



WERDEN SIE GIT EXPERTE!

René Preißel (eToSquare)

Nils Hartmann (Techniker Krankenkasse)

VORSTELLUNG



René Preißel | Freiberuflicher Softwarearchitekt, Entwickler und Trainer

Co-Autor des Buchs „Git: Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows“

Kontakt: rene.preissel@etosquare.de



Nils Hartmann | Java-Softwareentwickler, Techniker Krankenkasse

Schwerpunkte: OSGi, Eclipse und Build-Management, Co-Autor des Buches „Die OSGi Service Platform“

Kontakt: nils@nilshartmann.net



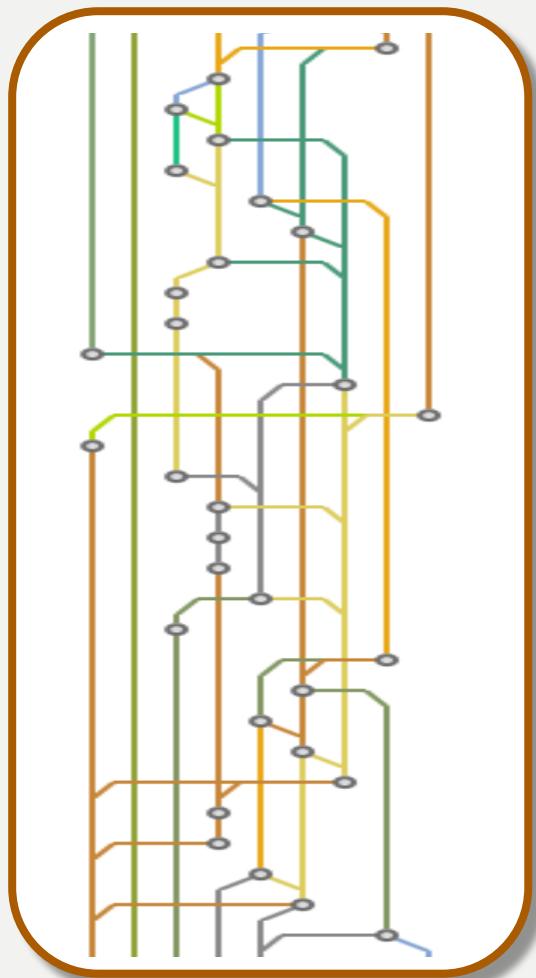
GIT – ZUM NACHLESEN



René Preisel, Bjørn Stachmann
Dezentrale Versionsverwaltung im Team
Grundlagen und Workflows
2. Auflage, dpunkt Verlag, 2013



AGENDA



- **(Vorbereitung)**
 - Start und Verwendung der VM
- **Git Internas**
 - Objekt-Datenbank, Branches, Remotes
- **Branch-Strategien**
 - Feature-Branches, Rebasing, Git Flow
- **Arbeiten mit großen Projekten**
 - submodules und subtrees
- **Git APIs: Alternativen zur Git Bash**
- **Verwendung von Build Tools**
 - maven und gradle

Fragen & Diskussionen: jederzeit!



VORBEREITUNG

DIE WORKSHOP-VM



Installation

Inhalt

INSTALLATION DER VM

1. Inhalt des Sticks auf Festplatte kopieren

2. Virtual Box installieren

Verzeichnis: **VirtualBoxInstall**

3. Virtual Box starten

4. VM-Image importieren

Maschine -> Hinzufügen...

Image: VMImage/wjax_gitworkshop_image.vbox

5. VM starten



VERWENDUNG DER VM



Username: **vagrant**

Passwort: **vagrant**

Repositories: **/home/vagrant/repos**

Sicherung: **/home/vagrant/_repos-backup.tar.gz**



VORBEREITUNG

DIE WORKSHOP-VM



Installation

Inhalt

WORKSHOP VM – GIT + GIT-TOOLS

Git 2.1

- <http://git-scm.com/>

Git cola 2.0.7 (Alternative GUI für Git)

- <http://git-cola.github.io/>

Git Flow (Bash Erweiterungen für Git Flow Workflow)

- <https://github.com/nvie/gitflow>

Meld – Mergetool für Git

- <http://meldmerge.org/>



WORKSHOP VM – ENTWICKLUNGSTOOLS

Eclipse 4.4SR1 mit EGit und JGit

- <http://www.eclipse.org/downloads/>

Atom Editor

- <https://atom.io/>

Maven 3.0.4

- <http://maven.apache.org/>

Gradle 2.0

- <http://www.gradle.org/>



TEIL 1

GIT INTERN

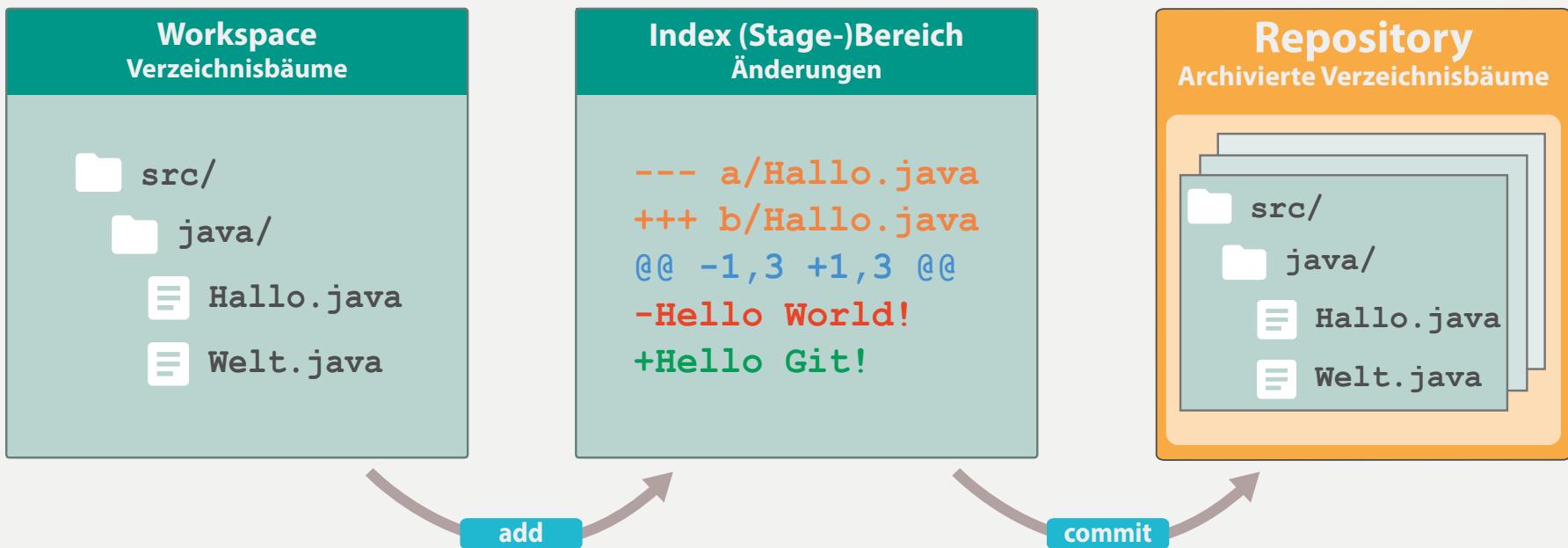


Objektdatenbanken und Referenzen

Merges

Remotes

GIT BESTANDTEILE



* aus „Git - Grundlagen und Workflows“



SCHNELLEINSTIEG

git init

Repository anlegen

git add <file>

Dateien zum nächsten Commit hinzufügen

git commit -m <message> / git gui

Commit durchführen

git rm <file>

Dateien im nächsten Commit als gelöscht markieren

git status

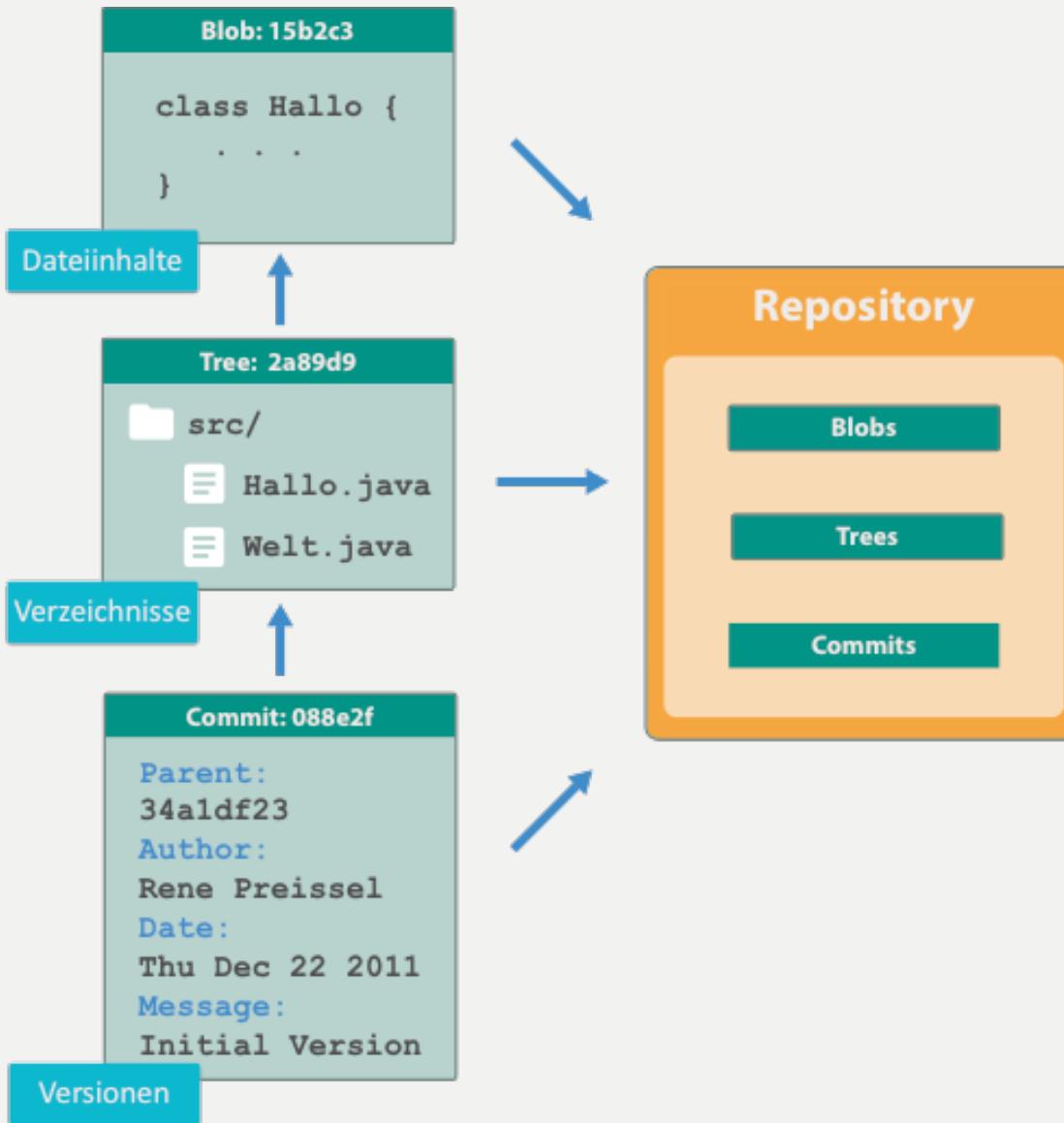
Aktuellen Zustand des Workspaces ansehen

git log --oneline --follow [-M<x>%]

Historie inklusive Umbenennungen ansehen



DAS REPOSITORY - EINE OBJEKTDATENBANK



Effizienter Objektspeicher

- Für alle Inhalte werden **Hash-Werte** als Schlüssel berechnet (SHA, 160 Bit)
- Trennung von Dateinhalt und Dateiname
- Alle Inhalte werden nur einmal gespeichert

Objekte

- **Blobs** - Dateinhalt
- **Trees** - Verzeichnisse mit Verweisen auf Inhalte
- **Commits** - Versionen von hierarchischen Verzeichnisstrukturen



OBJEKTDATENBANK – LOW LEVEL OPERATIONEN

git cat-file -p <sha>

Anzeige von beliebigen Objekten (blob, tree, commit, tag)

git ls-tree -r -t <tree-ish>

Anzeige eines Trees

git commit-tree -p <parent> -m <message> <tree>

Neues Commit zusammenstellen

git hash-object / git hash-object -w

Objekt Id erzeugen / Objekt speichern

git fsck --unreachable --no-reflogs

Garbage finden

git gc [--prune=all]

Garbage aufräumen



GEHEIMNISSE DES INDEX (STAGE, CACHE)

Stage, Cache

Alternative Namen für den Index

git ls-files --stage

Anzeige aller Dateien im Index

git add <file> / git update-index <file>

Datei/Änderungen zum Index hinzufügen

git read-tree --empty

Index leeren

git read-tree -i --prefix <path> / <tree-ish>

Index mit Tree füllen / vereinigen

git write-tree

Tree schreiben

git update-index --[no-]assume-unchanged -- <file>

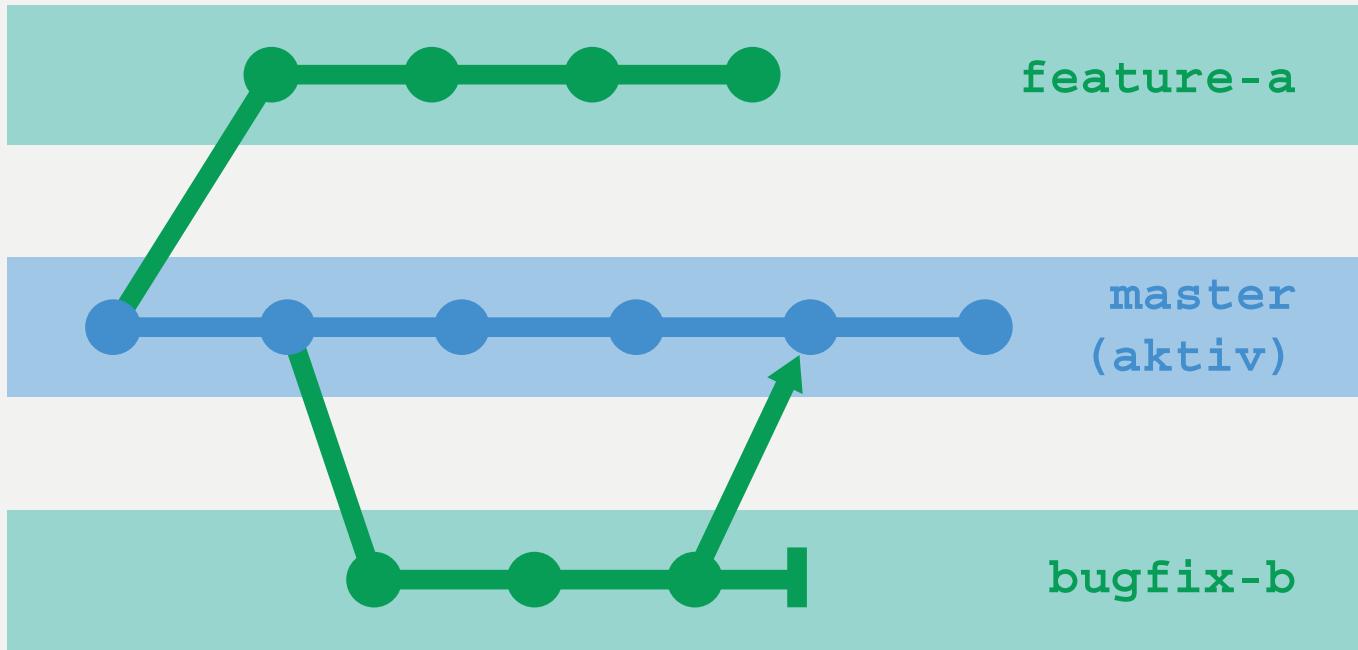
„Getrackte“ Dateien ignorieren

git ls-files -v

Ignorierte Dateien anzeigen



VERZWEIGUNGEN - BRANCHES



(aus „Git - Grundlagen und Workflows“)

- Branches können von jedem Entwickler lokal angelegt werden.
- Ein Branch ist im Workspace immer aktiv (Default: „master“).
- Ein Branch ist nichts weiter als der Zeiger (**Referenz**) auf ein Commit.
- Bei jedem neuen Commit wird der aktive Branch auf das neue Commit gesetzt.



BRANCHES & TAGS: ÜBERBLICK

git branch -v

Branch anzeigen

git branch <name> [<start-commit-oder-ref>]

Branch anlegen

git checkout <branch>

Branch wechseln

git branch -d <branch> / git branch -D <branch>

Branch löschen

git tag

Tag anzeigen

git tag -a <name> <commit-oder-ref>

Tag anlegen



REFERENZEN

git show-ref --head

Alle Refs anzeigen (siehe auch .git/refs)

git show-ref --tags --dereference

Annotated Tags dereferenzieren

git update-ref <full-ref> <sha>

Ref explizit aktualisieren

git reset --hard <sha-oder-ref> #Index + Workspace

git reset <sha-oder-ref> #Index

git reset --soft <sha-oder-ref> #Nur Ref

Ref [+Index +Workspace] des aktuellen Branches setzen

git reflog / git reflog <ref>

Ref-Änderungen nachvollziehen





ÜBUNG: OBJEKTDATENBANK

Repository: interna/einstieg

- Aufgabe 1:
Untersuchen Sie die Trees der beiden ersten Commits ([9a6686](#) und [a2e182](#)). Wie viele neue Blobs und Trees mussten für das zweite Commit angelegt werden?
- Aufgabe 2:
Ab wieviel Prozent erkennt Git das Unbenennen der Datei von `src/en>Hello.groovy` nach `src/en>HelloWorld.groovy`?
- Aufgabe 3:
Erzeugen Sie ein neues Commit auf Basis des Commits [f495c5](#), welches nur die zwei Dateien (`src/en>Hello.groovy` und `src/en>HelloWorld.groovy`) als Root-Tree beinhaltet und keinen Parent hat.
- Aufgabe 4:
Überprüfen Sie ob das angelegte Commit aus Aufgabe 3 als Garbage erkannt wird.
- Aufgabe 5 (optional):
Erzeugen Sie ein neues Tree-Objekt, welches nur die Dateien aus Aufgabe 3 im Verzeichnis „en/src“ beinhaltet.
Erzeugen Sie davon ausgehend ein neues Commit ohne Parent.
Verschieben Sie den Master-Branch auf dieses Commit.
Setzen Sie den Master-Branch wieder auf das vorherige Commit.



TEIL 1

GIT INTERN



Objektdatenbanken und Referenzen

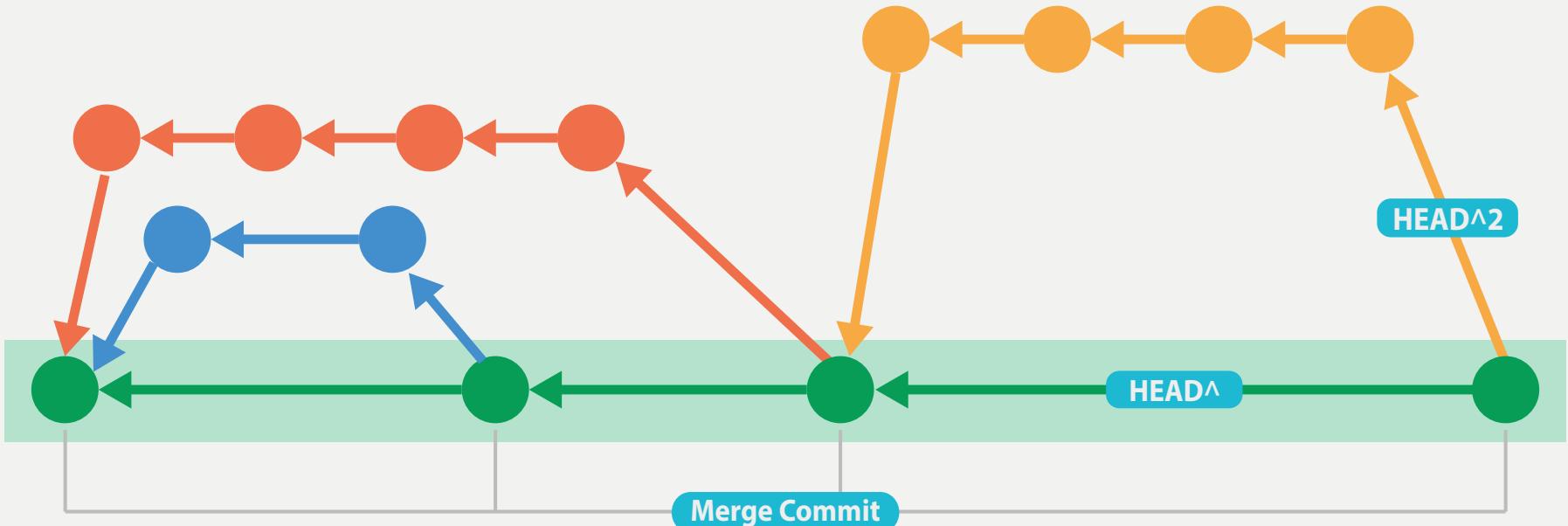
Merges

Remotes

GIT INTERN: MERGES 1

`git merge <branch>`

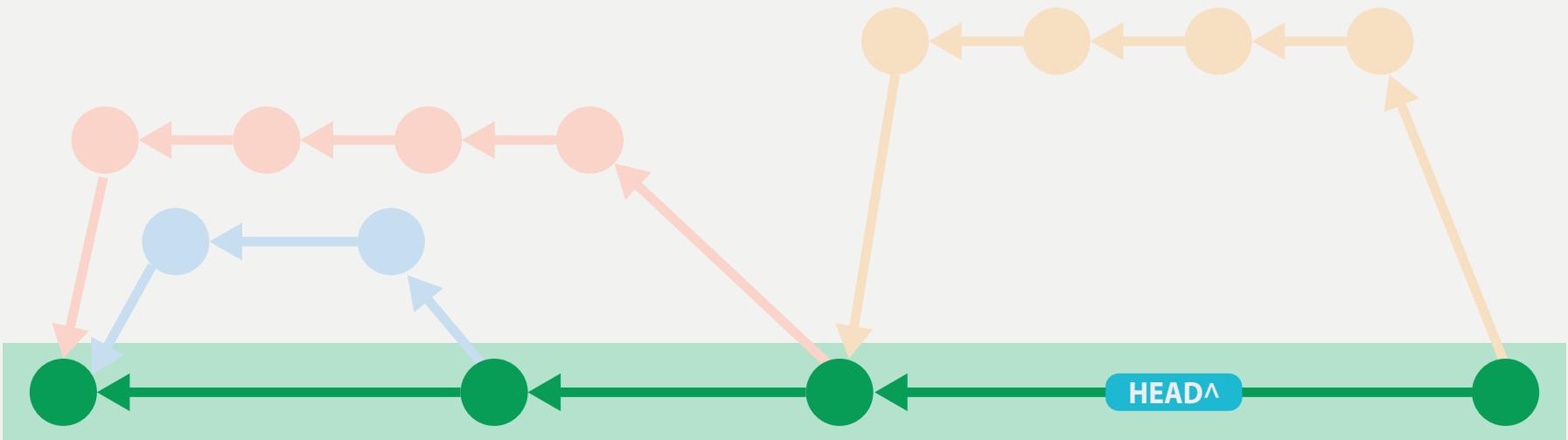
Merge-Commit: zwei oder mehr Parents



GIT INTERN: MERGES 2

`git log --first-parent [--oneline]`

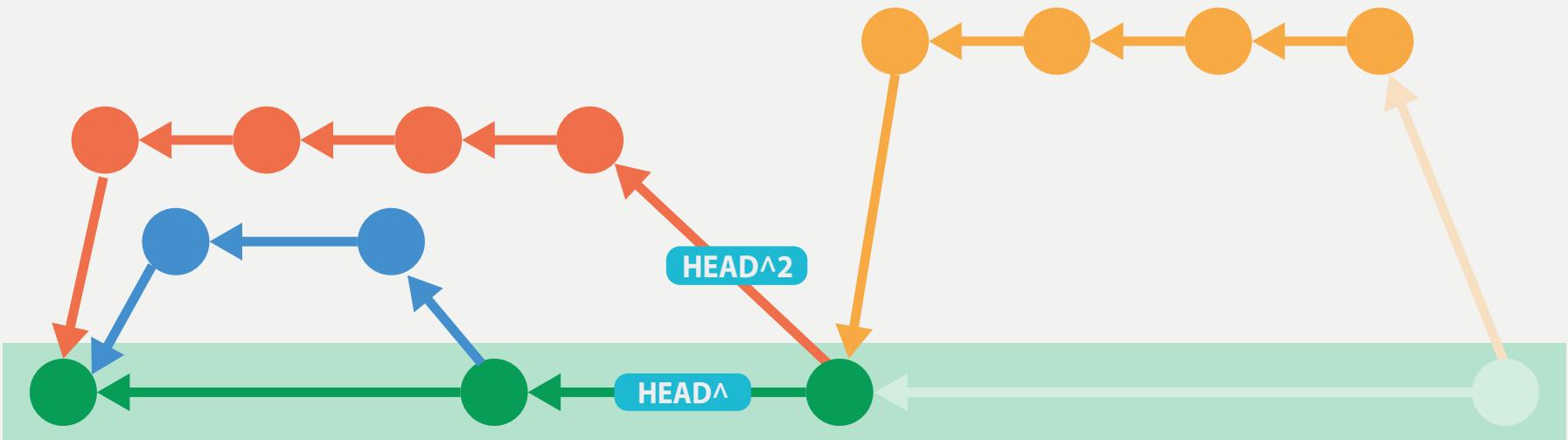
- First-Parent-History



GIT INTERN: MERGES 3

`git reset --hard HEAD^`

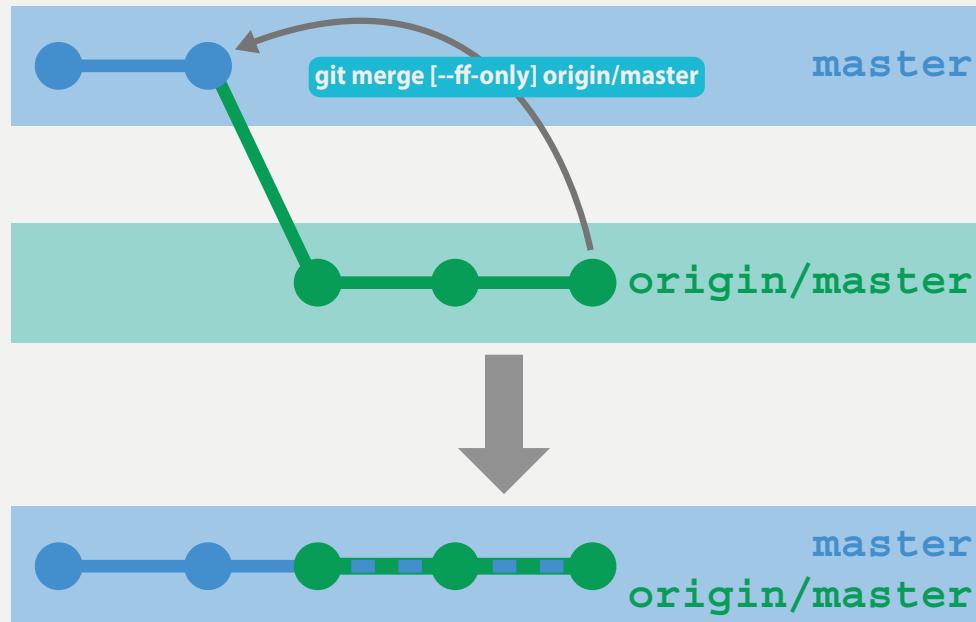
- Kann verwendet werden, um einen Merge zu verwerfen



GIT INTERN: FASTFORWARD-MERGE

Nur **ein** Branch hat sich verändert

- Kein (Merge-)Commit
- Voraussetzung für **push**



git merge --ff-only <branch>

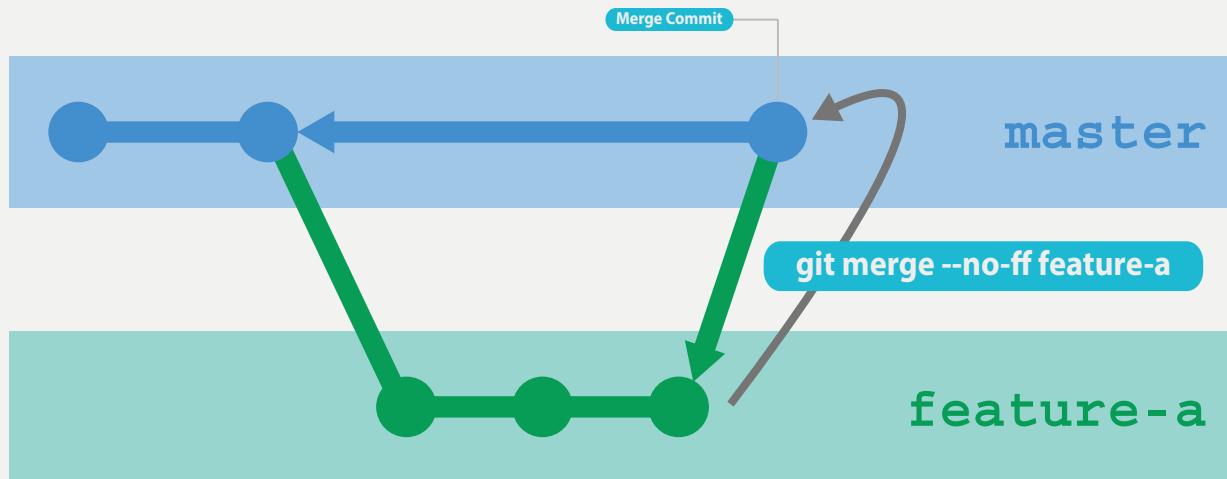
- Erzwingt FF-Merge
- Schlägt fehl, wenn nicht möglich



GIT INTERN: NO-FASTFORWARD

`git merge --no-ff <branch>`

- Erzwingt Merge-Commit



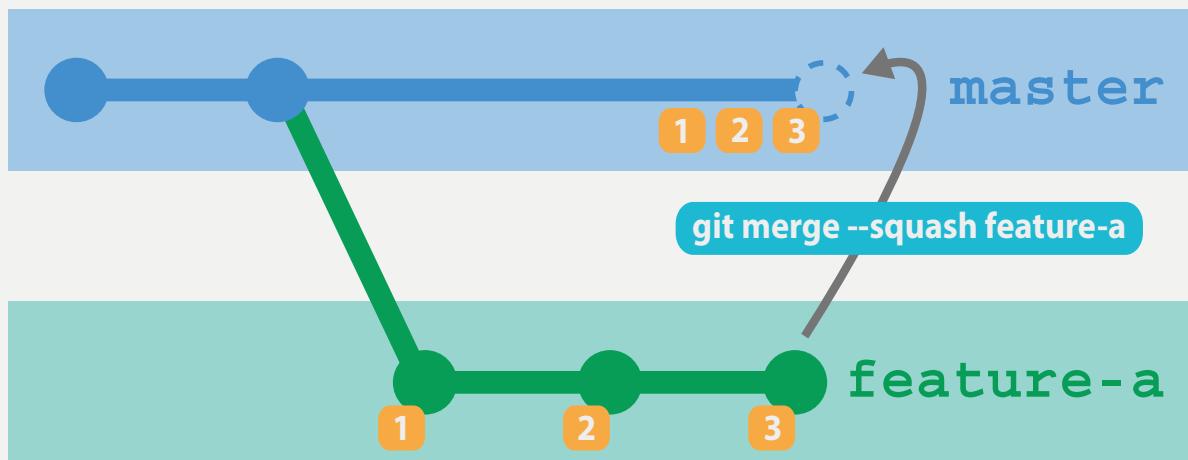
- Anwendungsfälle:
 - Dokumentation
 - First-Parent-History erzwingen



GIT INTERN: SQUASH

`git merge --squash <commit>`

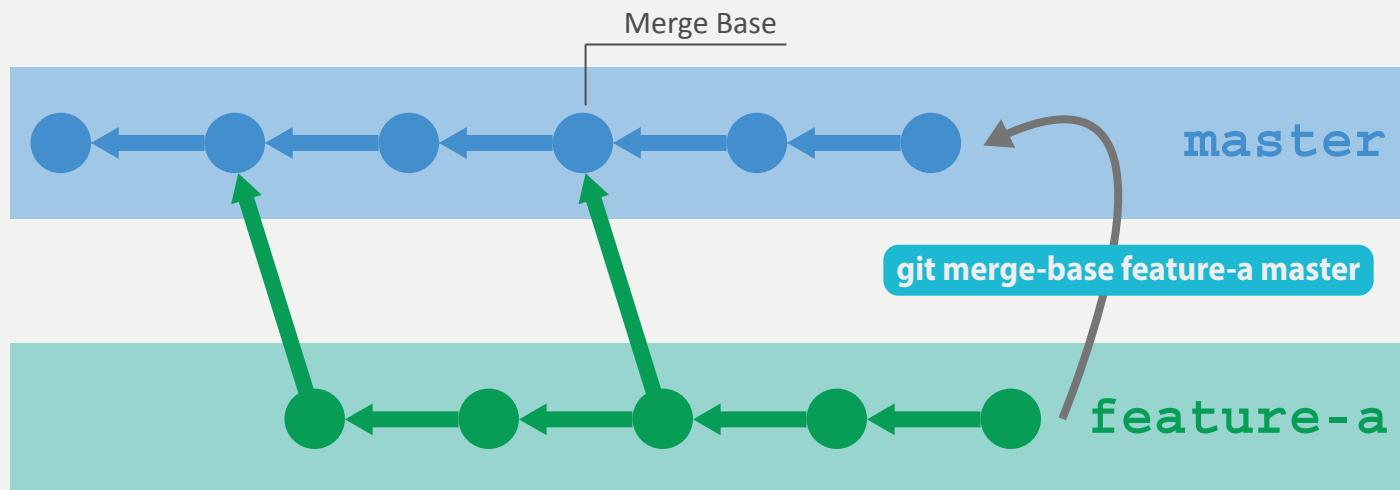
- Führt alle Änderungen des Branches im Workspace zusammen
- Aktualisiert den Index
- Führt noch keinen Commit durch
- Es entsteht **kein** Merge-Commit



GIT INTERN: MERGE-BASE

`git merge-base <commit> <commit>`

Ermittelt den letzten gemeinsamen Commit, an dem zwei Branches auseinanderlaufen



GIT INTERN: MERGE-FILE

`git merge-file <current> <base> <other>`

- Mergt alle Änderungen, die zwischen zwei Dateien (`base` und `other`) entstanden sind mit dem Stand einer dritten Datei (`current`)
- Wird von Git intern verwendet, wenn Dateien von verschiedenen Branches gemergt werden



GIT INTERN: MERGE-KONFLIKTE

- Änderungen an gleichem **Bereich** führen zu **Konflikten**
- Der Index hält mehrere Versionen der konfliktbehafteten Dateien

```
[master|MERGING] $ git ls-files --stage
```

100644 a25e03fffe0a11b42f865ea00f3c0570fa9a7292	0	install.txt
100644 827b968037791e8722959d13dac9d9a38b5bd3ca	1	readme.txt
100644 8c83f62c9b6bb1209e3142f9f227ffb0af8c07b2	2	readme.txt
100644 198618049f1593abbf46cd8a0fc577a77e02ae06	3	readme.txt
100644 f44f8c6b9a98205af69015113a3b7fdda0962e5b	0	version.txt

Stage
1 = Base
2 = Ours
3 = Theirs
(0 = „normal“)



GIT INTERN: MERGE KONFLIKTE - CHECKOUT

Bei Konflikten

git checkout --ours <pfad>

Version vom Ziel-Branch holen („Stage 2“)

git checkout --theirs <pfad>

Version vom Source-Branch holen („Stage 3“)

git checkout <commit> -- <pfad>

Beliebige Version einer Datei auschecken, wird sofort in den Index aufgenommen

(Auch außerhalb von Merge nutzbar)

git checkout --merge <pfad>

Merge zurücksetzen

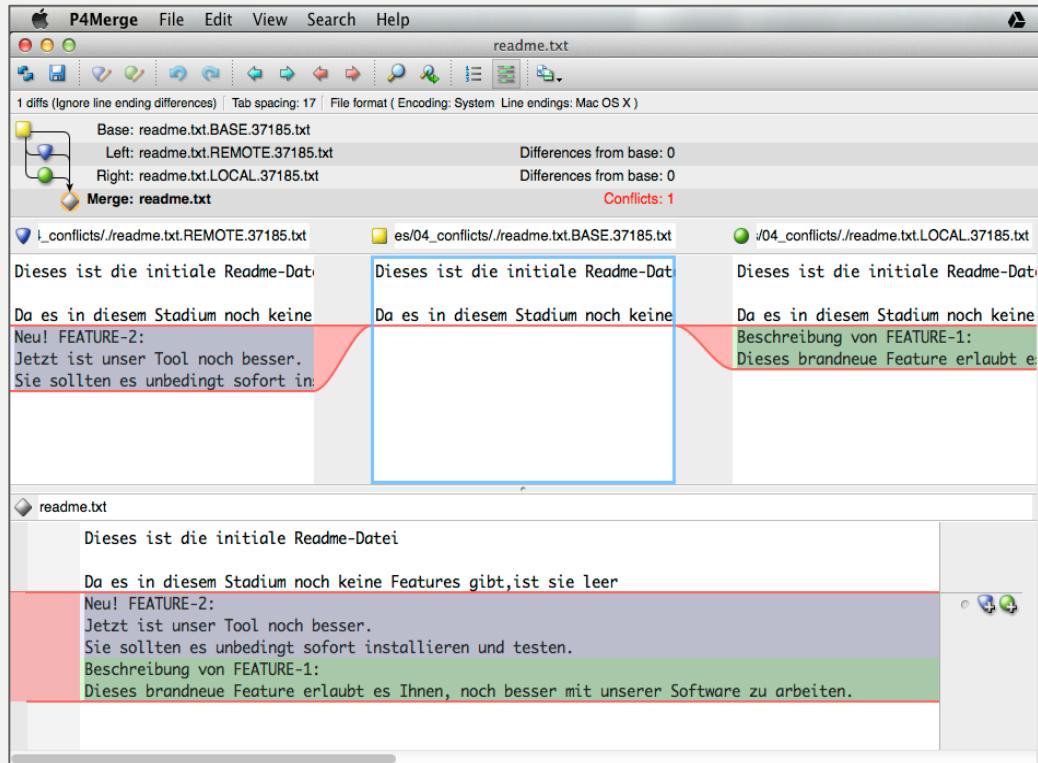
git checkout --conflict=<style> <pfad>

Merge zurücksetzen, Konfliktmarker wählbar



GIT INTERNAS: MERGE-TOOL

git mergetool [-t <mergetool>]



GIT INTERN: MERGES - STRATEGIEN

git merge -s ours <branch>

Merge-Strategie **ours**

Verwirft alle Änderungen von Branch **branch**

git merge -X ours <branch>

Option für Merge-Strategie **recursive**

Verwirft einkommende konfliktbehaftete Dateien

git merge -X theirs <branch>

Option für Merge-Strategie **recursive**

Konfliktbehaftete Dateien immer von Branch **branch** nehmen



GIT INTERN: MERGES - KONFLIKTE

```
git config --global merge.conflictstyle diff3
```

```
[master|MERGING] $ cat readme.txt
```

```
<<<<< ours
```

Dieses ist die erstmals angepasste Readme-Datei

Beschreibung von feature-1:

Dieses brandneue Feature erlaubt es Ihnen, noch besser...

```
|<<<<< base
```

Dieses ist die initiale Readme-Datei

Da es in diesem Stadium noch keine Features gibt, ist sie leer

```
=====
```

Dieses ist die Readme-Datei

Sie beschreibt alle Features Ihres gekauften Produktes

Neu! feature-2:

Jetzt ist unser Tool noch besser.

Sie sollten es unbedingt sofort installieren und testen.

```
>>>>> theirs
```

Basis-Version



GIT MERGES: RERERE

git rerere „Reuse recorded resolution“

- Konfliktlösungen abspeichern
- Problem: Lösungen werden nur lokal gespeichert
- Einschalten global oder pro Repository:
`git config [--global] rerere.enabled true`
- Nochmaliges ausführen
`git rerere forget`



TEIL 1

GIT INTERN



Objektdatenbanken und Referenzen

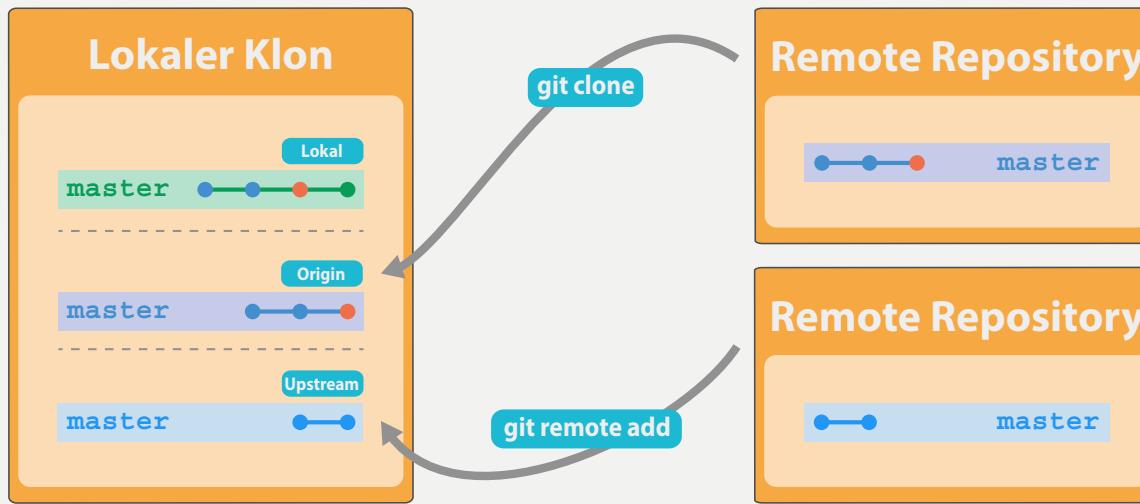
Merges

Remotes

GIT INTERN: REMOTES (I)

Verbinden lokale mit Remote-Repositories

- Durch `git clone` wird „origin“-Remote angelegt
- Mit `git remote add` können weitere Remotes hinzugefügt werden

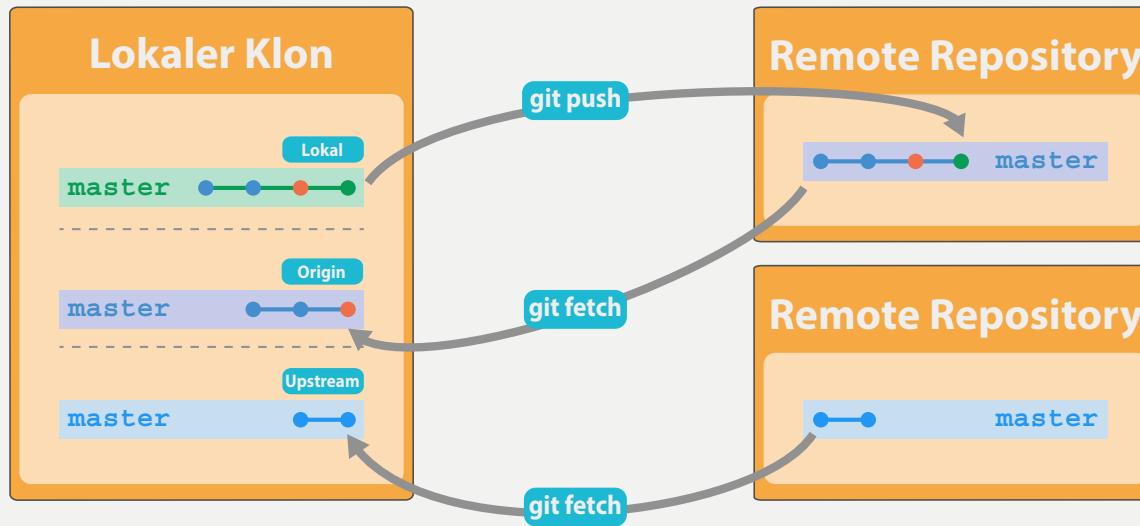


```
[remote "origin"]
  url = https://bitbucket.org/nilshartmann/wjax2014_workshop.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[remote "upstream"]
  url = https://bitbucket.org/rpreissel/wjax2014_workshop.git
  fetch = +refs/heads/*:refs/remotes/upstream/*
```



GIT INTERN: REMOTES (II)

Austausch von Objekten über **fetch** und **push**



```
[remote "origin"]
  url = https://bitbucket.org/nilshartmann/wjax2014_workshop.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[remote "upstream"]
  url = https://bitbucket.org/rpreissel/wjax2014_workshop.git
  fetch = +refs/heads/*:refs/remotes/upstream/*
```



GIT INTERN: REMOTES - KOMMANDOS

git remote add <name> <url>

Fügt ein neues Remote-Repository `name` mit der `url` hinzu

git remote -v

Zeigt alle Remote-Repositories an

git ls-remote <name>

Zeigt alle Referenzen an, die es in einem Remote-Repository gibt

git remote update <name>

Aktualisiert Referenzen aus einem Remote-Repository (entspricht `git fetch`)

git remote rm <name>

Löscht das angegebene Remote-Repository



GIT INTERN: REFERENZEN

Lokaler Klon

Lokal

refs/heads/master
refs/heads/feature/newUi
refs/tags/v4.0

Origin

refs/remotes/origin/master
refs/remotes/origin/feature/newUi
refs/remotes/origin/bugfix

Upstream

refs/remotes/upstream/master
refs/remotes/upstream/feature/java7
refs/remotes/upstream/notes/d3caa2



GIT INTERN: DIE REFSPEC (I)

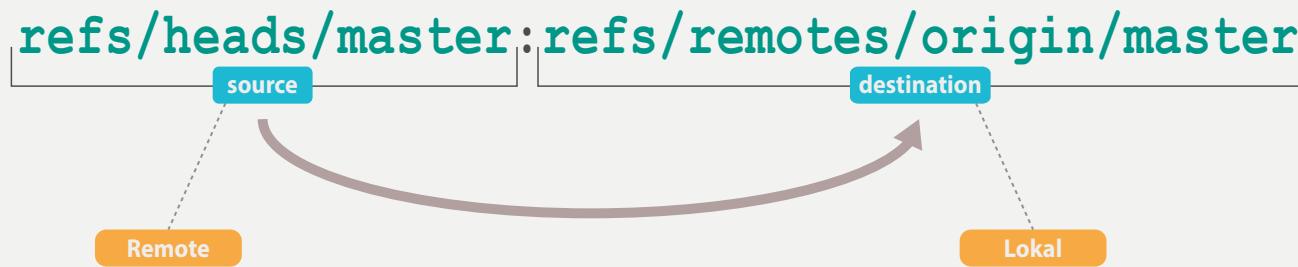
- Definiert, welche Referenzen ausgetauscht werden
 - Spezifiziert in Remote-Konfiguration oder auf der Kommandozeile
 - Mehrere Ref-Specs sind möglich



GIT INTERN: DIE REFSPEC (II)

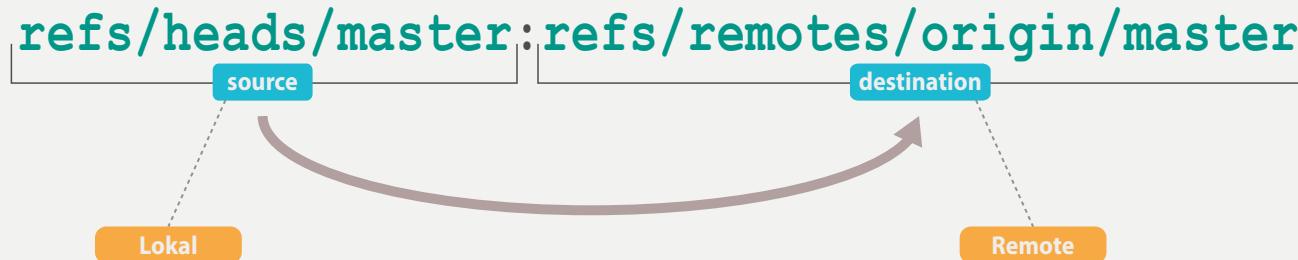
Fetch

Referenzen aus **Remote**-Repository (**source**) ins **lokale** Repository (**destination**) kopieren



Push

Referenzen aus lokalem Repository (**source**) ins Remote-Repository (**destination**) kopieren



GIT INTERN: DIE REFSPEC (III)

- * Platzhalter für (Branch-)Namen
 - Kann nicht für Substrings eingesetzt werden



GIT INTERN: DIE REFSPEC (IV)

- + Aktualisieren auch dann, wenn kein fast-forward möglich
- + Push: Führt einen ‚force-push‘ durch (`push -f`)



GIT INTERN: REFSPEC-KONFIGURATION

Verwendung in .git/config

```
[remote "origin"]
  url = https://bitbucket.org/nilshartmann/wjax2014_git.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/tags/*:refs/tags/*
# Nicht empfohlen, nur als Beispiel:
  push = refs/heads/*:refs/qa/*
```



GIT INTERN: REFSPEC-BEISPIELE

git fetch origin refs/pull/*/head:remotes/origin/pr/*

GitHub-Pullrequests nach origin/pr/Nummer fetchen

git fetch <name> refs/tags/*:refs/tags/*

Alle Tags fetchen, entspricht git fetch --tags

git fetch <name> refs/notes/*:refs/notes/*

Git Notes fetchen

git push <name> HEAD:refs/for/feature-1

Aktuellen Branch zum Review in Gerrit pushen



GIT INTERN: FETCH

git fetch

Holt Objekte aus allen konfigurierten Remote-Repositories. Ergebnis wird in `.git/FETCH_HEAD` gespeichert

git fetch <name>

Holt Objekte aus dem Remote-Repository name

git fetch <name> <refspec> <refspec>

Holt die auf die Refspecs passenden Objekte

git fetch --prune

Löscht Tracking-Banches, zu denen es keine Branches mehr im Remote-Repository gibt.

Konfiguration global möglich: `git config --global fetch.prune true`

git fetch --tags

Überträgt alle Tags aus dem Remote-Repository nach refs/tags



GIT INTERN: PULL

Objekte holen und neue Commits mergen

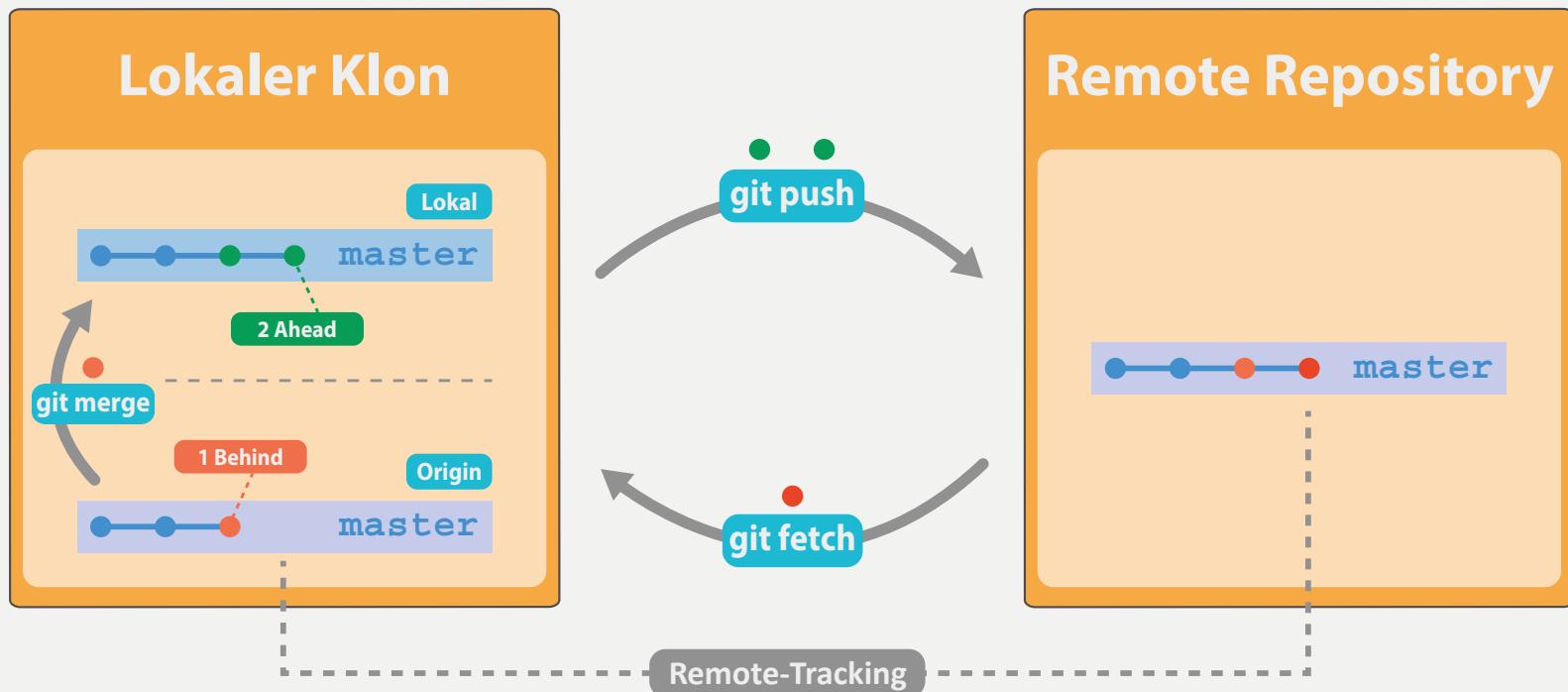
```
git fetch  
+ git merge FETCH_HEAD  
= git pull
```



GIT INTERN: TRACKING BRANCHES

Lokale Kopie der Remote-Banches

- `git branch -vv` zeigt Tracking Status
- `git branch -vva` zeigt alle Branches sowie Tracking Status



GIT INTERN: TRACKING BRANCH VERBINDEN

git checkout features/f4

Default-Verhalten: Wenn es einen Tracking-Branch gleichen Namens gibt, wird dieser automatisch verbunden

git checkout -b f4 --track origin/features/f4

Lokalen Branch „f4“ erstellen und mit Branch „features/f4“ im Remote „origin“ verbinden. Der Remote-Branch muss bereits vorhanden sein

git branch --set-upstream-to origin/features/f4

Aktiven Branch mit „features/f4“ im Remote-Repository origin verbinden. Der Remote-Branch muss bereits vorhanden sein

git push --set-upstream origin features/f4

Pusht den lokalen Branch auf den Branch „features/f4“ und speichert die Verbindung



GIT INTERN: PUSH

Objekte in ein Remote-Repository übertragen

- **git push origin**

Wenn Refspec in `remote.origin.push` gesetzt ist, gemäß dieser Refspec pushen (ggf. mehrere Branches!).
Ansonsten Strategie verwenden, die in `push.default` konfiguriert ist

- **git push origin master**

- **git push origin master:qa**

Überträgt Commits vom lokalen `master`-Branch auf einen Branch im Remote-Repository `origin`.

- **git push origin refs/heads/*:refs/heads/***

Überträgt Commits von allen lokalen Branches in das Remote-Repository `origin`.

- **git push -f origin master:qa**

Überträgt Commits auch dann, wenn im Remote-Repository kein fast-forward möglich ist („force push“). Im Remote-Repository können Commits verloren gehen! In anderen Klonen kommt es zu Merge-Konflikten!

- **git push --tags**

Überträgt die Tags aus dem lokalen Repository in das Remote-Repository.

- **git push origin :master**

Löscht den Branch „master“ im Remote-Repository („Übertrage *nichts* nach master“)



GIT INTERN: PUSH.DEFAULT

Mögliche Werte:

- **simple**: überträgt aktuellen Branch, wenn sein Name mit dem des Upstream-Branchs übereinstimmt (**Default seit Git 2.0**)
- **matching**: Überträgt *alle* lokalen Branches, zu denen es einen gleichnamigen Remote-Branch gibt (Default vor Git 2.0)
- **current**: überträgt den aktuellen Branch in ein Branch mit gleichem Namen im Remote-Repository (unabhängig davon, ob der Remote-Branch existiert und unabhängig vom Upstream-Branch)
- **upstream**: Pusht aktuellen Branch zum Upstream-Branch. Nur im „zentralen“ Workflow sinnvoll.
- **nothing**: Kein Push. Kann verhindert werden, um „versehentliche“ Pushes zu verhindern



GIT INTERN: PUSH EMPFEHLUNG

- Keine refspec in `remote.*.push` konfigurieren
- Lokaler Branchname sollte Remote-Branchnamen entsprechen
- `push.default` auf `simple` setzen
`git config --global push.default simple`
- Im Zweifelsfall beim Push genau angeben, was gepusht werden soll
`git push origin src:dest`



GIT... /-:

git checkout origin/branchname

git pull origin branchname

git push origin :branchname





ÜBUNG: REMOTE-REPOSITORIES

Arbeitsverzeichnis: remotes/uebung

Hinweis zum Arbeiten mit den „Remote-Repositories“ in der Übung: Sie können relative Pfad-URLs verwenden, bspw. git <...> a.git oder git <...> ../b.git.

1. Klonen Sie das Repository mein-spring.git
 - Welche Branches gibt es dort?
2. Führen Sie dort auf dem Branch „feature-1“ einen Commit durch.
 - Erzeugen Sie einen Tag für Ihren Commit
 - Pushen Sie Ihren Commit sowie den Tag zurück in das Remote-Repository
3. Fügen Sie das Repository spring.git als weiteres Remote hinzu
 - Welche Branches gibt es dort?
4. Übertragen Sie Ihren Branch „feature-1“ unter der Referenz „refs/for/review/feature-1“ in das Repository spring.git
5. Übertragen Sie Ihre „feature-1“-Änderungen auch auf den „feature-1“-Branch in das Remote-Repository spring.git.



TEIL 2

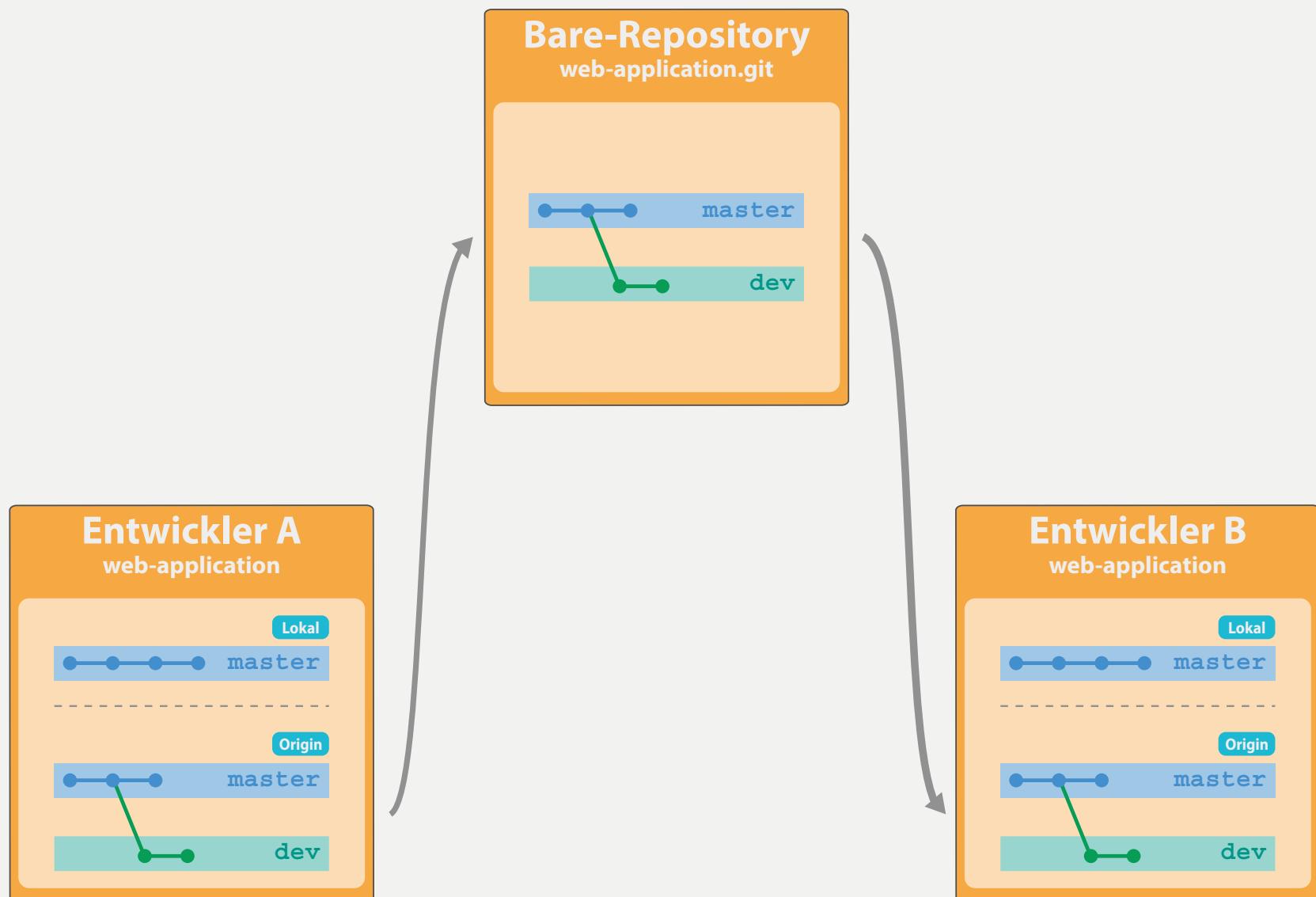
BRANCH MODELLE



Workflows in der Entwicklung

Releaseprozesse

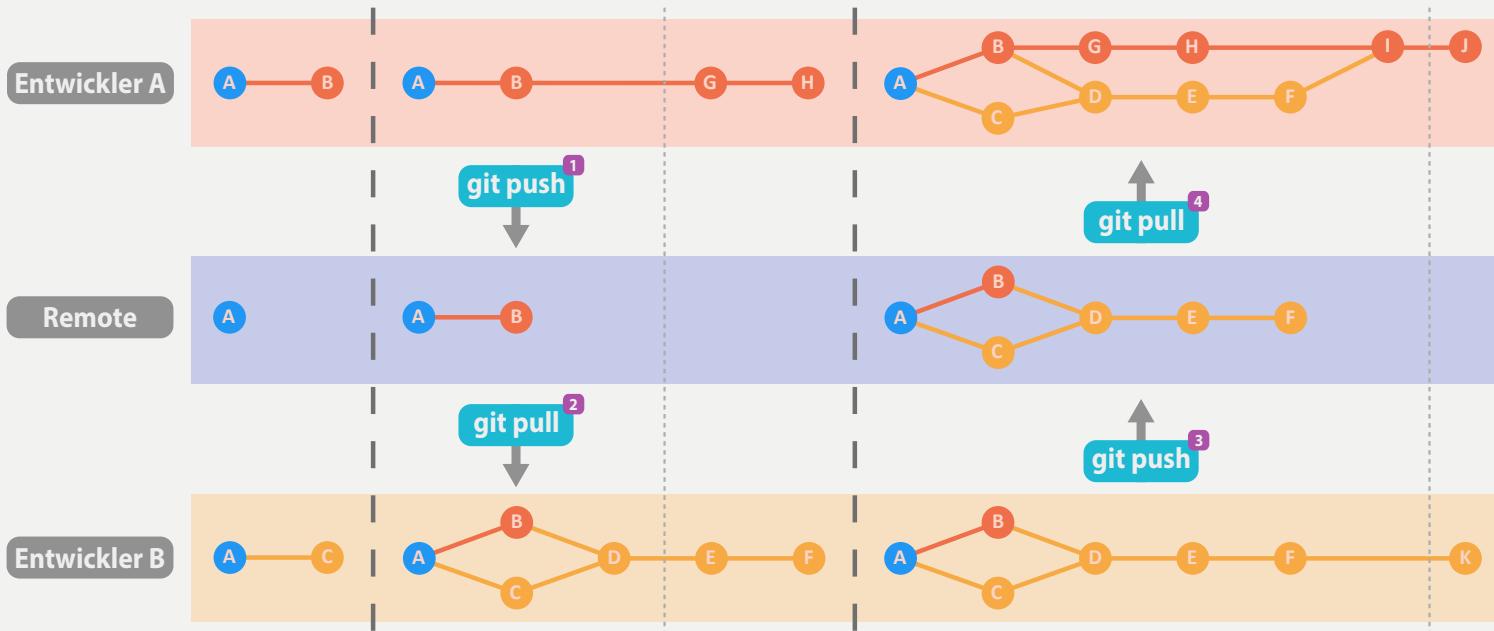
TYPISCHE WORKFLOWS IN DER ENTWICKLUNG



ARBEITEN AUF GEMEINSAMEN BRANCH (MERGE)

Alle Änderungen für alle Tasks werden auf dem **master**-Branch durchgeführt

- orientiert sich stark an zentralen Versionsverwaltungen
- einfache Umsetzung

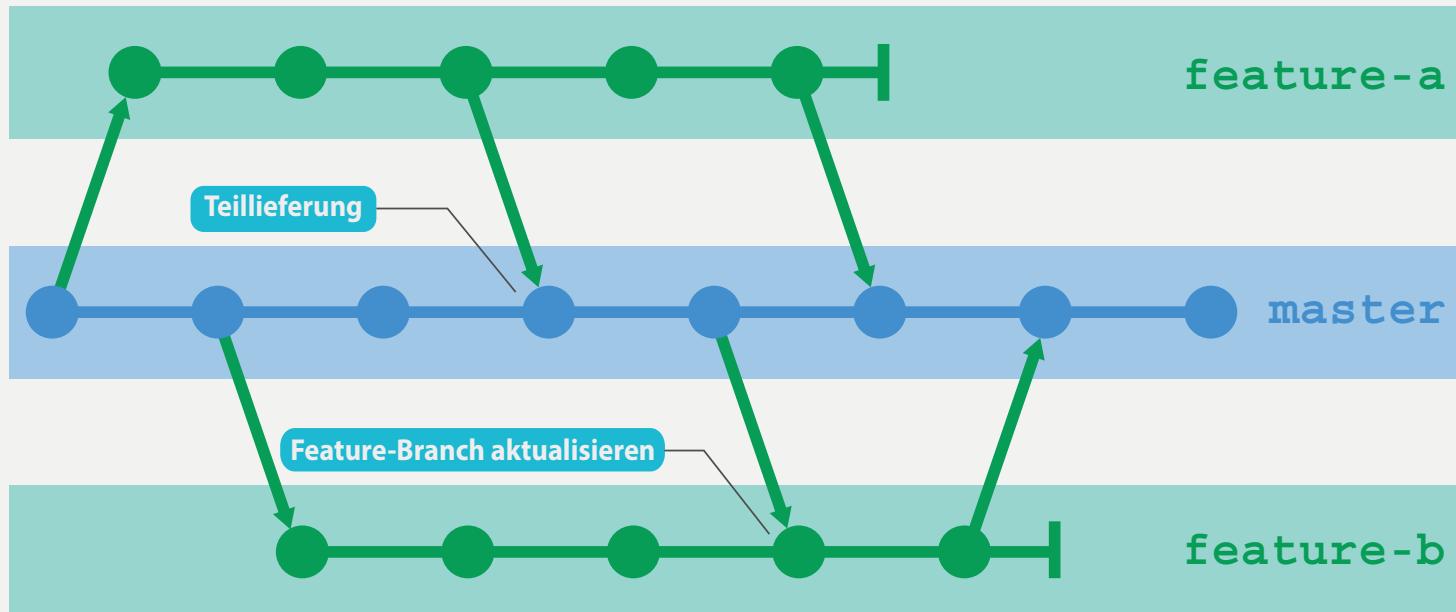


Historie mit vielen Merge-Commits

- keine strukturelle Zuordnung zu Features
- in jeder Commit-MESSAGE kann die Task-ID hinterlegt werden, um die Commits zu unterscheiden und z.B. Release-Dokumentation zu erzeugen



ARBEITEN MIT FEATURE-BRANCHES



* aus „Git - Grundlagen und Workflows“

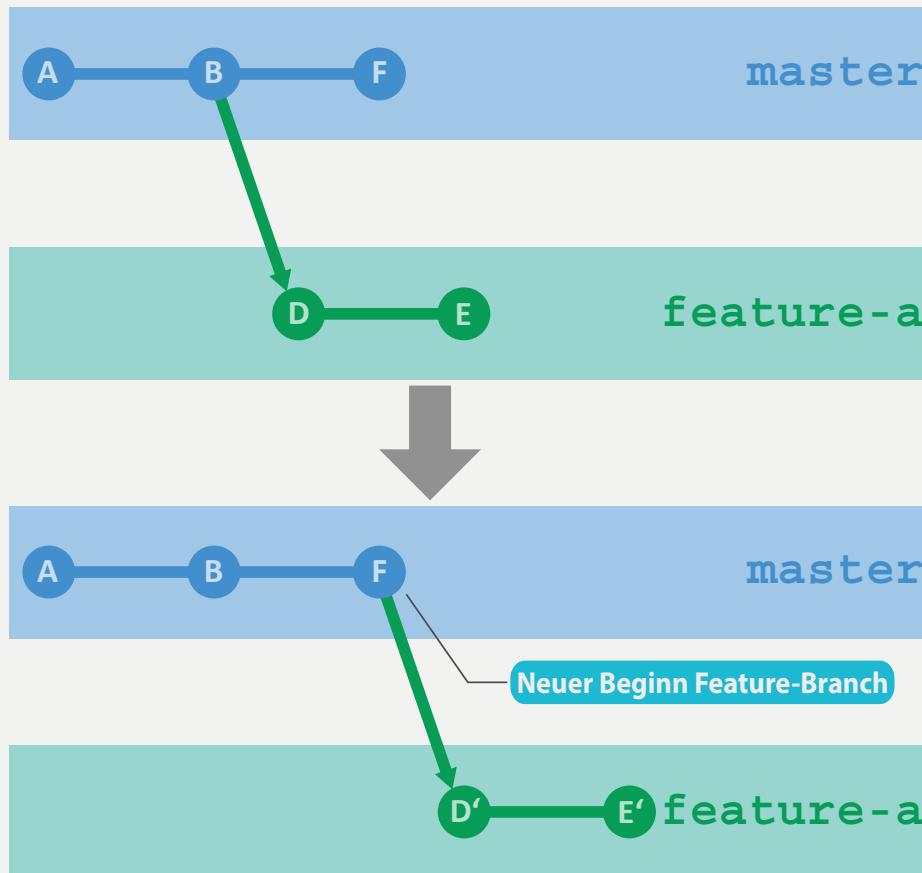
- Jedes Feature wird auf einen eigenen Branch entwickelt
- Bei allen Merges auf dem „master“-Branch werden Fast-Forward-Merges unterdrückt (Eindeutige First-Parent-Historie erzeugen)
- Austausch zwischen Features findet immer über den „master“-Branch statt
- Komplexerer Ablauf
- Gute Nachvollziehbarkeit der Änderungen für ein Feature



REBASING - ÜBERBLICK

`git rebase origin/master`

`git pull --rebase`

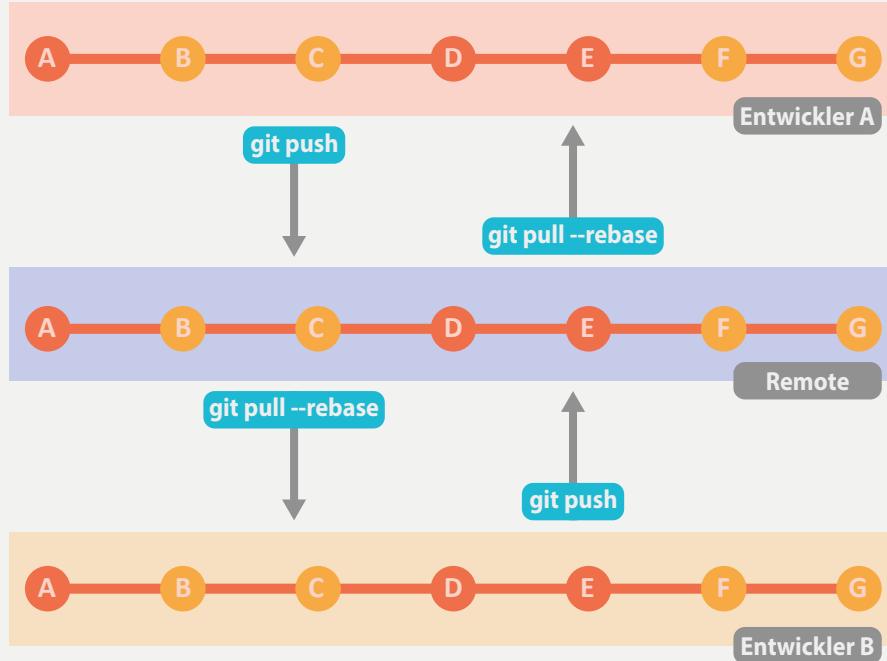


- Änderungen von Commits werden in neue Commits kopiert
- Der aktuelle Branch wird verschoben

* aus „Git - Grundlagen und Workflows“



ARBEITEN AUF GEMEINSAMEN BRANCH (REBASE)



Lineare Historie

- Commit-Reihenfolge entspricht nicht der zeitlichen Reihenfolge
- Rebase führt u.U. zu mehreren Konfliktbehandlungen, die dafür aber kleiner ausfallen

Rebase aktivieren

```
git config pull.rebase true  
git config branch.master.rebase true  
git config branch.autosetuprebase true
```

#Immer rebase
#Nur dieser Branch
#Neue Branches



REBASE DURCHFÜHREN

git rebase origin/master

Rebase mit dem Remote-Tracking-Branch

Bei Konflikten:

git add foo.txt

Dateien editieren und zum Index hinzufügen

git rebase --continue

Anschließend Rebase fortsetzen

git rebase --skip

Alternativ das Commit überspringen

git rebase --abort

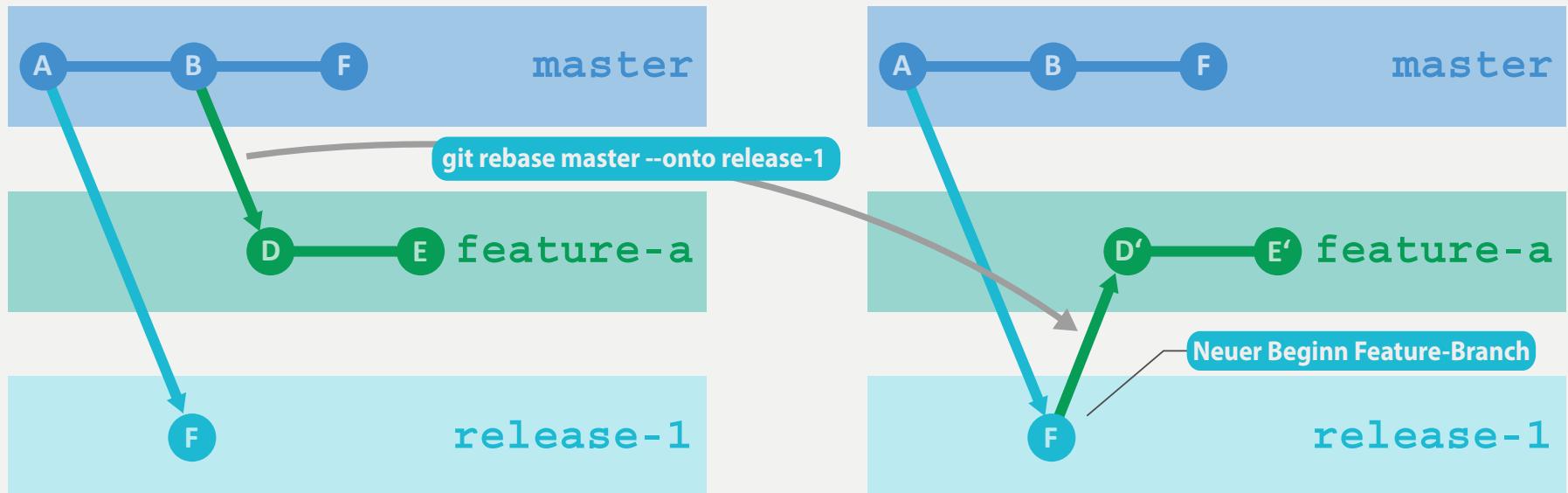
Oder das Rebase ganz abbrechen



NOCH MEHR REBASE

git rebase master --onto <branch>

- „Verschiebt“ die Commits eines Branches

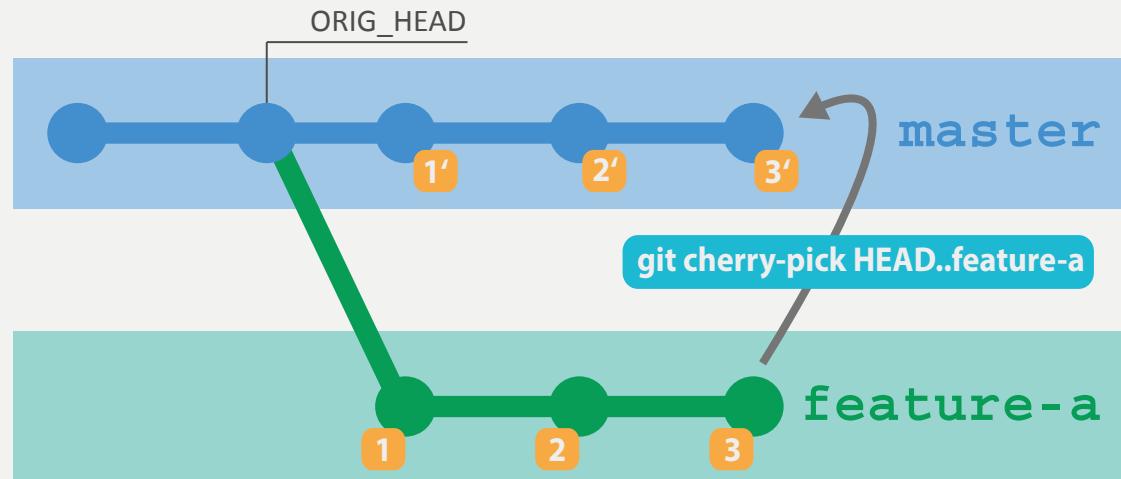


CHERRY-PICK

```
git cherry-pick <commit>
```

```
git cherry-pick <ref1>..<ref2>
```

Kopiert ein oder mehrere Commits



- Jedes Commit hat **Author** und **Committer** um bei Rebase und Cherry-Pick den originalen **Author** und den aktuellen **Committer** zu unterscheiden



INTERAKTIVES REBASE

```
git rebase -i HEAD~4
```

```
pick ca40bf8 Erster Entwurf
pick 9342d1f JUnit-Tests
pick 0f5232f JavaDoc
pick 18ba83d JavaDoc korrigiert
```

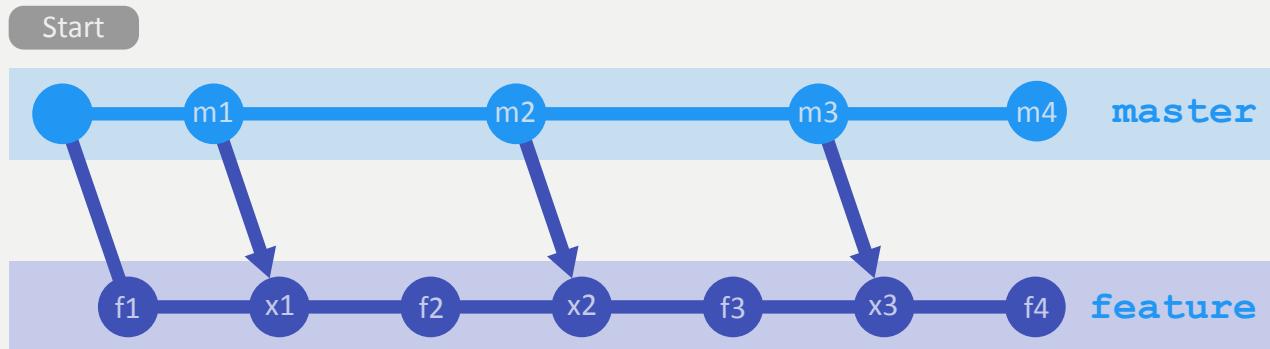
```
# Rebase e5d686f..add00c1 onto e5d686f
# Commands:
#   p, pick = use commit
#   r, reword = use commit, but edit the commit message
#   e, edit = use commit, but stop for amending
#   s, squash = use commit, but meld into previous commit
#   f, fixup = like "squash", but discard this commit's log message
#   x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
# Note that empty commits are commented out
```



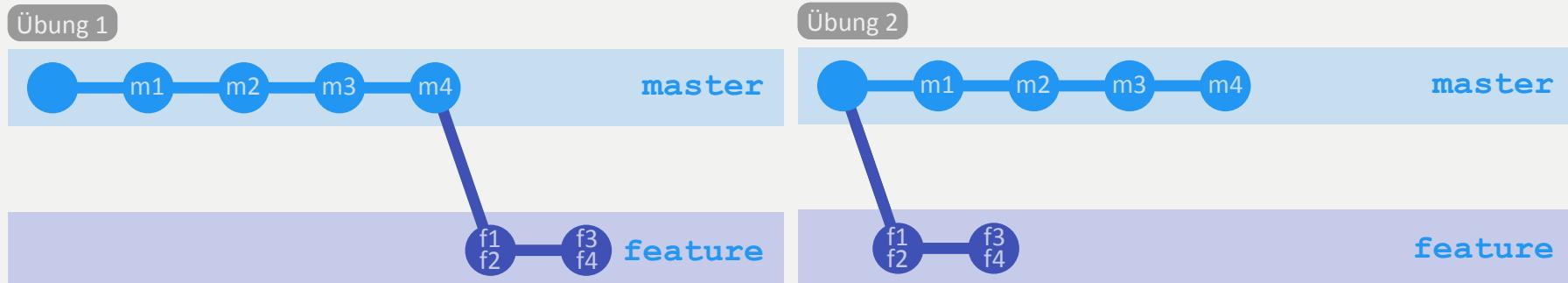


ÜBUNG: REBASE - 1

Repository: rebase/rebase-<aufgabe>



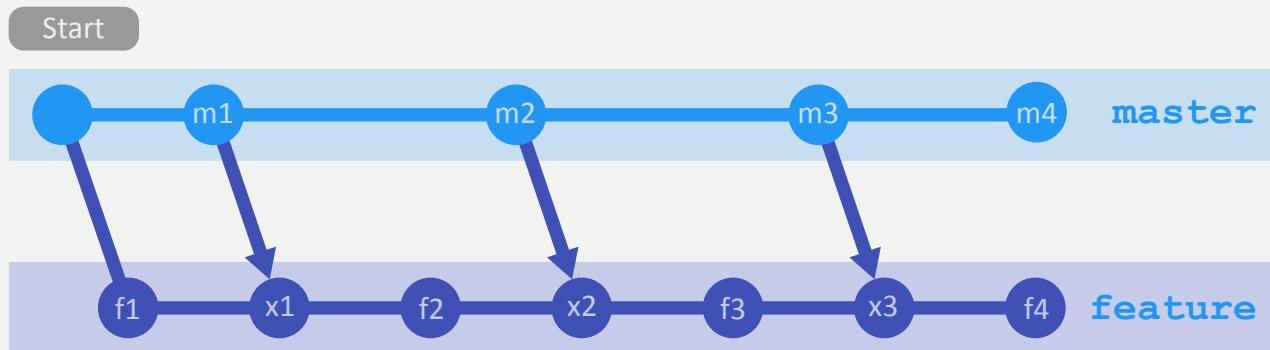
Aufgabe 1 + 2: Führen Sie ein interaktives Rebasing durch, so dass das Ergebnis wie folgt dargestellt aussieht:





ÜBUNG: REBASE - 2

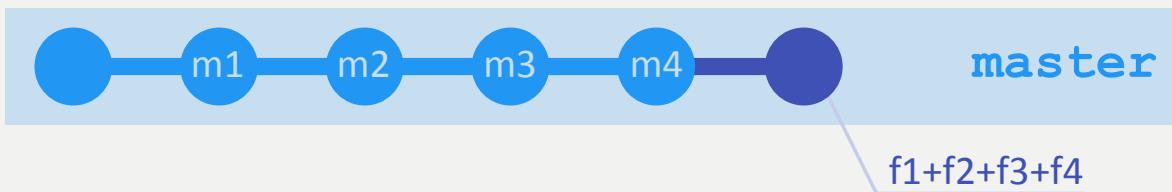
Repository: rebase/rebase-<aufgabe>



Aufgabe 3:

Welche Möglichkeiten / Befehle gibt es um die Historie, so wie unten dargestellt umzubauen?

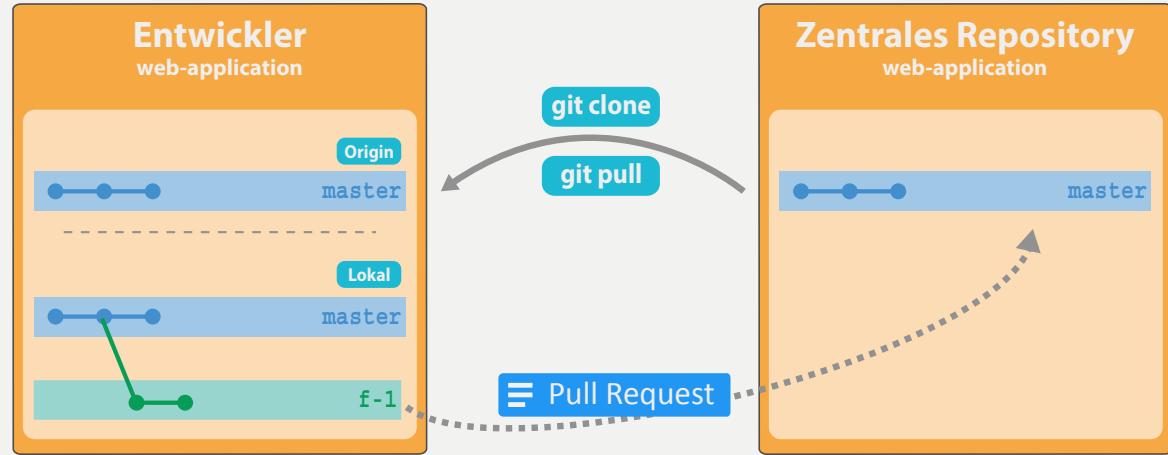
Übung 3



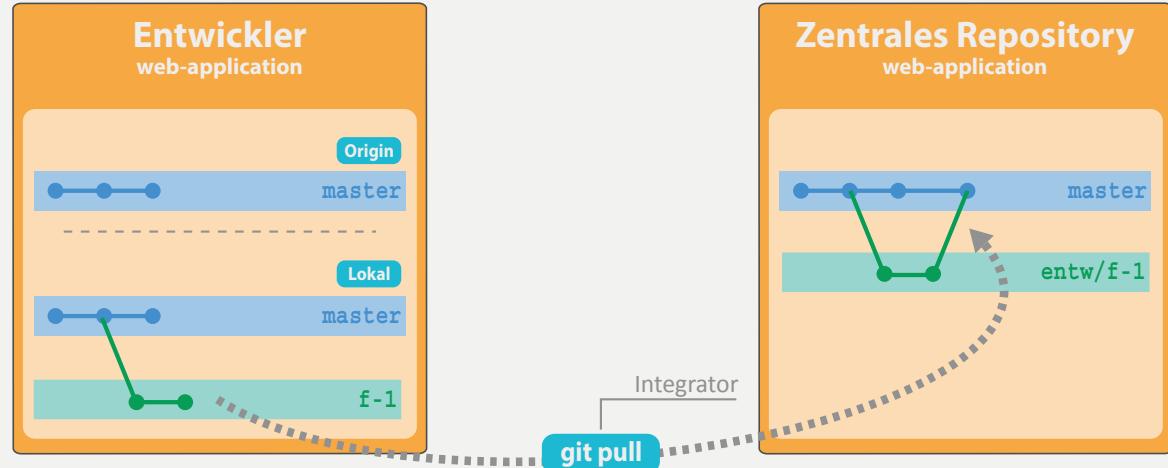
PULL-REQUESTS

Zugriff auf **Repositories** oder einzelne **Branches** ist reglementiert.

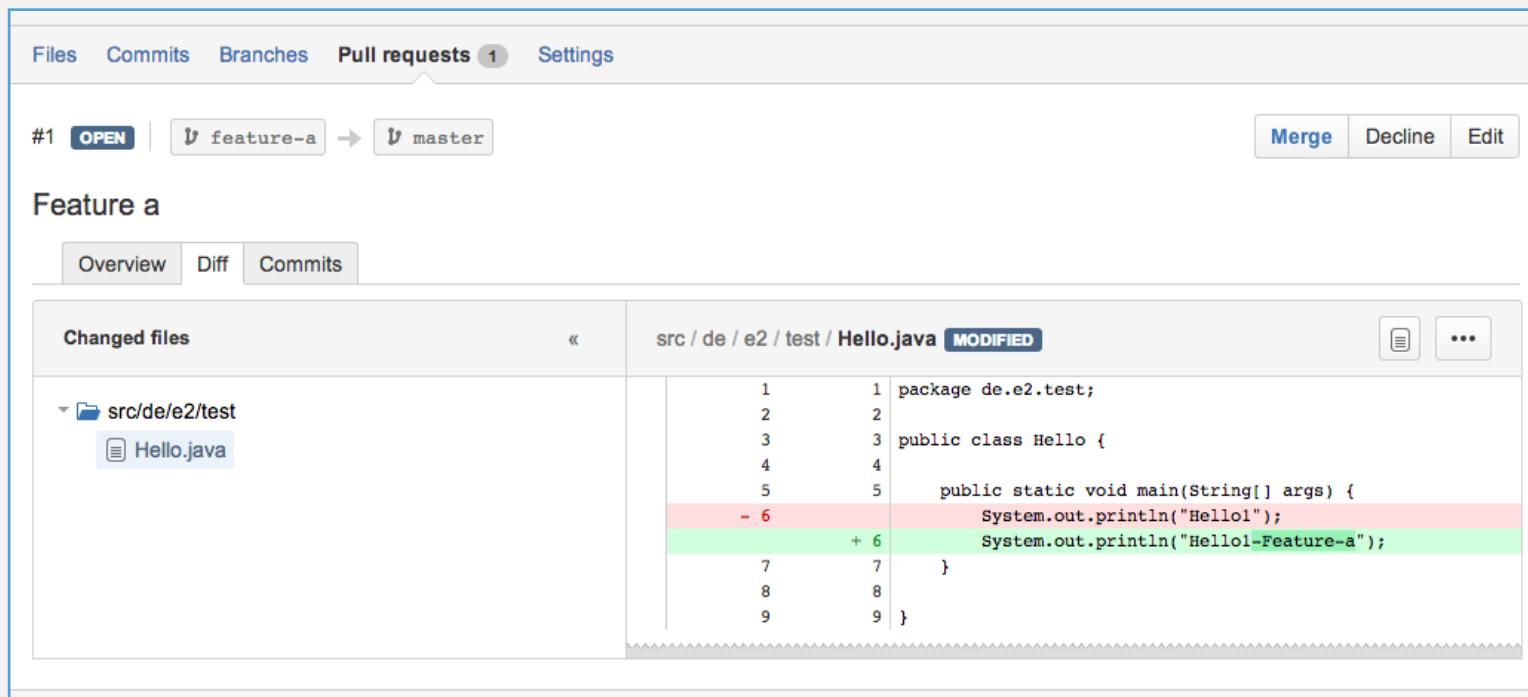
1. Entwickler führt Änderungen in seinem Repository durch und informiert Integrator über den Wunsch zur Abgabe (Pull-Request).



2. Integrator holt sich die Änderungen (Pull) und führt diese zusammen.



PULL-REQUEST FÜR REVIEW-WORKFLOWS



- Pull-Requests können gut als Review-Werkzeug benutzt werden
- Continuous Integration Server (Jenkins, etc) unterstützen das Bauen von Pull-Requests (temporäres Merge-Commit)



TEIL 2

BRANCH MODELLE



Workflows in der Entwicklung

Releaseprozesse

RELEASEPROZESS - GITFLOW

<http://nvie.com/posts/a-successful-git-branching-model/>

- Etabliertes Branch-Modell für Git

Historische Branches

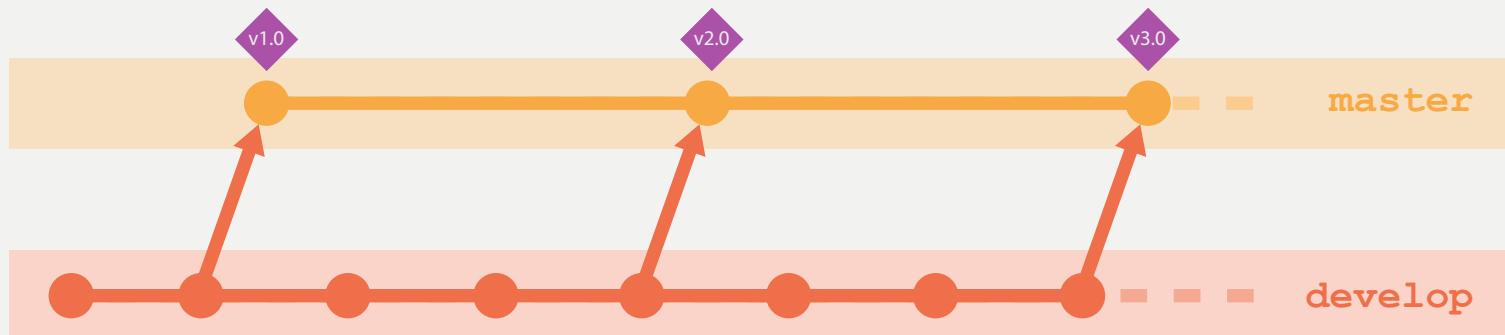
- **master**: Fertige Releases
- **develop**: für die Entwicklung
- Entwicklung erfolgt auf **feature**-Branches



RELEASEPROZESS - GITFLOW

„Historische“ Branches

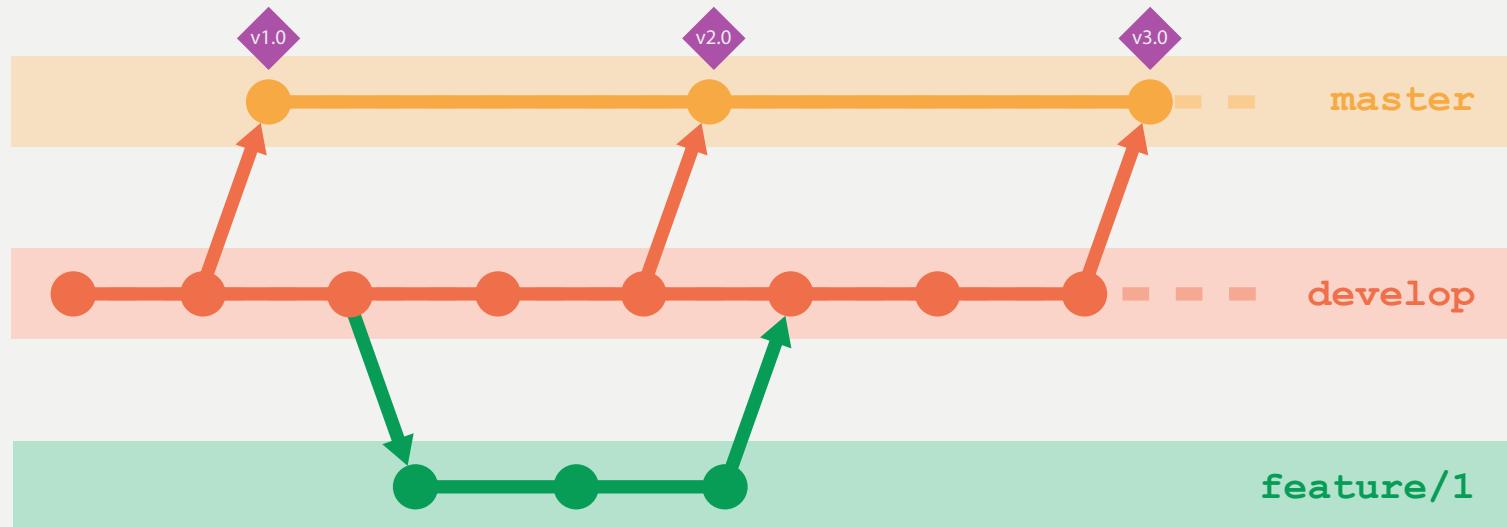
- **master**: Fertige Releases
- **develop**: Integrationsbranch für Features



RELEASEPROZESS - FEATURE ENTWICKELN (I)

Entwicklung erfolgt über **feature**-Branches

- Feature-Branches starten und enden auf dem **develop**-Branch
- Namensprefix: **feature/featureBezeichnung**
- Mergen mit **--no-ff**, um First-Parent-Historie zu erzwingen



RELEASEPROZESS – FEATURE ENTWICKELN (II)

Praxis

```
# Feature Branch erzeugen  
git checkout -b feature/restApiErweitern
```

```
# Änderungen durchführen, Commits erzeugen  
. . .
```

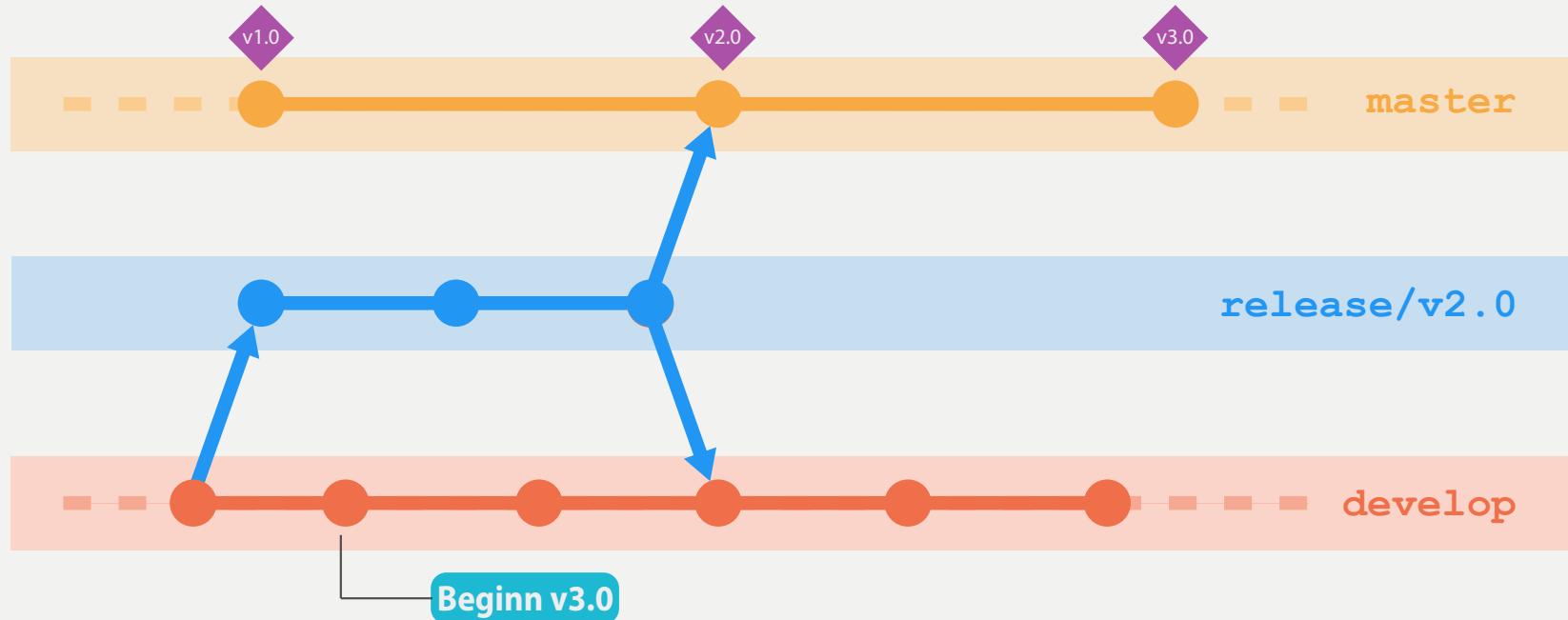
```
# Feature-Branch mergen und löschen  
git checkout develop  
git merge --no-ff feature/restApiErweitern  
git branch -d feature/restApiErweitern
```



RELEASEPROZESS – RELEASE ERZEUGEN (I)

Stabilisierung auf dem **release**-Branch

- Prefix: **release**/**releaseBezeichnung**
- Nach Abschluss auf **master** und **develop** mergen



RELEASEPROZESS – RELEASE ERZEUGEN (II)

Praxis

```
# Release-Branch erzeugen
git checkout -b release/v1.0 develop

. . .

# Release abschließen
git checkout master
git merge --no-ff release/v1.0
git tag [-s] -m „Release 1.0 fertiggestellt“ v1.0

# Änderungen in den develop-Branch integrieren
git checkout develop
git merge --no-ff release/v1.0

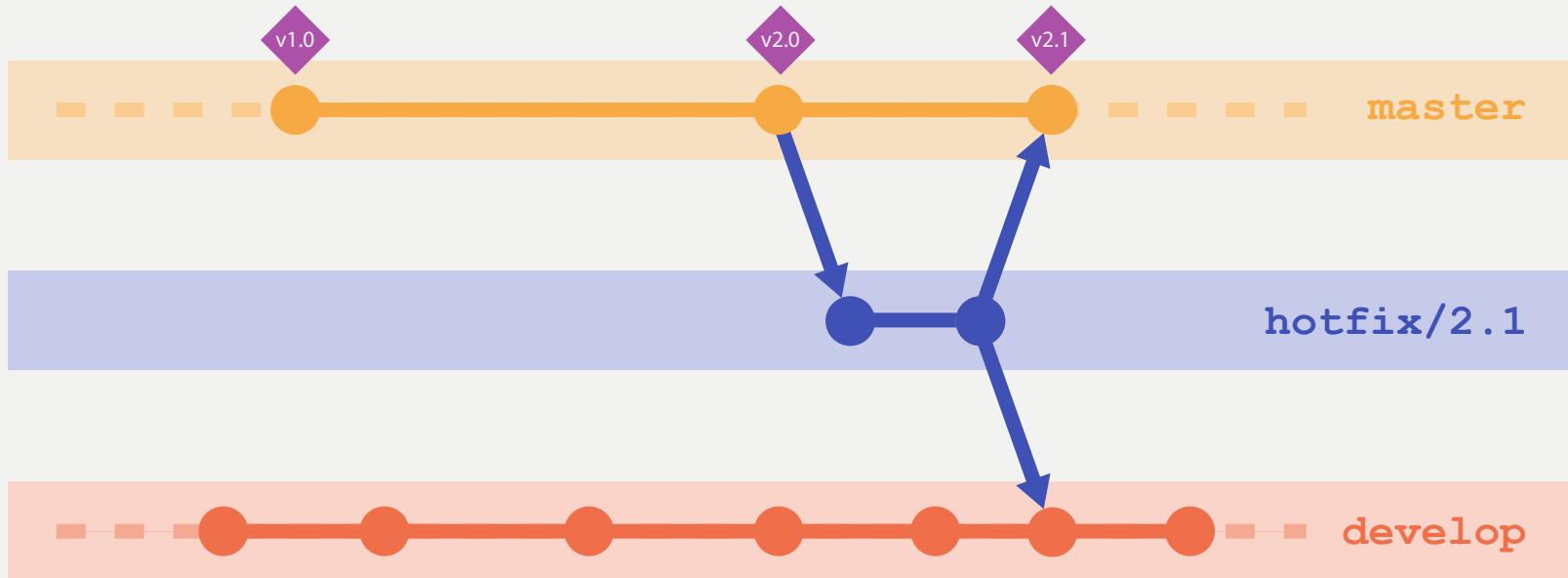
# Release-Branch löschen
git branch -d release/v1.0
```



RELEASEPROZESS – HOTFIX ERSTELLEN (I)

Hotfixes: Reparaturen für aktuelles Release

- Prefix: **hotfix/problemBezeichnung**
- Zweigt vom neusten Release ab
- Nach **master** und **develop** mergern



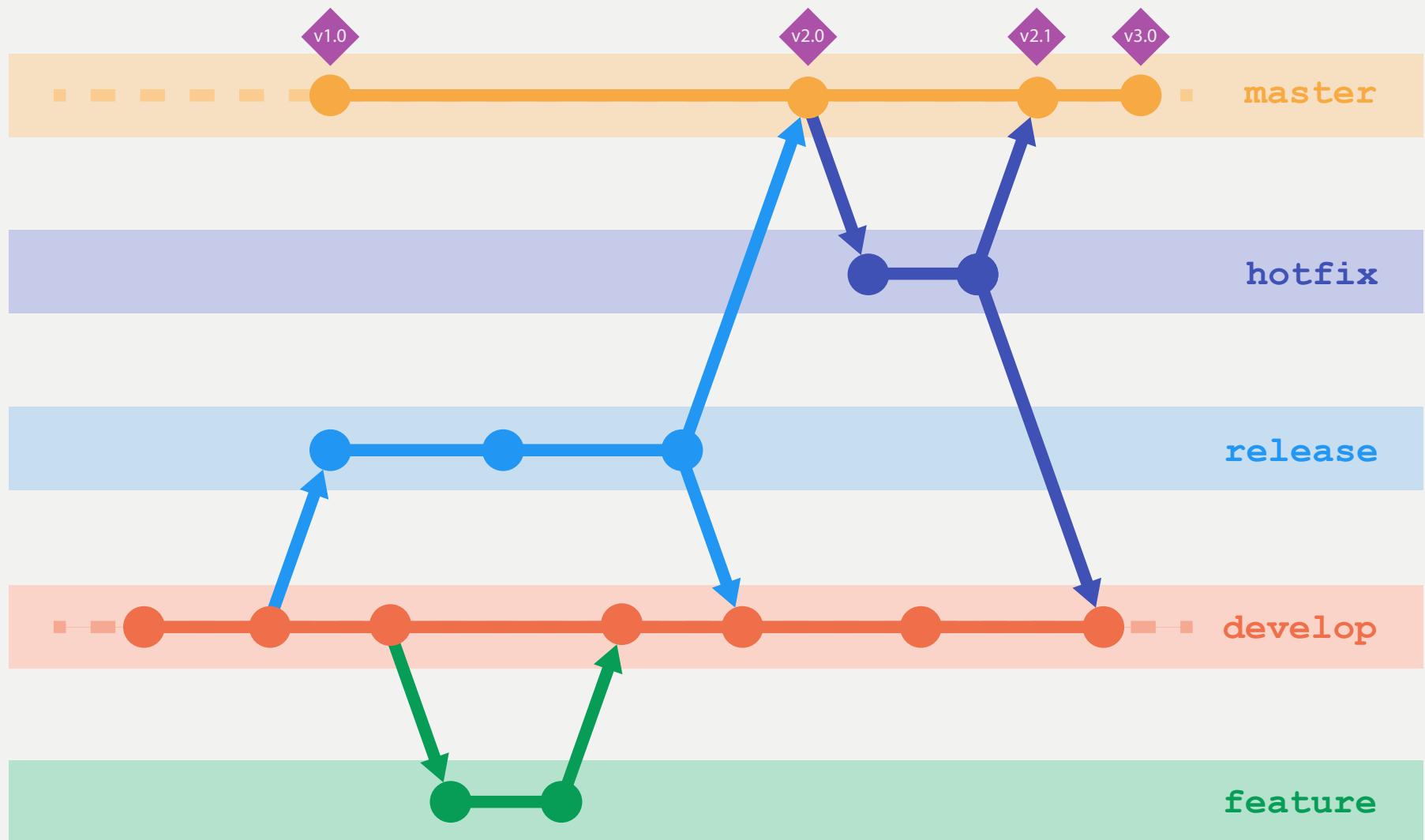
RELEASEPROZESS – HOTFIX ERSTELLEN (II)

Praxis

```
# Hotfix-Branch vom aktuellen master erzeugen  
git checkout -b hotfix/exceptionInRESTCall master  
  
# Änderungen durchführen, Commits erzeugen  
.  
.  
. .  
  
# Hotfix-Branch zurück mergen und löschen  
git checkout master  
git merge --no-ff hotfix/exceptionInRESTCall  
git tag [-s] -m „Release v1.1“ v1.1  
  
git checkout develop  
git merge --no-ff hotfix/exceptionInRESTCall  
  
git branch -d hotfix/exceptionInRESTCall
```

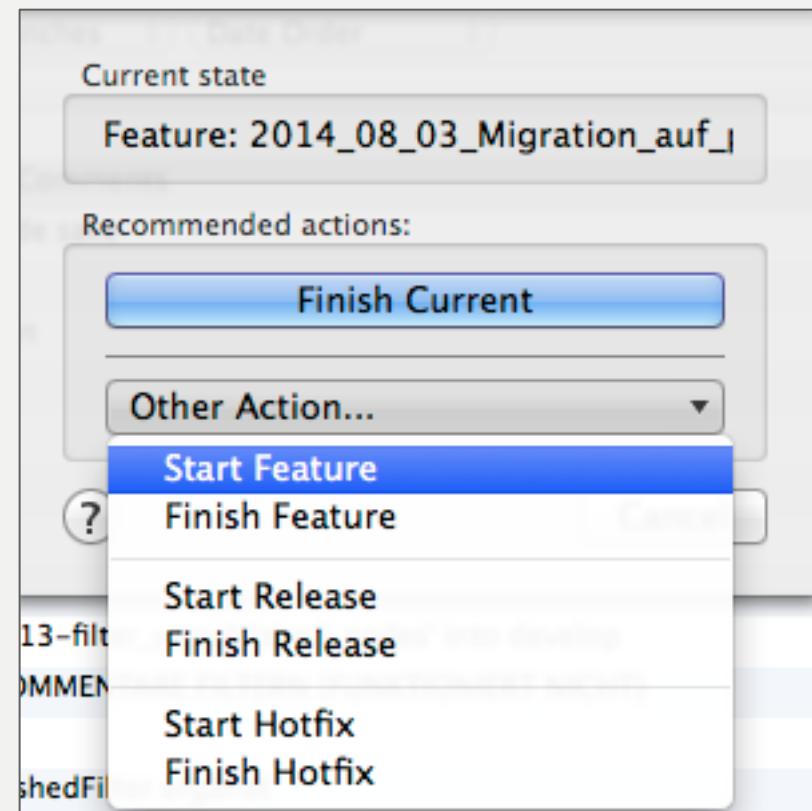


RELEASEPROZESS - GITFLOW ÜBERSICHT



RELEASEPROZESS - GITFLOW-TOOLS

- Bash Script:
 - <https://github.com/nvie/gitflow>
- SmartGit, SourceTree, IDEA
- Atlassian
 - Maven
 - JGit



RELEASEPROZESS – GITFLOW GOODIES

- Release-Notes erzeugen

```
$ git log --oneline --first-parent v0.3^..v0.3^2  
2388f93 Release 0.3 fertig  
09120b4 Finished feature/inspectPerformance (Merged into develop)  
a945780 Finished feature/upgradeSpringVersion (Merged into develop)  
6120529 Merge Branch 'release/v0.2' into 'develop'
```

- An welchen Features wird gearbeitet?

```
git fetch origin refs/heads/feature/*:refs/current/features/*  
git ls-remote origin feature/*
```

- Alle Commits zu einem Feature

```
git log --oneline --first-parent --grep MEIN-FEATURE  
(Nur bei entsprechenden Commit-Message-Konventionen)
```

- Zurücknehmen von Änderungen

```
git revert <Merge-Commit>
```



RELEASEPROZESS - GITFLOW PROBLEME

- Recht viele Merges
 - `release` nach `master` und `develop`
 - `hotfix` nach `master` und `develop`
- Was passiert wenn `release` *und* `hotfix` in Arbeit sind?
- Keine Behandlung älterer Releases





ÜBUNG: GIT FLOW

Repository: **gitflow-uebung**

1. Welche Releases sind bereits veröffentlicht worden?
2. Welche Features sind gerade in Arbeit für das nächste Release?
3. Welche Features wurden bereits für das nächste Release abgeschlossen?
4. Erzeugen Sie mit den bereits abgeschlossenen Features das nächste Release.
 - Verwenden Sie dazu bitte die Git-Kommandos und nicht die GitFlow Erweiterung!
 - Setzen Sie dabei in der Datei `version.txt` die korrekte Versionsnummer für das Release.
5. Zugabe: Machen Sie für das Release 0.2 einen *Backport* des Features `fixShellshockSecurityIssue`.



TEIL 3

GROSSE PROJEKTE

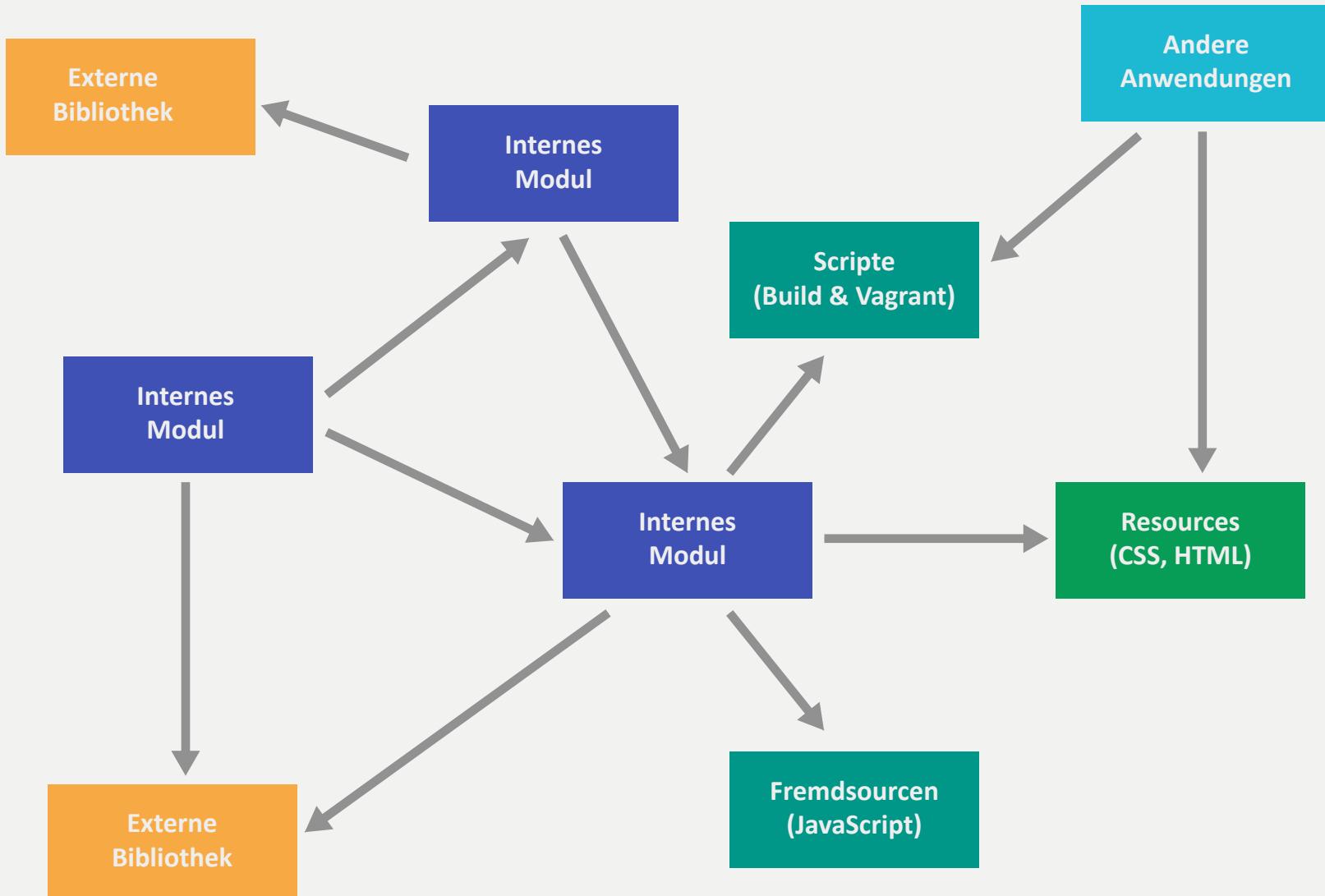


Repositoryaufteilung

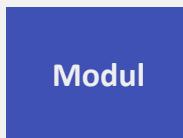
Submodules

Subtrees

HINTERGRUND: KOMPLEXE PROJEKTE



REPOSITORY AUFTHEILUNG: MODULE



Ein Modul...



Release-Einheit

- eigene Version
- eigenen Lebenszyklus



Git-Repository

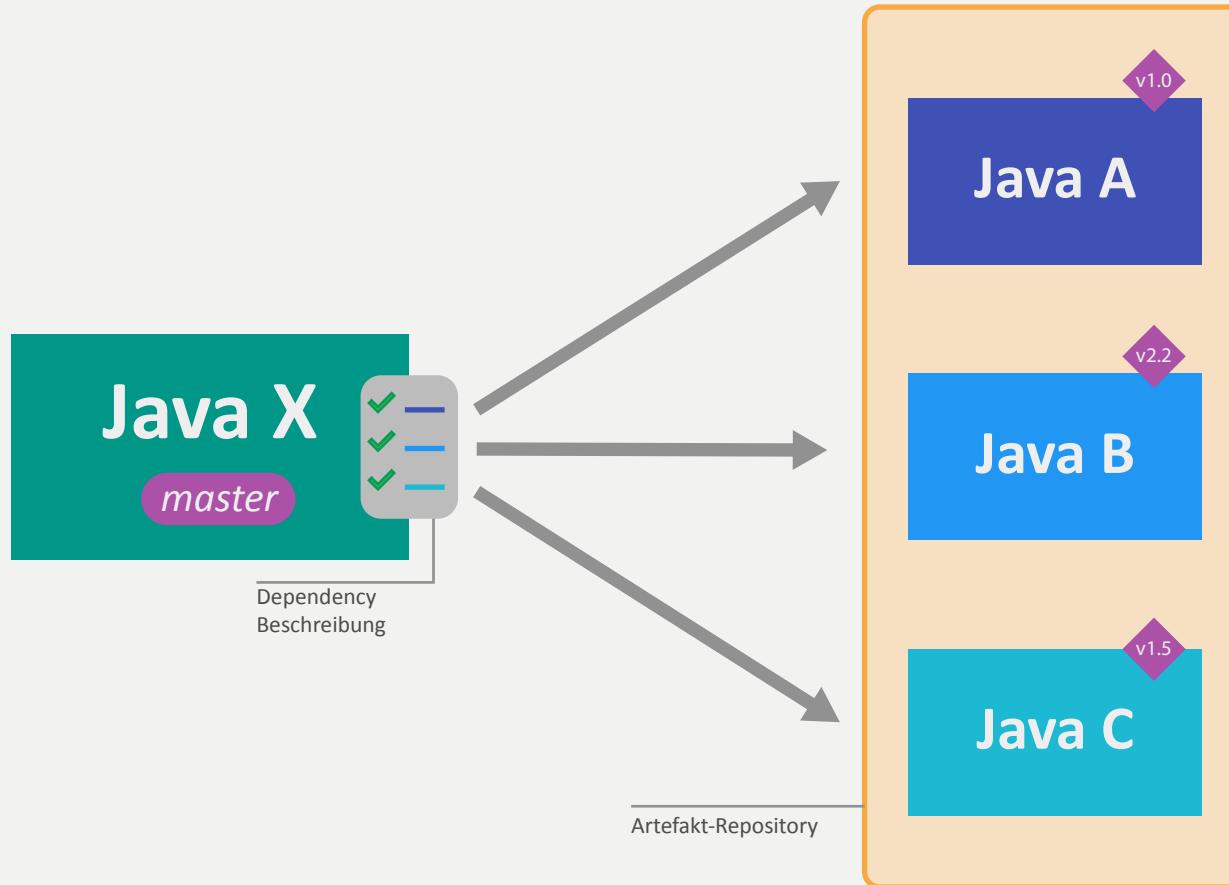
- eigene Branches
- eigene Tags



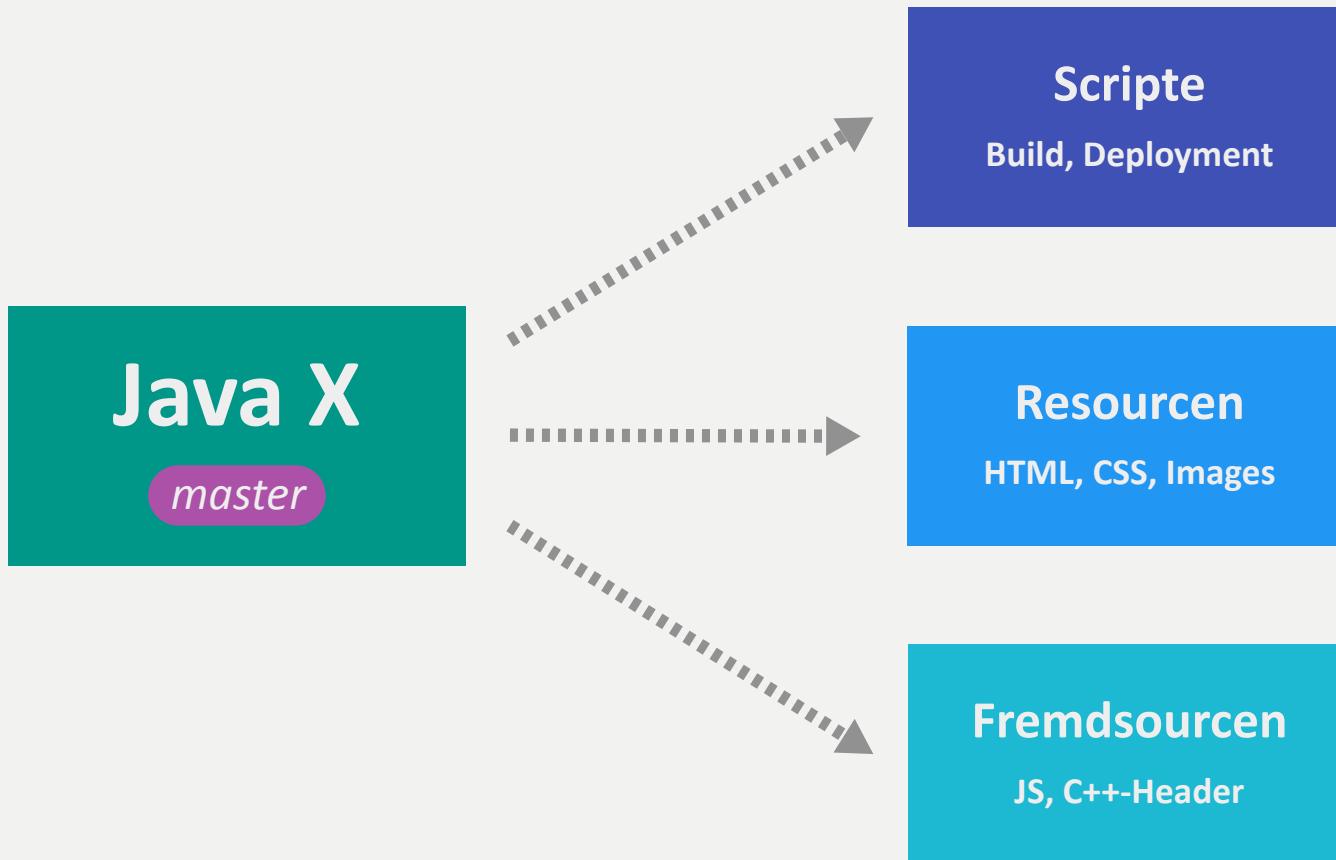
HOMOGENE MODULE

Dependency Manager

Maven, Ivy, Gradle, P2 | npm, RequireJS | Leiningen, SBT



INHOMOGENE MODULE (I)



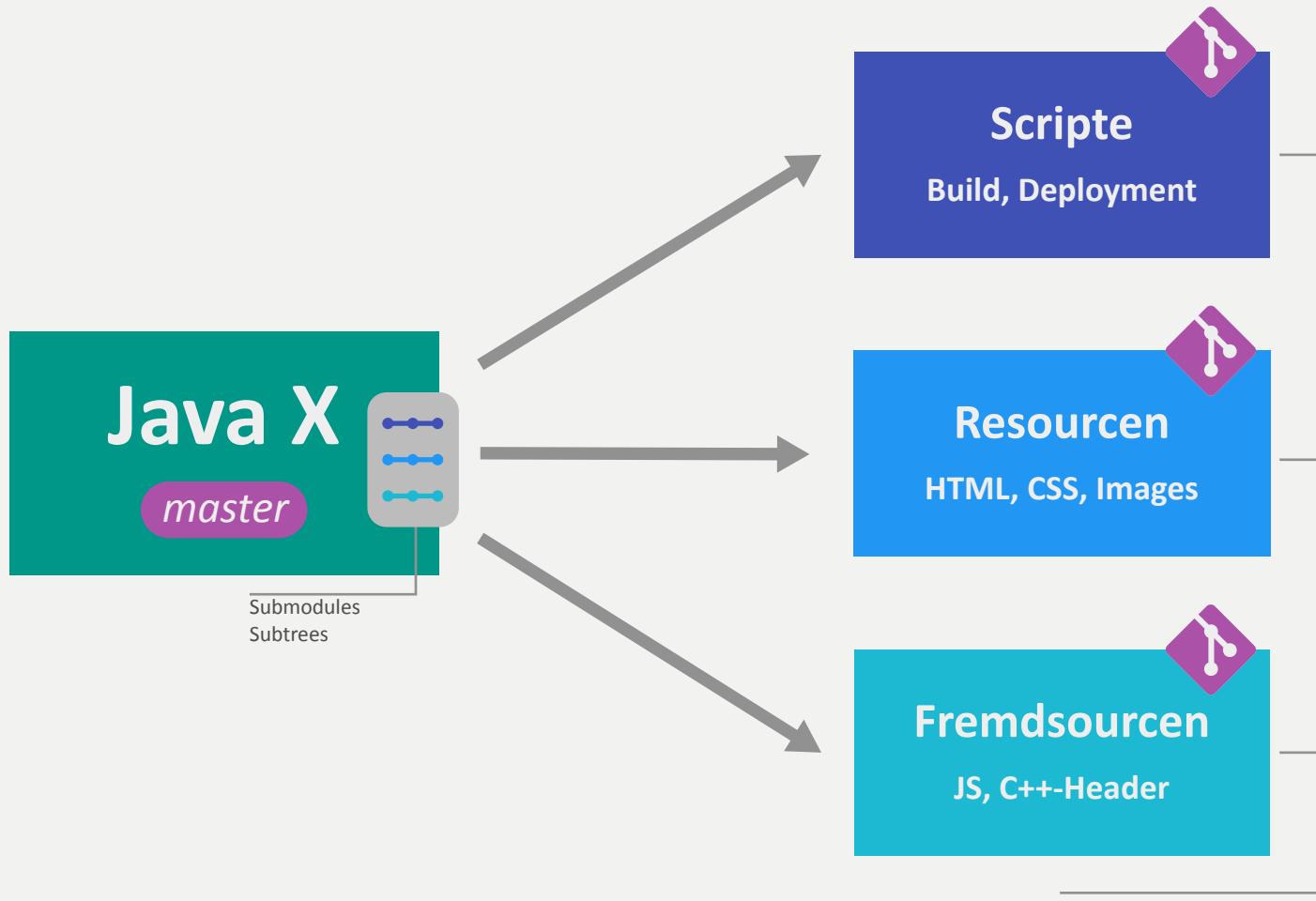
Inhomogene Infrastruktur

- Einbinden von (externen) Sourcen und Ressourcen erforderlich
- Globales Build notwendig



INHOMOGENE MODULE (II)

v1.0



Submodule oder Subtree

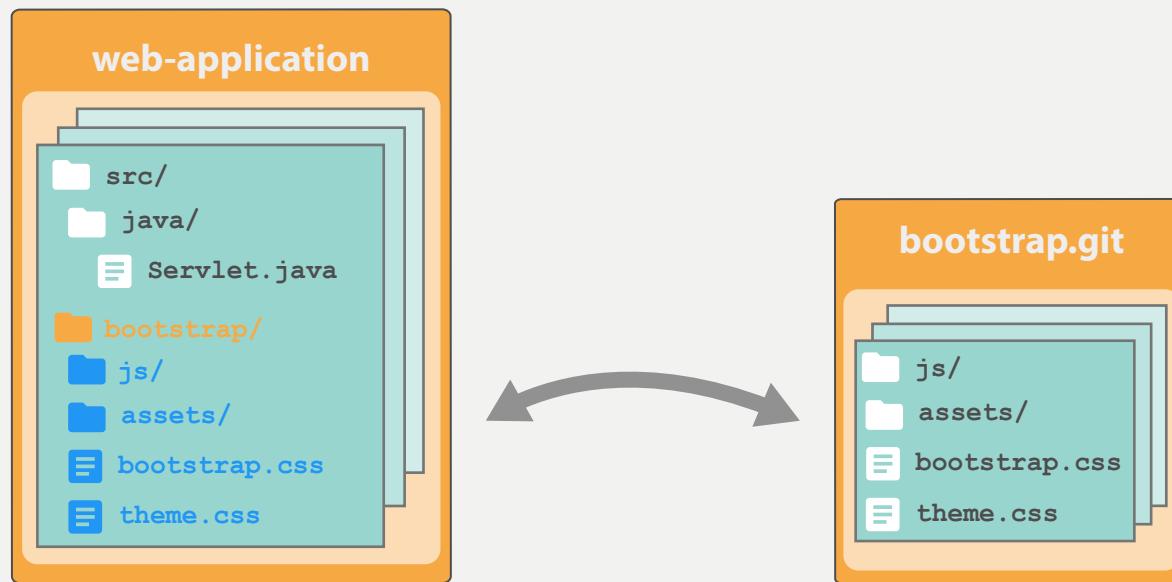
- Einbinden von externen Git-Repositories
- Exakter Stand wird versioniert



EINGEBUNDENE REPOSITORIES

Referenziertes Repository wird unterhalb eines Verzeichnisses eingebunden

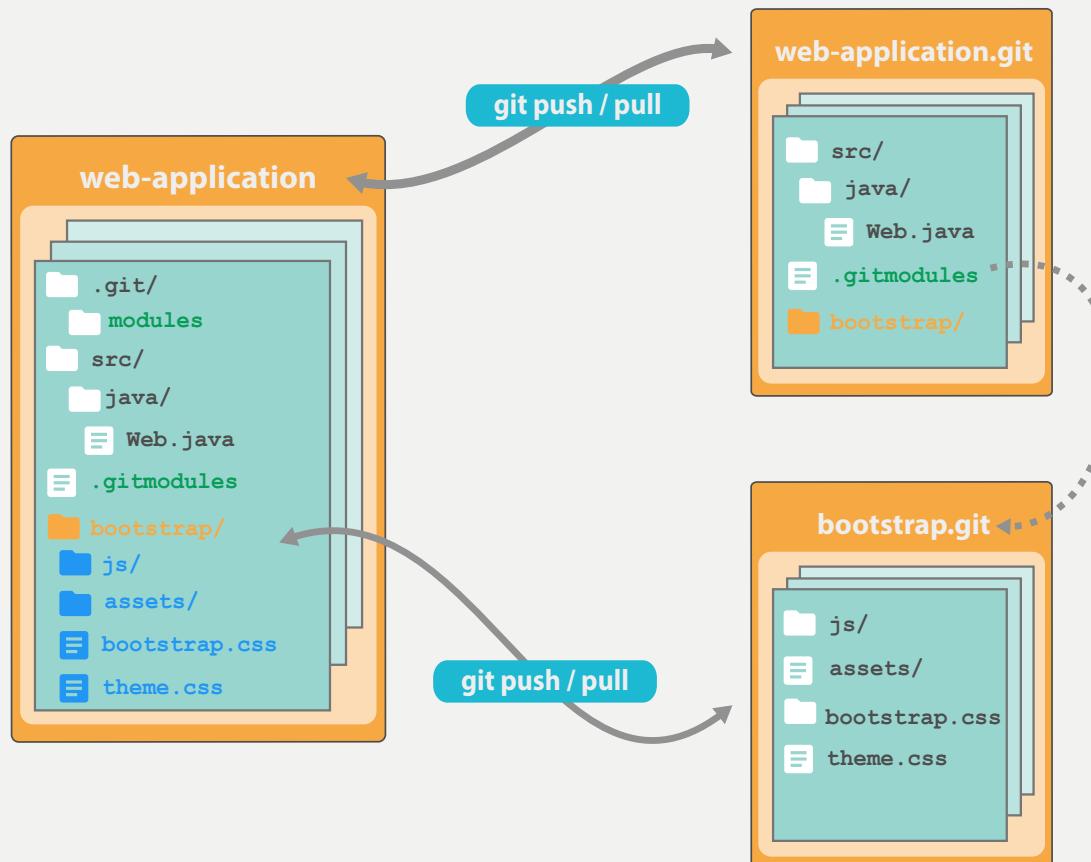
- Zwei mögliche Varianten **git submodule** und **git subtree**



GIT SUBMODULE

Fremdes Repository wird „verlinkt“

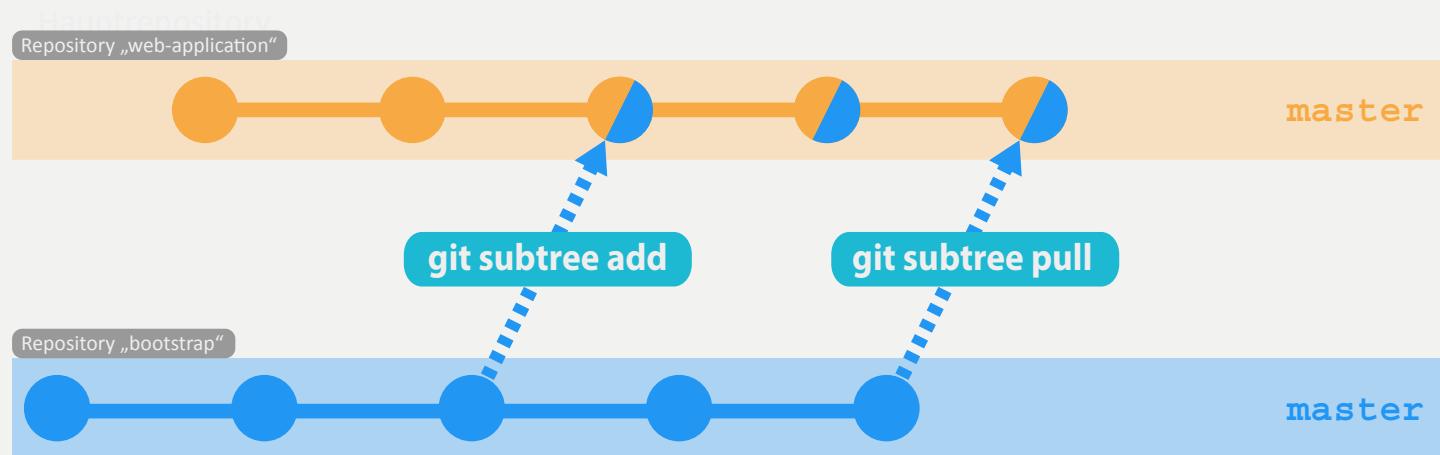
- URL und Commit des fremden Repositories werden im Haupt-Repository hinterlegt
- Beide Repositories existieren unabhängig voneinander



GIT SUBTREE

Fremdes Repository wird über einen Subtree Merge eingebunden

- Historie wird in Ziel-Repository eingebunden
- Objekte und Referenzen werden in Ziel-Repository übernommen
- Tree des Original-Repositories wird in Ziel-Repository als Unterverzeichnis eingebunden



* aus „Git - Grundlagen und Workflows“



PRAXIS-BEISPIEL



1. Modul hinzufügen
2. Repository mit Modul klonen
3. Neue Version eines Moduls einbinden
4. Änderungen in einem Modul durchführen



TEIL 3

GROSSE PROJEKTE

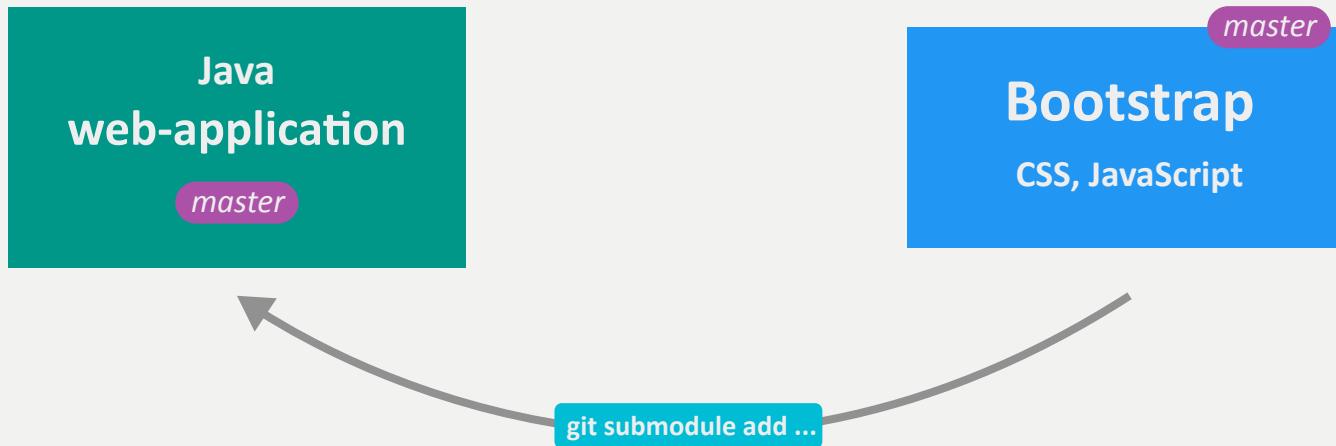


Repositoryaufteilung

Submodules

Subtrees

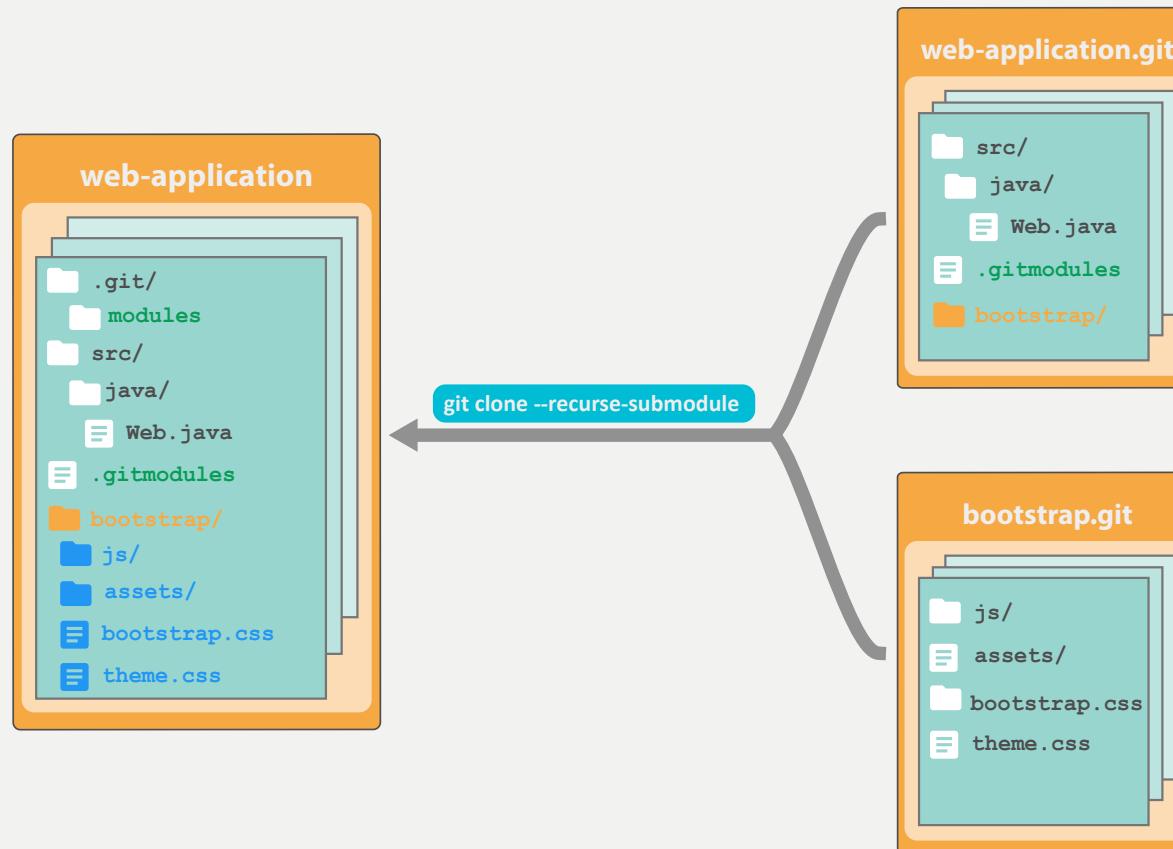
SUBMODULE HINZUFÜGEN



```
web-app$ git submodule add bootstrap.git
web-app$ git commit -m "Bootstrap hinzugefügt"
[master 7377297] Bootstrap hinzugefügt
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 src/main/webapp/bootstrap
```



KLONEN MIT SUBMODULES (I) - NEUER KLON

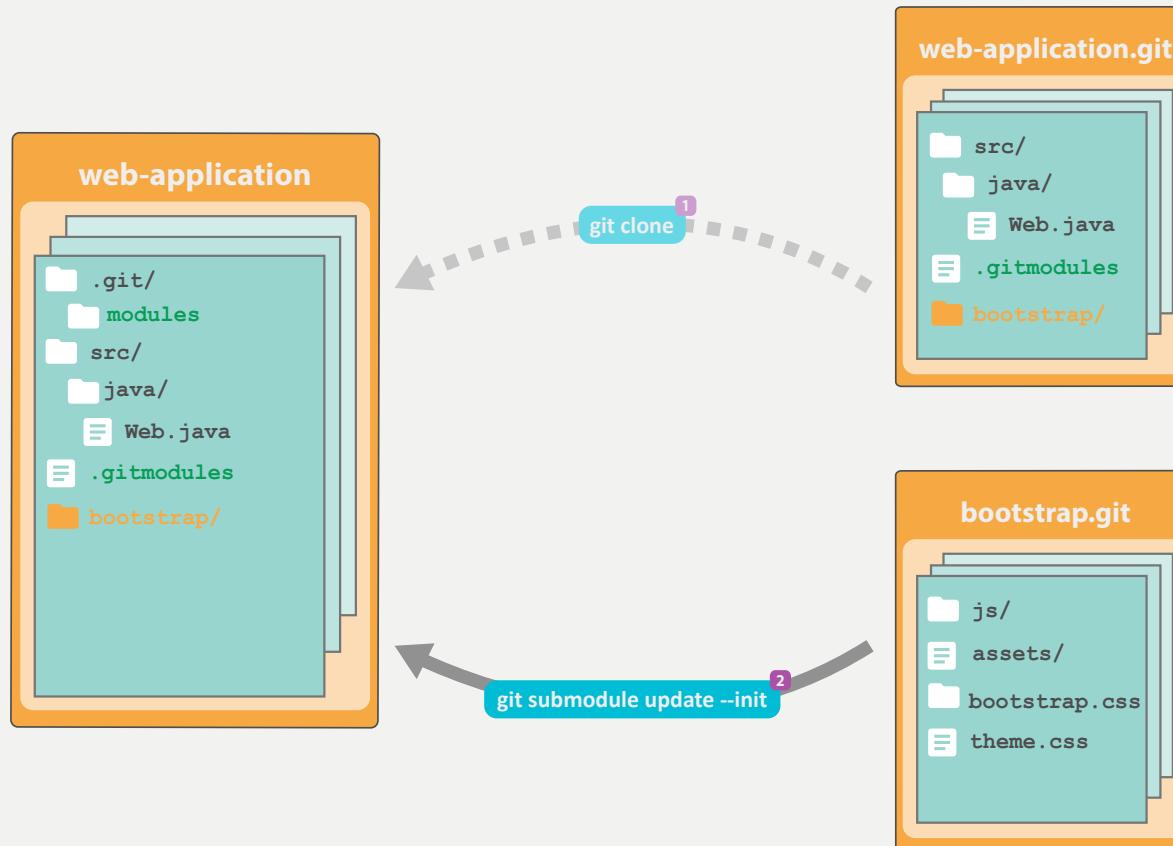


```
$ git clone --recurse-submodule web-application.git
```

```
...
Cloning into 'src/main/webapp/bootstrap'...
Submodule path 'src/main/webapp/bootstrap': checked out
```



KLONEN MIT SUBMODULES (II) BESTEHENDER KLON



```
# Bestehenden Klon mit Submodule initialisieren
```

```
web-app$ git submodule update --init
```

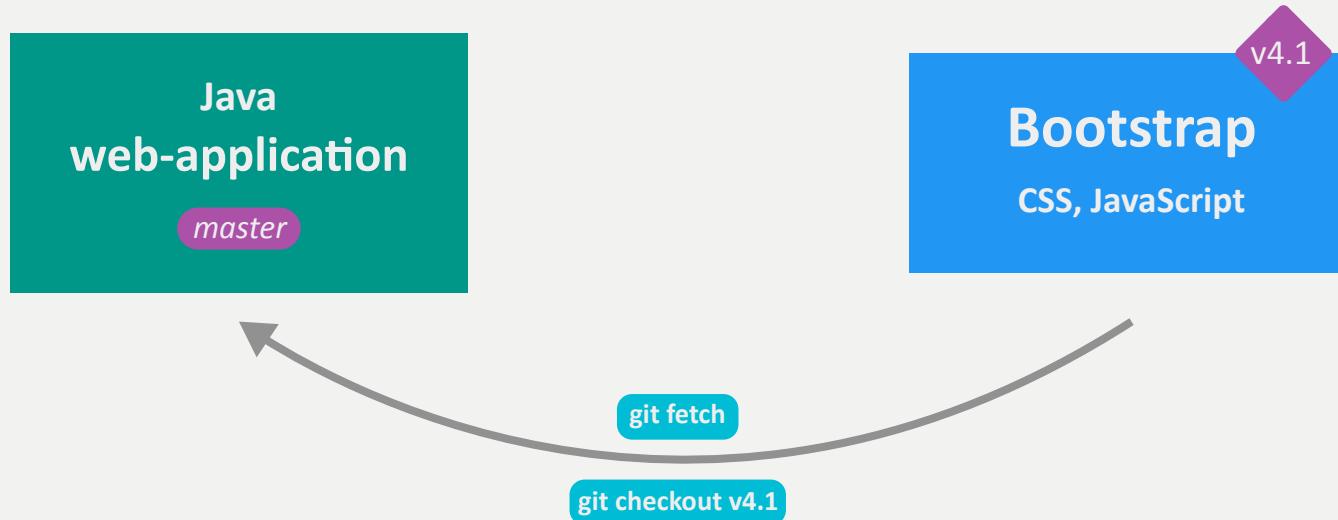
```
...
```

```
Cloning into 'src/main/webapp/bootstrap'...
```

```
Submodule path 'src/main/webapp/bootstrap': checked out
```



NEUE VERSION EINBINDEN



```
web-app/bootstrap$ git fetch
```

```
web-app/bootstrap$ git checkout v4.1
```

Previous HEAD position was 9876b4d...

HEAD is now at 61bb387...

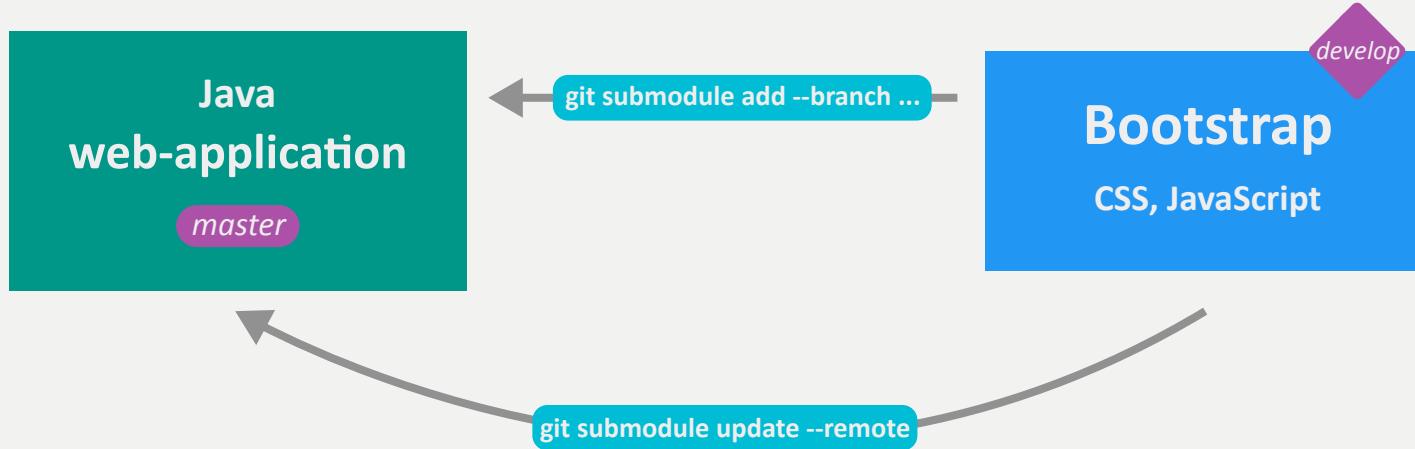
```
web-app$ git commit -m „Neue Version eingebunden“ bootstrap
```

```
[master d6b2223] Neue Version eingebunden
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```



ALTERNATIVE: BRANCH EINBINDEN (SEIT GIT 1.8.2)

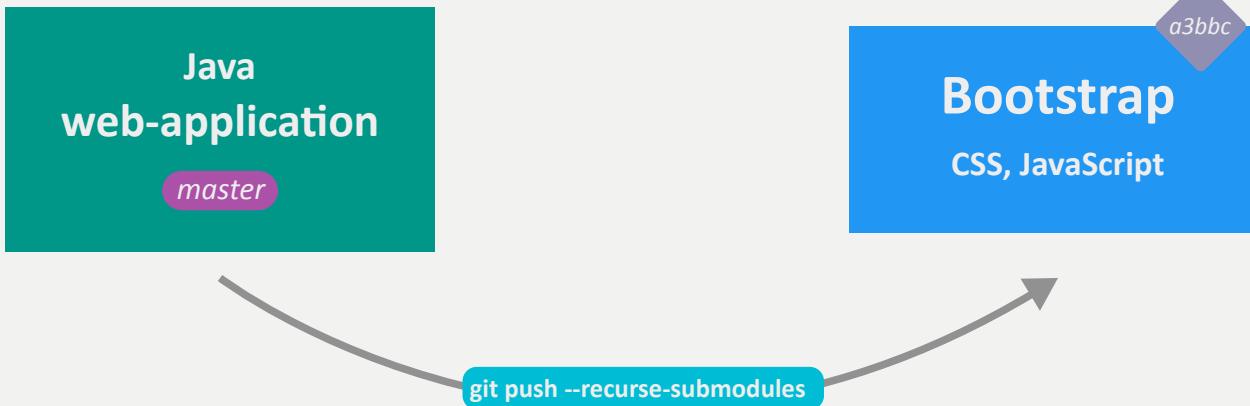


```
# Einmaliges Hinzufügen des Submoduls
web-app$ git submodule add --branch develop bootstrap.git
web-app$ git commit -m "Bootstrap hinzugefügt"

# Aktualisieren mit neuster Version des Branches develop
web-app$ git submodule update --remote
web-app$ git commit -m „Neue Version vom develop-Branch“ bootstrap
[master 92598cc] Neue Version vom develop-Branch
 1 file changed, 1 insertion(+), 1 deletion(-)
```



IN SUBMODULES ÄNDERN



```
web-app/bs$ git checkout master
web-app/bs$ git pull
# ...Dateien im Submodule ändern...
web-app/bs$ git commit -am „Direkt im Submodule geändert“

# Neue Version einbinden und in beide Repositories pushen
web-app$ git commit -am „Geänderte bootstrap Version“
web-app$ git push --recurse-submodules=on-demand
Pushing submodule 'src/main/webapp/bootstrap'
To bootstrap.git
f33af09..1032148  master -> master
To web-application.git
cd9de05..f0bb790  master -> master
```



TEIL 3

GROSSE PROJEKTE



Repositoryaufteilung

Submodules

Subtrees

SUBTREE HINZUFÜGEN

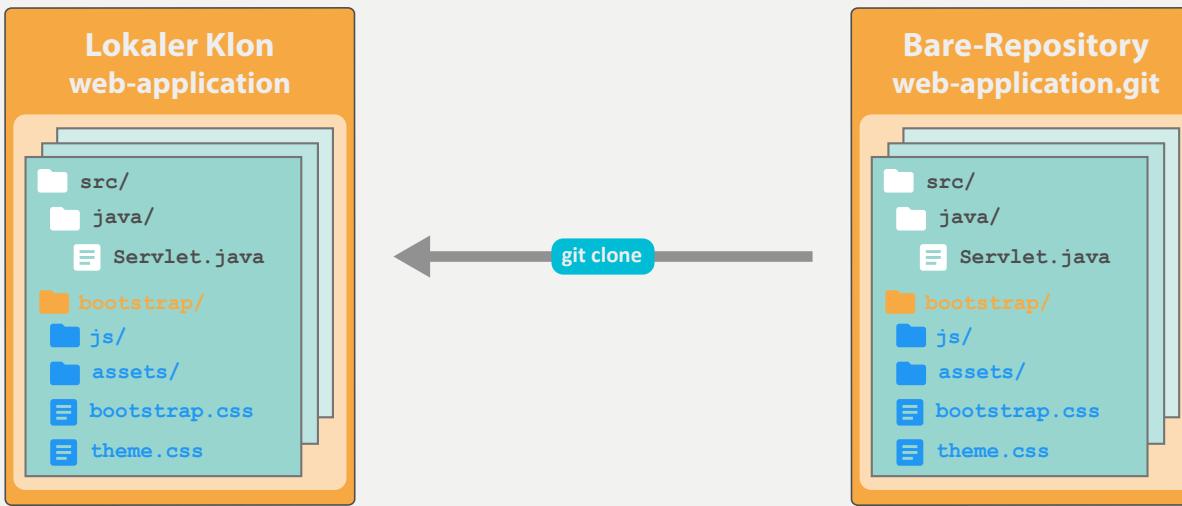


```
web-app$ git subtree add  
--prefix src/main/webapp/bootstrap  
--squash bootstrap.git  
master
```

```
git fetch ../bootstrap.git/ master  
warning: no common commits  
From ../bootstrap  
 * branch master      -> FETCH_HEAD  
Added dir 'src/main/webapp/bootstrap'
```



KLONEN MIT SUBTREES



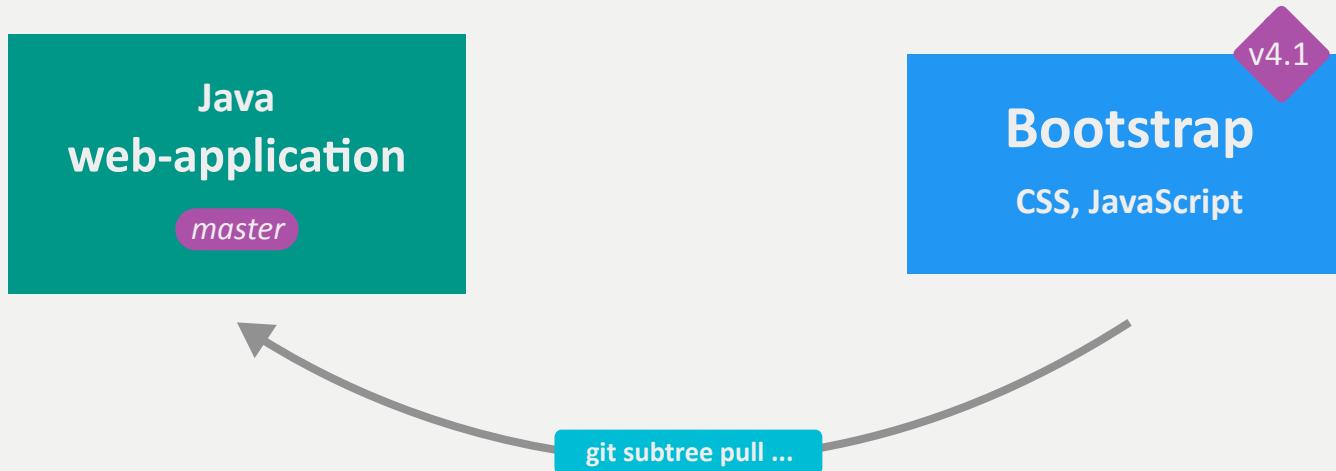
```
web-app$ git clone web-application.git
```

Cloning into 'web-application'...

Done.



NEUE VERSION EINBINDEN



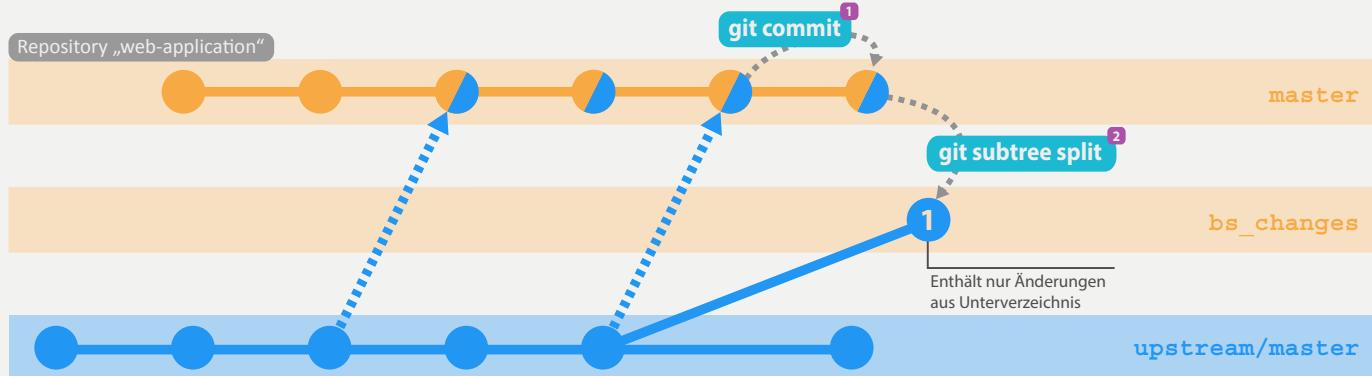
```
web-app$ git subtree pull
--prefix src/main/webapp/bootstrap
--squash bootstrap.git
-m „Version v4.1 eingebunden“
v4.1
```

```
From ../bootstrap
 * tag          v4.1      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
src/main/webapp/bootstrap/dist/css/bootstrap-theme.css | 6 +++++-
1 file changed, 4 insertions(+), 2 deletions(-)
```

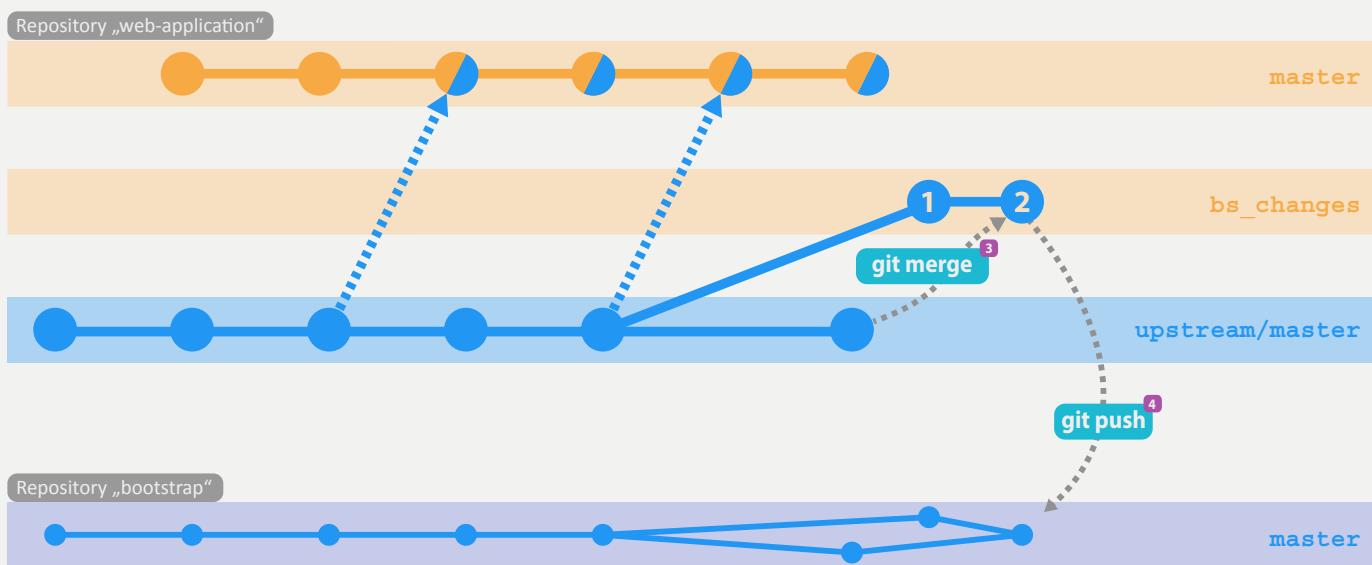


IN SUBTREES ÄNDERN - HINTERGRUND

1. Herauslösen der betreffenden Commits in isolierten Branch



2. Zurückführen in das Original-Repository



IN SUBTREES ÄNDERN (I)



Schritt 1: Dateien lokal ändern und committen

```
web-app$ git commit  
  -m "Eine Änderung im bootstrap-Verzeichnis"  
  src/main/webapp/bootstrap/
```

```
[master d3ca4] Eine Änderung im bootstrap-Verzeichnis  
1 file changed, 1 insertion(+), 1 deletion(-)
```



IN SUBTREES ÄNDERN (II)



Schritt 2: Änderungen extrahieren und in Original-Repository pushen

```
# Branch erzeugen und aktivieren
```

```
web-app$ git subtree split --prefix src/main/webapp/bootstrap/  
--branch bs_changes
```

```
Created branch 'bs_changes'
```

```
web-app$ git checkout bs_changes
```

```
# Remote anlegen, um auf Original-Repository zugreifen zu können
```

```
web-app$ git remote upstream bootstrap.git && git fetch upstream
```

```
# Ggf. Änderungen aus dem Original-Repository mergen
```

```
web-app$ git merge upstream/master
```

```
# Änderungen in Original-Repository pushen
```

```
web-app$ git push upstream HEAD:master
```



Repository: `submodule_subtree/subtree`

1. Klonen Sie das Repository `web-application` („Klon 1“) und fügen dort mittels Subtree das Repository „`bootstrap.git`“ (Tag: v4.1) hinzu (in das Unterverzeichnis `src/main/webapp/bootstrap`)
2. Klonen Sie das Repository `web-application` erneut („Klon 2“) und fügen dort mittels Submodule das Repository „`bootstrap.git`“ (Tag: v4.1) hinzu hinzu (in das Unterverzeichnis `src/main/webapp/bootstrap`)
3. Ändern Sie im Klon 2 eine Datei im `bootstrap`-Verzeichnis, committen Sie diese und schreiben Sie die Änderung zurück in das Original-Repository „`bootstrap.git`“
4. Aktualisieren Sie im Klon 1 das eingebundene Bootstrap-Repository mit dem Commit aus Schritt 3



TEIL 4

GIT APIs: BASH ALTERNATIVEN



libgit2

JGit

Weitere Java APIs

- Portable C Implementierung der Git-Core-Methoden
 - Low-Level-Api
- Wird von GitHub, Microsoft, Plastic SCM unterstützt und benutzt
- Version 0.21.1 ist aktuell.
Version 1.0 sollte bereits vor einem Jahr kommen.
- Es gibt Bindings für verschiedene Sprachen:
 - Rugged – Ruby
 - LibGit2Sharp - .Net
 - Objectivec-git – Objective-C
 - ...
- Bindings sind meistens nicht vollständig und nicht aktuell

```
//Open Repo
git_repository *repo = NULL;
int error = git_repository_open(&repo, ".");

//Convert sha to oid
const char *sha =
"4a202b346bb0fb0db7eff3cffeb3c70babbd2045";
git_oid oid = 0;
int error = git_oid_fromstr(&oid, sha);

//Lookup Commit and Blob
git_commit *commit;
int error = git_commit_lookup(&commit, repo, &oid);

git_blob *blob;
error = git_blob_lookup(&blob, repo, &oid);
```



```
require 'rugged'

#Open Repo
repo = Rugged::Repository.new('.')
Object = repo.lookup('0f4433...')

puts object.message

File.open("foo.txt", 'a+') {
  |f| f.write("next line\n")}

#Work with Index
Index = repo.index
index.add('foo.txt')
index.each { |i| puts i.inspect }
Tree = index.write_tree()
index.write()
```



LIBGIT2 - RUGGED BEISPIEL (2)



```
#Create new Commit
Author = {
  :email=>'rp@etosquare.de',
  :time=>Time.now,
  :name=>'Rene Preisel' }
Parents = [ repo.head.target.oid ]

commit = Rugged::Commit.create(
  repo,
  :author=>author,
  :message=>"Hello world",
  :committer=>author,
  :parents=>parents,
  :tree=>tree,
  :update_ref=>'HEAD' )

puts commit
```



TEIL 4

GIT APIs: BASH ALTERNATIVEN



libgit2

JGit

Weitere Java APIs

JGIT – JAVA GIT IMPLEMENTIERUNG

<http://eclipse.org/jgit>

- Java-Implementierung von Git
- Open Source, EDL
- API und Kommandozeile
- Anwendungen
 - Eclipse (EGit)
 - Jenkins Git Plug-in 2.0
 - Gerrit
 - SmartGit (teilweise)



JGIT - EINBINDEN

- **Download:** <http://www.eclipse.org/jgit/download/>
- **Maven POM:**

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.jgit</groupId>
      <artifactId>org.eclipse.jgit</artifactId>
      <version>3.5.0.201409260305-r</version>
    </dependency>
  </dependencies>
</project>
```



JGIT – KOMMANDOZEILE

Kann als Ersatz von CGit verwendet werden

```
# Ausführung als Bash-Script
```

```
jgit.sh commit -m "Initial Import"
```

```
# Alternativ: Verwendung als executable Jar
```

```
java -jar jgit.sh commit -m „Initial Import“
```



JGIT - PORCELAIN API

Porcelain API

- *High-Level API* für Git „Porcelain“ Kommandos
- Einstieg über die Klasse org.eclipse.jgit.api.Git
- Factory-Methoden für Git Befehle

```
Git git = Git.open(new File("/opt/repos/my-repository"));
```

```
git.checkout()  
    .setName("feature-1")  
    .setStartPoint("master")  
    .call();  
  
git.add()  
    .addFilepattern("readme.txt")  
    .call();  
  
git.commit()  
    .setMessage("Readme hinzugefügt")  
    .call();
```



JGIT – REPOSITORY API

Repository

- Low-Level API zur Arbeit mit dem Repository
- Zugriff auf Object- und Reference-Datenbanken

```
// Repository erzeugen
Repository repository = FileRepositoryBuilder.create(new File("mein-repo/.git"));

// Liefert den aktuellen Branch
String currentBranch = repository.getBranch();

// Referenzen auflösen
ObjectId masterCommit = repository.resolve("refs/heads/master");

// Zugriff auf Konfiguration
StoredConfig config = repository.getConfig();
config.setString("push", null, "default", "simple");
config.save();
```



JGIT – WALKER (I)

RevWalk

- „Iterator“ über eine Menge von Commits
- Erlaubt das Wandern über einen Graphen von Commits

```
// Ausgabe von Commits analog zu „git log --oneline master..feature-1“
RevWalk revWalk = new RevWalk(repository);

revWalk.markStart(revWalk.parseCommit(repository.resolve("feature-1")));
revWalk.markUninteresting(revWalk.parseCommit(repository.resolve("master")));

Iterator<RevCommit> iterator = revWalk.iterator();

while(iterator.hasNext()) {
    RevCommit revCommit = iterator.next();
    System.out.println(
        revCommit.abbreviate(6).name() + " " + revCommit.getShortMessage());
}
```



JGIT – WALKER (II)

TreeWalk

- Wandert über ein oder mehrere Tree-Objekte
- Kann z.B. für diffs verwendet werden

```
// Alle Dateien im Tree des HEAD-Commits ausgeben
TreeWalk treeWalk = new TreeWalk(repository);
ObjectId headTree = repository.resolve("HEAD^{tree}");
treeWalk.addTree(headTree);
treeWalk.setRecursive(true);

while (treeWalk.next()) {
    System.out.printf("%s %s/%s %s%n",
                      treeWalk.getObjectId(0).abbreviate(6).name(),
                      treeWalk.getPathString() , treeWalk.getNameString(),
                      treeWalk.getFileMode(0));
}
}
```



JGIT – OBJEKTE LESEN UND SCHREIBEN

ObjectReader und ObjectInserter

- Low-Level Klassen zum Lesen und Schreiben von Objekten
- Es werden *keine Referenzen* auf die Objekte erzeugt

```
// readme.txt einlesen und auf Stdout ausgeben
ObjectId readmeId = repository.resolve(":readme.txt");
ObjectReader objectReader = repository.newObjectReader();
objectReader.open(readmeId).copyTo(System.out);

// Ein Tag-Objekt schreiben (schreibt nur das OBJEKT, keine REFERENZ)
TagBuilder tagBuilder = new TagBuilder();
tagBuilder.setTag("v2.0");
tagBuilder.setObjectId(repository.resolve("HEAD"), Constants.OBJ_COMMIT);

ObjectInserter inserter = repository.newObjectInserter();
ObjectId tagId = inserter.insert(tagBuilder);
```



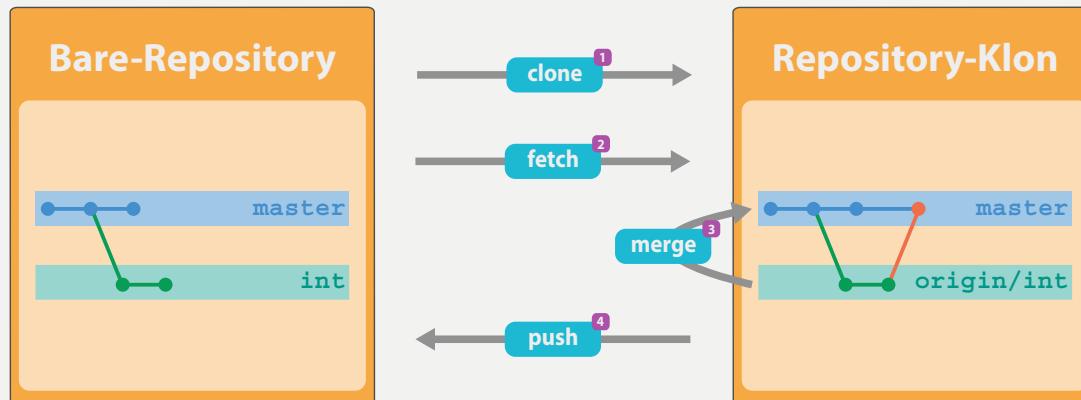


ÜBUNG: EIN MERGETOOL MIT JGIT

Repository/Eclipse-Projekt: api/mergetool

Hinweis: Sie können das Repository „api/mergetool“ als Projekt in Eclipse importieren (File -> Import... -> General -> Existing Projects into Workspace)

- Vervollständigen Sie die vorhandene Klasse **de.e2.mergetool.MergeTool**, so dass alle Änderungen des Integrations-Branches (**int**) in den master-Branch übernommen werden.
- Dazu muss das externe Repository geklont bzw. aktualisiert werden. Dann wird der Merge durchgeführt und am Ende das Ergebnis zurückgeschrieben.
- Nutzen Sie den vorhanden Testcase **de.e2.mergetool.MergeToolTest** als Überprüfung, ob die Implementierung vollständig ist.



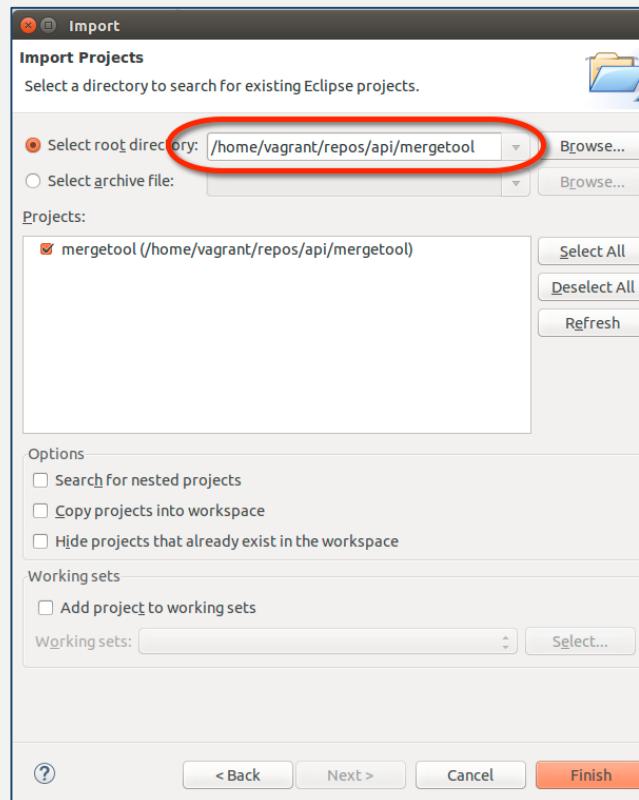


ÜBUNG: EIN MERGETOOL MIT JGIT

Repository/Eclipse-Projekt: api/mergetool

Importieren des Projektes in den Eclipse Workspace:

File -> Import... -> General -> Existing Projects into Workspace



TEIL 4

GIT APIs: BASH ALTERNATIVEN



libgit2

JGit

Weitere Java APIs

JGIT ADD-ON: GROOVY GRGIT

Groovy API für JGit

- <https://github.com/ajoberstar/grgit>
- Groovy-like Wrapper um JGit
- Unterstützt nicht alle JGit Features
- Basis für Gradle Git Plug-in

```
def grgit = Grgit.open('path/to/my/repo')
grgit.checkout(branch: 'feature-1')

grgit.add(patterns: ['readme.txt'])
grgit.commit(message: 'Initial Import')
```



JGIT ADD-ON: ATlassian GITFLOW API

GitFlow Implementierung für Java

- <https://bitbucket.org/atlassian/jgit-flow>
- Basiert auf JGit
- Unterstützt auch das Remote-Repository

```
JGitFlow gitFlow = JGitFlow.getOrInit(new File("/my/repository/dir"));
// Feature beginnen
gitFlow.featureStart("feature-1").setPush(true).call();

// [JGit] Dateien ändern, hinzufügen, löschen
gitFlow.git().add().addFilepattern(".").call();
gitFlow.git().commit().setMessage("...").call();

// Feature abschliessen
gitFlow.featureFinish("feature-1")
    .setKeepBranch(true)
    .setMessage("Feature 1 beendet")
    .call();
```



GITHUB REST API

Java-API für GitHub REST Schnittstelle

- <https://github.com/eclipse/egit-github/>
- Zugriff auf GitHub und GitHub Enterprise
- Unterstützung für die meisten GitHub-Aktionen

```
GitHubClient gitHubClient = new GitHubClient();
gitHubClient.setCredentials("me", "secret");
```

```
RepositoryId repository = new RepositoryId("username", "repository");
```

```
Tag tag = new Tag();
tag.setMessage("Release 1.0");
tag.setTag("v1.0");
tag.setSha("a4ff3bcd1");
```

```
DataService dataService = new DataService(gitHubClient);
dataService.createTag(repository, tag);
```



TEIL 5

BUILD-WERKZEUGE



Gradle

Maven – Release Plug-in

Maven – JGit Flow Plug-in

GRADLE UND GIT

- Verschiedene Plugins:
- **townsfolk/gradle-release**
 - Basiert auf der Git-Kommandozeile
 - Orientiert sich am Maven-Plugin
 - Versionen werden in der gradle.properties gehalten und selber versioniert
- **ajoberstar/grgit**
 - Basiert auf JGit
 - Reine Groovy-API um Git-Kommandos auszuführen, keine Tasks
- **ajoberstar/gradle-git**
 - Basiert auf ajoberstar/grgit / Jgit
 - Release-Task zum Taggen von Releases
 - Versionen werden aus vorhanden Tags und Kommando-Parameter ermittelt (Semantische Versionen)



TOWNSFOLK / GRADLE-RELEASE

- Überprüft ob:
 - das Repo auf dem richtigen Branch ist
 - keine unversionierten Dateien vorhanden sind
 - es keine weiteren Änderungen auf dem Remote-Tracking-Branch gibt
 - alle lokalen Commits „ge-pushed“ wurden
 - keine SNAPSHOT-Abhängigkeiten vorhanden sind
- Erzeugt ein neues Commit mit einer „nicht“-Snapshot-Version
- Erzeugt ein Release-Tag
- Erzeugt ein neues Commit mit der nächsten (Snapshot-)Version
- Kann durch eigene Tasks angepasst werden



AJOBERSTAR / GRADLE-GIT

- Erzeugt Versions-Label die sich an „Semantic Versioning“ halten
- Versionen werden anhand
 - der **Vorgängerversion**,
 - des **Scopes** (Major, Minor Patch) und
 - definierten **Stages** (dev, rc, milestone, **final**) ermittelt
- gradle release -Prelease.scope=major -Prelease.stage=final



TEIL 5

BUILD-WERKZEUGE



Gradle

Maven – Release Plug-in

Maven – JGit Flow Plug-in

MAVEN-RELEASE-PLUGIN

<http://maven.apache.org/maven-release/maven-release-plugin/>

Ziel: Automatisierung vieler (Maven-)Aufgaben

- Aktualisiert die POMs
- Erzeugt Tags
- Führt Build und Tests aus
- Installiert die Maven-Artefakte im Maven-Repository

- Nicht Git-spezifisch
- Kein Release-Prozess im Sinne von Git-Flow



MAVEN-RELEASE-PLUGIN – KONFIGURATION 1

Konfiguration der Git- und Maven-Repositories

- Git-Repository kann vom „origin“ abweichen
- In das Maven-Repository werden die fertigen Artefakte installiert

```
<project ...>
  <scm>
    <developerConnection>scm:git:GIT_REPO_URL</developerConnection>
  </scm>

  <distributionManagement>
    <repository>
      <id>maven-repository</id>
      <url>http://mycompany.com/my/maven/centralrepo</url>
    </repository>
  </distributionManagement>
</project>
```



MAVEN-RELEASE-PLUGIN – KONFIGURATION 2

Einbinden des Plug-ins

- Versionsnummer erforderlich
- Optional: Format des Tags

```
<project ...>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.5</version>
        <configuration>
          <tagNameFormat>v@{project.version}</tagNameFormat>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```



MAVEN-RELEASE-PLUGIN – SCHRITT 1

mvn [--batch-mode] release:prepare

- Aktualisiert und committet das POM mit Release-Version
 - Erzeugt ein Release-Tag im Repository
 - Aktualisiert das POM mit nächster SNAPSHOT-Version
-
- Alle Änderungen werden ins konfigurierte Repository gepusht
 - Hinterlässt temporäre Dateien für Schritt 2



MAVEN-RELEASE-PLUGIN – SCHRITT 2

`mvn release:perform`

- Klont das Repository in temp-Verzeichnis
- Ruft darin 2. Maven auf
- Führt erneut Build + Tests aus
- Installiert die Artefakte im zentralen Maven-Repository



MAVEN-RELEASE-PLUGIN - ANWENDUNG

`mvn release:rollback`

- Verwirft die in der Prepare-Phase gemachten Änderungen
- Erzeugt ein „Rollback-Commit“
- Änderungen bleiben im Repository



TEIL 5

BUILD-WERKZEUGE



Gradle

Maven – Release Plug-in

Maven – JGit Flow Plug-in

ATLASSIAN JGIT-FLOW

<https://bitbucket.org/atlassian/jgit-flow/wiki/Home>

Ziel: GitFlow-Unterstützung für Maven

- Unterstützung für Entwicklungs- und Releaseprozess
 - Feature
 - Release
 - Hotfix
- Mehr als nur POM-Pflege
- Git-spezifisch
- Gute Unterstützung von Remote-Repositories



ATLASSIAN JGIT-FLOW – KONFIGURATION 1

Konfiguration des Maven-Repositories

- Als Git-Repository wird das „origin“ verwendet
- Für das Deployment muss Maven-Repository konfiguriert werden

```
<project ...>
  <distributionManagement>
    <repository>
      <id>maven-repository</id>
      <url>http://mycompany.com/my/maven/centralrepo</url>
    </repository>
  </distributionManagement>
</project>
```



ATLASSIAN JGIT-FLOW – KONFIGURATION 2

Einbinden des JGit-Flow Plug-ins

- Zahlreiche Konfigurationsmöglichkeiten

```
<project ...>
  <plugins>
    <plugin>
      <groupId>external.atlassian.jgitflow</groupId>
      <artifactId>jgitflow-maven-plugin</artifactId>
      <version>1.0-m4.3</version>
      <configuration>
        <pushFeatures>true</pushFeatures>
        <pushReleases>true</pushReleases>
      </configuration>
    </plugin>
  </plugins>
</project>
```



ATLASSIAN JGIT-FLOW - ANWENDUNG

`mvn jgitflow:feature-start`

- Erzeugt einen neuen `feature`-Branch

`mvn jgitflow:feature-finish`

- Führt Build und Tests auf `feature`-Branch aus
- Installiert Artefakte (vom `feature`-Branch ?!)
- Mergt Änderungen auf `develop`-Branch



ATLASSIAN JGIT-FLOW - ANWENDUNG

`mvn jgitflow:release-start`

- Erzeugt einen neuen `release`-Branch
- Aktualisiert POM auf `develop`-Branch für nächstes Release

`mvn jgitflow:release-finish`

- Führt Build und Tests auf `release`-Branch aus
- Aktualisiert POM auf `release`-Branch
- Mergt Änderungen auf `master`- und `develop`-Branch
- Deployed Maven Artefakte in zentrales Repository



VIELEN DANK!

NOCH FRAGEN?



`git help cmd`
`git cmd --help`
`man git-cmd`

`rene.preissel@etosquare.de`

`nils@nilshartmann.net`



COPYRIGHT



Licensed under the Creative Commons Attribution 3.0
Unported License:



Git Logo

<http://git-scm.com/downloads/logos>

- by Jason Long



Icons

<http://shreyasachar.github.io/AndroidAssetStudio/>

- by shreyasachar

