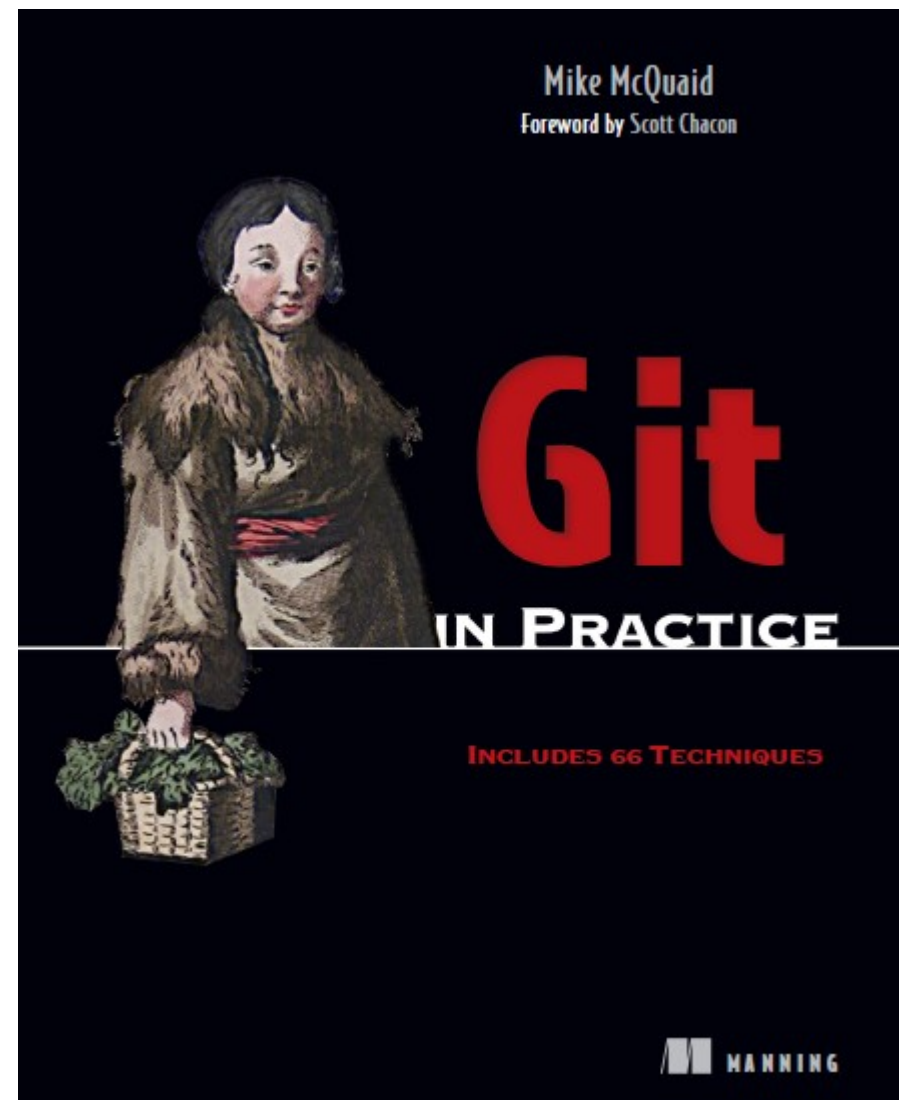
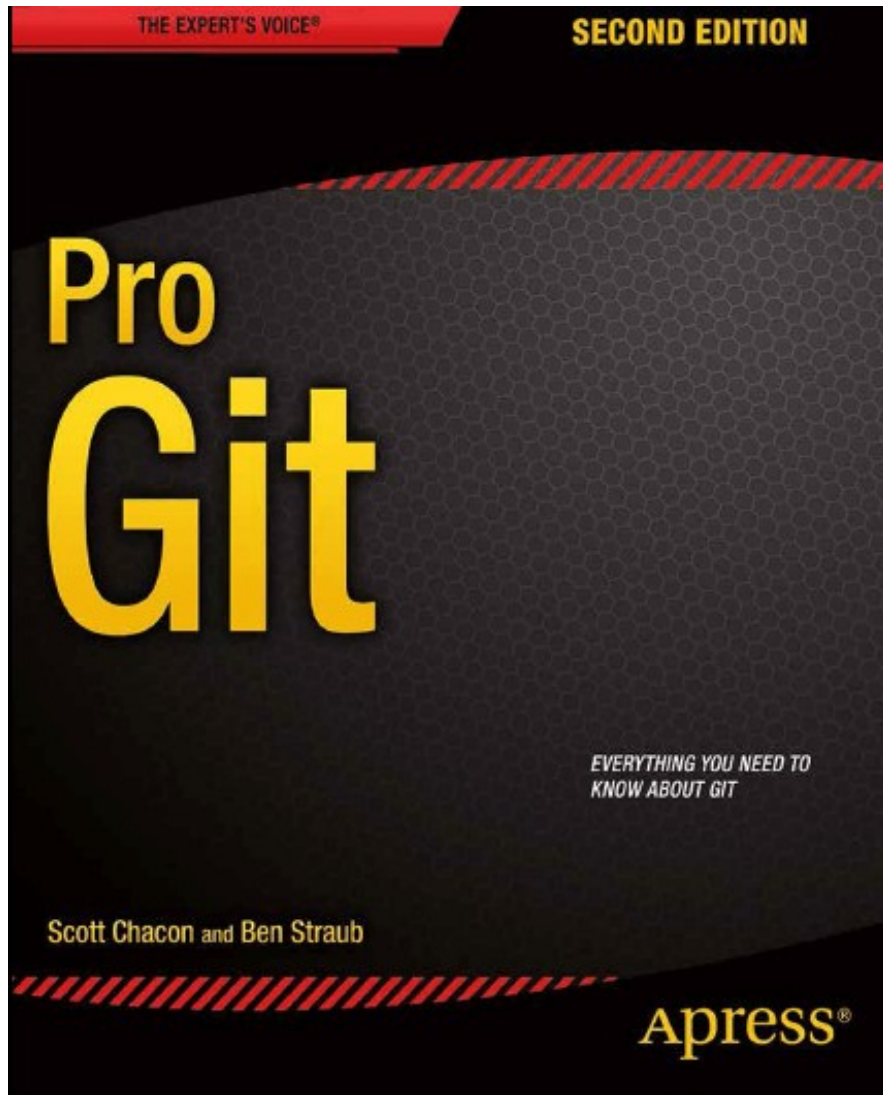




integrata  
cegos

# Git

Ein dezentrales Versionsverwaltungssystem



© Integrata Cegos GmbH

Integrata Cegos GmbH  
Zettachring 4  
70567 Stuttgart

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.**

- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
  - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
  - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
  - Musterbeispiele werden zur Verfügung gestellt
  - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
    - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
- Konventionen
  - Befehle werden in `Courier-Schriftart` dargestellt
  - Dateinamen werden in *`Courier-Schriftart`* dargestellt
  - Links werden in `unterstrichener Courier-Schriftart` dargestellt

Einführung	6
Erste Schritte	27
Arbeiten mit Git	48
Distributed Repositories	80
Git auf dem Server	90
Workflows	105
Git-Clients	122

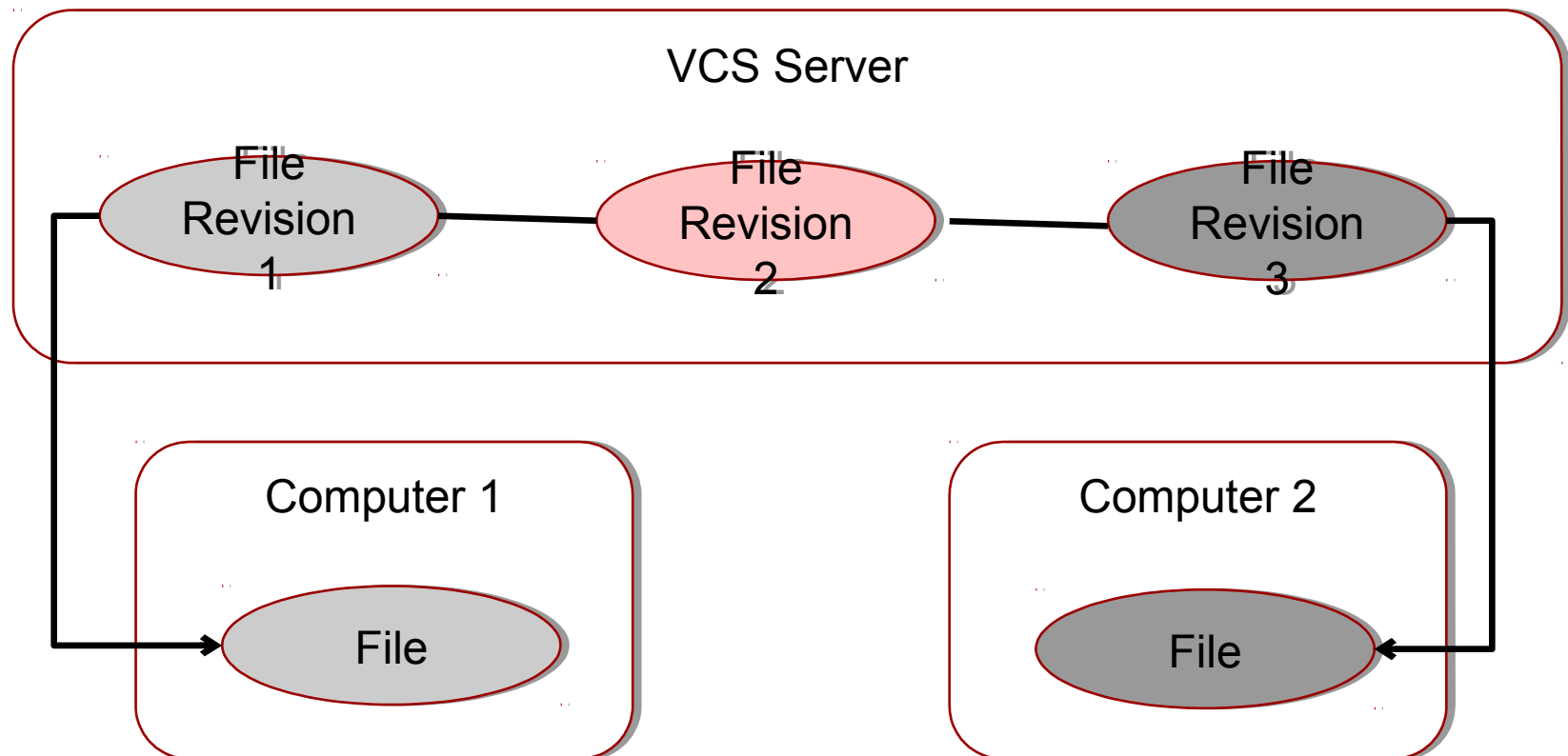
1

# EINFÜHRUNG

1.1

# ÜBERSICHT VERSIONSVERWALTUNG

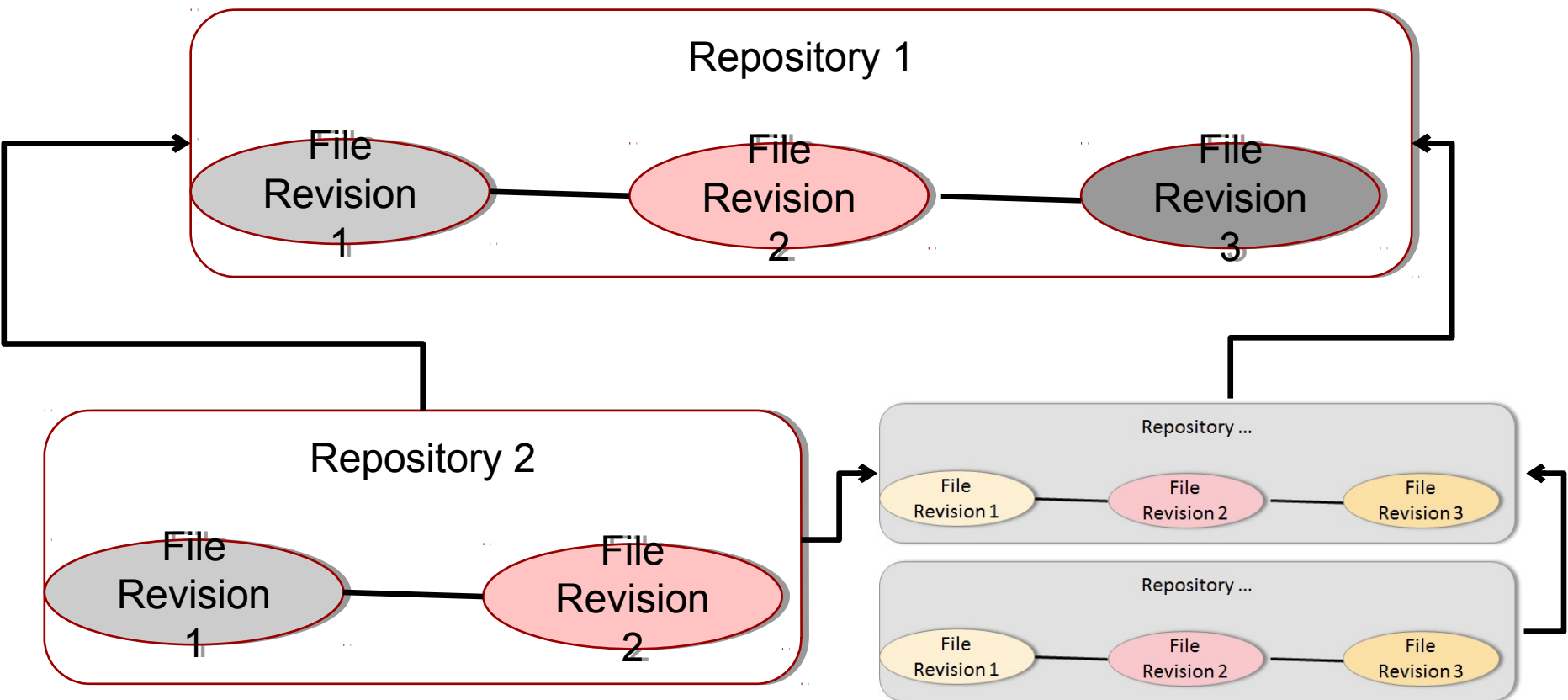
- Zentrale Ablage auf dem Server des Version Control Systems (VCS)
- Beispiel
  - Subversion, CVS, Clearcase





- 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

- Dateiablage in gleichberechtigten Repositories
- Beispiel
  - Git, Mercurial

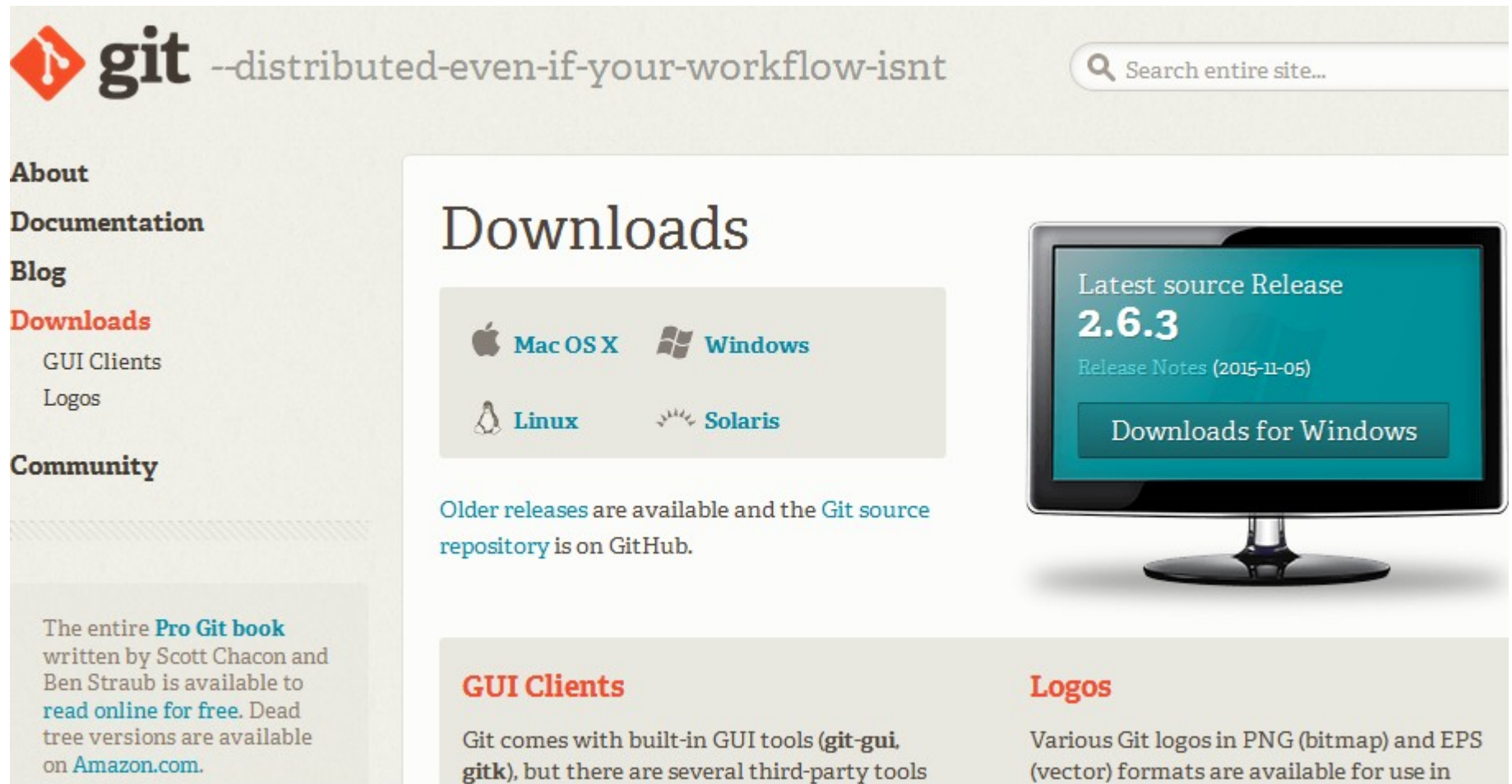


- Jedes Repository ist prinzipiell gleichberechtigt
- Alle Funktionen können lokal ausgeführt werden
- Synchronisation mit anderen Repositories nur bei Bedarf
- Keine Sperrmechanismen
- Authentifizierung und Autorisierung nur bei Kommunikation mit anderen Repositories notwendig
- Zentrale Server-Lösungen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

1.2

## **INSTALLATION VON GIT**

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



The screenshot shows the Git website's Downloads page. At the top, the Git logo is followed by the tagline "--distributed-even-if-your-workflow-isnt". A search bar on the right says "Search entire site...". The left sidebar contains links for "About", "Documentation", "Blog", "Downloads" (highlighted in red), "GUI Clients", "Logos", and "Community". The main content area has a large "Downloads" heading. Below it, a box displays icons for Mac OS X, Windows, Linux, and Solaris. A text block states that older releases are available and the source repository is on GitHub. To the right, a monitor graphic displays the "Latest source Release 2.6.3" with a "Downloads for Windows" button. At the bottom, two sections titled "GUI Clients" and "Logos" provide additional information.

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

Search entire site...

**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.

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

### GUI Clients

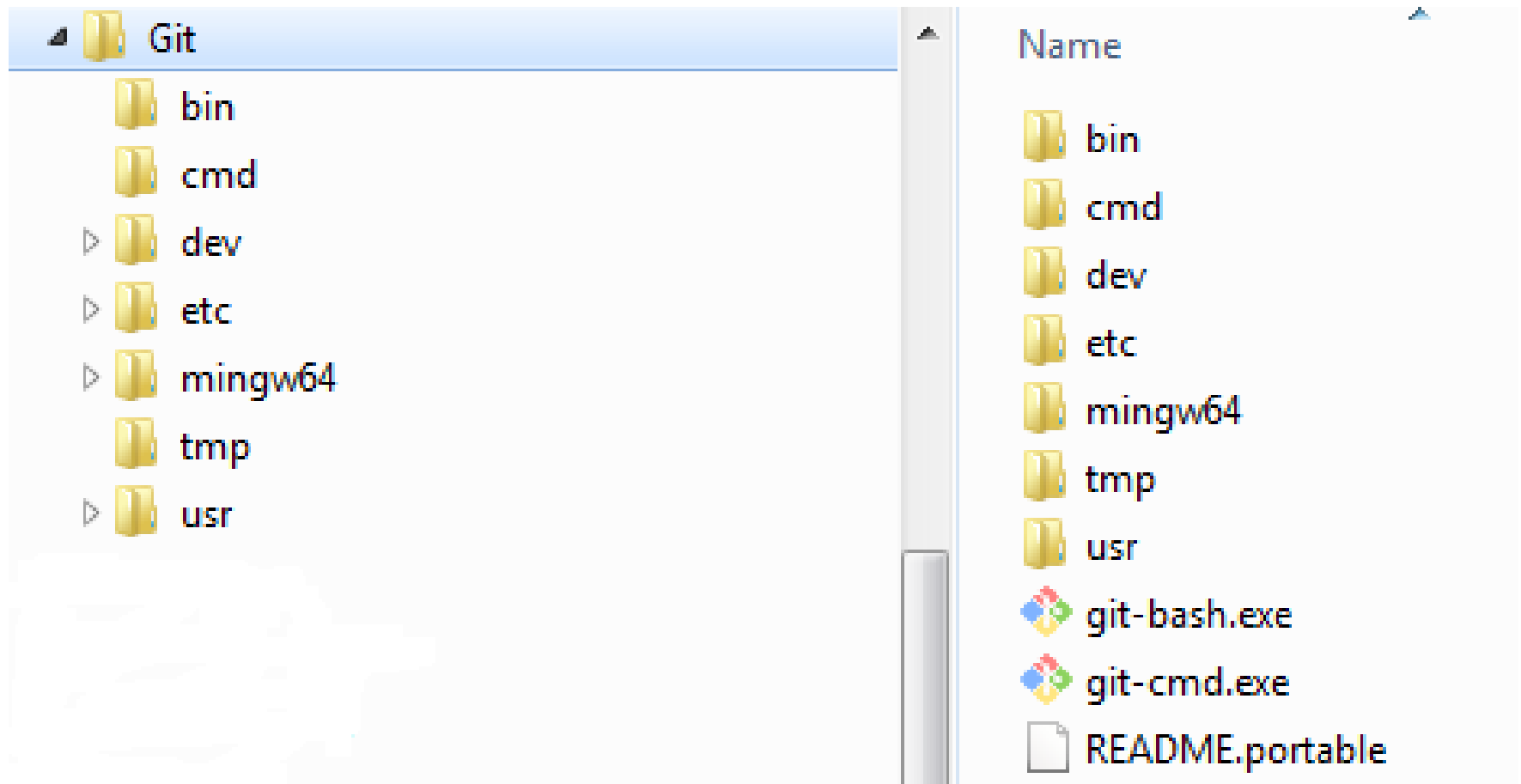
Git comes with built-in GUI tools ([git-gui](#), [gitk](#)), but there are several third-party tools

### Logos

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

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

- 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



1.3

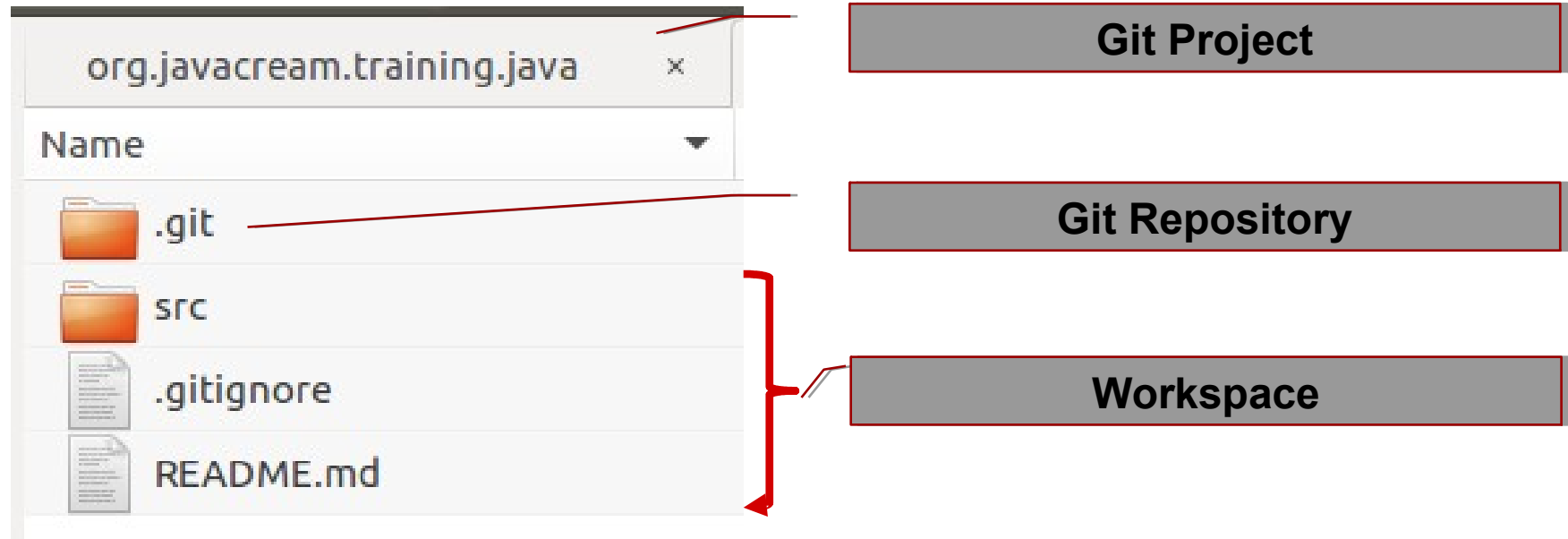
## **EINRICHTUNG**



- 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`

- 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



- 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

1.4

## **DOKUMENTATION UND REFERENZ**

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

## 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*.

## 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

```
$ git config --global user.name "[name]"
```

Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```

Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```

Enables helpful colorization of command line output

## CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```

Creates a new local repository with the specified name

```
$ git clone [url]
```

## MAKE CHANGES

Review edits and craft a commit transaction

```
$ git status
```

Lists all new or modified files to be committed

```
$ git diff
```

Shows file differences not yet staged

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git diff --staged
```

Shows file differences between staging and the last file version

```
$ git reset [file]
```

Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history

## GROUP CHANGES

Name a series of commits and combine completed efforts

```
$ git branch
```

Lists all local branches in the current repository

```
$ git branch [branch-name]
```

Creates a new branch

```
$ git checkout [branch-name]
```

Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```

Combines the specified branch's history into the current branch

```
$ git branch -d [branch-name]
```



## 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

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

2

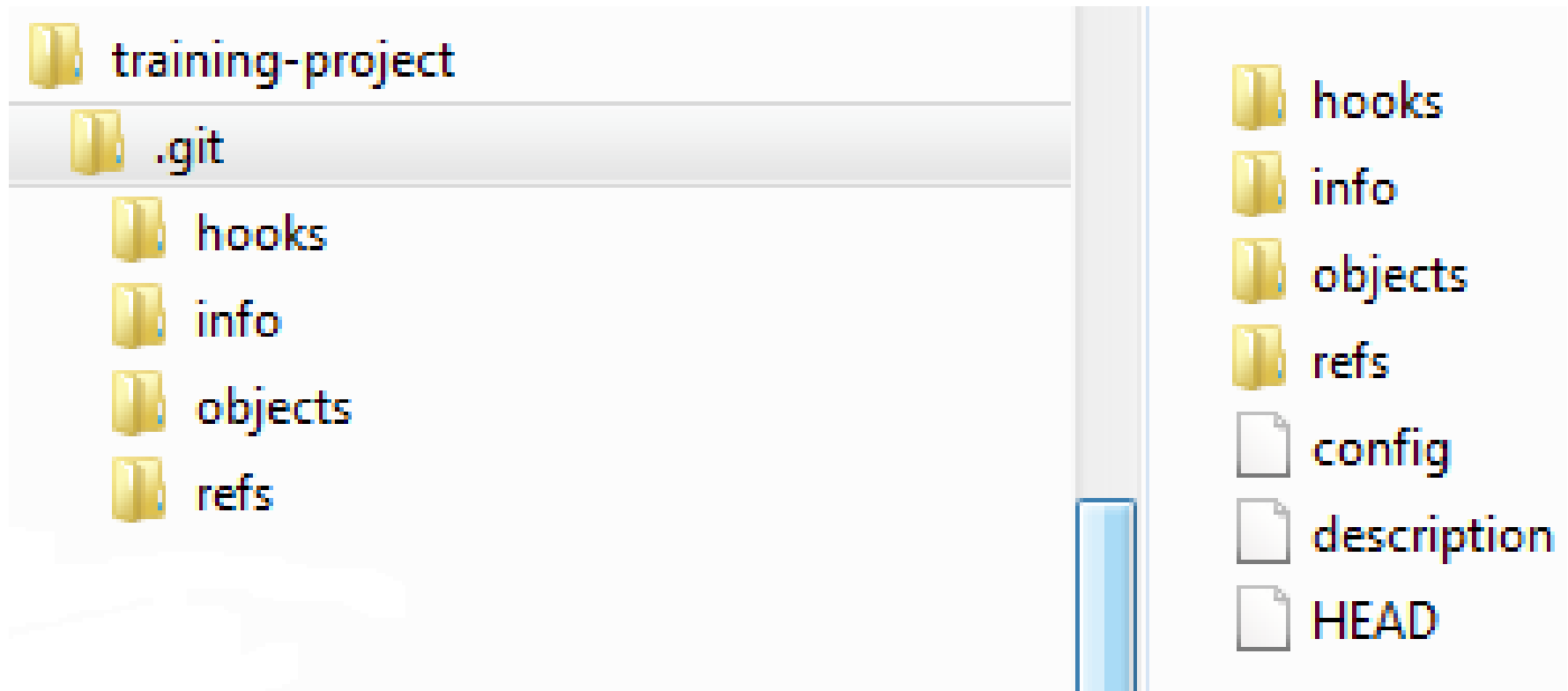
## ERSTE SCHRITTE

2.1

## **DAS REPOSITORY**

- 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



- 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

- **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



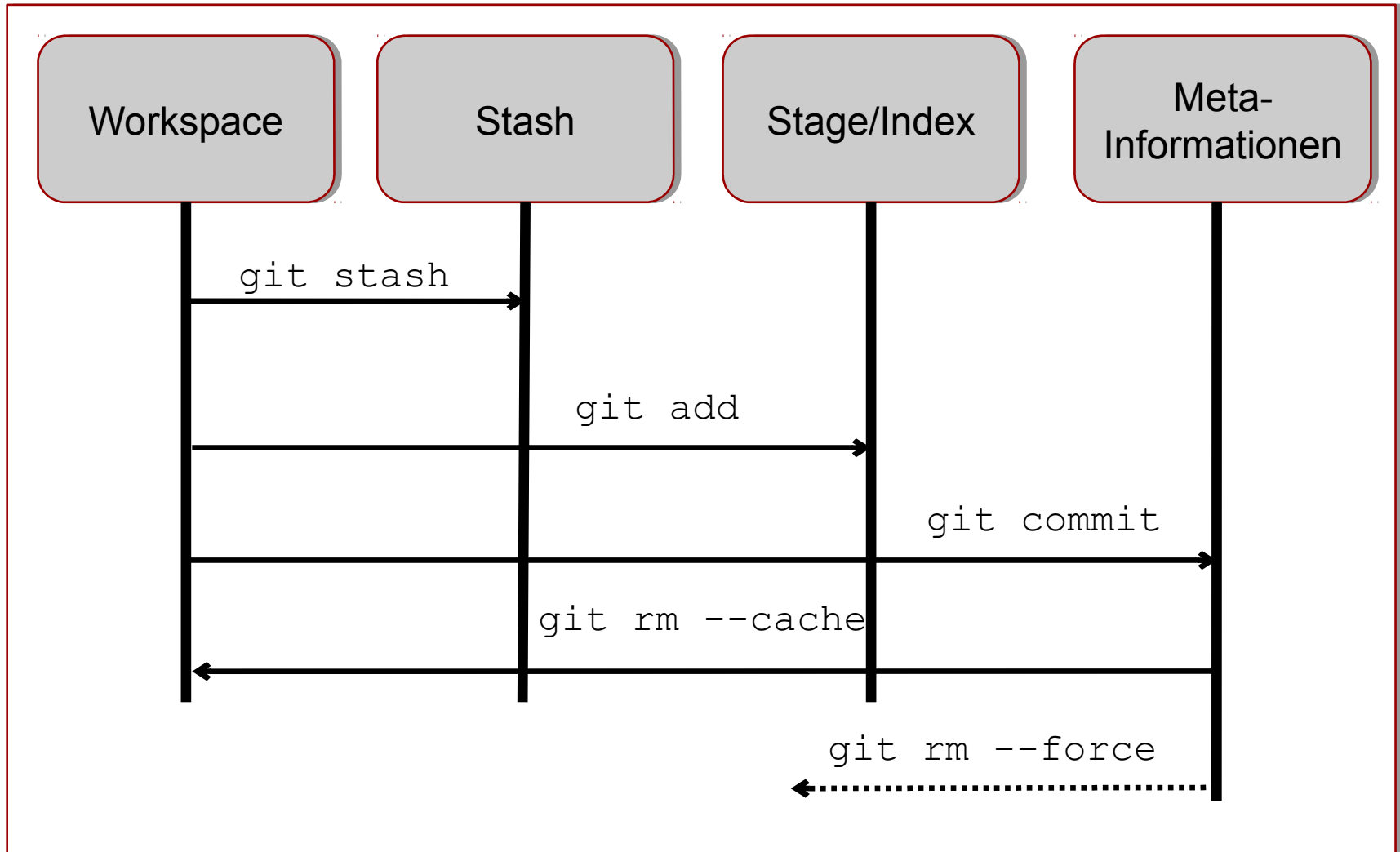
## 2.2

# **GIT KOMMANDOS**

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

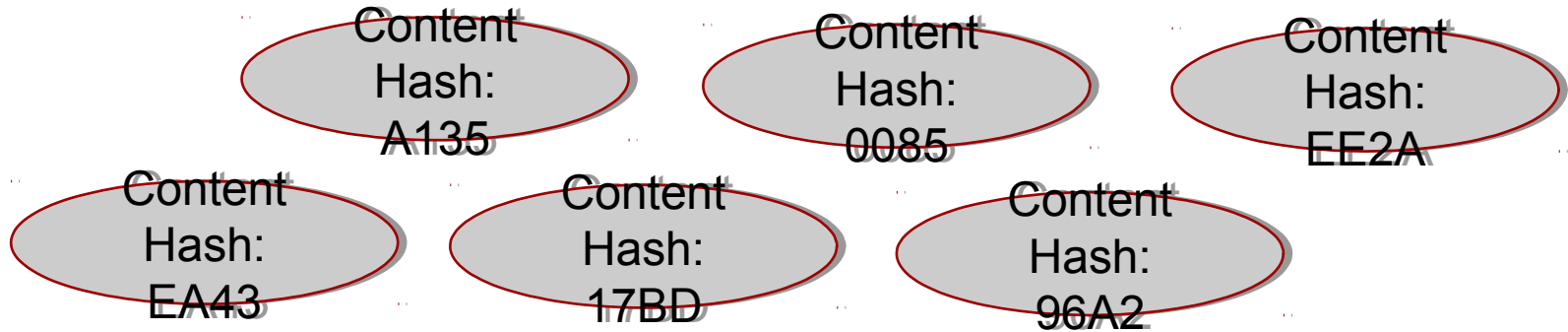
2.3

## **ABLÄUFE IM DETAIL**



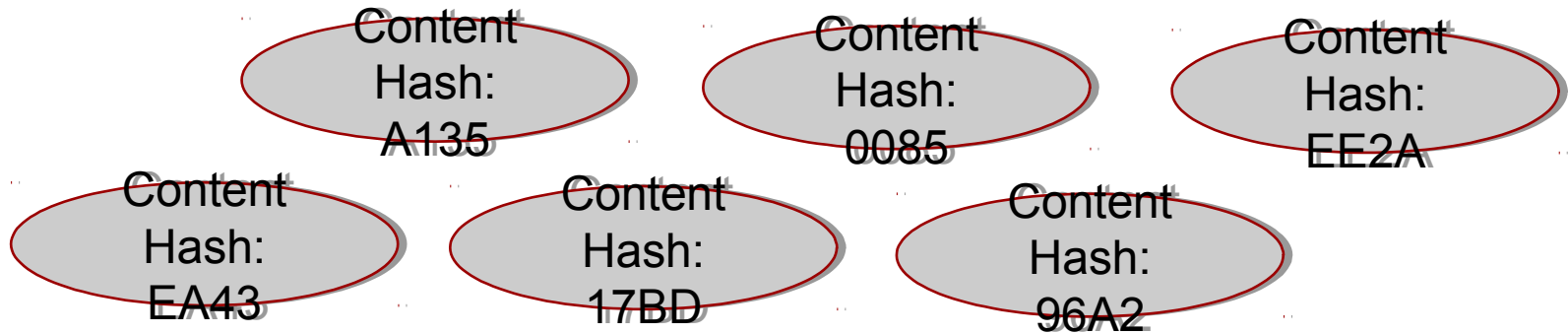
- `git stash`
  - Der gesamte Workspace wird unter einem Stash-Namen im Stash-Directory abgelegt
  - Ein simpler Backup

- `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



- `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



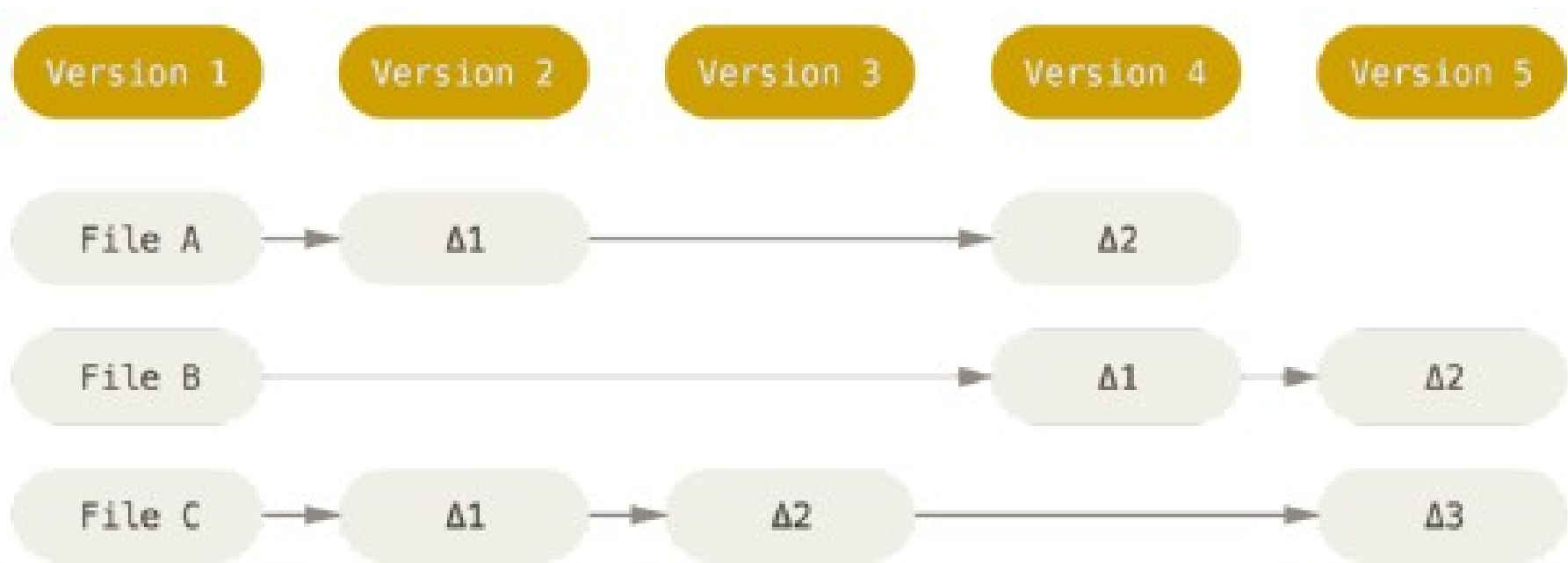


Commit  
Hash: 11E5  
Message: Init  
Refs:  
EA43  
17BD  
A135  
0085

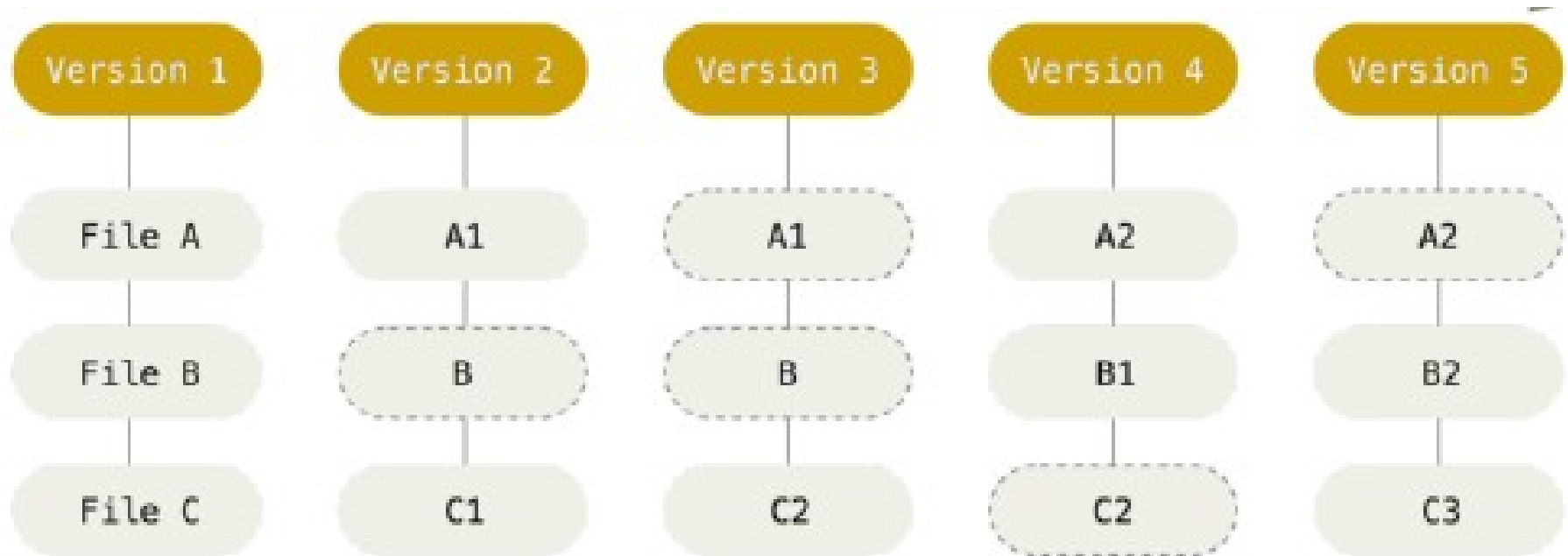
Commit  
Hash: FA17  
Message: Changed  
Refs:  
EA43  
96A2  
A135  
EE2A

- `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

- Delta-Informationen

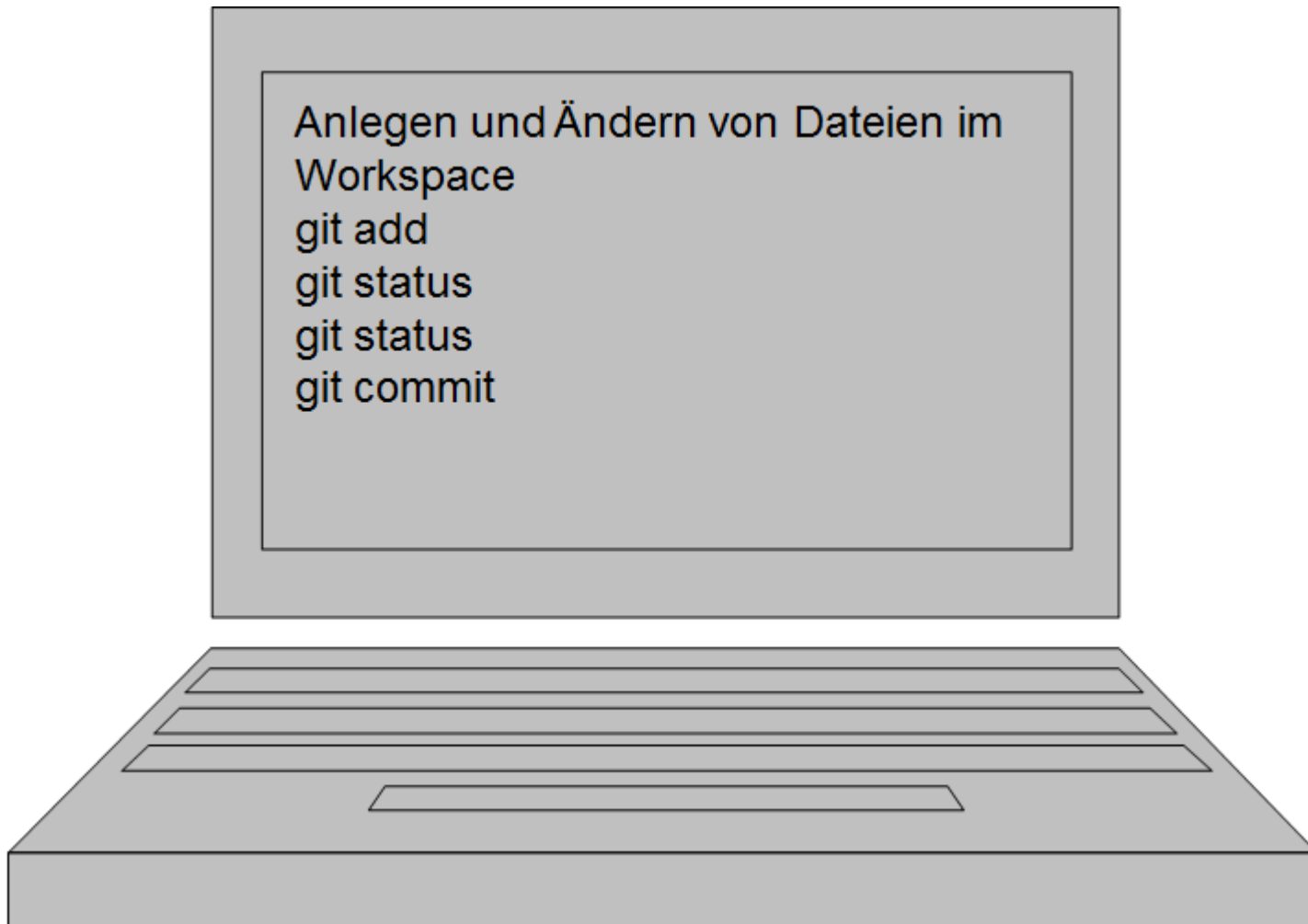


- Jeder Commit ist ein vollständiger Snapshot



- `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

- `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`



# 3

## **ARBEITEN MIT GIT**



3.1

## **TAGS**

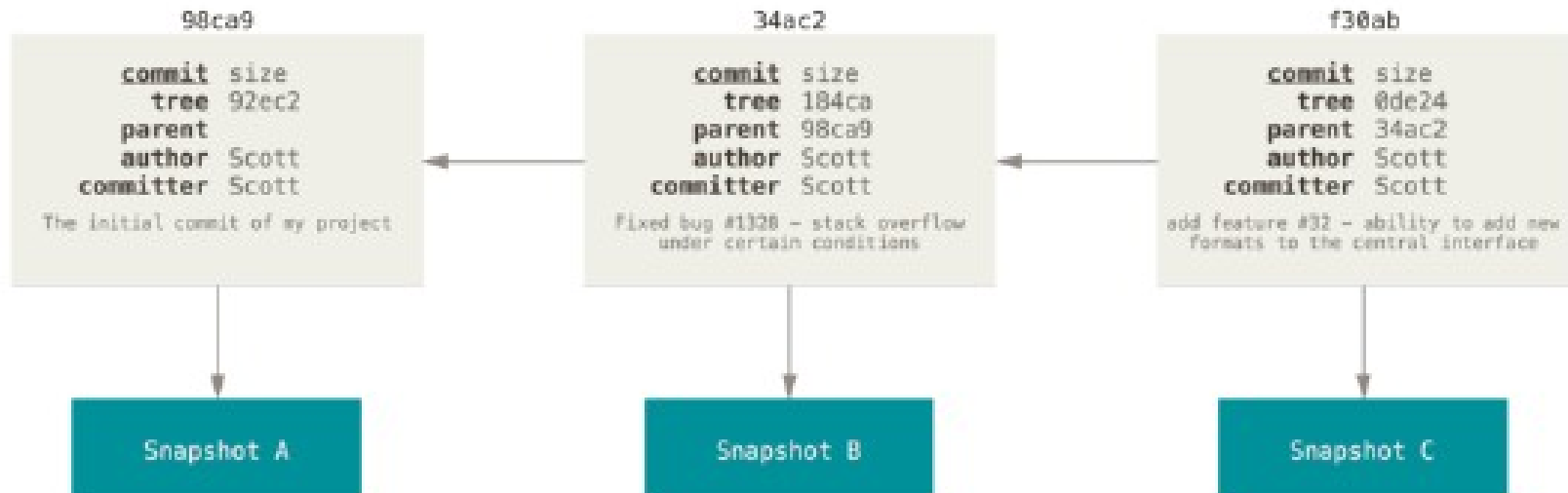
- Ein Tag ist eine interne, unveränderbare Referenz auf einen Commit
  - Auf Grund der Snapshot-Technik ist ein Tag in Git damit extrem einfach
- Zwei Kategorien
  - Lightweight
    - Nur die Referenz
  - Annotated
    - Zusätzliche Meta-Informationen
      - Message
      - Committer
      - Timestamp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

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

3.2

## **BRANCHES**

- Jeder Commit kennt seinen Parent



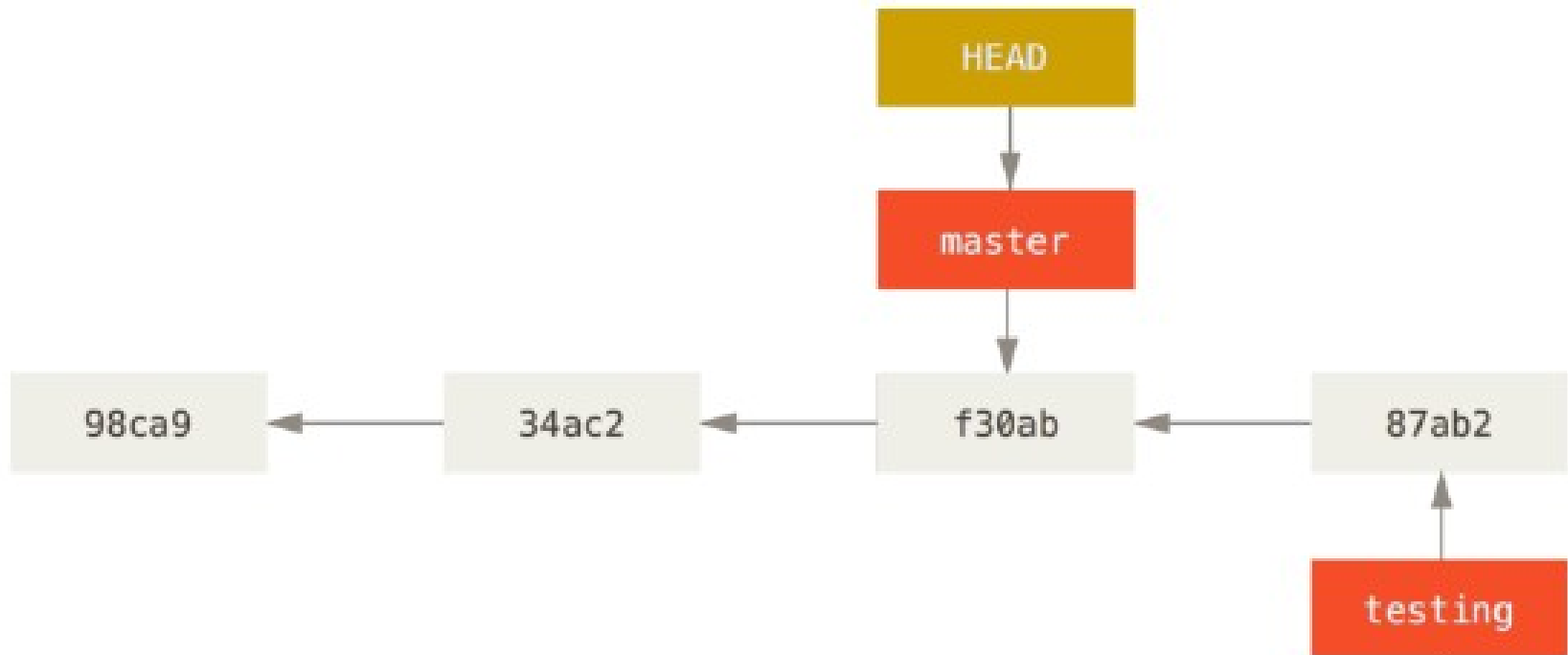
- Ein Commit wird identifiziert über
  - Einen SHA-Hash
  - ein Tag
- Ein Branch referenziert ein Commit-Objekt
- HEAD referenziert den aktuell ausgecheckten Branch
- Falls ein Commit-Objekt über einen Hash-Wert oder ein Tag ausgecheckt worden ist, befindet sich Git in einem Ausnahmestand
  - "Detached 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

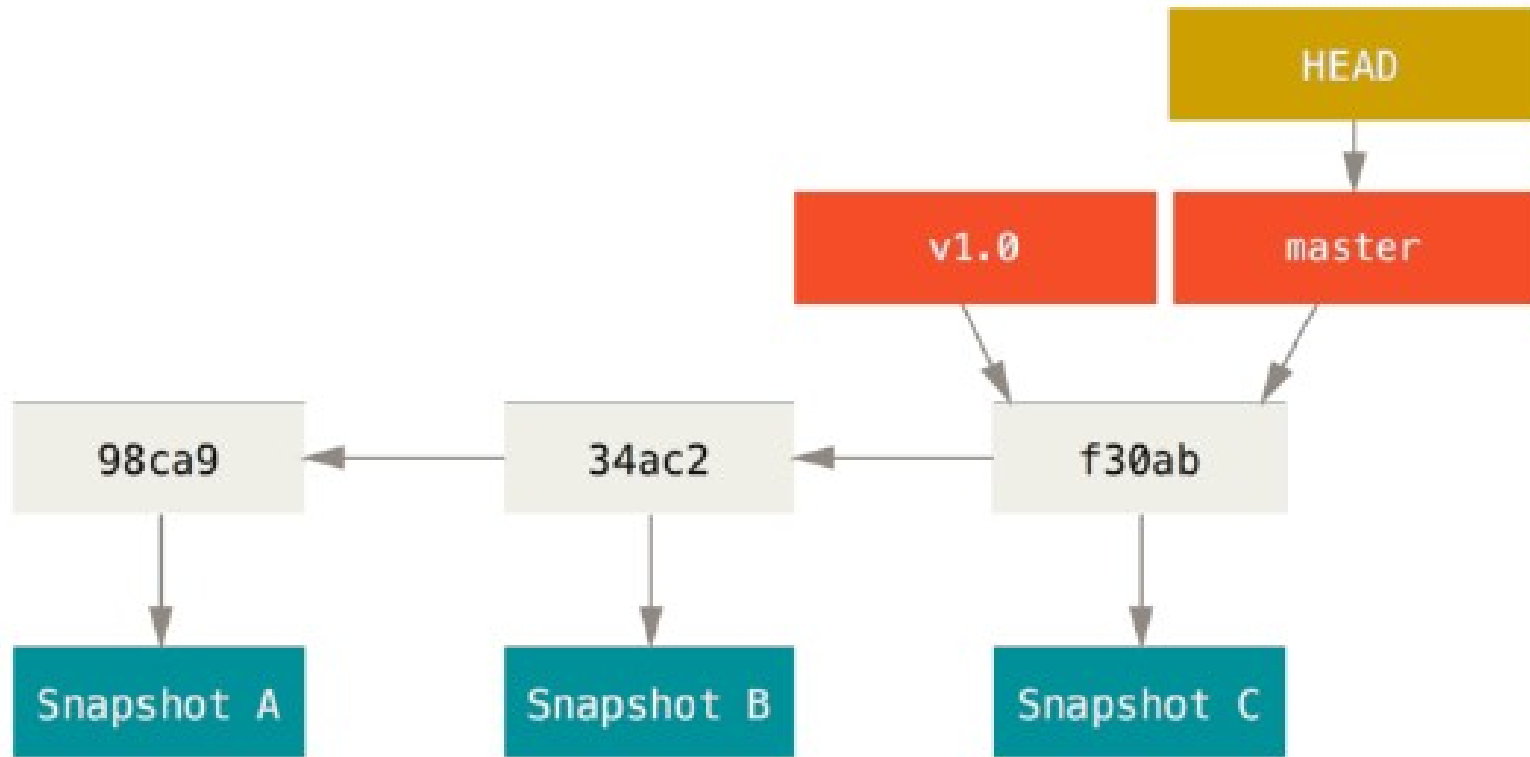
- `git branch <new-branch-name>`
  - z.B. `testing`



- `git checkout master`





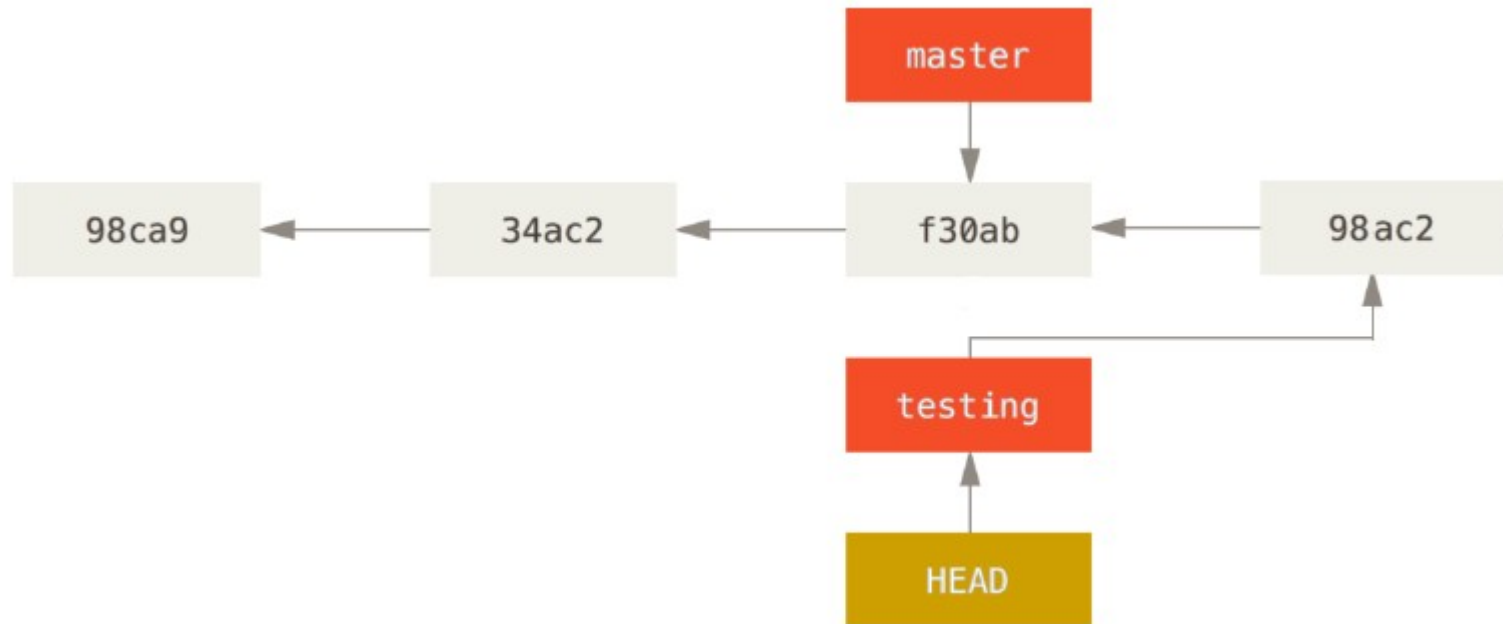


- `git checkout <branch-name>`
  - z.B. `testing`



- 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

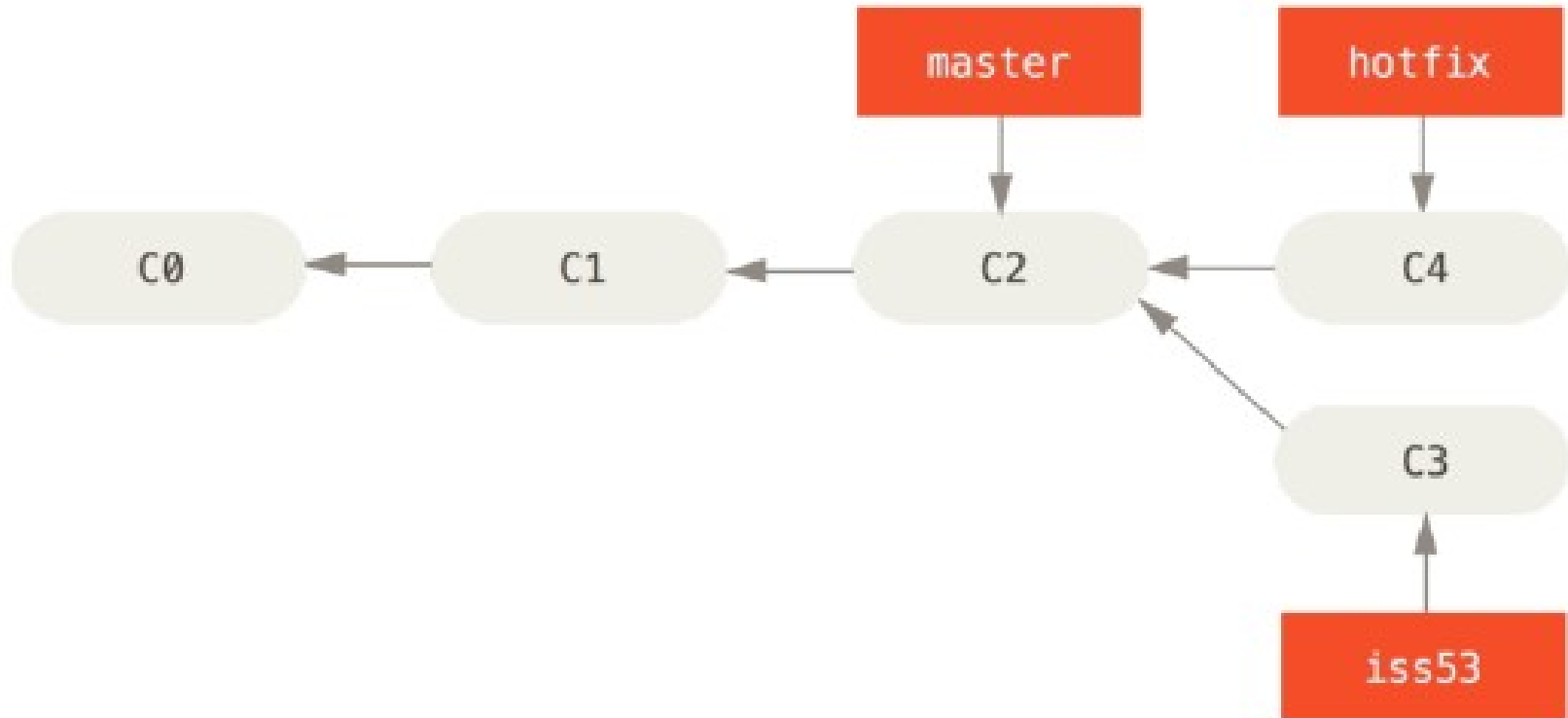




3.3

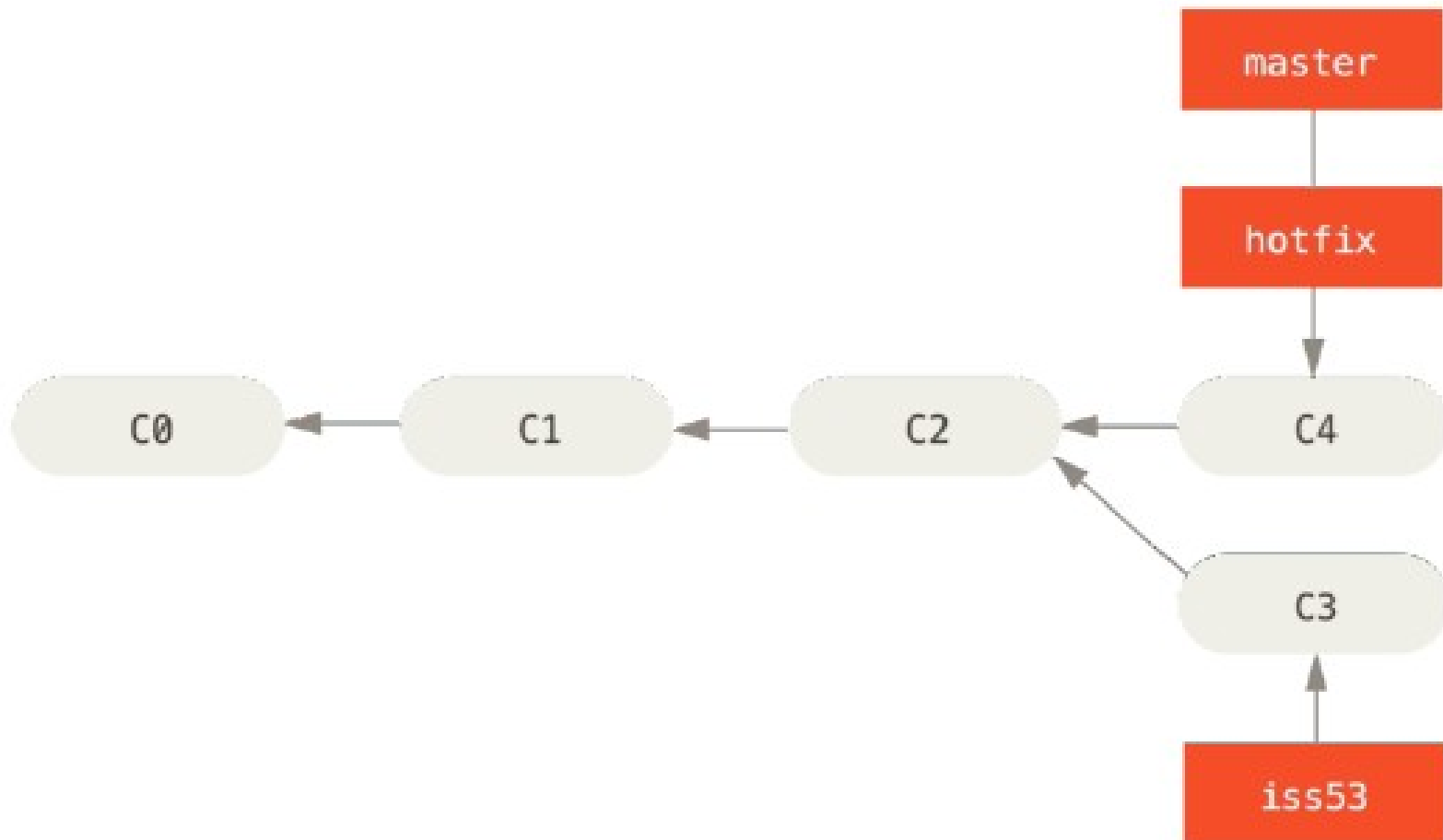
## MERGING

# Fast Forward Merge: Ausgangssituation



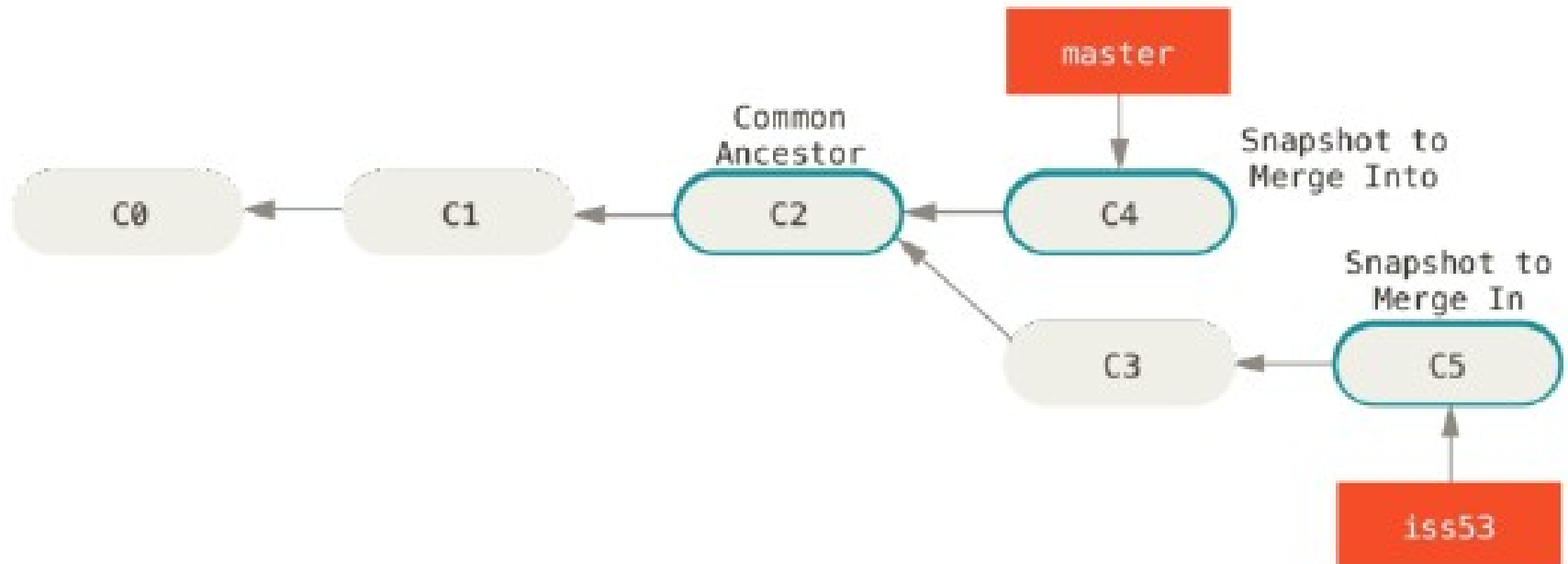
# Fast Forward Merges: Nach merge

- `git merge hotfix`



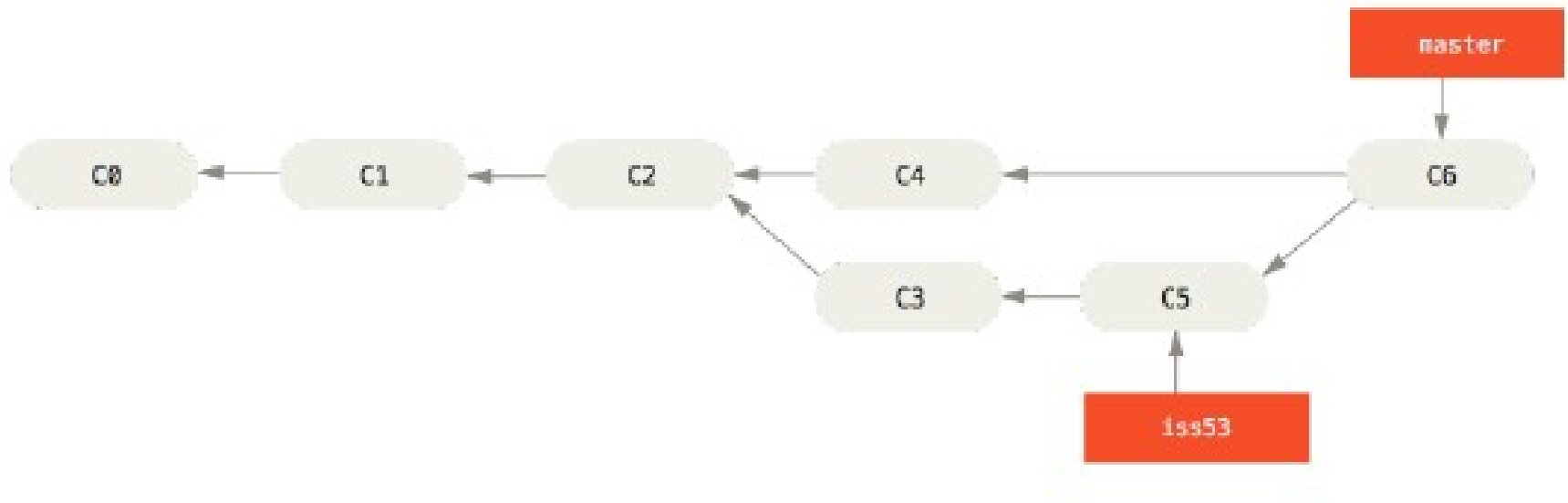


# Recursive merge: Ausgangssituation



# Recursive merge: Nach merge

- `git merge iss53`



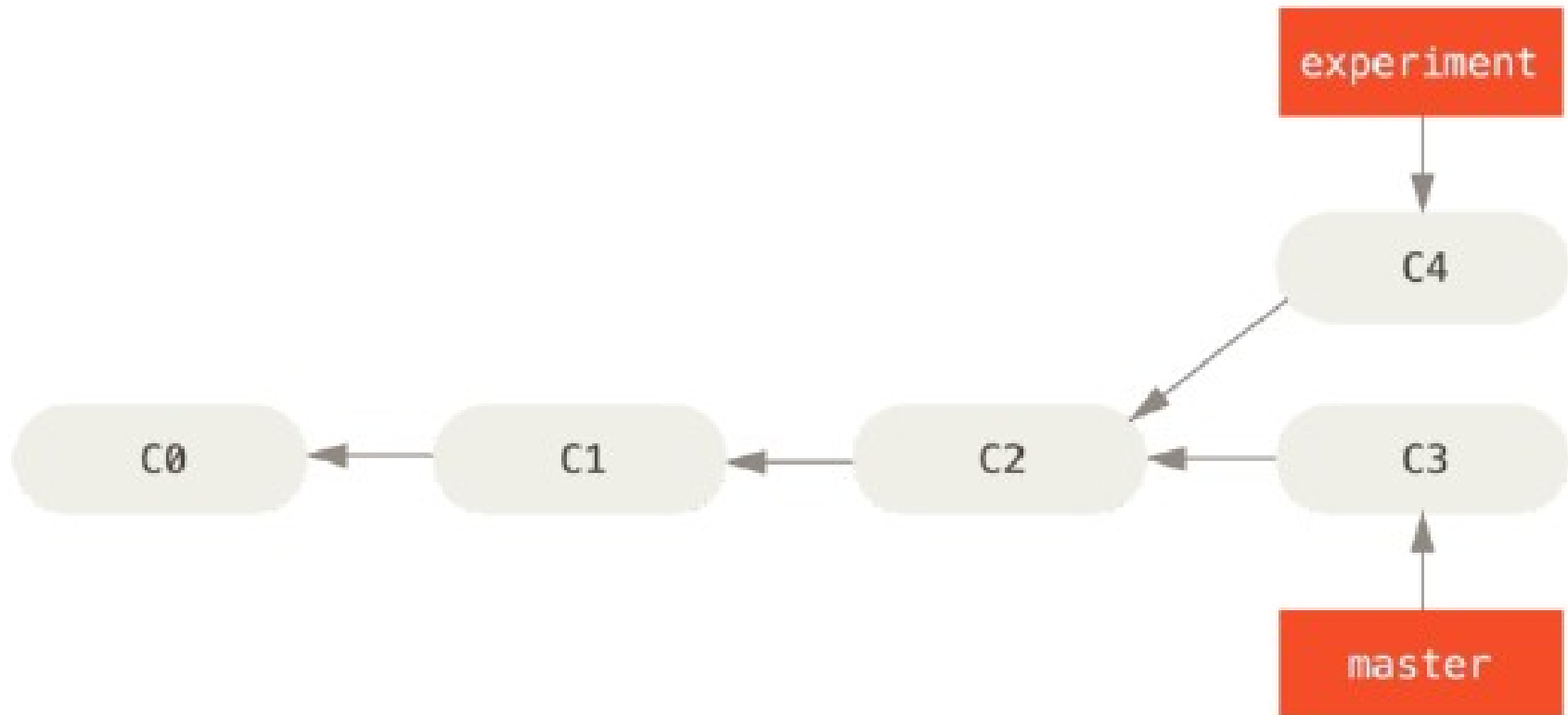
- Git hat bereits ein sehr ausgefeiltes Konzept, um Merge-Konflikte zu erkennen
  - Einfache Konflikte werden automatisch korrigiert
- Nicht-auflösbare Konflikte müssen händisch behoben werden
  - Was auch sonst...
- Git: "conflict resolution markers"

```
<<<<<<< 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
```

3.4

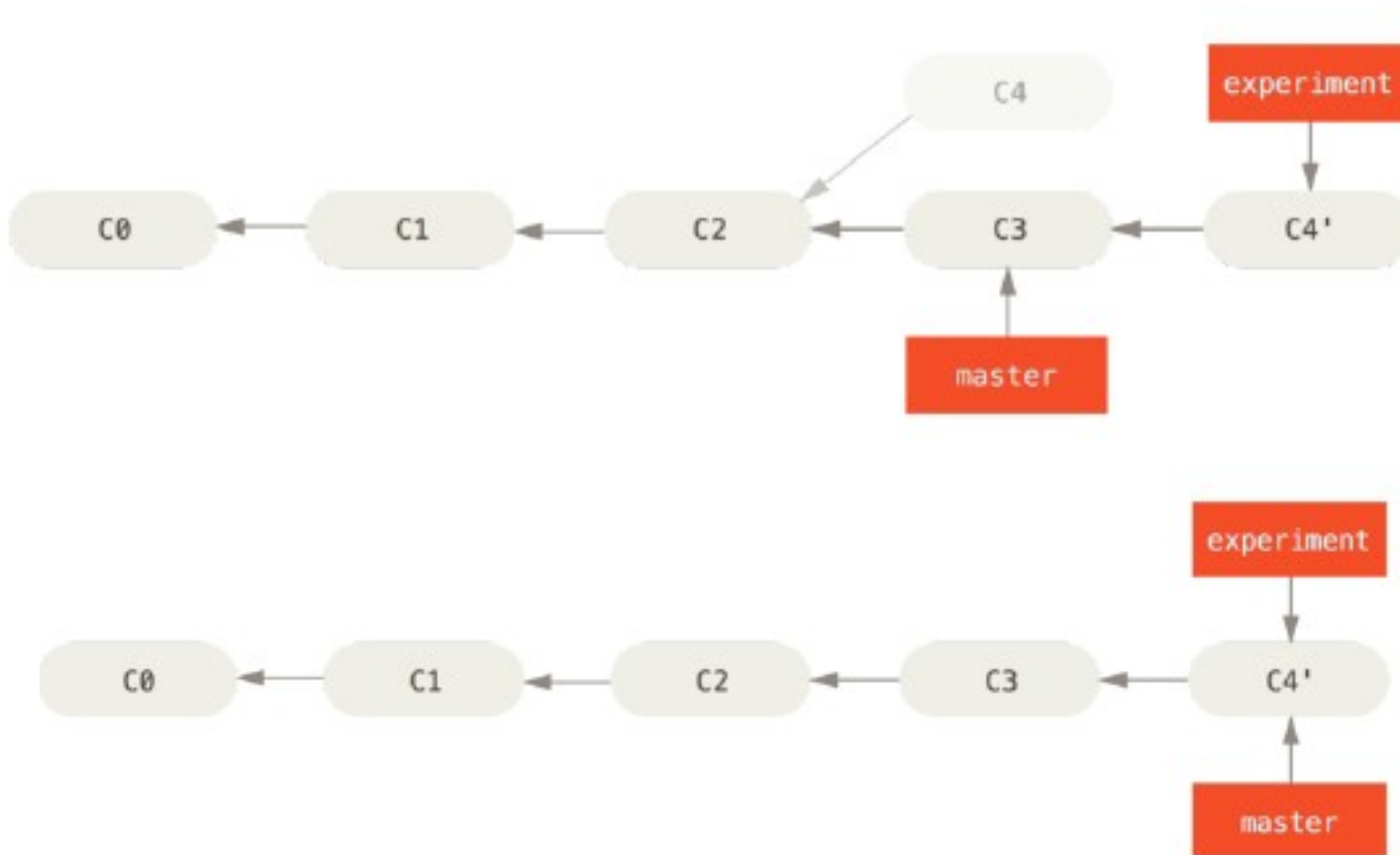
## REBASING

- 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: nach rebase

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



- 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`



- 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



3.5

## **CUSTOMIZING**

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

- 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>`

- `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

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

## 4

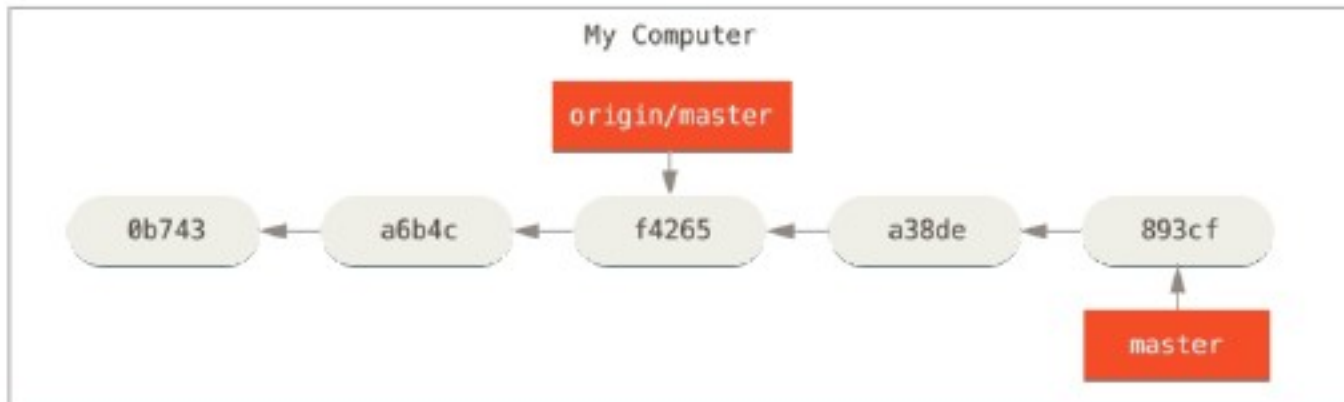
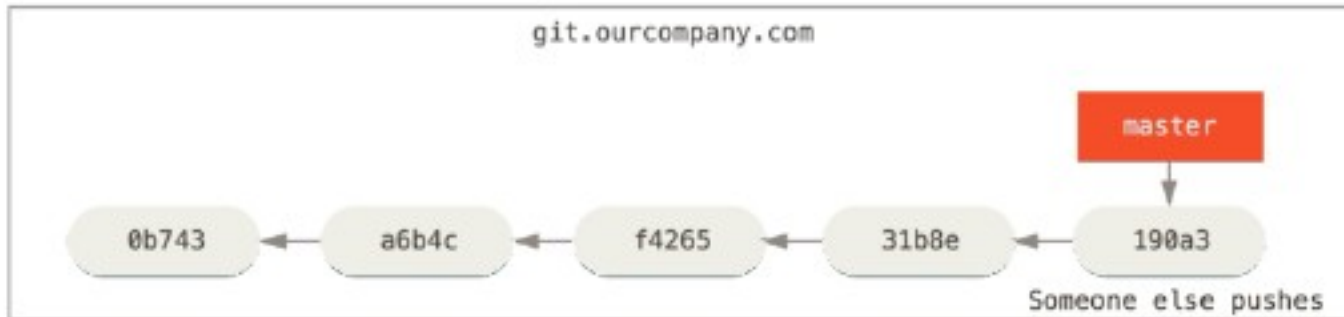
# DISTRIBUTED REPOSITORIES



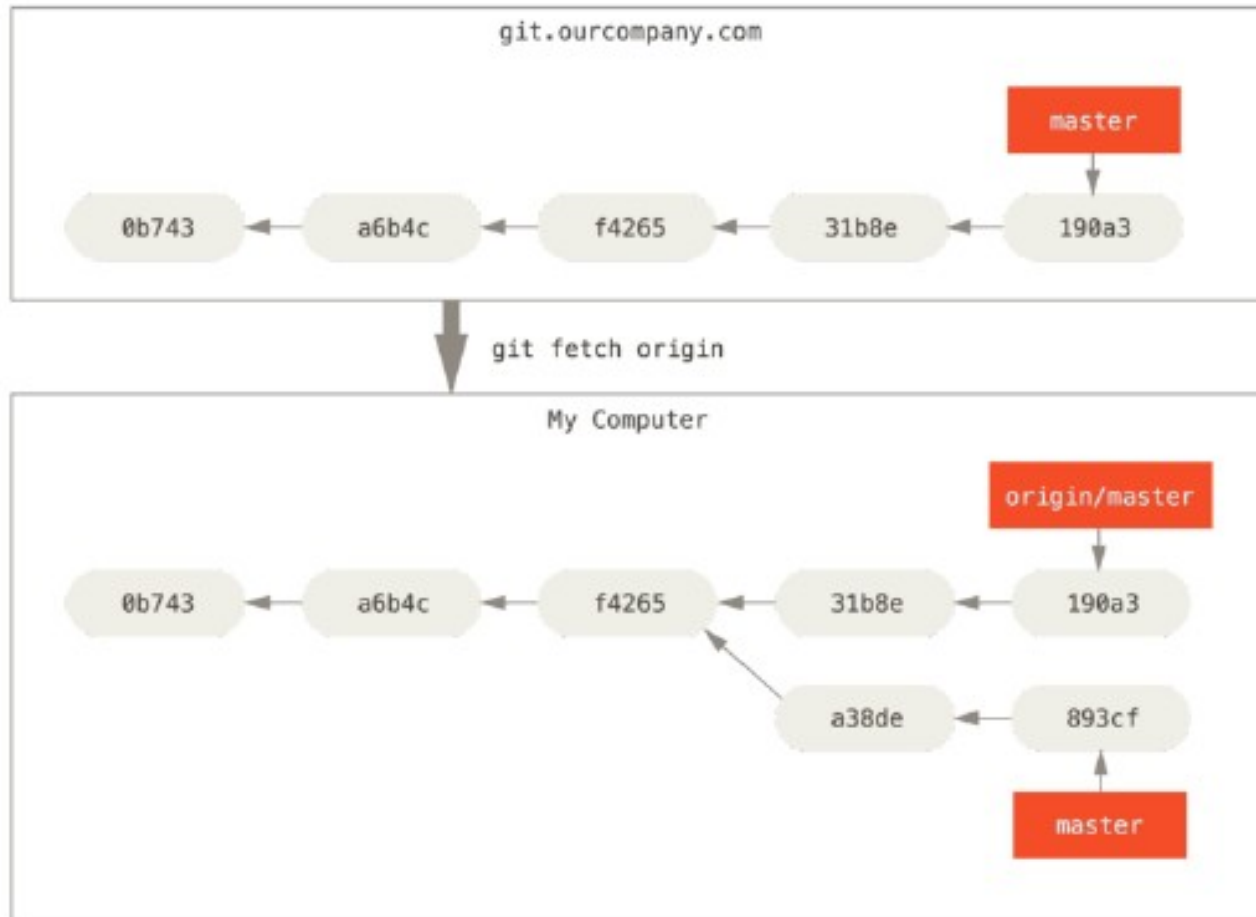
4.1

## **PULL UND PUSH**

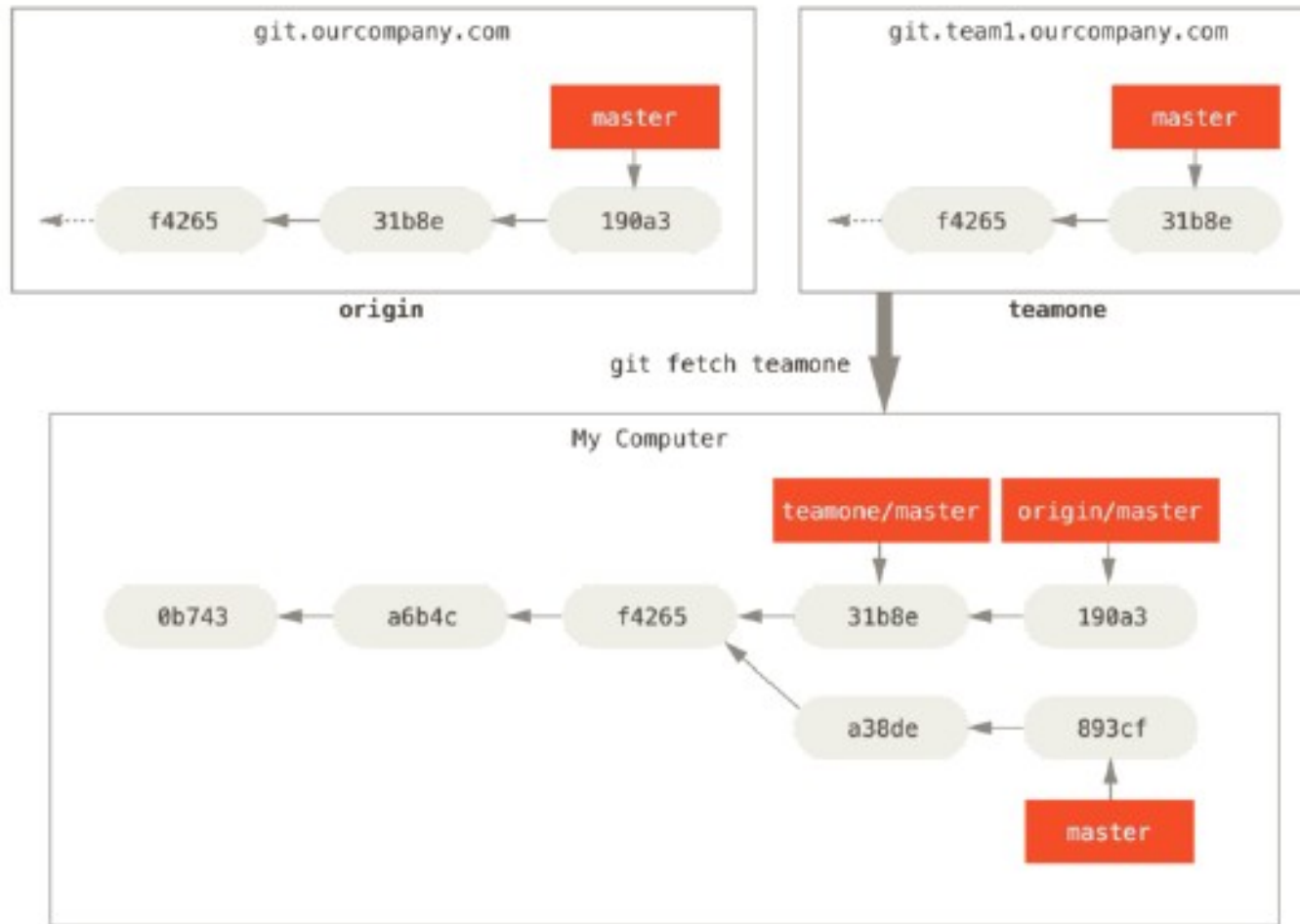
- Zur Unterscheidung werden Repositories durch einen eigenen Namespace identifiziert
  - Geclonetes Repository: `origin`



# Fetching



# Fetching mit mehreren Remote Servern



- 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`

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

4.2

## 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"



- 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>`

5

## **GIT AUF DEM SERVER**

5.1

## ÜBERBLICK GIT SERVER

- 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

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

- 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

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

5.2


## **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 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** ✕

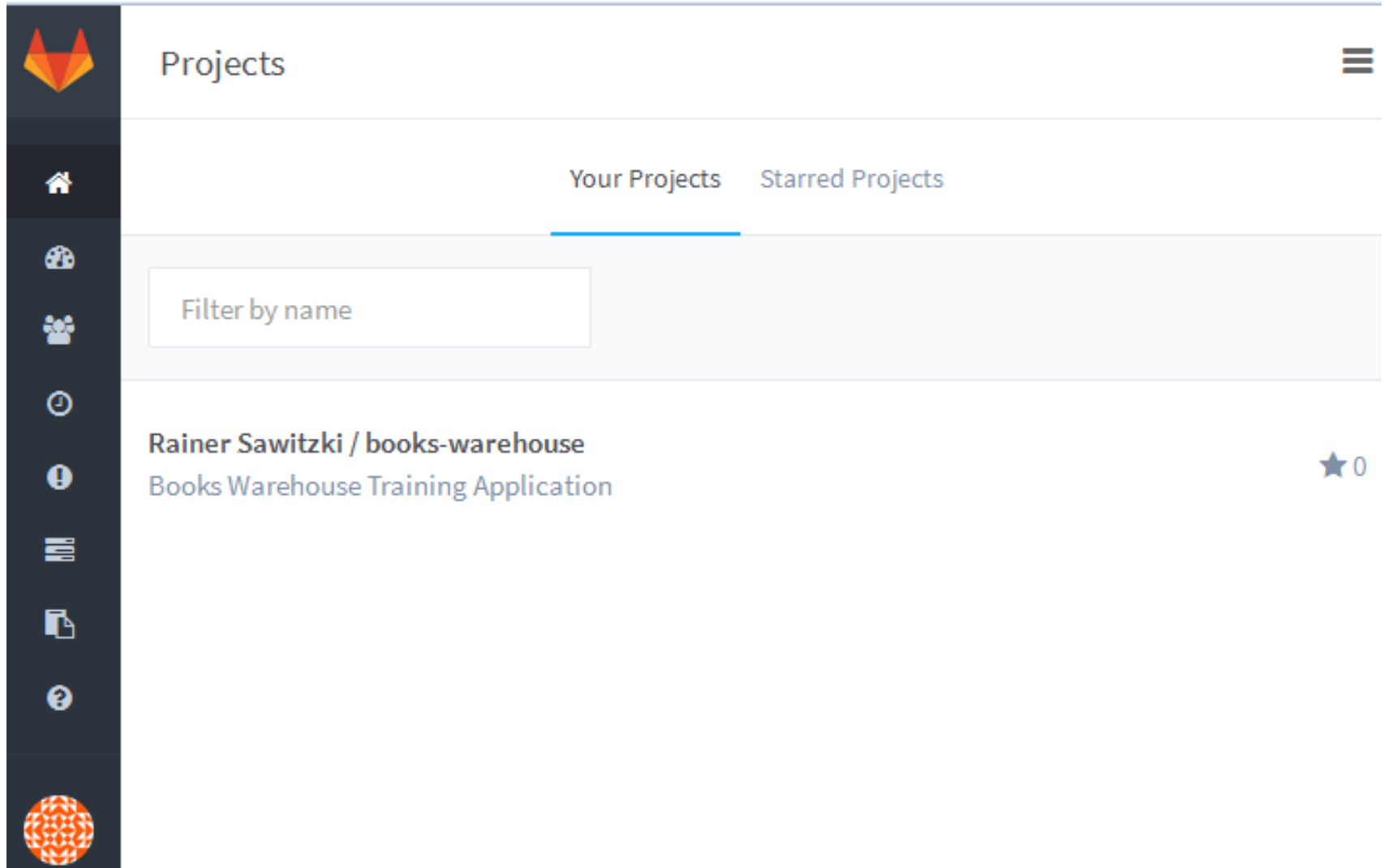
Our on-demand training option will have you contributing on GitHub quicker than you can say Pull Request!

5.3

## **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`

- 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/>



- 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

## Web hooks

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

URL	<input type="text" value="http://example.com/trigger-ci.json"/>
Trigger	<div><input checked="" type="checkbox"/> <b>Push events</b> This url will be triggered by a push to the repository</div> <div><input type="checkbox"/> <b>Tag push events</b> This url will be triggered when a new tag is pushed to the repository</div> <div><input type="checkbox"/> <b>Comments</b> This url will be triggered when someone adds a comment</div> <div><input type="checkbox"/> <b>Issues events</b> This url will be triggered when an issue is created</div> <div><input type="checkbox"/> <b>Merge Request events</b> This url will be triggered when a merge request is created</div>



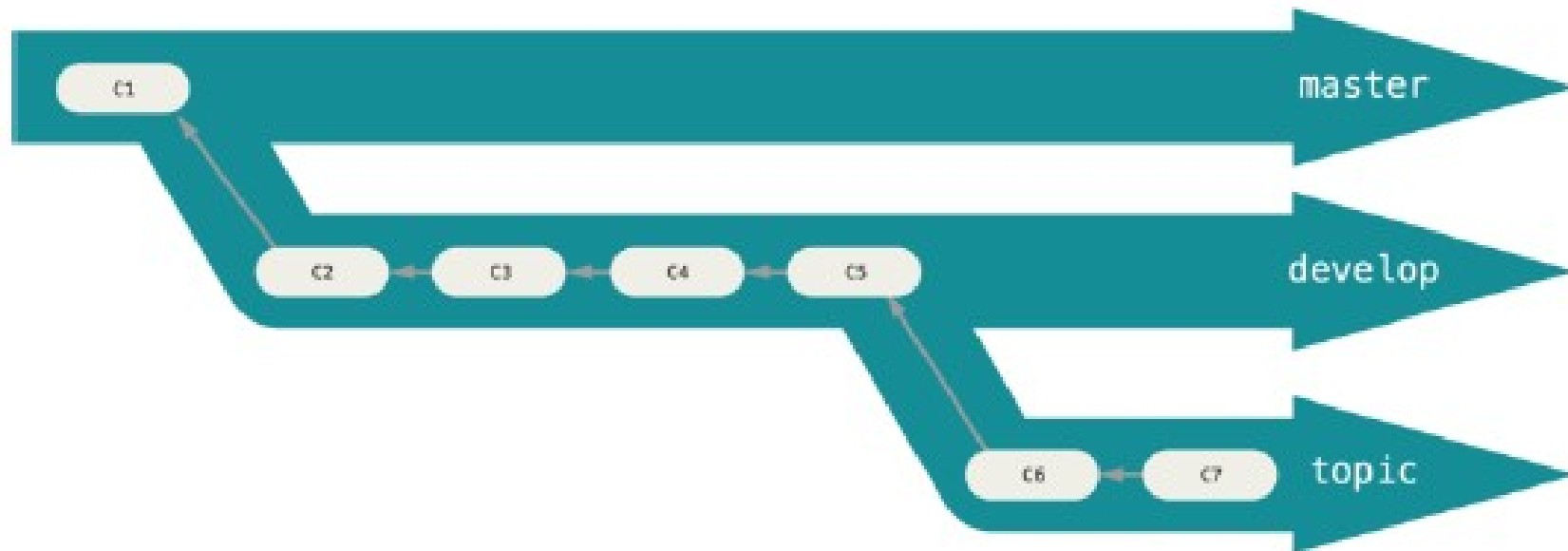
6

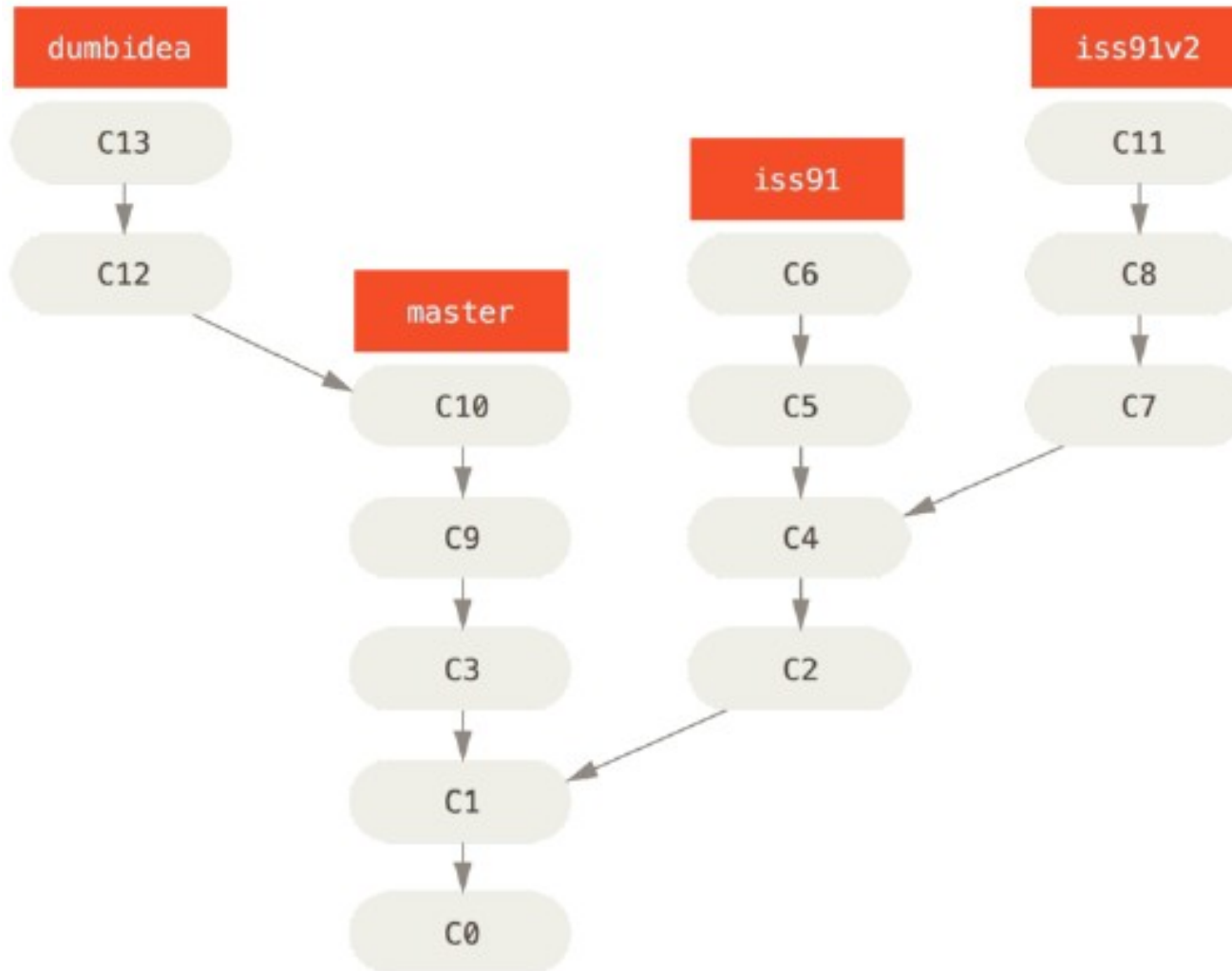
## WORKFLOWS

6.1

## ÜBERSICHT

- 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





6.2

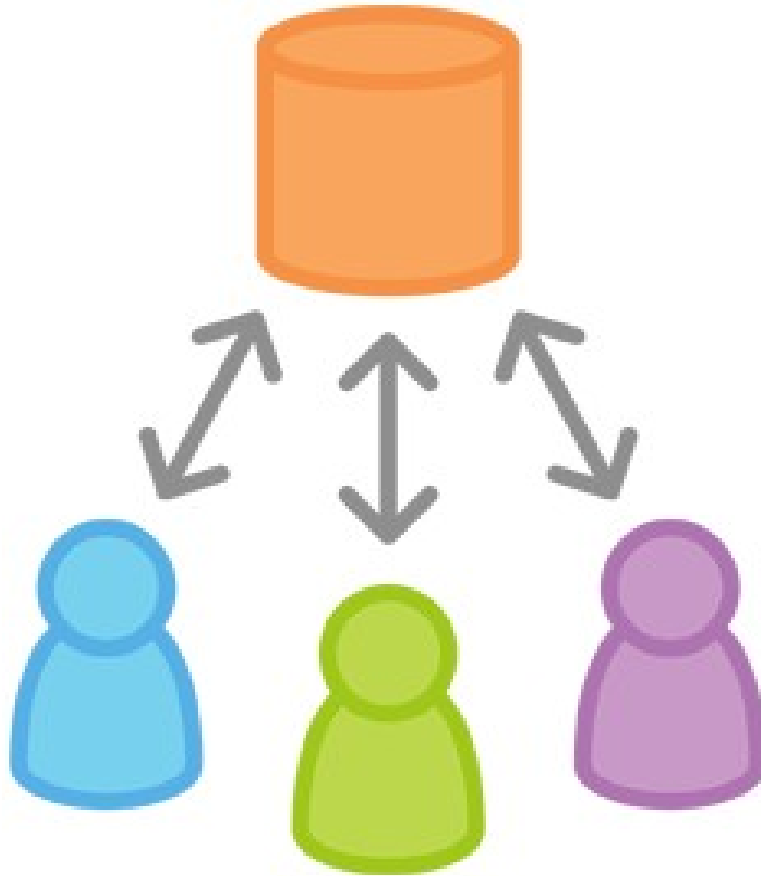
## **ATLASSIAN FLOWS**

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

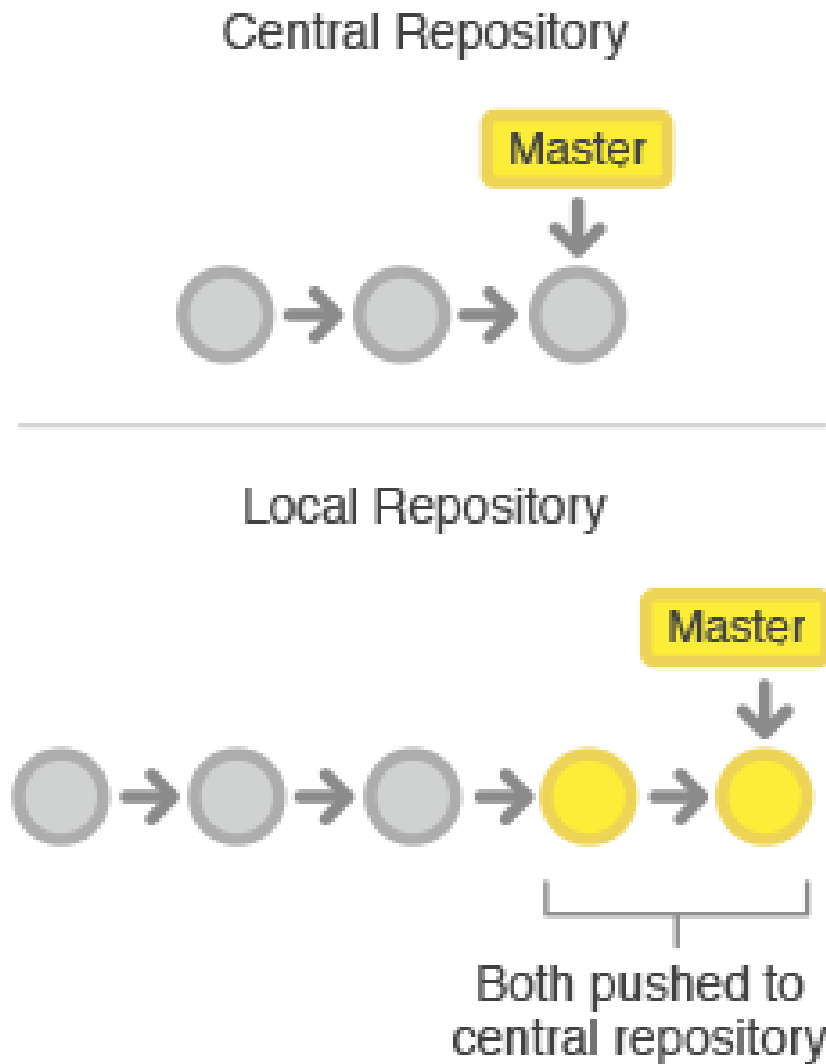
- Whitespace Prüfungen zur Vermeidung unnötiger Diffs
  - `git diff -check`
- One commit per Issue
  - Insbesondere bei Anbindung an ein Ticket-System wie Jira
- Sprechende Commit Messages
  - Maximal 50 Zeichen für beschreibendes Kommando
  - Detailbeschreibung mit Motivation (Issue) und Abgrenzung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



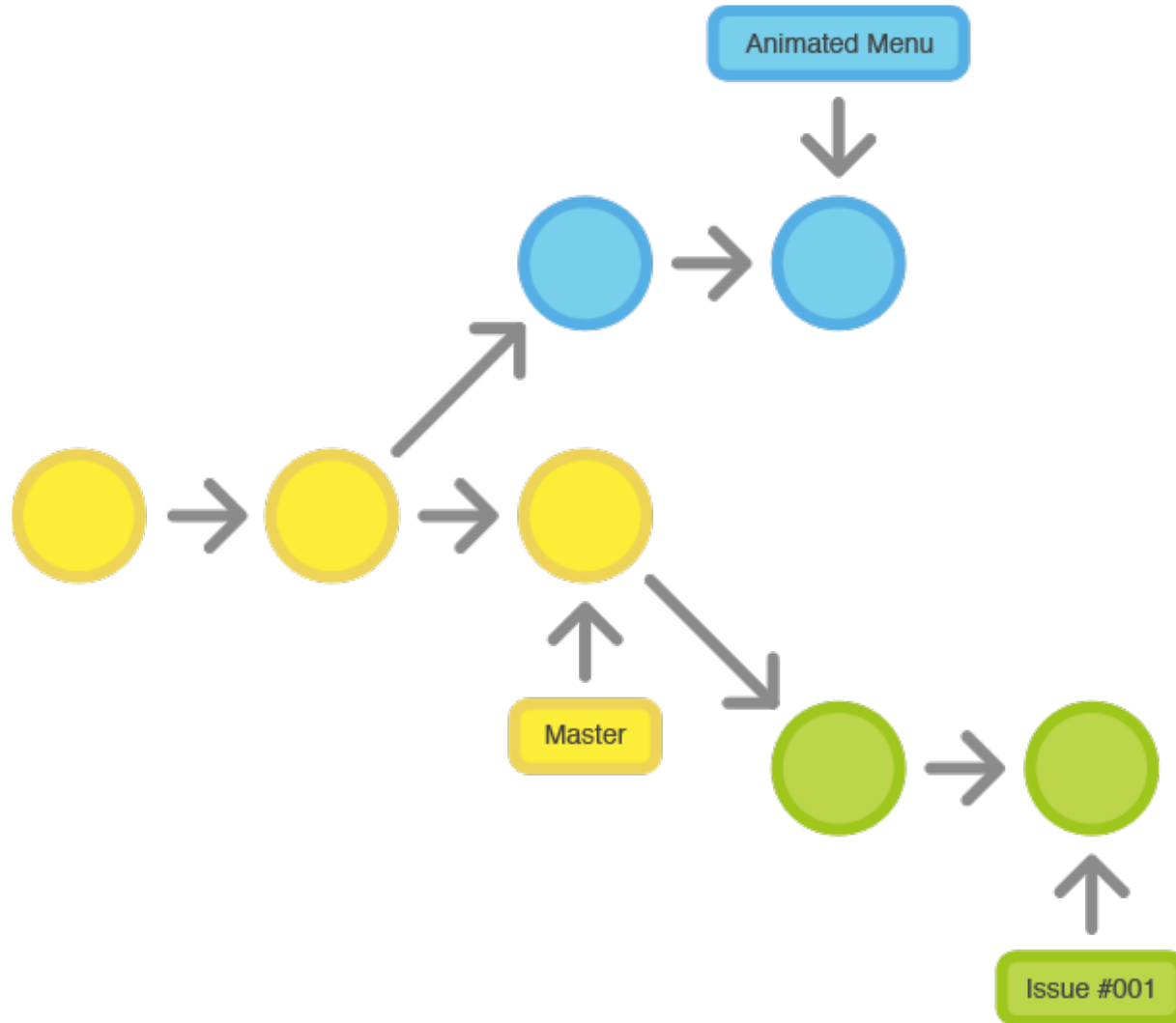
- Mehrere Benutzer teilen ein gemeinsames Repository



# Centralized Workflow: Commit

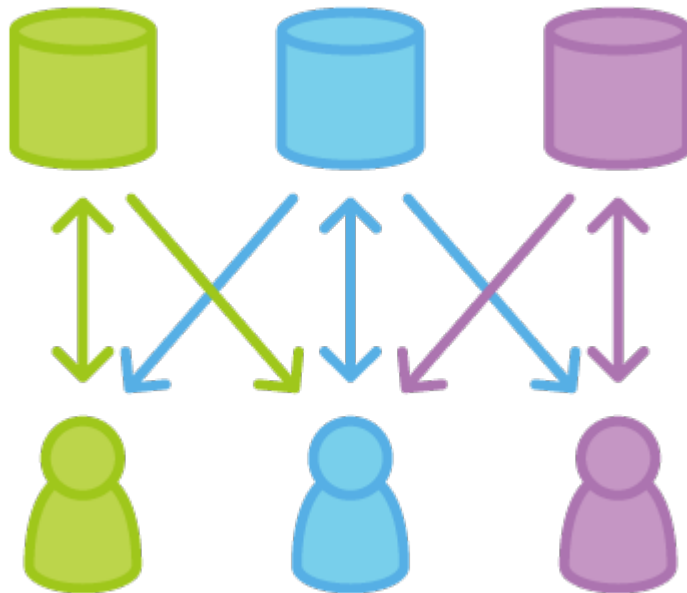


# Feature Branch Workflow

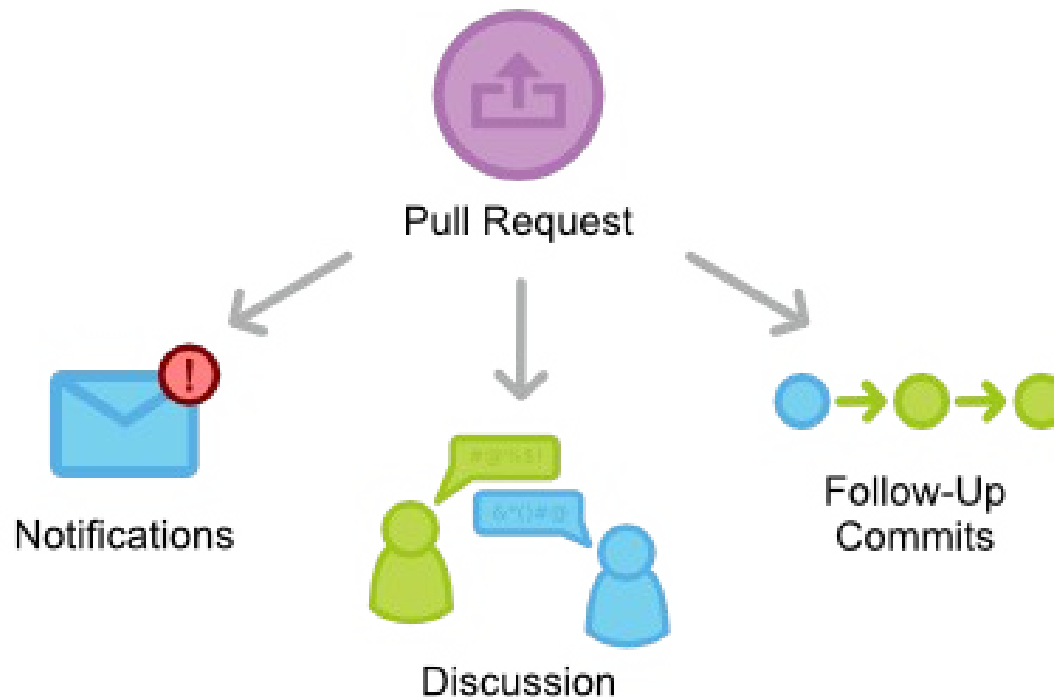




- Jeder Developer besitzt zwei Repositories
  - Ein lokales
  - Ein Remote



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



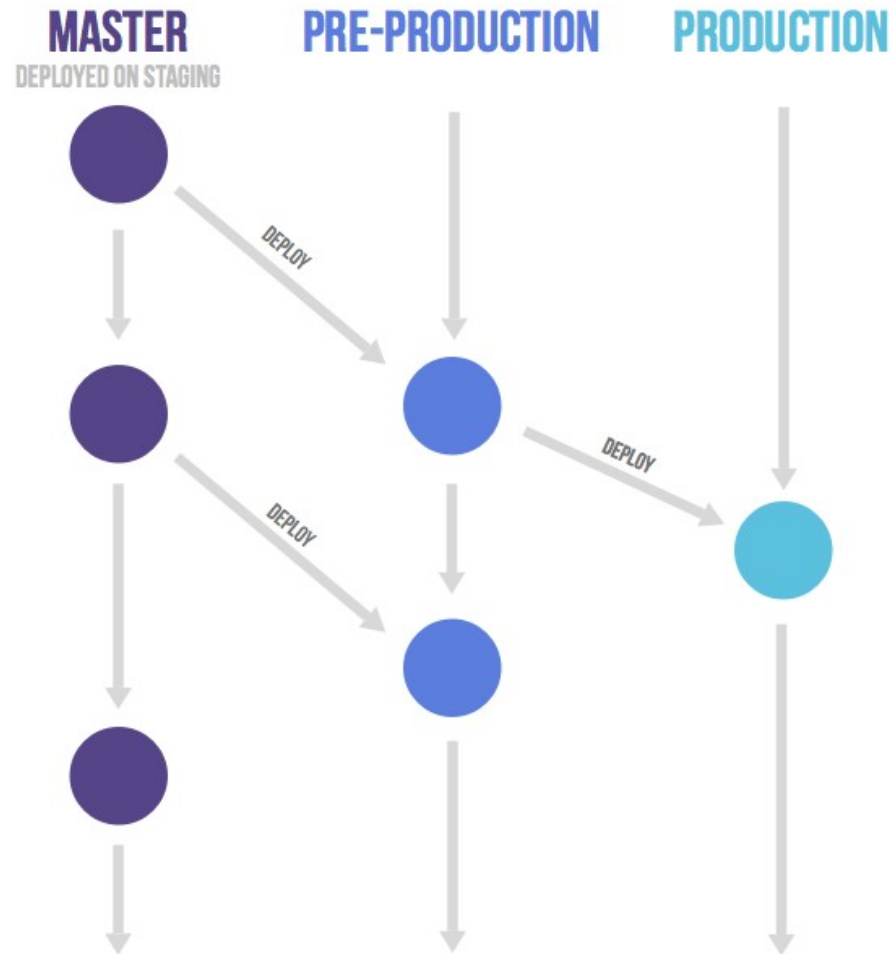
6.3

## **GITLAB FLOW**

- 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



7

## **GIT-CLIENTS**

7.1

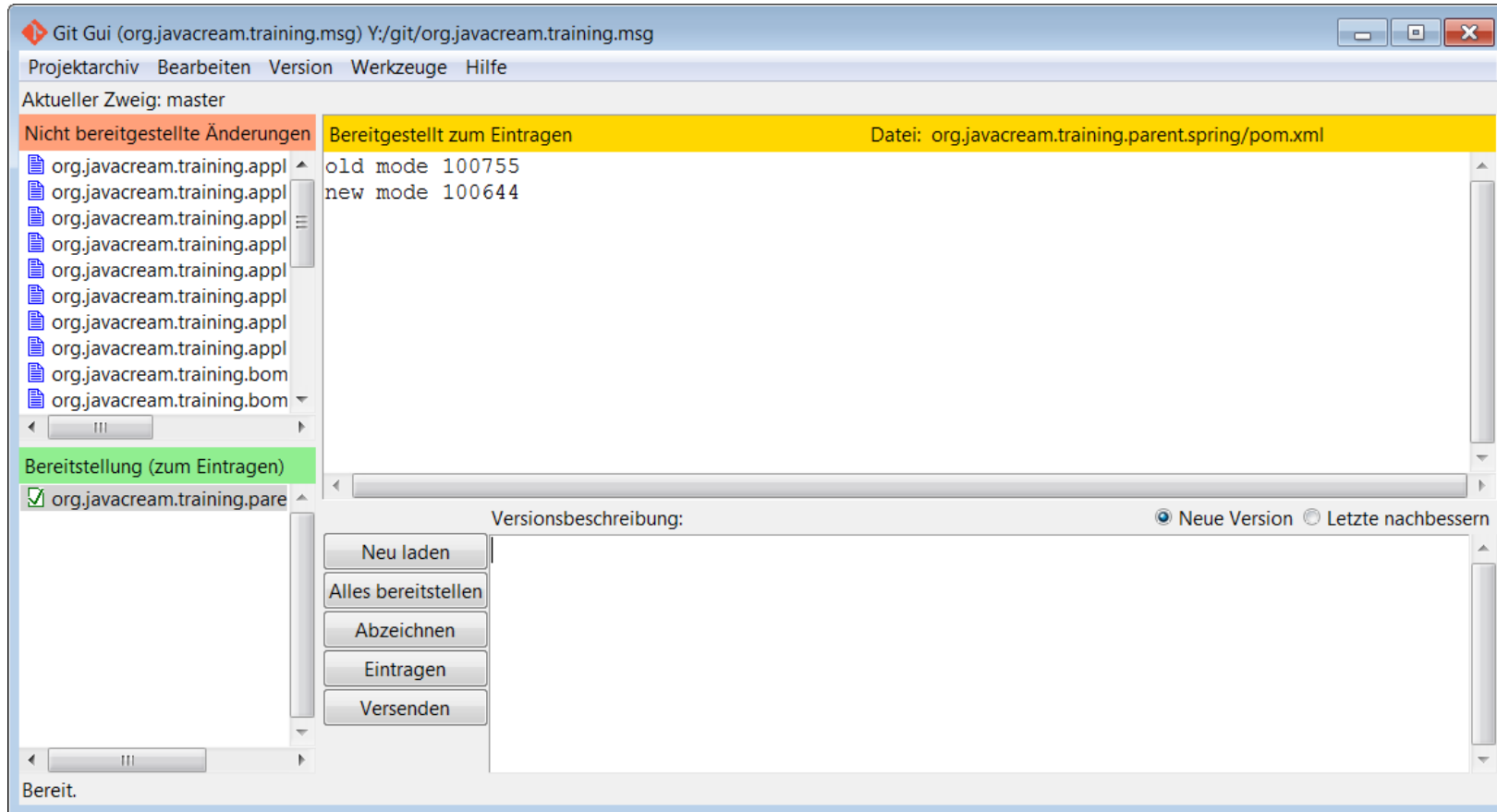
## AUFGABEN

- 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
  - ...

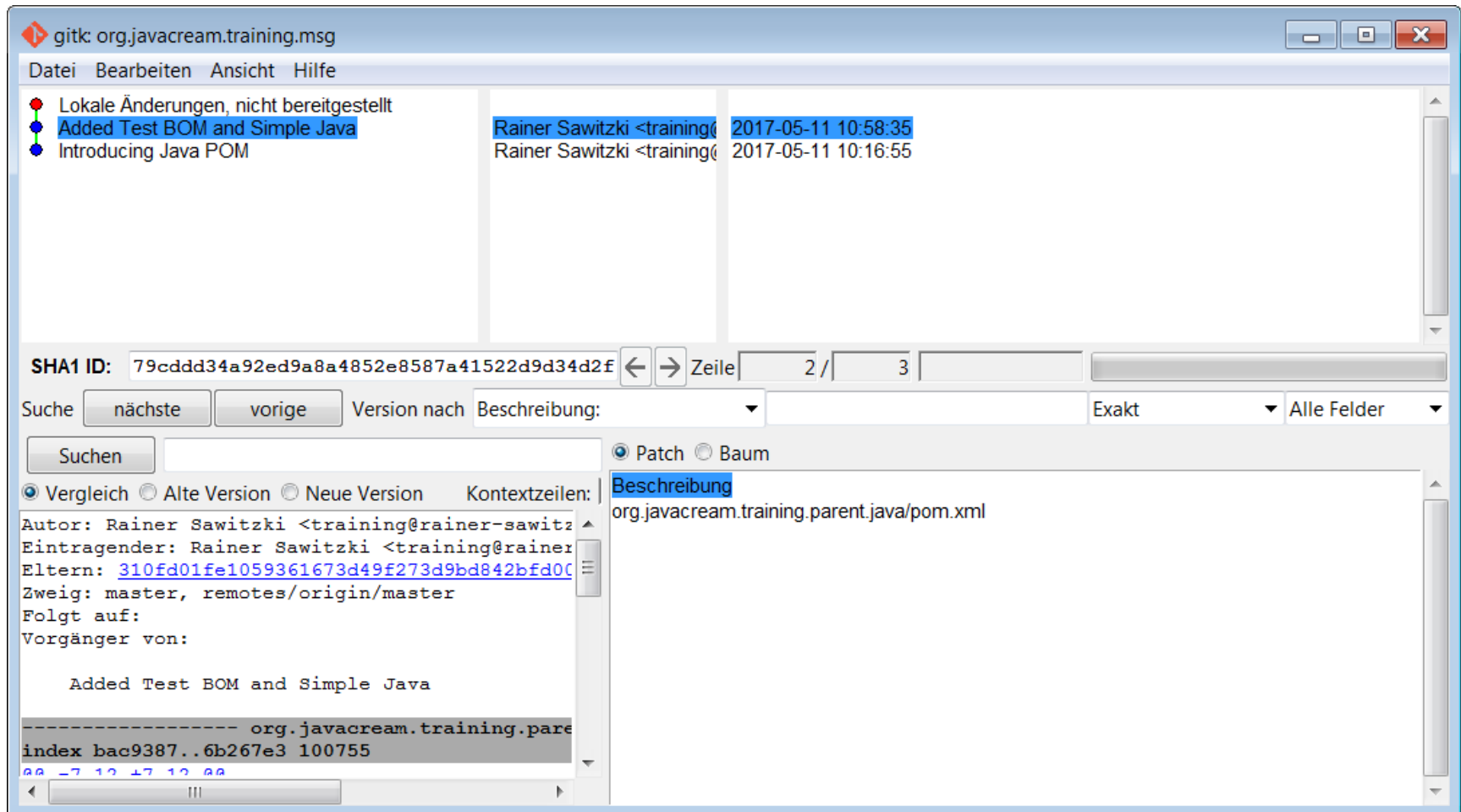
7.2

## **DIE GIT GUI ALS EINFACHER CLIENT**

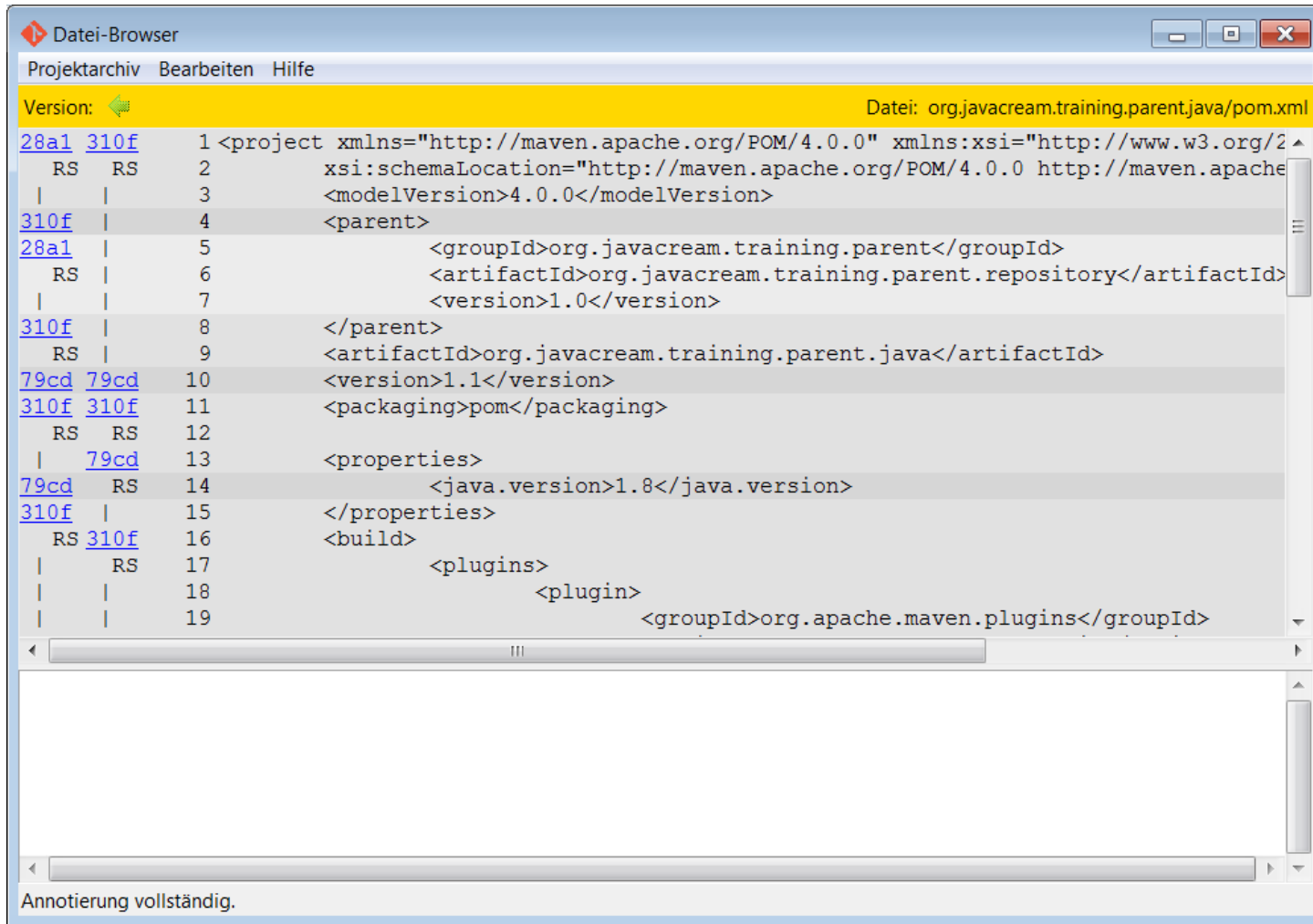
# Git Client Windows: Commit



# Git Client Windows: History



# Git Client Windows: Blame





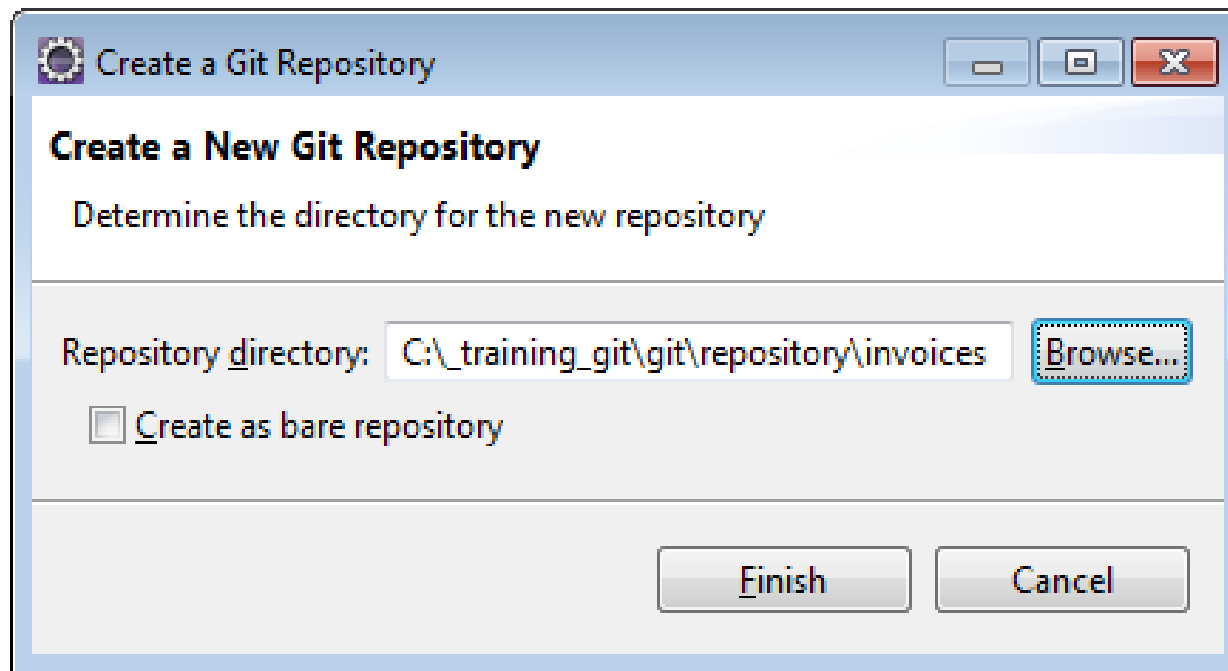
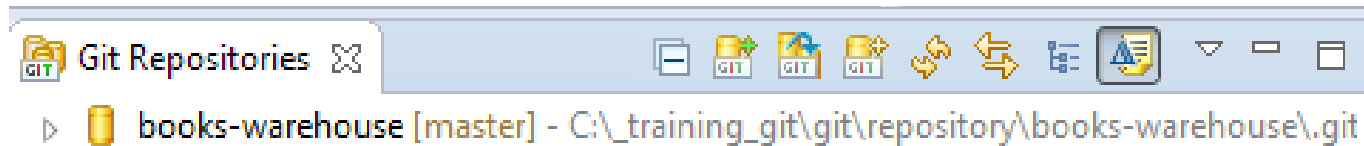
7.3

## **ECLIPSE ALS BEISPIEL FÜR EINE ENTWICKLUNGSUMGEBUNG**

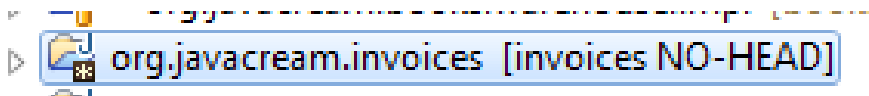
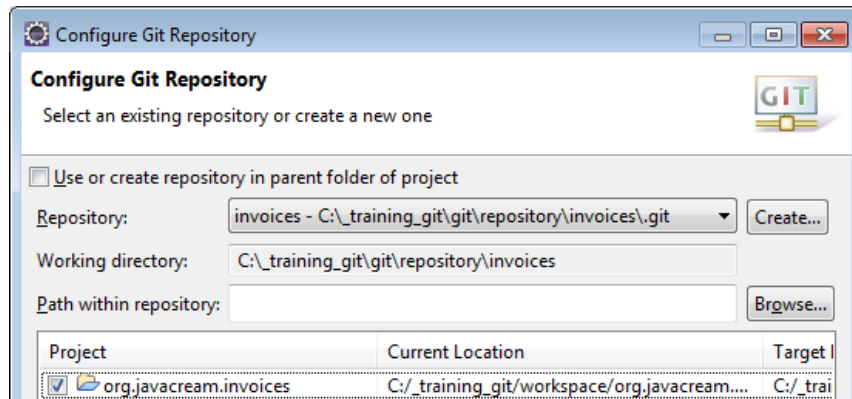
- Weit verbreitete Entwicklungsumgebung für Java, C, ...
- Eclipse bringt Git in der Standard-Installation bereits mit
- Alternativ können natürlich auch andere Clients oder Entwickler-Werkzeuge benutzt werden
  - Übersicht unter <https://git-scm.com/downloads/guis/>

# Eclipse: Git-Konfiguration

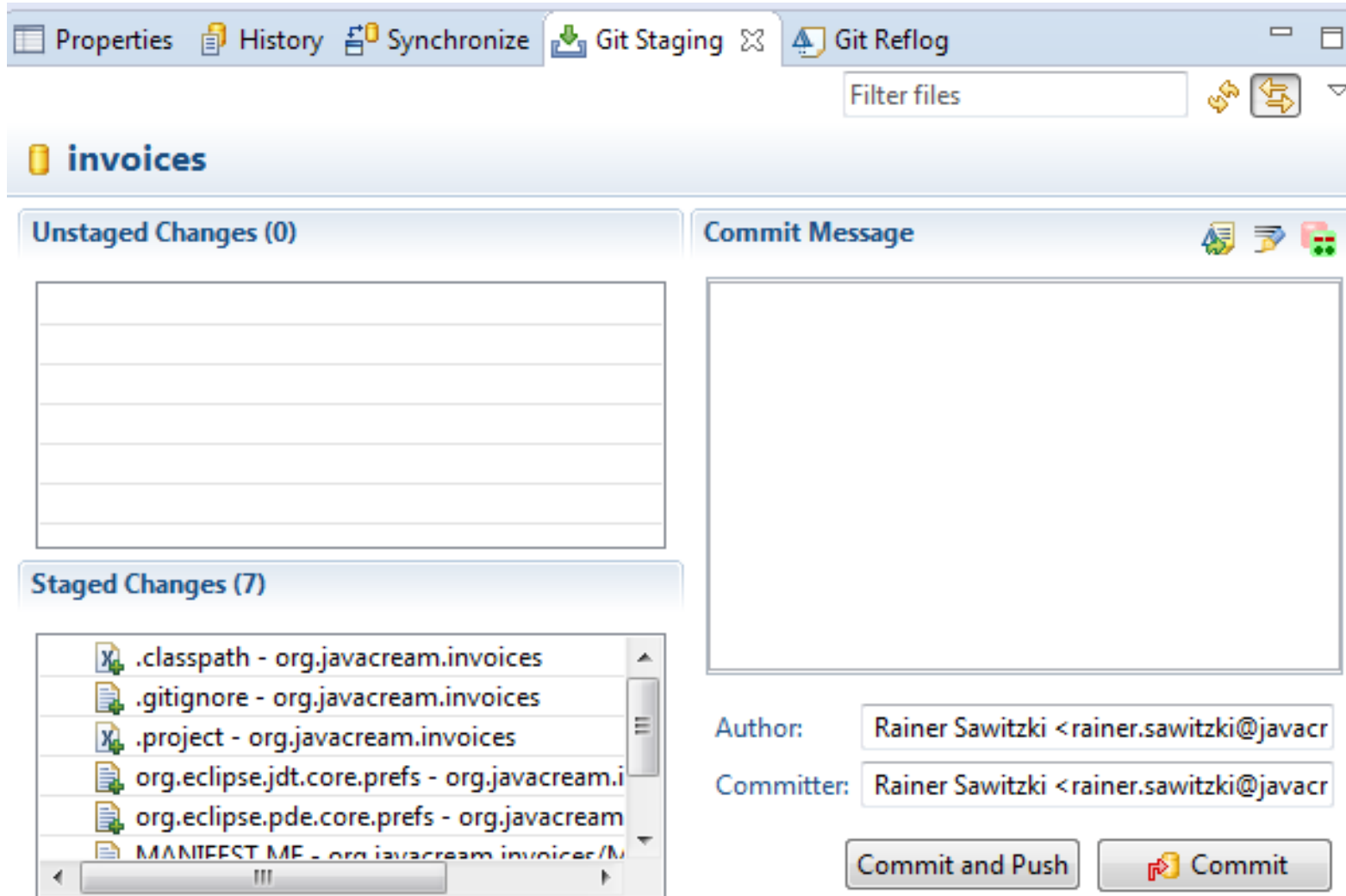
[illegible]



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



# Eclipse: Staging



The screenshot shows the Eclipse IDE's Git Staging view. At the top, there's a toolbar with buttons for Properties, History, Synchronize, Git Staging (active), and Git Reflog. Below the toolbar is a search bar labeled "Filter files". The main area is divided into two panes. The left pane, titled "invoices", contains two sections: "Unstaged Changes (0)" which is empty, and "Staged Changes (7)" which lists seven files: .classpath - org.javacream.invoices, .gitignore - org.javacream.invoices, .project - org.javacream.invoices, org.eclipse.jdt.core.prefs - org.javacream.i, org.eclipse.pde.core.prefs - org.javacream, and MANIFEST.MF - org.javacream.invoices/. The right pane, titled "Commit Message", contains a large text area for writing the commit message. Below the text area, there are fields for "Author:" and "Committer:", both containing the text "Rainer Sawitzki <rainer.sawitzki@javacr". At the bottom right, there are two buttons: "Commit and Push" and "Commit".

Properties History Synchronize Git Staging Git Reflog

Filter files

**invoices**

Unstaged Changes (0)

Staged Changes (7)

- .classpath - org.javacream.invoices
- .gitignore - org.javacream.invoices
- .project - org.javacream.invoices
- org.eclipse.jdt.core.prefs - org.javacream.i
- org.eclipse.pde.core.prefs - org.javacream
- MANIFEST.MF - org.javacream.invoices/

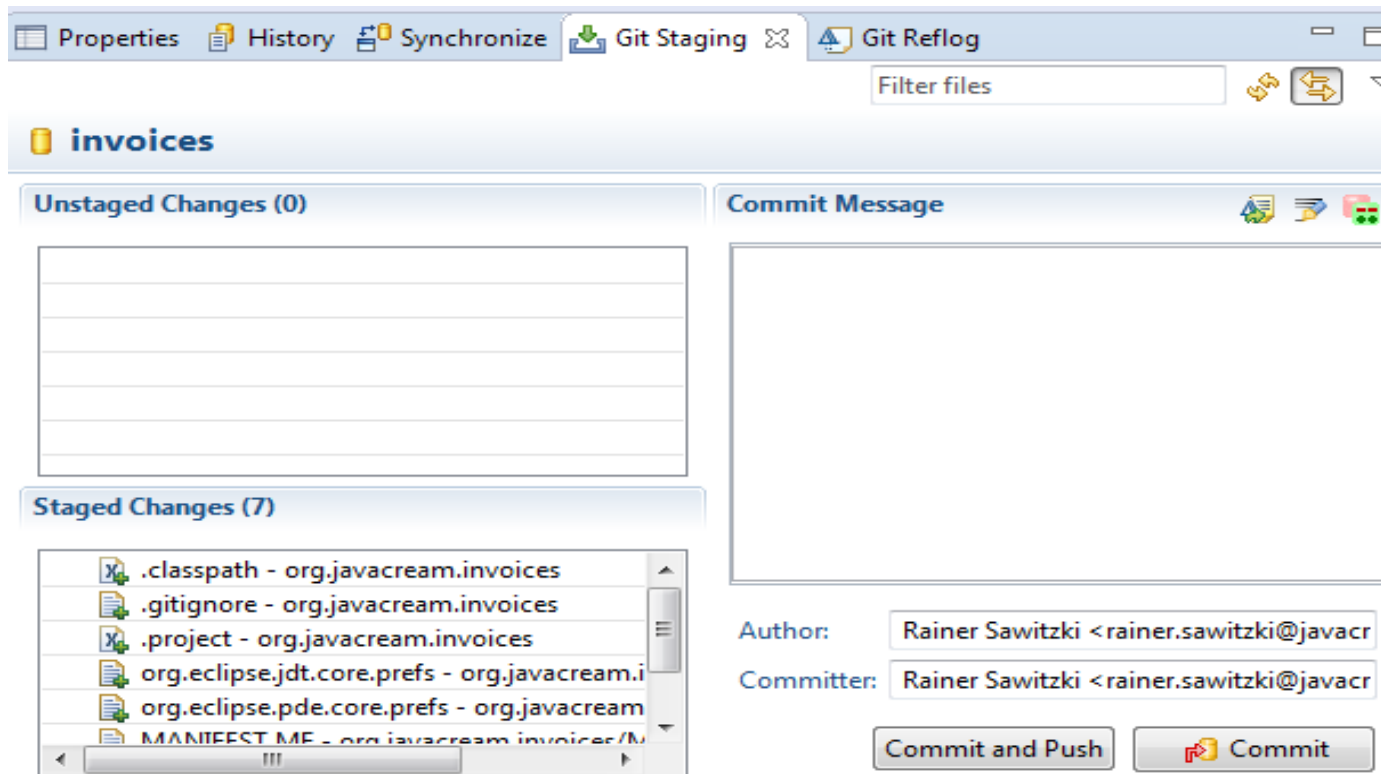
Commit Message

Author: Rainer Sawitzki <rainer.sawitzki@javacr

Committer: Rainer Sawitzki <rainer.sawitzki@javacr

Commit and Push Commit

# Eclipse: add und commit



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



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
```