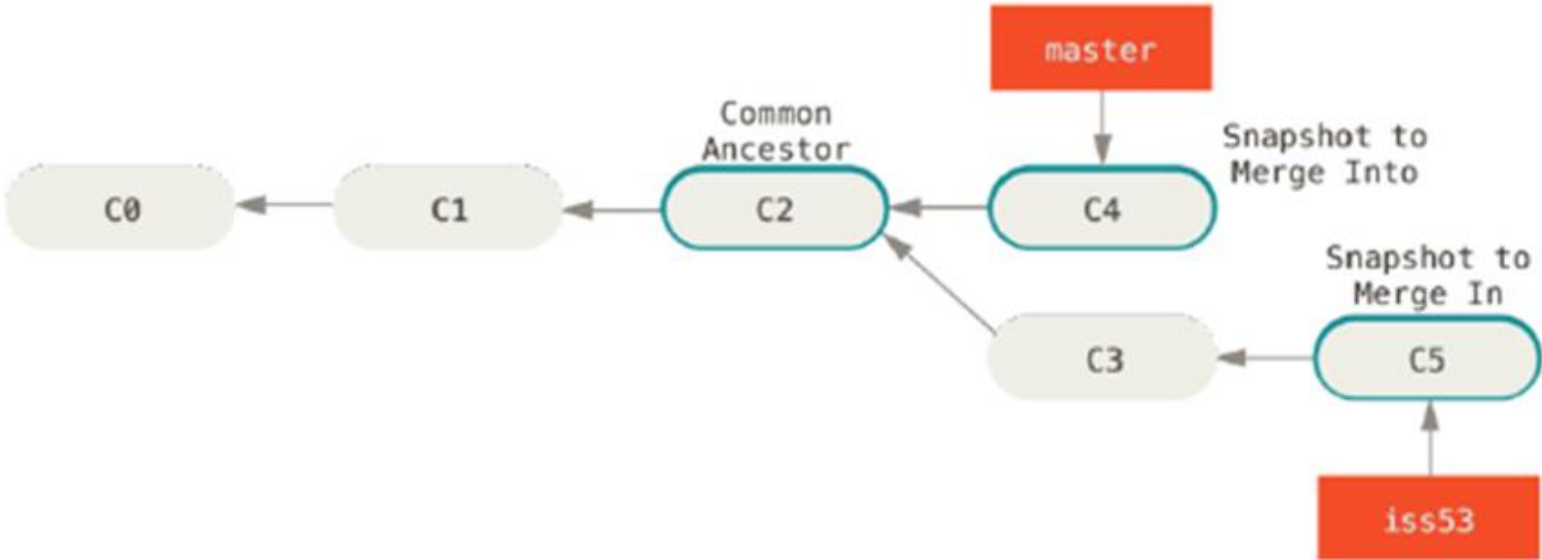
Fast Forward

Merges: Nach merge

- `git merge hotfix`



Recursive merge: Ausgangssituation





Recursive merge: Nach merge

- `git merge iss53`



Merge Konflikte

- Git hat bereits ein sehr ausgefeiltes Konzept, um Merge-Konflikte zu erkennen
 - Einfache Konflikte werden automatisch korrigiert
- Nicht-auflösbare Konflikte müssen händisch behoben werden
 - Was auch sonst...
- Git: "conflict resolution markers"

```
<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```



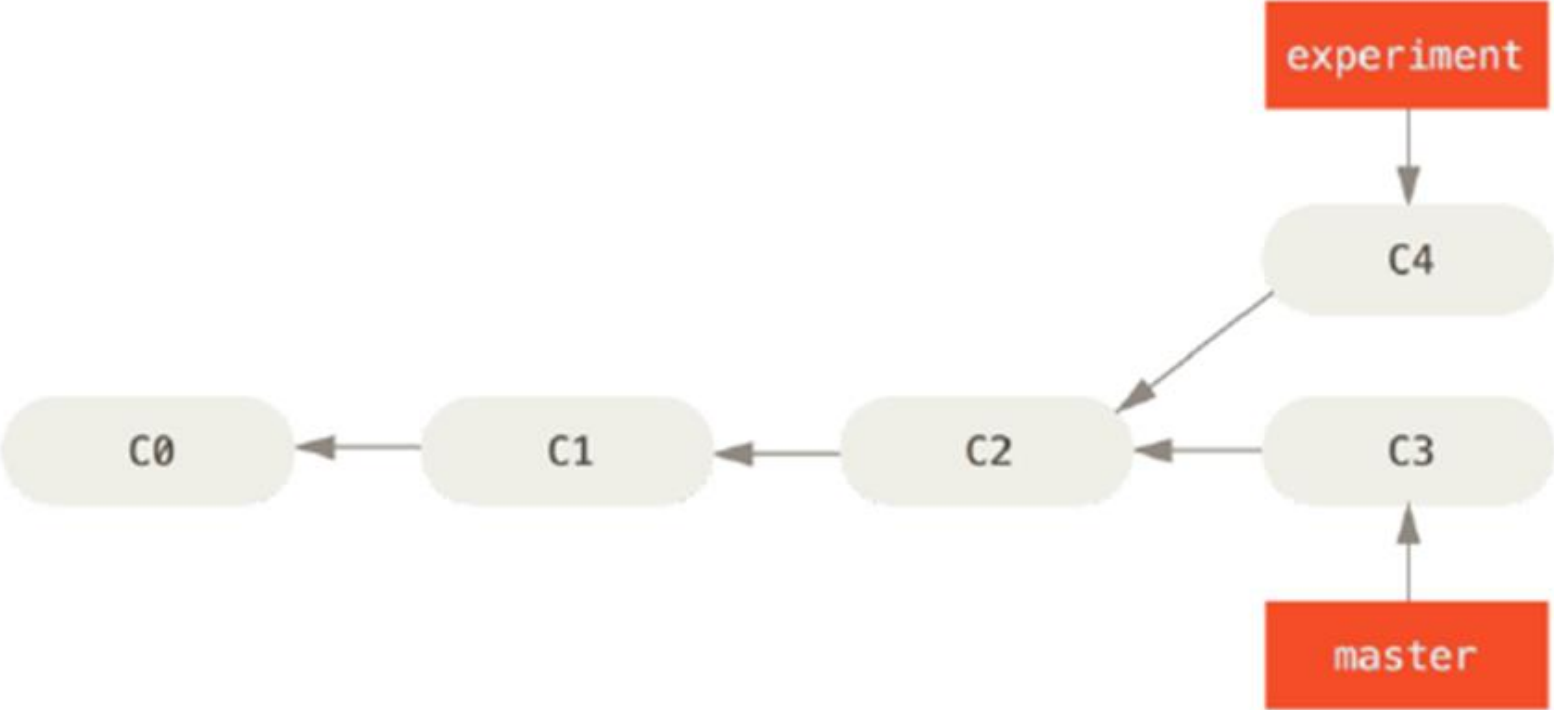

Rebasing



Arbeitsweise

- Statt eines rekursiven Mergings werden commits auf dem zweiten Branch "nachgespielt"
- Im Vergleich zum Merging wird kein neuer Snapshot erstellt, sondern es werden vorhandene Commits abgeändert
- VORSICHT
 - "Kein Rebase für Commits, die bereits außerhalb eines lokalen Repositories bekannt sind!"

Rebasing: Ausgangssituation





Rebasing: nach rebase

- `git rebase master`
- Anschließend: Simpler Fast Forward



Rebasing-Sequenz

- Treten bei einem Rebase Konflikte auf müssen diese analog zum Merge gelöst werden
- Durch das Ändern des Commit-Objekts ist dieser Vorgang jedoch komplexer und wird von Git "transaktionell" gesteuert
 - `git rebase`
 - `git rebase --continue`
 - `git rebase --skip`
 - `git rebase --abort`



Weitere Merging-Verfahren

- Cherry Picking
 - Ein Commit wird in einen beliebigen anderen Commit integriert
 - `git cherry-pick <hash>`
 - Damit ähnlich zum Rebasing
- Patches
 - Patches sind exportierte Commits
 - Diese können an beliebiger Stelle eingespielt werden



Hands on!





Customizing



Alias für Befehle

- In der Git-Konfiguration können für Befehle Alias-Namen definiert werden
- Insbesondere interessant für Kommandozeilen-Befehle mit (aufwändiger) Parametrisierung



Custom Kommandos

- Git kann jedes vom Betriebssystem ausführbare Skript als eigenes Kommando ausführen
 - Es muss also nur ein Skript-Interpreter gefunden werden
 - Die Programmiersprache, in der das Skript geschrieben wird, ist damit egal
- Name des Skripts: `git-<Kommando>`



Hooks

- `git` ruft bei bestimmten Aktionen Callback-Funktionen auf: "Hooks"
- Hooks werden von `git` als Skript-Programme aufgerufen
 - Unter Linux beginnt das Skript damit mit einer Shebang-Anweisung
 - Unter Windows ist die Bash-Shell Bestandteil der Distribution
- Beispiele
 - `pre-commit`
 - `commit-message`
 - `post-commit`
- Parametrisierung
 - `git` ruft die Skripte mit Aufrufparametern auf
 - Außerdem wird ein Satz von Git-typischen Environment-Variablen gesetzt
- Die Exit-Codes des Scripts werden von Git zur weiteren Verarbeitung ausgewertet



Übersicht

- <https://git-scm.com/docs/githooks>
- <https://www.digitalocean.com/community/tutorials/how-to-use-git-hooks-to-automate-development-and-deployment-tasks>



Distributed Repositories



Pull und Push



Remote Konfiguration



Pull und Push

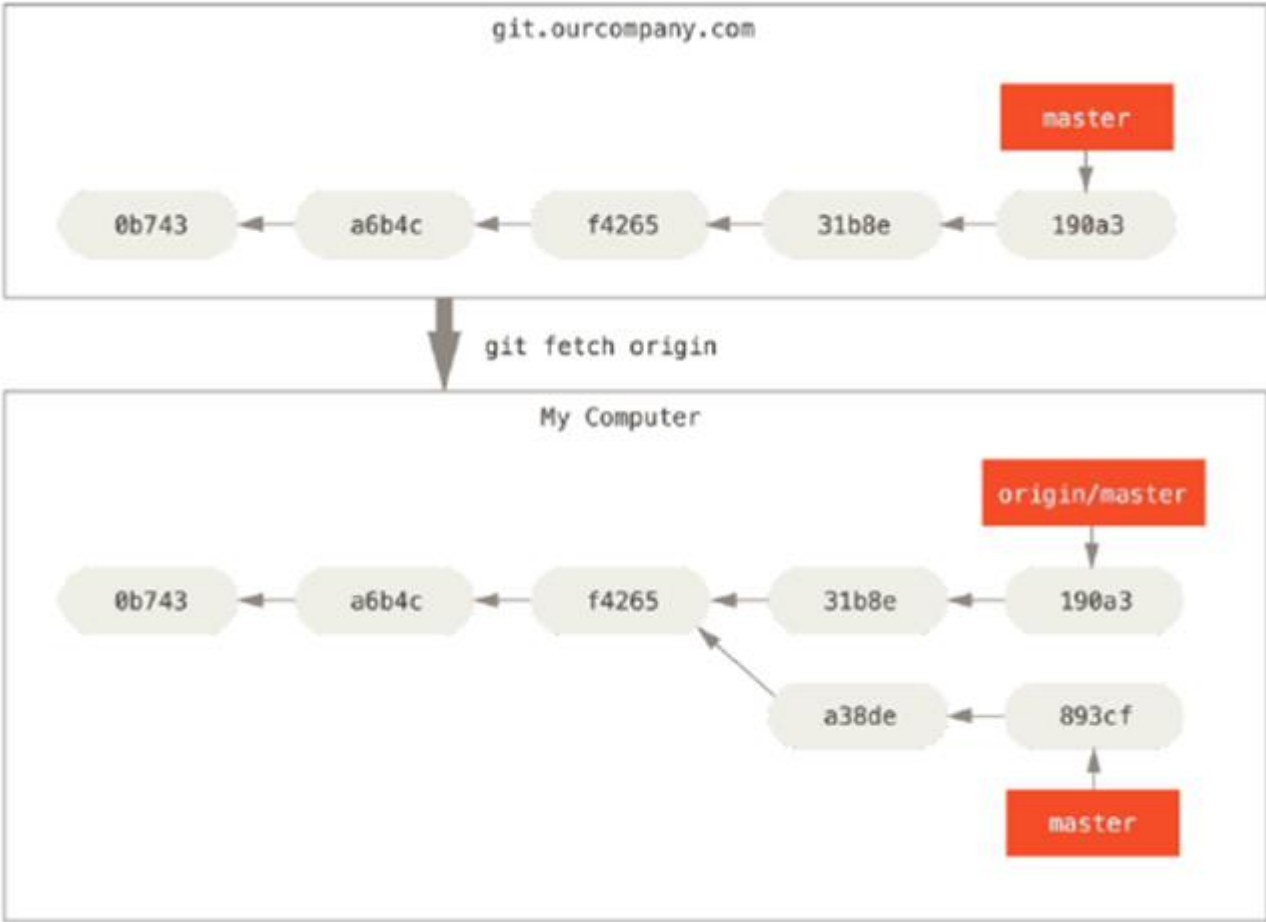


Remote Branches

- Zur Unterscheidung werden Repositories durch einen eigenen Namespace identifiziert
 - Geclonetes Repository: `origin`

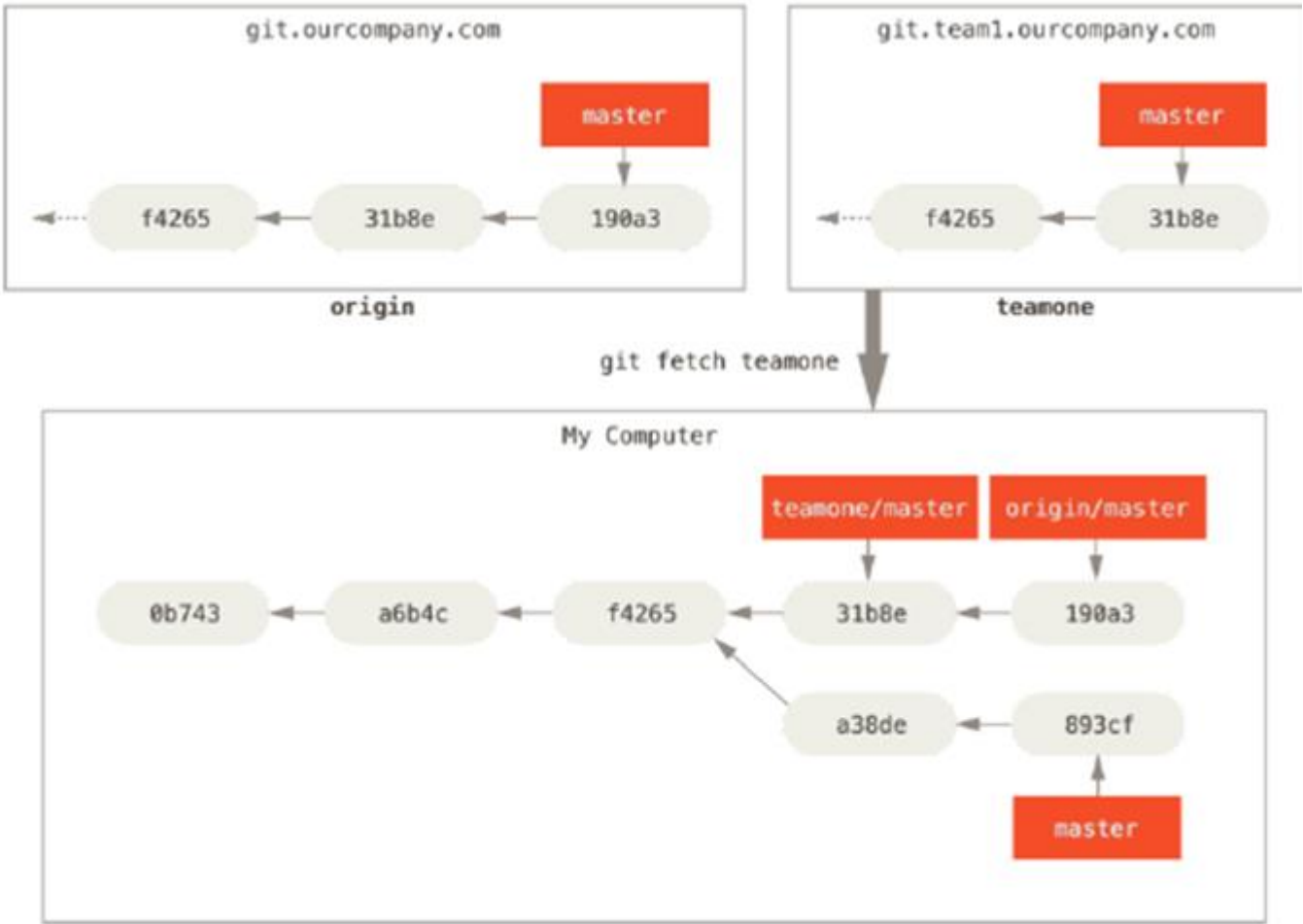


Fetching





Fetching mit mehreren Remote Servern





Pulling

- Im Unterschied zum `fetch` versucht `git pull`, die vom entfernten Repository gezogenen Änderungen direkt in den aktuellen Branch zu mergen
 - Falls ein Rebase notwendig ist
 - `git pull --rebase`



Pushing

- Entfernte Branches kennen die URL des entfernten Repositories
 - Damit können Änderungen in das entfernte Repository gesendet werden
 - `git push local remote`



Remote Konfiguration



Remote Konfiguration

- Ein Repository kann beliebig viele Konfigurationen zu anderen Repositories enthalten
 - Eindeutigkeit der Branch-Namen durch eindeutigen Namespace
- Eine Anbindung muss nicht dauerhaft sein!
 - So kann beispielsweise nach dem Holen eines Branches von einem entfernten Repository ein lokaler Branch erstellt werden und anschließend die Remote-Konfiguration entfernt werden
 - Grundprinzip des "Forkings"






git remote

- Verwalten der Remote-Konfiguration
 - Diese wird in der Git-Konfiguration des Repositories abgespeichert
- Basis-Kommandos
 - `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>`
 - `git remote rename <old> <new>`
 - `git remote remove <name>`



Git auf dem Server



-  Überblick Git Server
-  GitHub
-  GitLab



Überblick Git Server



Server-Repositories

- Besitzen kein Working-Directory
 - Auschecken der Ressource ist nicht notwendig
- "Bare Repositories"
- Das Aufsetzen des Servers ist sehr einfach:
 - Kopieren des Bare Repositories auf die Server-Maschine
 - Definition der Protokolle



Klonen eines Repositories

- `git clone https://github.com/...`
- Unterstützte Protokolle
 - Lokal
 - Lokale oder auch shared Directories
 - http/https
 - smart
 - dumb
 - SSH
 - Git



Git Server: Produkte

- Für die Verwaltung mehrerer Repositories in Software-Projekten sind Server-Produkte praktisch unerlässlich
- Aufgaben:
 - Repository-Verwaltung
 - Benutzer-Verwaltung
 - Lokal
 - Anbindung an vorhandene LDAP-Server
 - ...
 - Einfache Benutzerführung
 - Forking von Repositories
 - Benutzer-Registrierung
 - Administration
- Einbinden in die restliche Infrastruktur der Software-Entwicklung
 - Ticketsystem
 - Build-Server
 - Continuous Integration



GitWeb

- Ein simpler Web Server als Bestandteil der Git-Distribution
- Notwendig ist nur noch ein installierter Web Server



GitHub



Übersicht: GitHub

- Wiki
 - "GitHub ist ein webbasierter Filehosting-Dienst für Software-Entwicklungsprojekte. Namensgebend ist das Versionsverwaltungssystem Git."
- Freier und kommerzieller Repository-Support
- GitHub-Server kann auf eigenen Servern installiert werden

GitHub Web Anwendung

GitHub Bootcamp

1

Set up Git
A quick guide to help you get started with Git.

2

Create repositories
Repositories are where you'll work and collaborate on projects.

3

Fork repositories
Forking creates a new, unique project from an existing one.

4

Work together
Send pull requests, follow friends. Star and watch projects.

rainersawitzki

You've been added to the **Javacream** organization!

Start Learning Git and GitHub Today with Self-Paced Training
Our on-demand training option will have you contributing on GitHub quicker than you can say Pull Request!



GitLab



Übersicht: GitLab

- Produkt mit kommerziellem Support
 - Community Edition frei verfügbar
- Einfache Installation auf dem Ubuntu-Server
 - Web Server
 - Ruby-Interpreter
 - Datenbank
- Steuerung
 - `gitlab-ctl <Options>`
 - Gültige Optionen mit `gitlab-ctl --help`

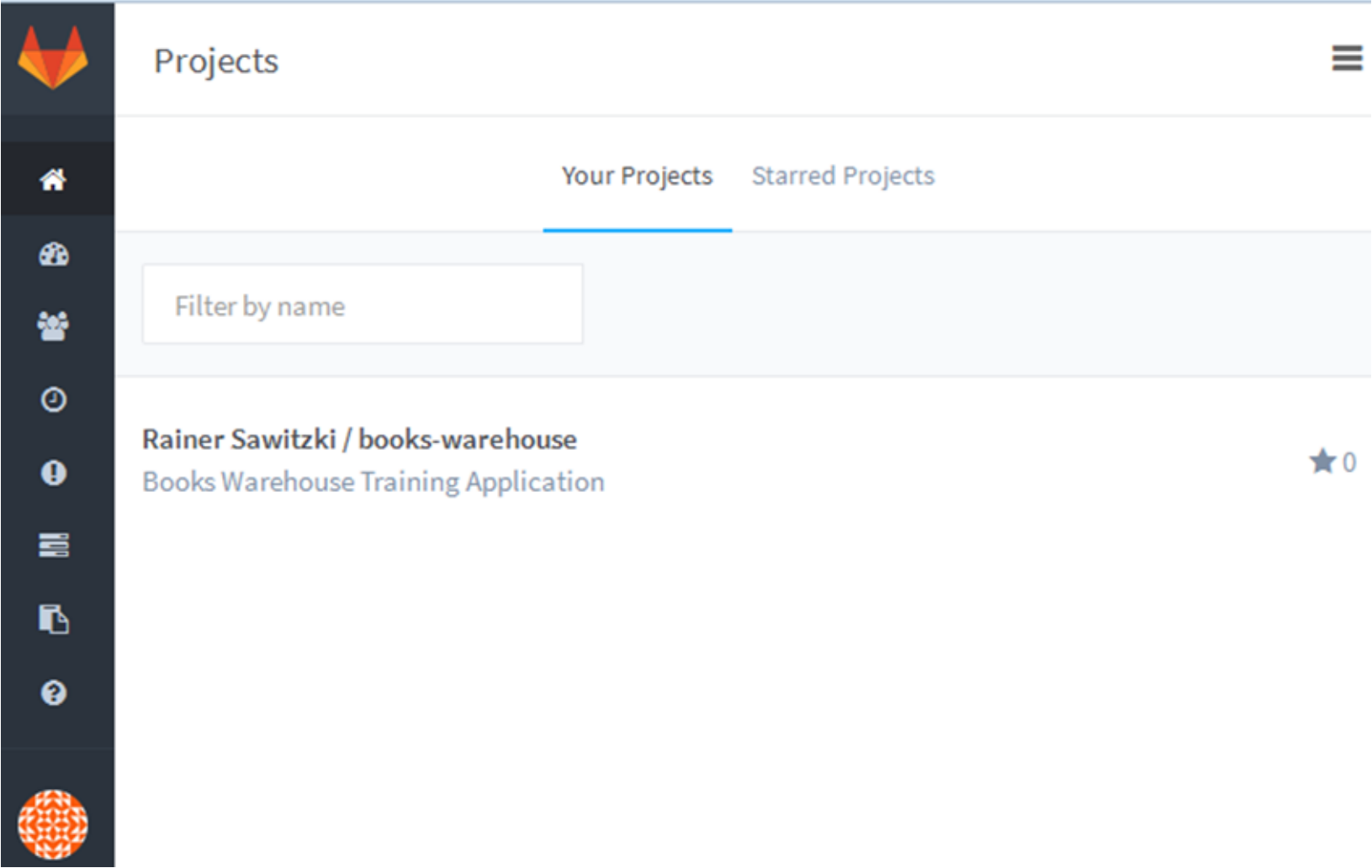


Installation von GitLab

- GitLab wird nur für Linux unterstützt
- Bitnami stellt fertig konfigurierte Images für VMWare Player und Virtual VBox zur Verfügung
 - <https://bitnami.com/stack/gitlab/virtual-machine>
- Auch ein Docker-Image ist mittlerweile auf DockerHub vorhanden
 - <https://hub.docker.com/r/gitlab/gitlab-ce/>

GitLab

Weboberfläche





GitLab Hooks

- Mit GitLab Hooks wird der Repository-Server mit anderen Produkten verbunden
 - Haben nichts mit Git-Hooks zu tun!
- Bei bestimmten Aktionen werden Http-Requests abgesetzt
 - Ziel-URL ist definierbar
 - Die übermittelten Daten werden von GitLab festgelegt und sind nicht veränderbar



Beispiel: Web Hooks

Web hooks

Web hooks can be used for binding events when something is happening within the project.

URL

`http://example.com/trigger-ci.json`

Trigger



Push events

This url will be triggered by a push to the repository



Tag push events

This url will be triggered when a new tag is pushed to the repository



Comments

This url will be triggered when someone adds a comment



Issues events

This url will be triggered when an issue is created



Merge Request events

This url will be triggered when a merge request is created



Workflows



Übersicht



Atlassian Flows



GitLab Flow



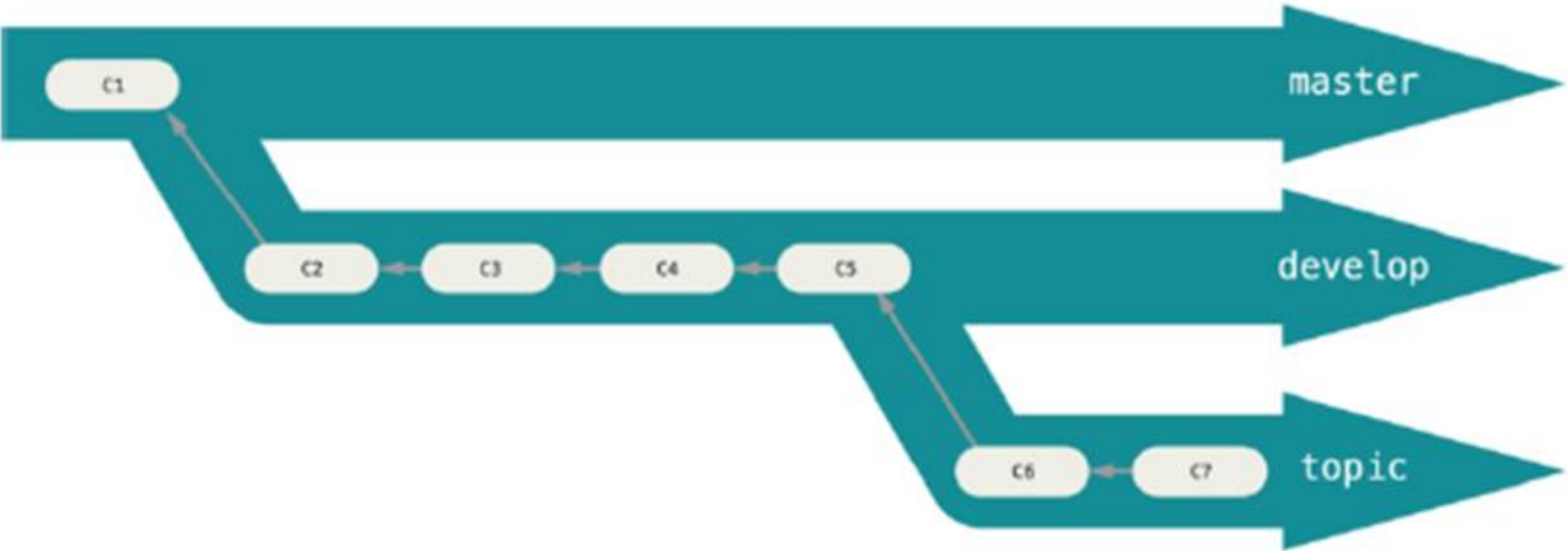
Übersicht



Was ist ein Flow?

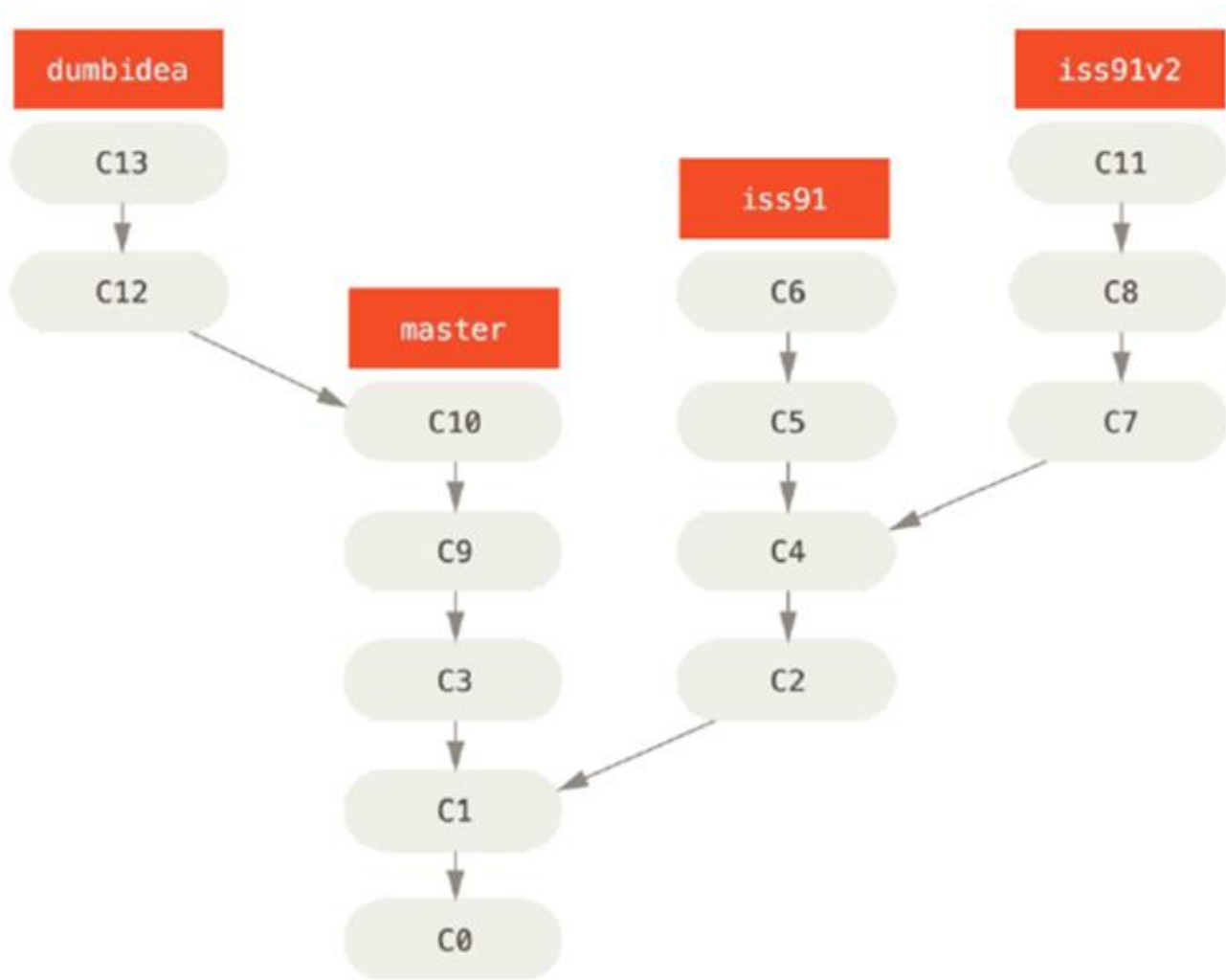
- Git selber ist nur ein Revisionsverwaltungssystem
 - und stellt damit Basis-Befehle zur Verfügung
- Wie genau Git am Besten benutzt werden kann wird durch einen (Work) Flow beschrieben
 - Im Wesentlichen eine Vorgehensweise, die aus den Erfahrungen vieler Projekte gewonnen wurden
 - Damit eine "Best Practice"
- Beispiele
 - Git Flow
 - Vorgestellt und dokumentiert von Atlassian
 - GitHub Flow
 - GitLab Flow

Langlebige Branches





Kurzlebige Topic-Branches





Atlassian Flows



Atlassian.com

- Die folgenden Workflow-Patterns und Bilder entstammen der Atlassian-Community
- Details unter <https://www.atlassian.com/pt/git/workflows>



Commit Guidelines

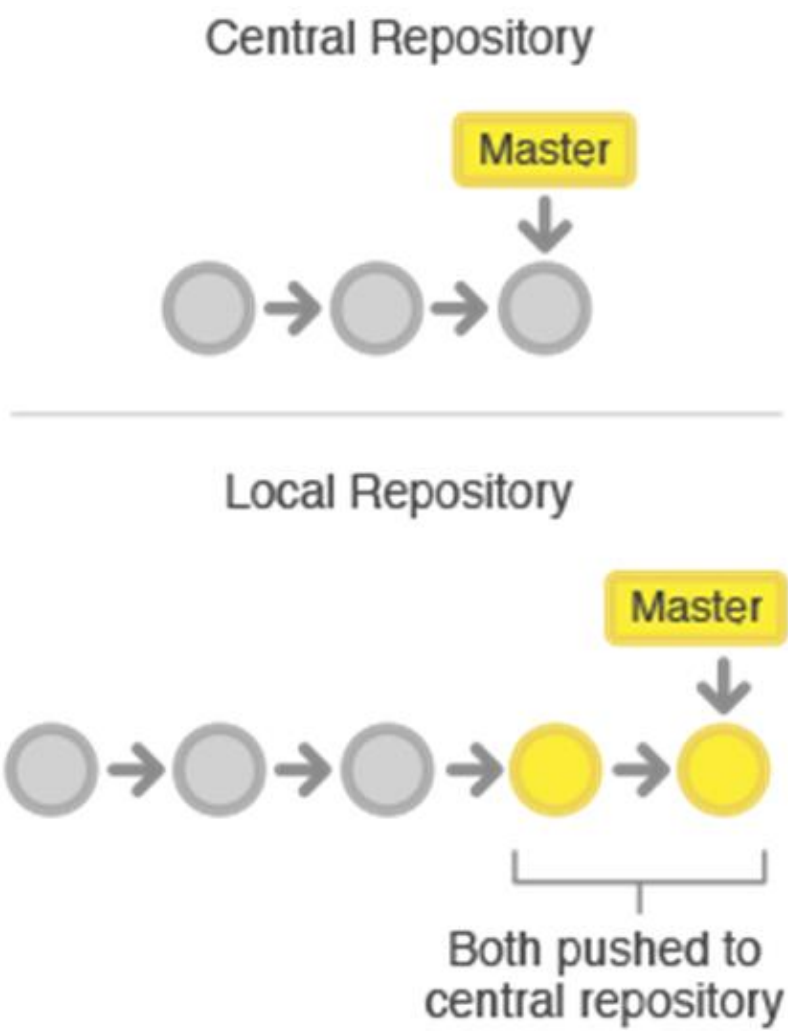
- Whitespace Prüfungen zur Vermeidung unnötiger Diffs
 - `git diff -check`
- One commit per Issue
 - Insbesondere bei Anbindung an ein Ticket-System wie Jira
- Sprechende Commit Messages
 - Maximal 50 Zeichen für beschreibendes Kommando
 - Detailbeschreibung mit Motivation (Issue) und Abgrenzung zur bestehenden Version



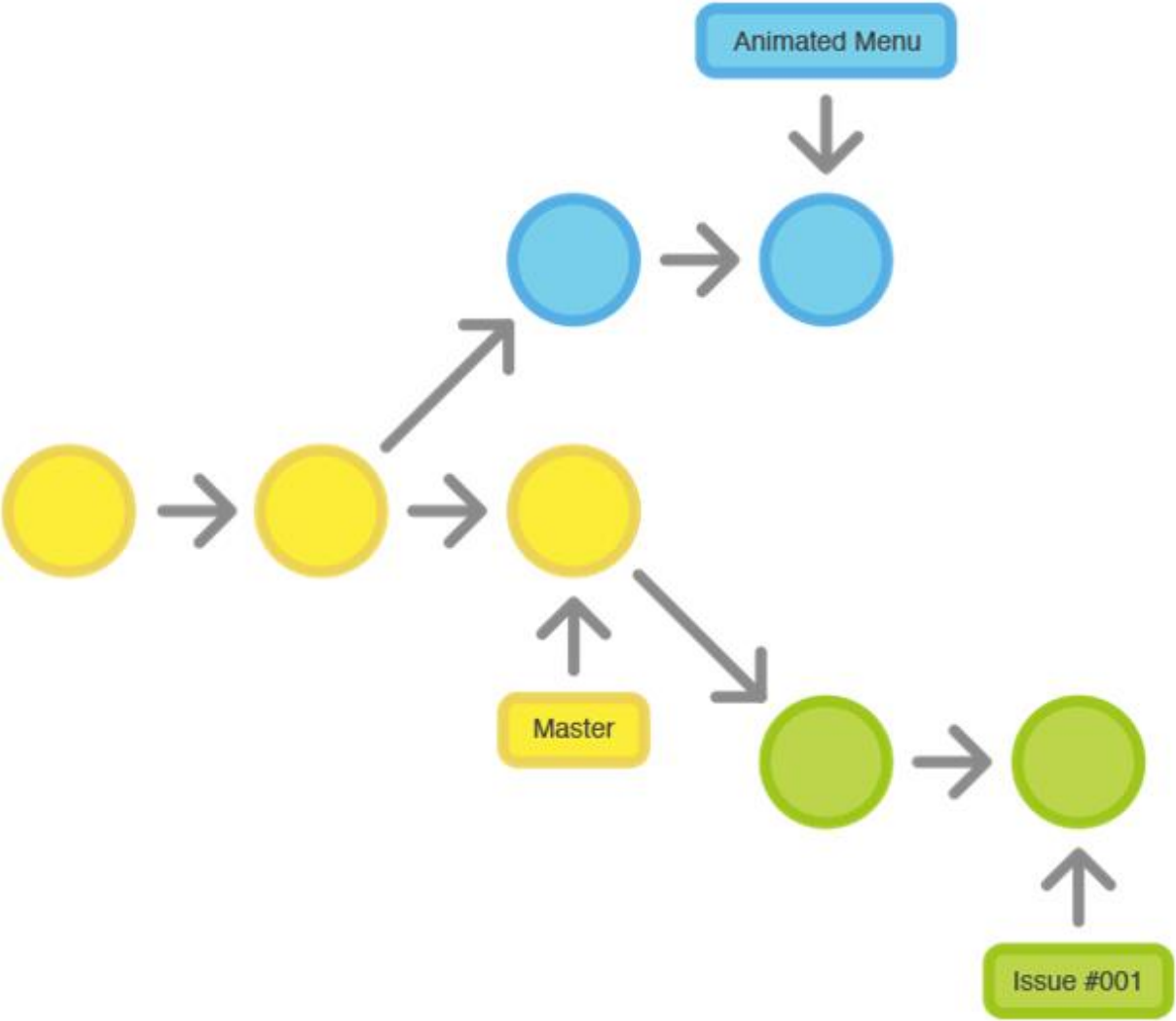
Centralized Workflow: Benutzer

- Mehrere Benutzer teilen ein gemeinsames Repository

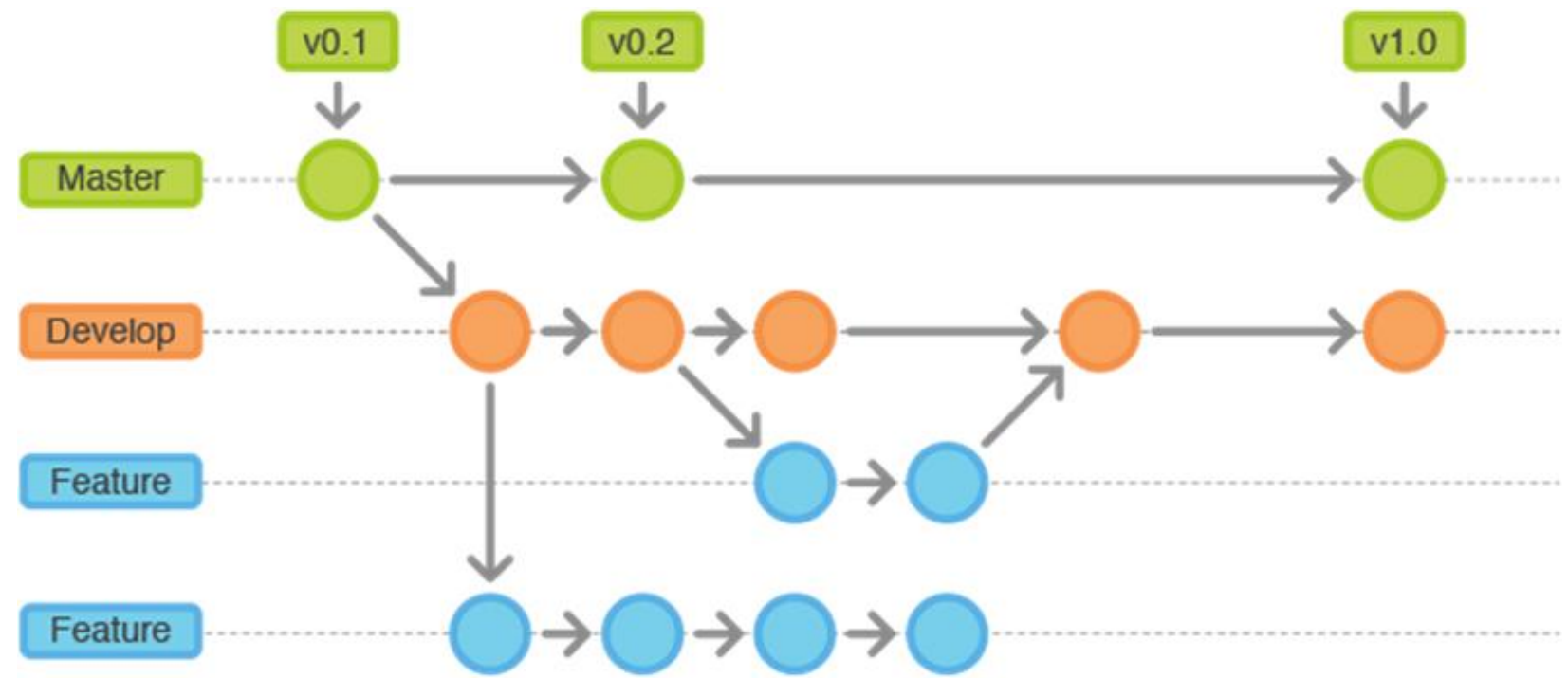
Centralized Workflow: Commit



Feature Branch Workflow



Gitflow





Forking Workflow

- Jeder Developer besitzt zwei Repositories
 - Ein lokales
 - Ein Remote



Pull Requests

- Ein Developer forked ein Projekt
- Änderungen werden durch einen Pull Request vom Projektverantwortlichen gemerged



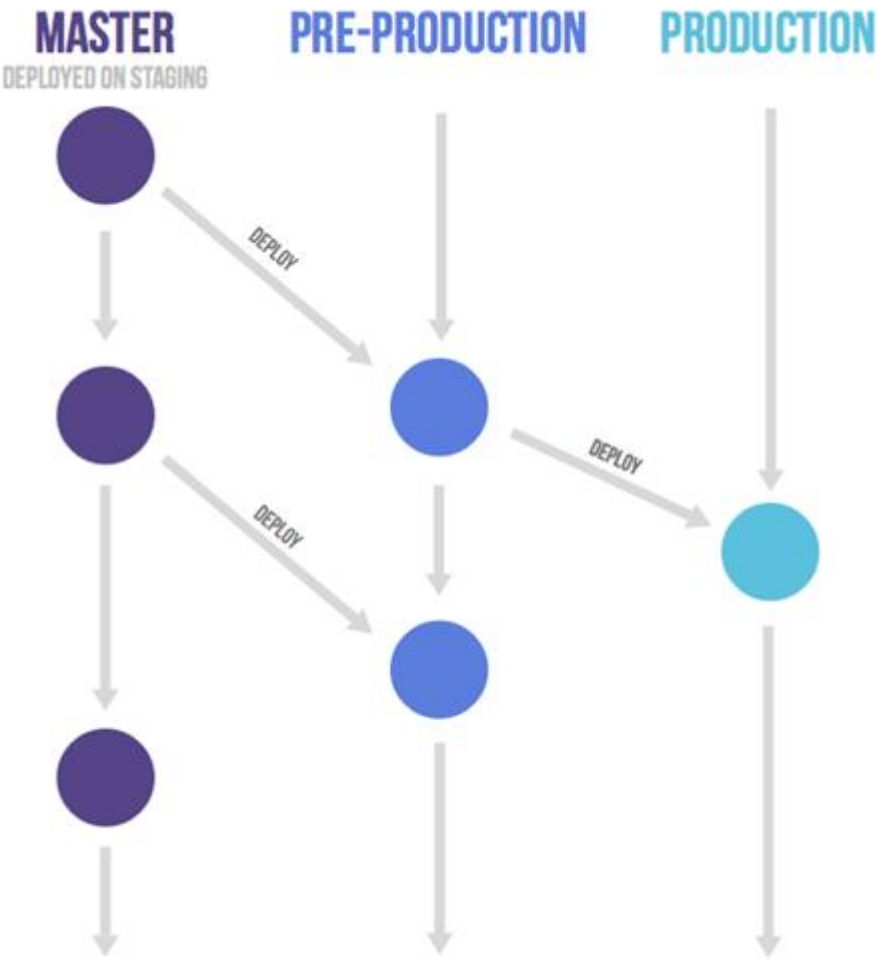
GitLab Flow



Grundprinzip des GitLab Flows

- Basiert auf Merge Requests
 - Es existieren somit geschützte Branches, die nur von speziellen Rollen benutzt werden dürfen
- Die geschützten Branches definieren ein Environment bestehend aus verschiedenen Stages
 - Test und QS
 - Preproduction
 - Production
 - ...
- Auch Releases können damit verwaltet werden

Beispiel für den GitLab Flow





Git-Clients



Aufgaben



Die Git GUI als einfacher Client



Eclipse als Beispiel für eine
Entwicklungsumgebung



Aufgaben



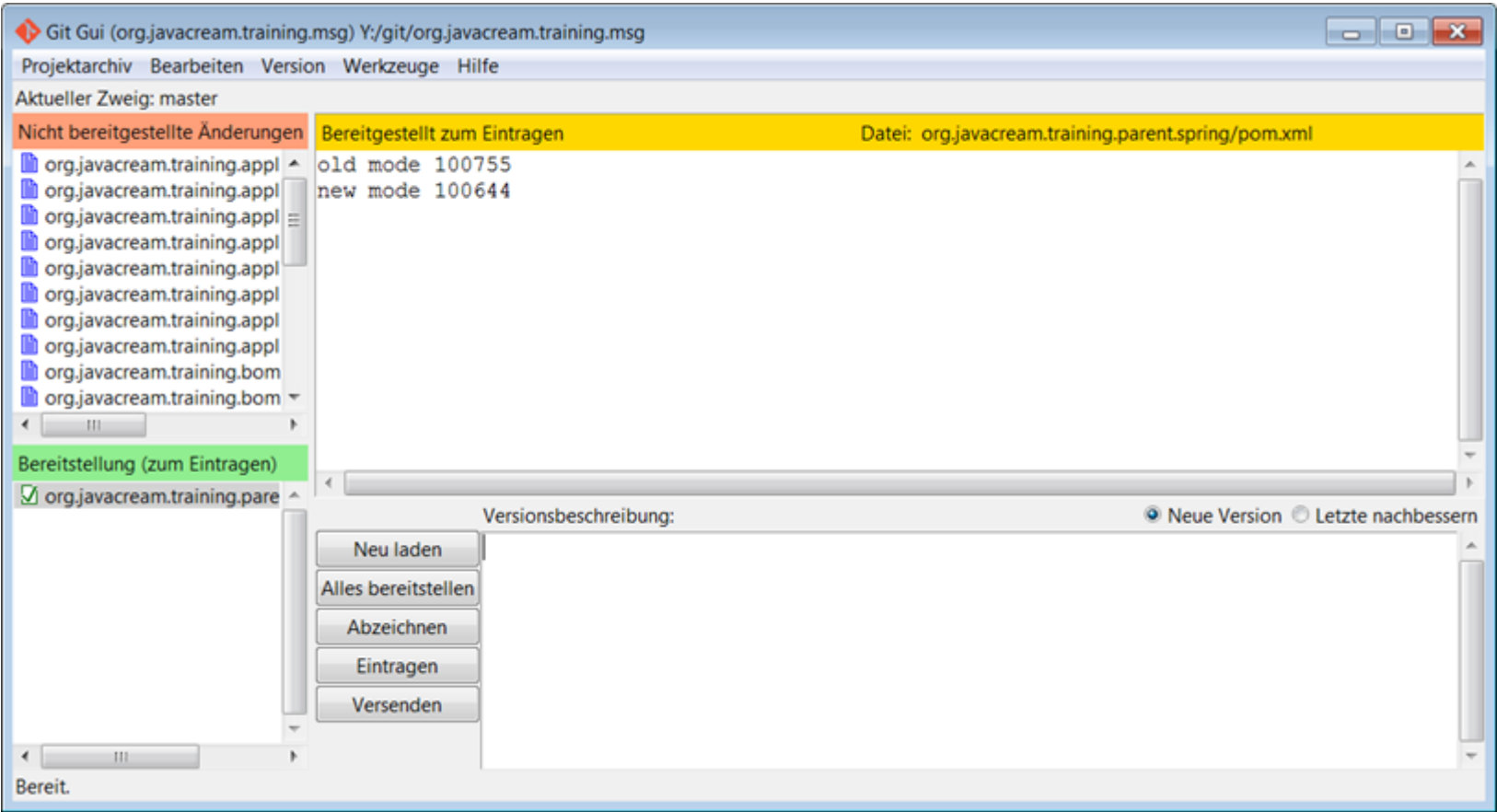
Übersicht: Git-Clients

- Die Kommandozeilen-Befehle sind für ein technisches Verständnis der Abläufe sehr interessant
- In der Praxis werden jedoch häufig Git-Clients mit grafischer Unterstützung verwendet
- Standalone-Programme
 - Tortoise
 - SourceTree
- Integration in Entwickler-Werkzeuge
 - Eclipse
 - XCode
 - Visual Studio
 - Atom
 - ...



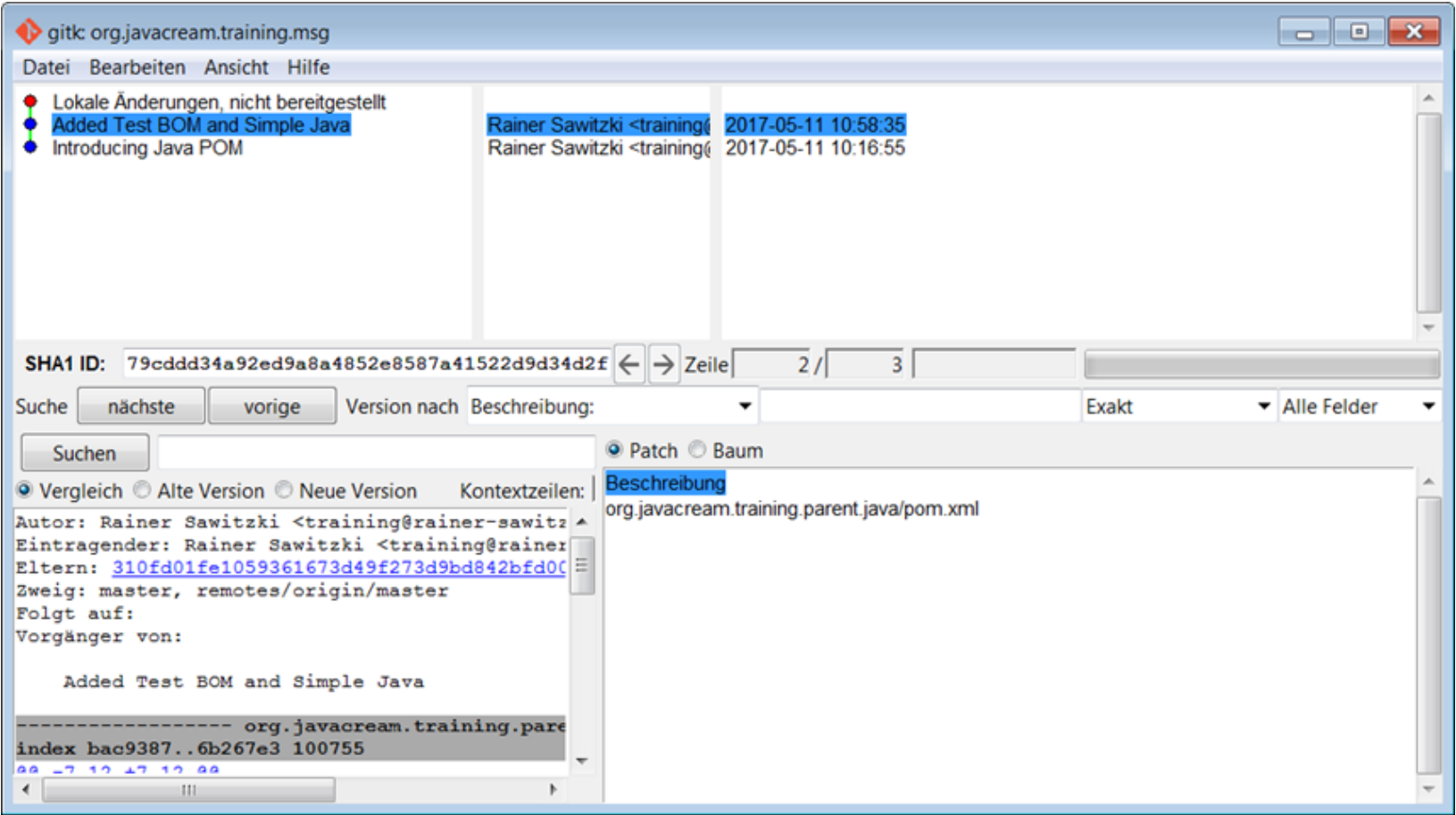
Die Git GUI als einfacher Client

Git Client Windows: Commit

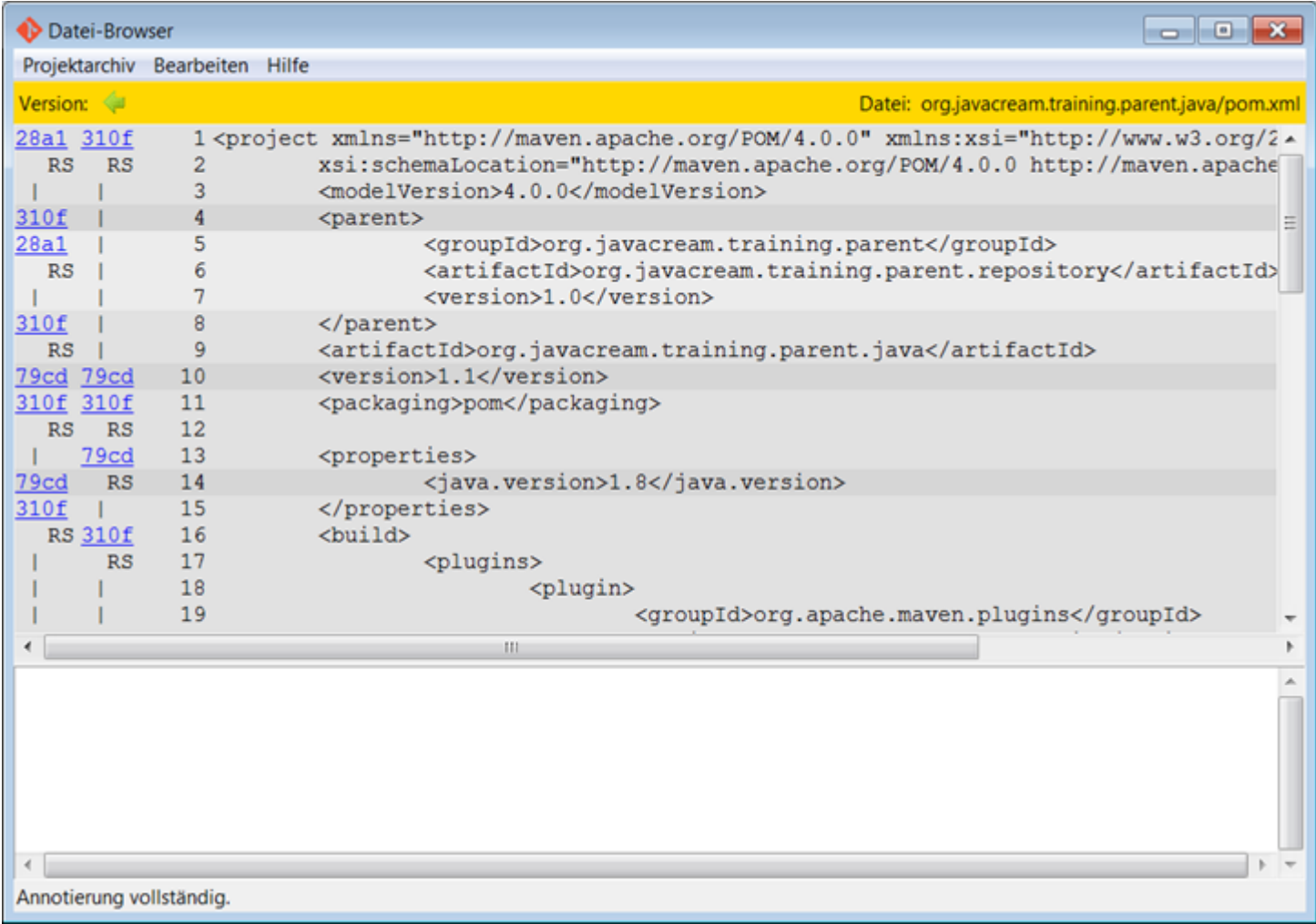




Git Client Windows: History



Git Client Windows: Blame





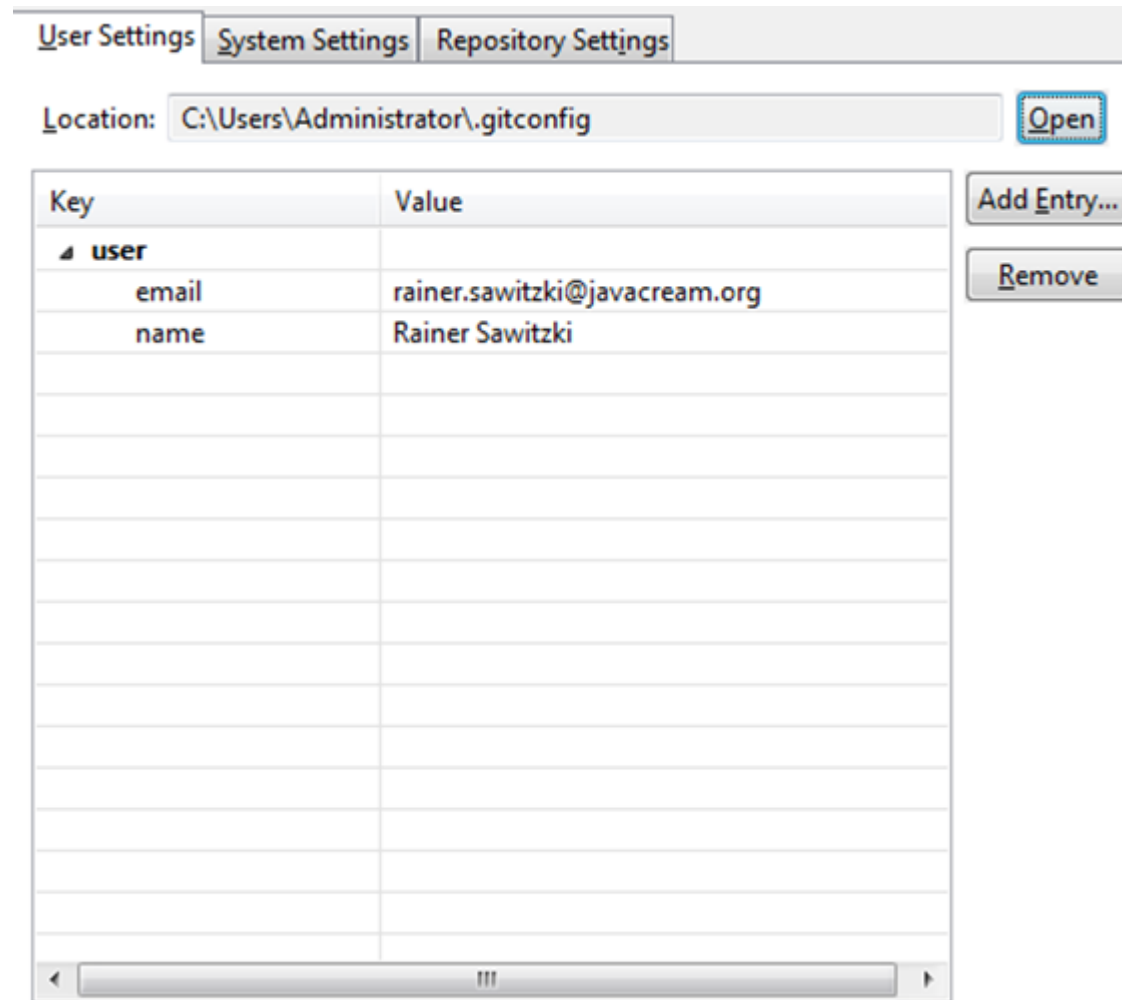
Eclipse als Beispiel für eine Entwicklungsumgebun g



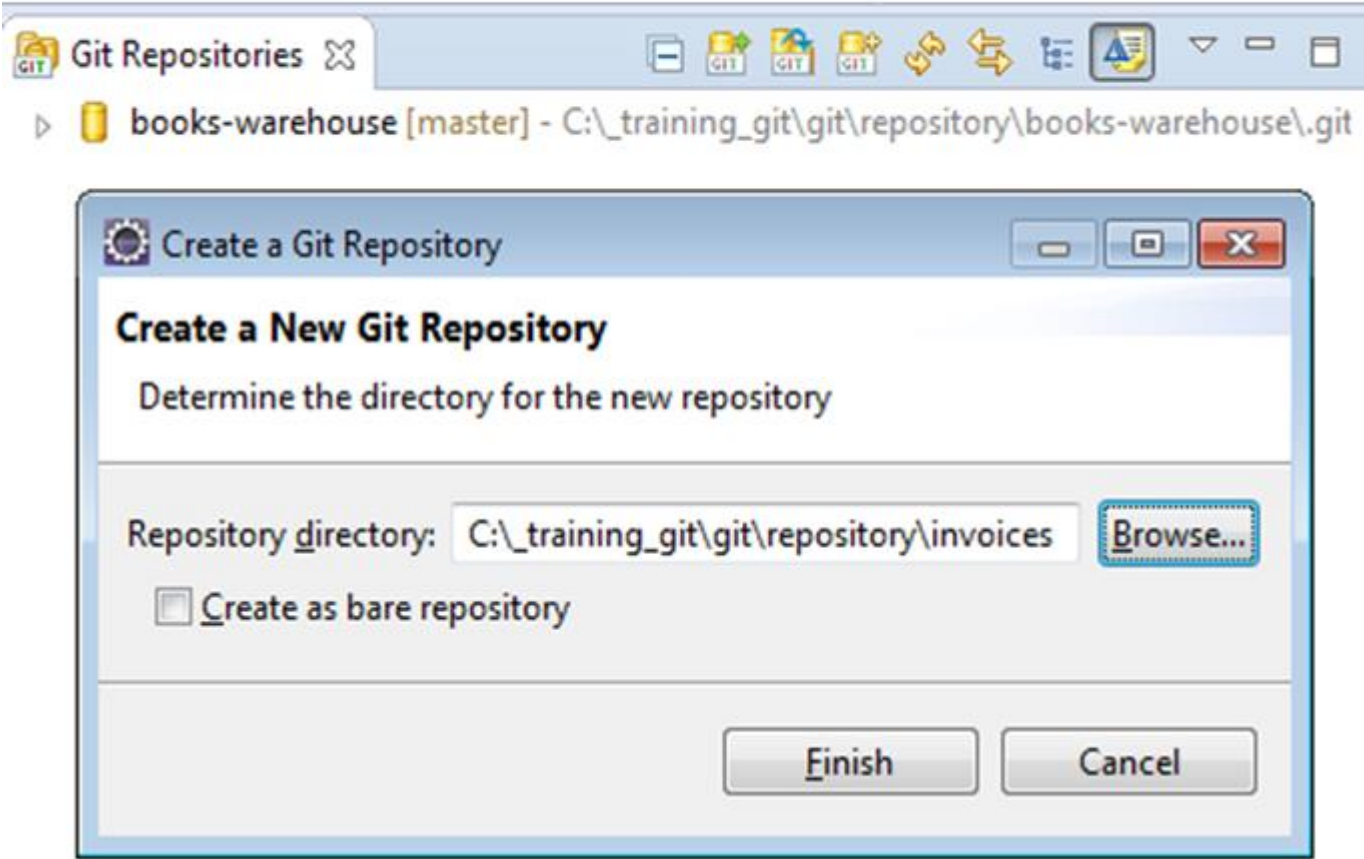
Beispiel: Eclipse

- Weit verbreitete Entwicklungsumgebung für Java, C, ...
- Eclipse bringt Git in der Standard-Installation bereits mit
- Alternativ können natürlich auch andere Clients oder Entwickler-Werkzeuge benutzt werden
 - Übersicht unter <https://git-scm.com/downloads/guis/>

Eclipse: Git-Konfiguration



Eclipse: Initialisierung eines Repositories

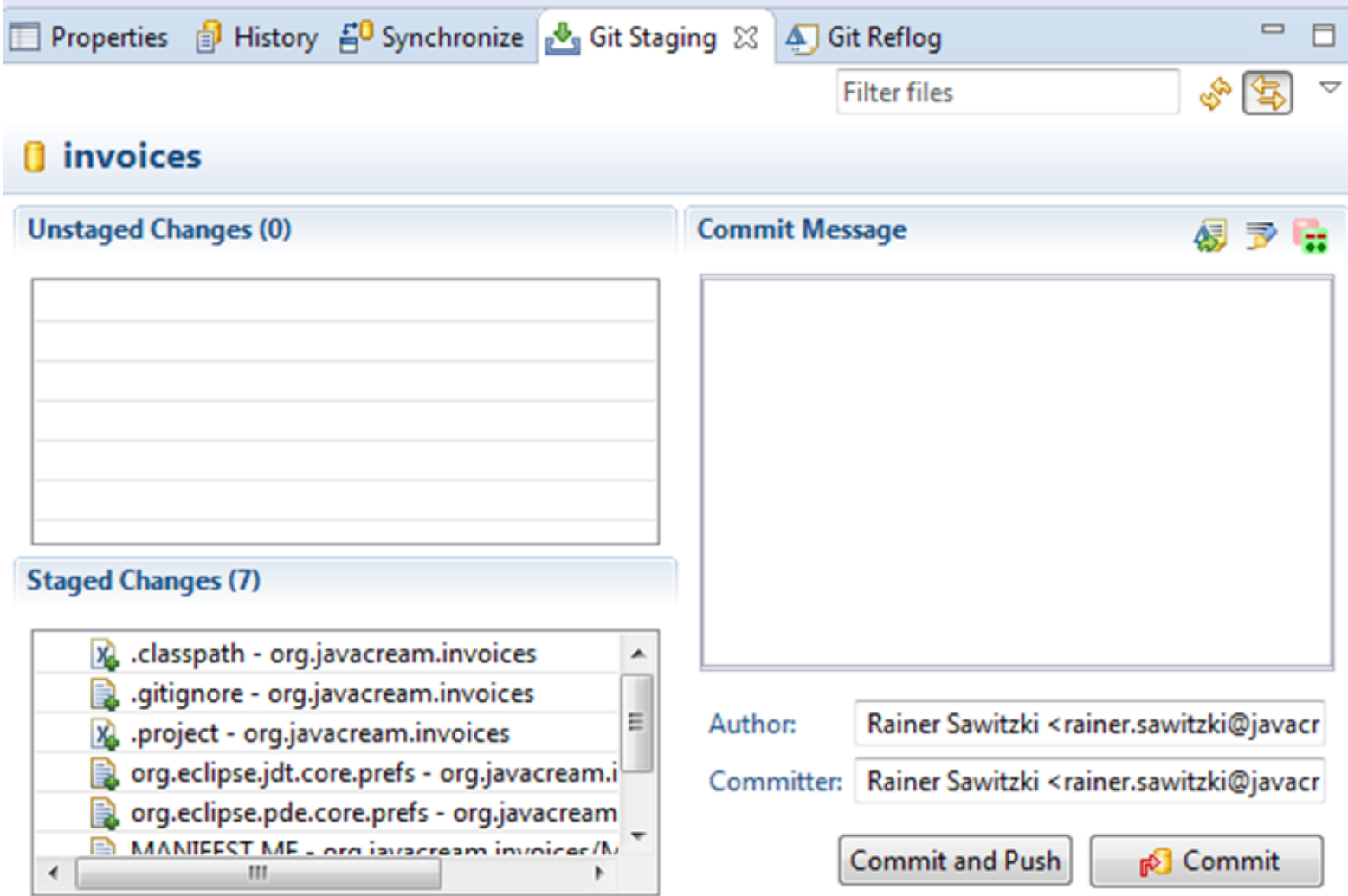




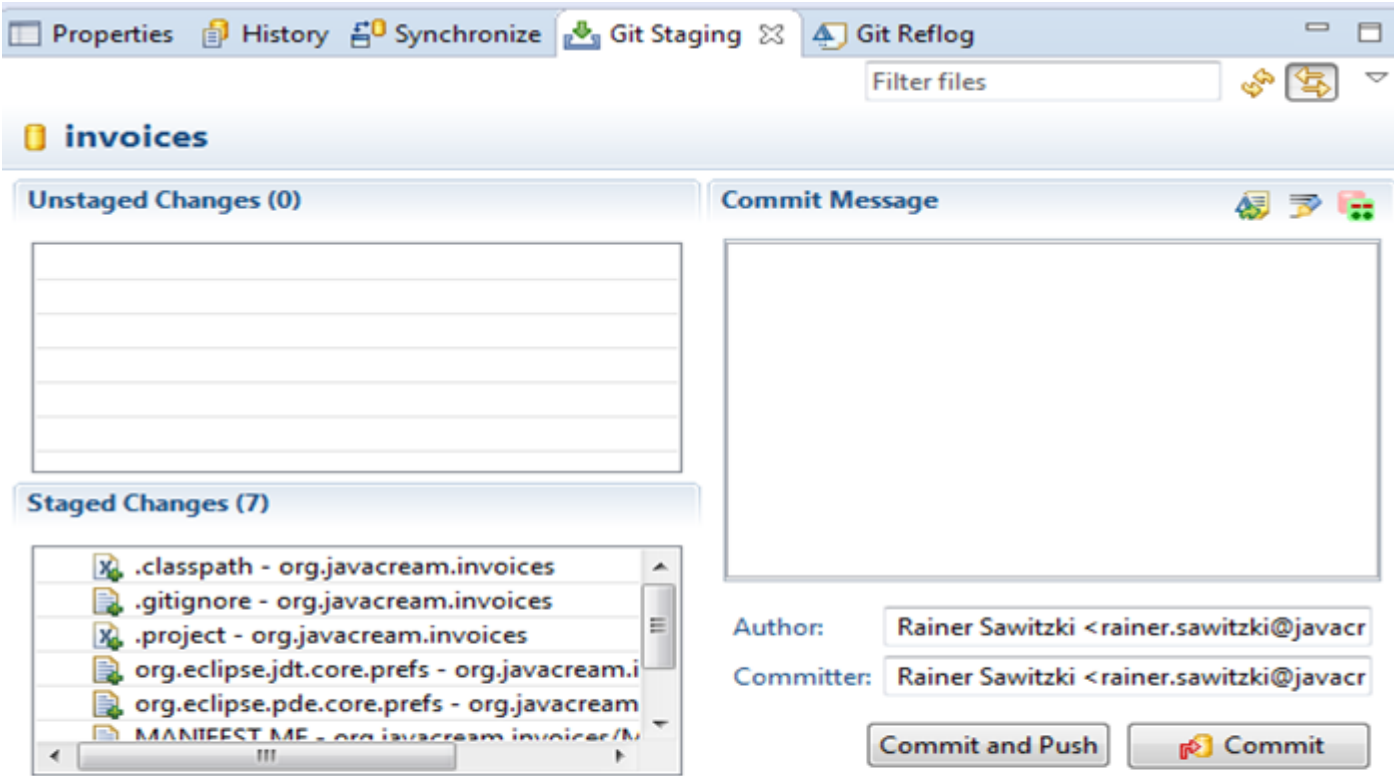
Eclipse : Hinzufügen von Inhalt

- Team – Share Project... Git
- Anschließend das Projekt dem Repository hinzufügen
 - Team – Add to Index












Eclipse: Staging



Eclipse: add und commit



Eclipse: History

Id	Message
d07df54	 master HEAD ok
dc966e1	 ok
dd8970e	 ok
9ffd6dd	 OK
24b0edc	 Changed gitignore
0ed0a89	 changed gitignore
61e33fc	 Added Manifest
7f0aa98	 origin/master Version 2
7dae1bc	 1.1 Changed IsbnGeneratorImpl to use KeyGeneratorStrategy
9a31544	 1.0 Add gitignore
e1bfc7f	 Initial Project



Eclipse: Diffs

Java Structure Compare

- Compilation Unit
 - Import Declarations
 - org.javacream.util.KeyGeneratorStrategy
 - IsbnGeneratorImpl
 - prefix : String
 - strategy : KeyGeneratorStrategy
 - suffix : String
 - nextIsbn()

Java Source Compare

Local: IsbnGeneratorImpl.java

```

4 import org.javacream.util.KeyGeneratorStrategy;
5
6 public class IsbnGeneratorImpl implements IsbnGenerator{
7
8     private KeyGeneratorStrategy strategy;
9     private String suffix;
10    private String prefix;
11
12    public void setStrategy(KeyGeneratorStrategy strategy) {
13        this.strategy = strategy;
14    }
15
16    public void setSuffix(String suffix) {
17        this.suffix = suffix;
18    }
19

```

IsbnGeneratorImpl.java 9166b55... (Rainer Sawitzki)

```

1 package org.javacream.isbngenerator.impl;
2
3 import org.javacream.isbngenerator.IsbnGenerator;
4
5 public class IsbnGeneratorImpl implements IsbnGenerator{
6
7     /* (non-Javadoc)
8      * @see org.javacream.isbngenerator.impl.IsbnGenerator
9      */
10    @Override
11    public String nextIsbn(){
12        return "ISBN:" + System.currentTimeMillis();
13    }
14 }
15

```