

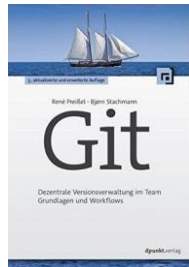
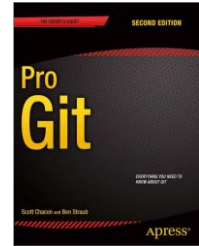
Es gibt verschiedenste Tools zur Unterstützung der Konfigurationsprozesse.

Hier werden einige Open Source Tools vorgestellt, die weit verbreitet sind:


- **Subversion, GIT** - Versionskontrolle
- **Maven** - Build Prozess
- **Hudson** - Continuous Integration
- **Redmine** - Kollaboration

Literatur:

- Chacon, Straub: Pro Git, <https://git-scm.com/book/de/v2>
- Preißel, Stachmann; Git Dezentrale Versionsverwaltung im Team Grundlagen und workflows, dpunkt.verlag, 3. Auflage 2016
- <https://git-scm.com/>

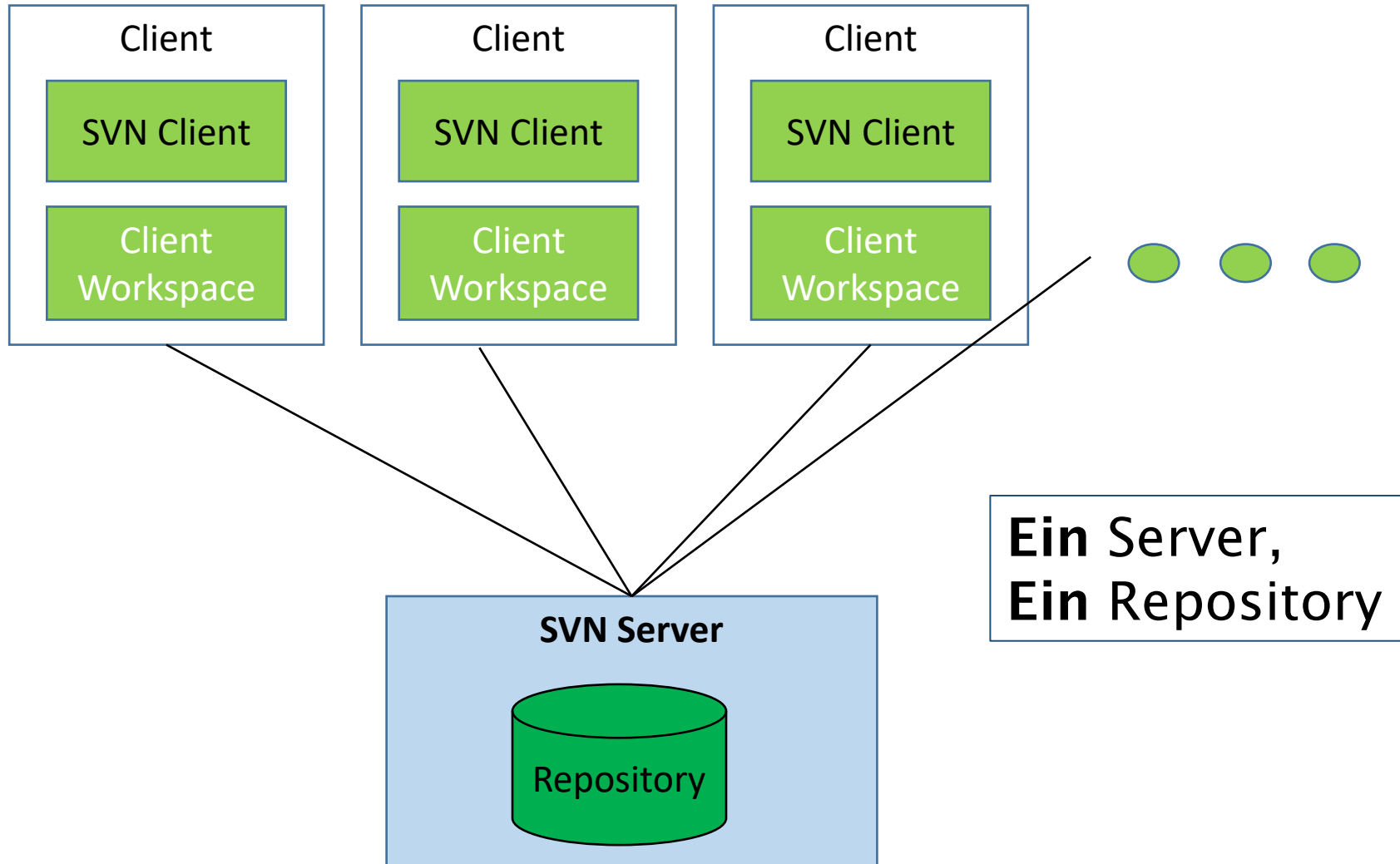


Agenda

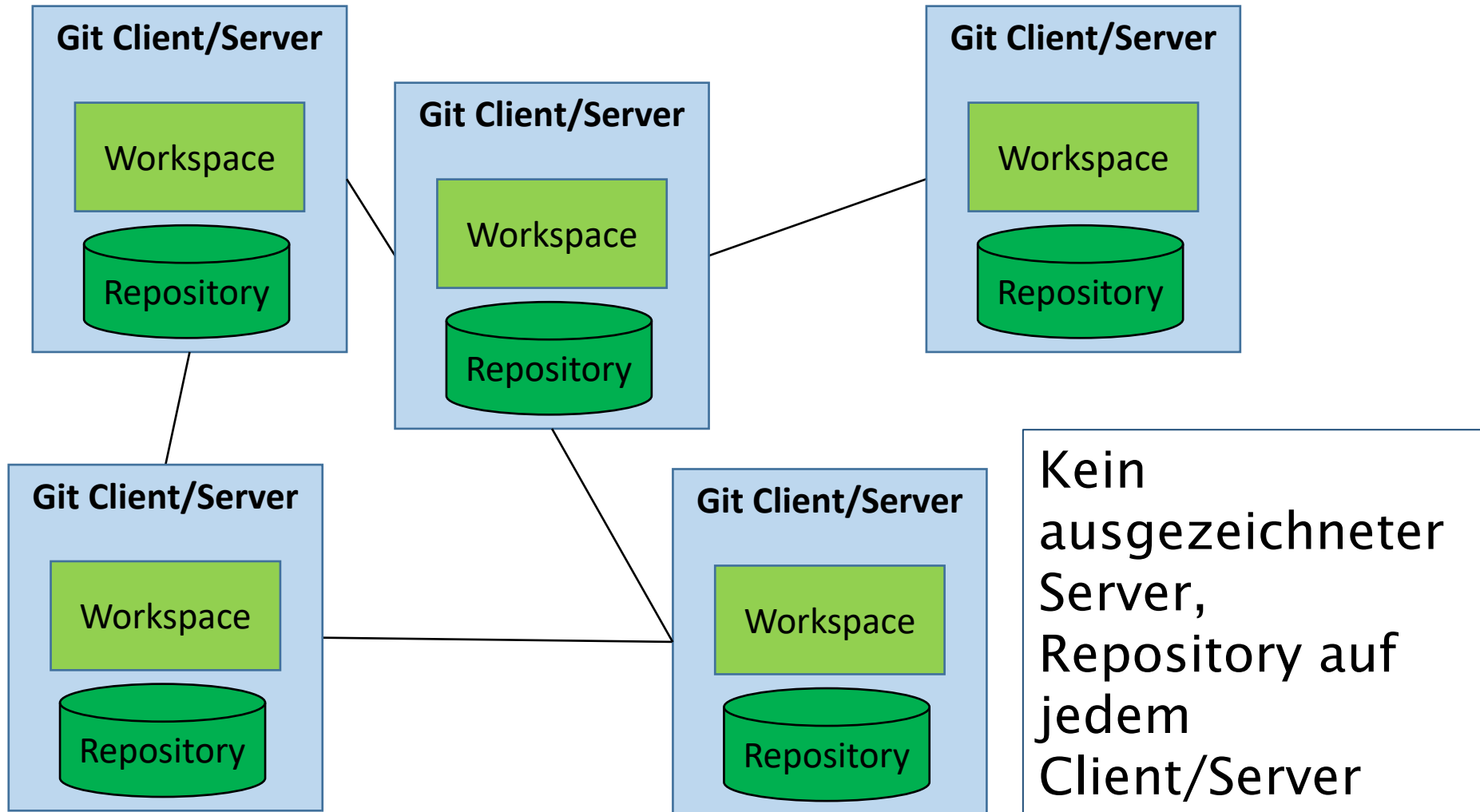
- 
- Einleitung
 - Erste Schritte
 - Commits
 - Projekthistorie
 - Staging
 - Branching
 - Merging
 - Rebase
 - Repositories
 - Austausch zw repositories
 - workflow

- GIT ist sehr weit verbreitet.
- Löst zunehmend SVN ab. Neue Projekte speziell im Open Source Bereich nutzen häufig GIT (unterstützt auch durch Plattformen wie Github).

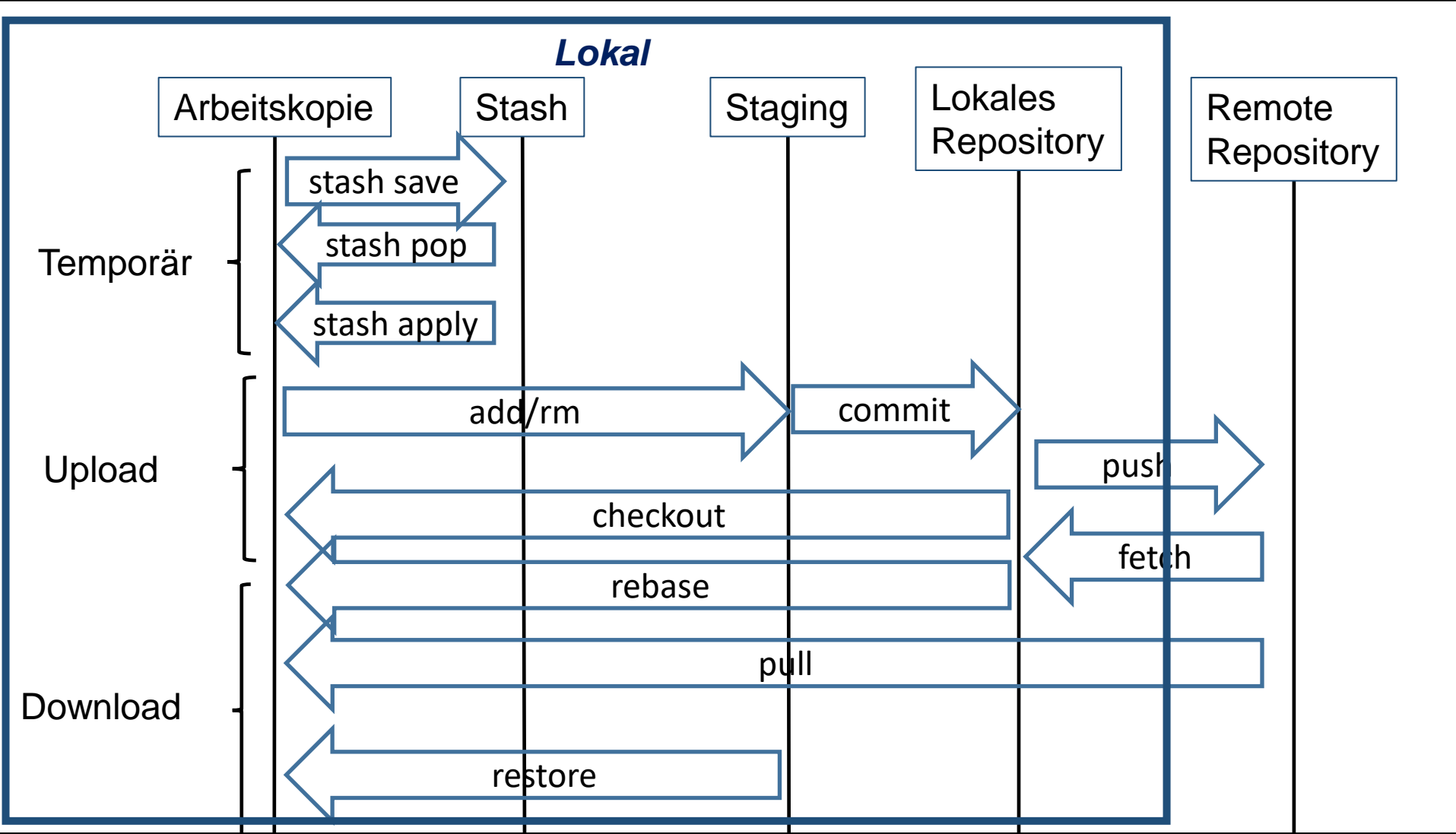
Erinnerung: SVN als zentrale Versionsverwaltung



Git als dezentrale Versionsverwaltung



Git Übergänge



Laut Preißel/Stachmann Vorteile der dezentralen Versionsverwaltung:

- Hohe Performance
- Effiziente Arbeitsweisen
- Offline Fähigkeiten
- Flexibilität der Entwicklungsprozesse
- Backup
- Wartbarkeit

Agenda

- Einleitung
-  ▪ Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

Voraussetzungen:

- GIT installieren → siehe unter <https://www.git-scm.com/downloads>
- Benutzernamen eintragen
 - *git config --global user.name „mustermann“*
- Email Adresse eintragen
 - *git config --global user.email „lena@mustermann.de“*

Erstes Projekt:

1. Projektfolder mit Verzeichnissen und Dateien anlegen.
2. Repository anlegen:
 1. Im Projektfolder *git init*
→ *repository unter <projektfolder>/.**git* (noch leer, eventuell in Windows nicht sichtbar)

▪ Erstes Commit

- Erstmal hinzufügen, dh. unter Sourceverwaltung stellen:
 - *git add <dateiname> <dateiname> ...*
- Dann ins Repository übertragen
 - *git commit -message „Beispielprojekt importiert“*

▪ Status abfragen

- Mit *git status* erhalten Sie Informationen über alle Änderungen seit dem letzten Commit (auf Ihrem Computer)

▪ Commit nach Änderungen

- Änderungen anmelden: *git add* oder *git rm*
- Änderungen ins Repository commiten

▪ Historie betrachten

- *git log*

Bisher: Alles auf dem lokalen Rechner, keine Zusammenarbeit.

Jetzt: Zusammenarbeit mit git

Motivation: Austausch zwischen Workspaces verschiedener Entwickler

Zum Vergleich: SVN hat nur ein einziges zentrales Repository, das zum Austausch dient.

Repository klonen

Befehl *git clone <quelle> <ziel>*

- Kopieren des Quellrepositorys
- Checkout des aktuellen heads in den Arbeitsbereich
- Link zur Quelle speichern

- Repository klonen
 - *git clone <quelle> <ziel>*
- Änderungen aus einem anderen Repository holen
 - *git fetch*: holt die Änderungen aus dem Repository, von dem es geklont wurde, in das lokale repository. Keine Änderungen im workspace.
 - *git pull <quelle><branch>*
git holt die Änderungen aus dem Repository, von dem es geklont wurde, in den lokalen Arbeitsbereich. Vorsicht!
- Änderungen in ein anderes Repository schreiben
 - Typischerweise in ein Repository nur für den Austausch (bare repository ohne workspace – erzeugt durch Option – bare beim Klonen)
 - *git push <ziel><branch>*

Agenda

- Einleitung
- Erste Schritte
-  ▪ Commits
- Projekthistorie
- Staging
- Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

Commit

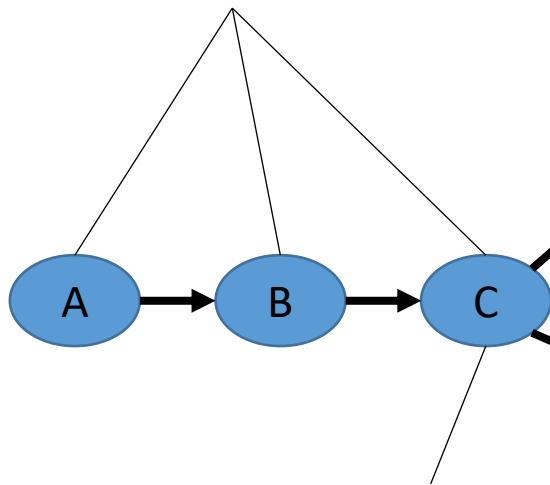
- git verwaltet Versionen von Software.
- Jede Version wird als Commit in einem Repository abgelegt.
- Ein Commit umfasst immer das ganze Projekt.
- Mit jedem Commit wird für jede einzelne Datei im Projekt eine Kopie im Repository angelegt (logisch).

Commit Hash

- Für jedes Commit berechnet Git einen 40stelligen Commit Hash
 - Erinnerung: in SVN: fortlaufende Nummern
 - Die Hashes können lokal erzeugt werden.
 - Die Hashes dienen als Prüfsummen.
-
- Wiederherstellen einer Version in Git:
„Checkout“

Historie von Commits (logisch)

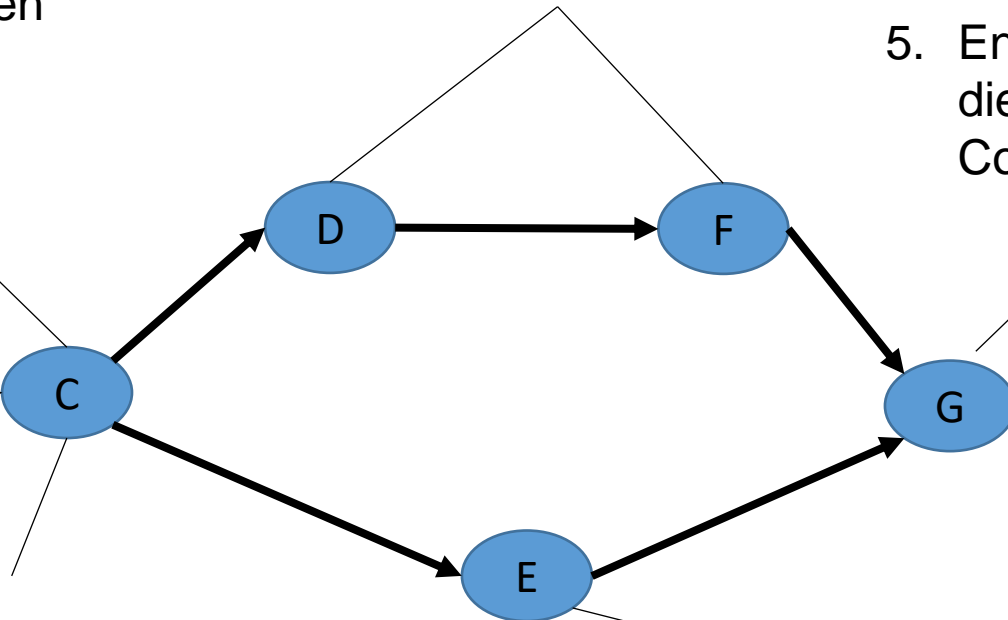
1. Entwickler 1 beginnt Projekt mit eigenen Commits



2. Entwickler 2 beginnt am Projekt zu arbeiten

3. Entwickler 1 arbeitet weiter

5. Entwickler 1 führt die Zweige in einem Commit zusammen

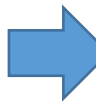


4. Entwickler 2 behebt einen Fehler

- Commits entsprechen Versionsständen.
- Unterschied zum Vorgängercommit:
Changeset

git diff ermöglicht, Unterschiede zwischen beliebigen Commits anzuzeigen.

Agenda

- Einleitung
- Erste Schritte
- Commits
-  ▪ Projekthistorie
- Staging
- Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

In git sind die Historien lokal
git log Befehl hilft, die (lokale) Historie anzusehen.

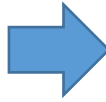
→ Frage: Was ist die offizielle Historie des Projekts?

Lösung häufig: Einrichtung eines per Konvention ausgezeichneten Repositories, das die offizielle Historie enthält. Die lokalen Historien werden per merge Befehl zusammengeführt.

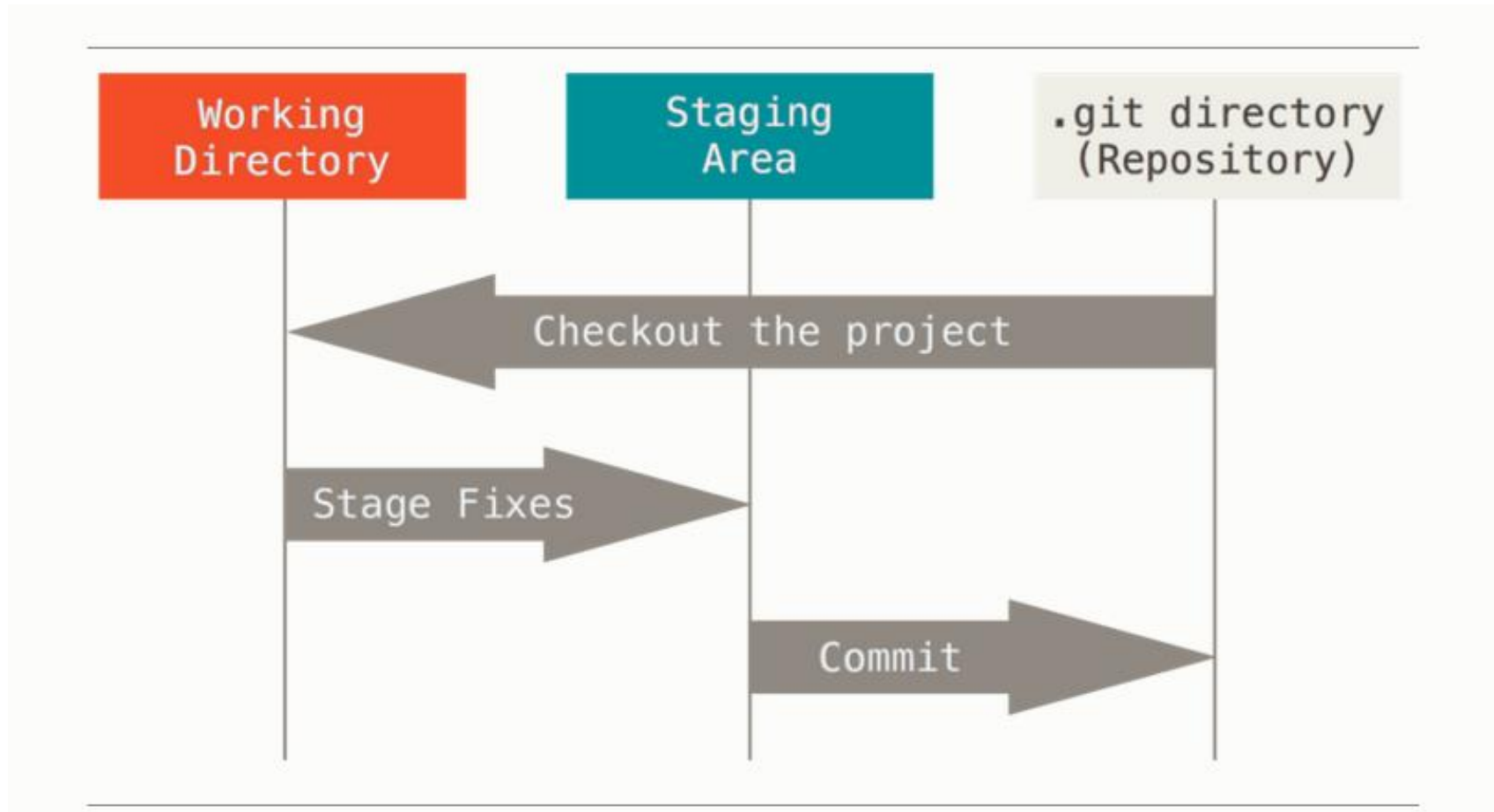
Hinweis:

Durch die Einrichtung eines blessed Repositories führen Sie logisch wieder eine zentrale Architektur ein.

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
-  ▪ Staging
- Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

Lokales Staging



Quelle: Chacon, Straub: Pro Git

Das commit in das lokale repository ist ein zweistufiger Prozess:

1. In den Stage Bereich schreiben
2. Von dort ins repository schreiben

➔ Unterschied zwischen Workspace und Staging Area möglich.

➔ Detaillierte Definition, was auch wirklich mit einem Commit ins repository geschrieben werden soll.

Git status

Mit dem *git status* Befehl sieht man sich die Änderungen an, d.h.

- welche Files sind im Staging Bereich,
- Welche sind verändert, aber noch nicht in den Staging Bereich überführt.
- Welche Dateien sind neu oder gelöscht?

Hinweis: Im Staging Bereich liegen wirklich Snapshots, d.h. die selbe Datei kann in verschiedenen Bearbeitungsständen im Workspace und im Staging Bereich liegen.

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
-  ▪ Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

Branching

Branches: Verschiedene Entwicklungszweige

Gründe:

- Parallel Entwicklung verschiedener Produktlinien/Releases
→ ebenso wie bei svn
- Speziell bei Git aufgrund der dezentralen Architektur:
Mehrere Entwickler

Unterschied: Im Fall 1 werden zwei verschiedene Ergebnisse ausgeliefert, im Fall zwei werden die branches immer vor der Auslieferung gemergt.

Pro Git Repository immer genau ein aktiver branch

Anzeigen der branches im Repository: *git branch*
Der aktive branch ist mit * gekennzeichnet.


- Commit geht immer auf den aktiven branch
- Mit *git checkout* kann man den aktiven branch wechseln. Dadurch werden die Dateien im Workspace ausgetauscht.

Explizit branches in Git erstellen mithilfe des branch Befehls:

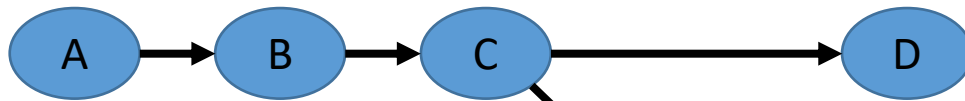
- Branches vom aktuellen Commit
- Branches von einem beliebigen vorhandenen Commit
- Branches von beliebigen Zweig

Wechseln zu einem anderen branch mit *checkout*.

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
-  ▪ Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow

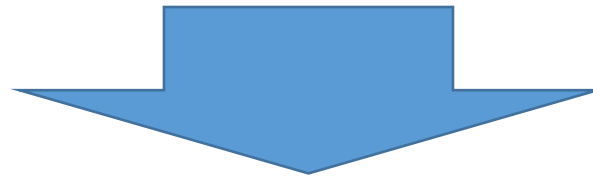
Merge in den master



master



feature



master




feature

Wie läuft der merge?

- ➔ Zunächst automatisch: Zusammenführen und **automatisch** commit
- ➔ Bei nicht auflösbaren Konflikten
 - ➔ automatisch gemergte Dateien im Workspace und im Staging Bereich
 - ➔ Konfliktstellen markiert nur im Workspace

Manuell mergen – von git unterstützt.

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
- Merging
- ▪ Rebase
- Repositories
- Austausch zw repositories
- workflow

Rebasing → Historie glätten indem man einen branch an einem anderen Knoten ansetzen lässt.

→ Vermeiden von merges

Wann?

- Commits versehentlich im falschen branch ausgeführt
- Mehrere Entwickler an der gleichen Software
→ viele kleine Verzweigungen → glätten zu einer linearen Historie

Prinzip:

Git nimmt eine Folge von Commits und spielt sie auf dem Zielbranch genau so ein.

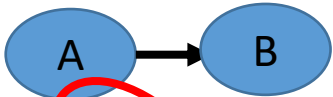
➔ Rebasing erzeugt immer neue Commits

➔ Ggfs entstehen dadurch Konflikte, die genau wie merge Konflikte gelöst werden müssen.

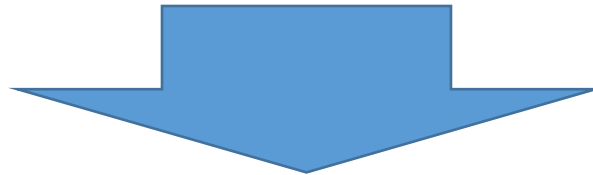
Nicht für bereits gepushte branches machen!

Bsp für rebase

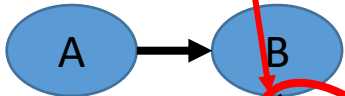
master



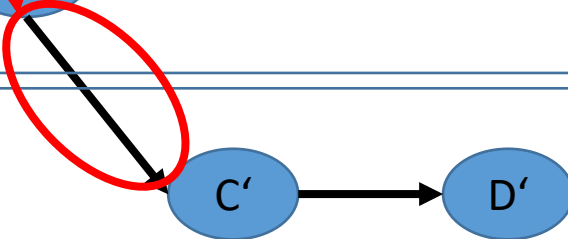
feature



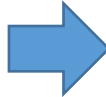
master



feature




Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
- Merging
- Rebase
-  ▪ Repositories
- Austausch zw repositories
- workflow

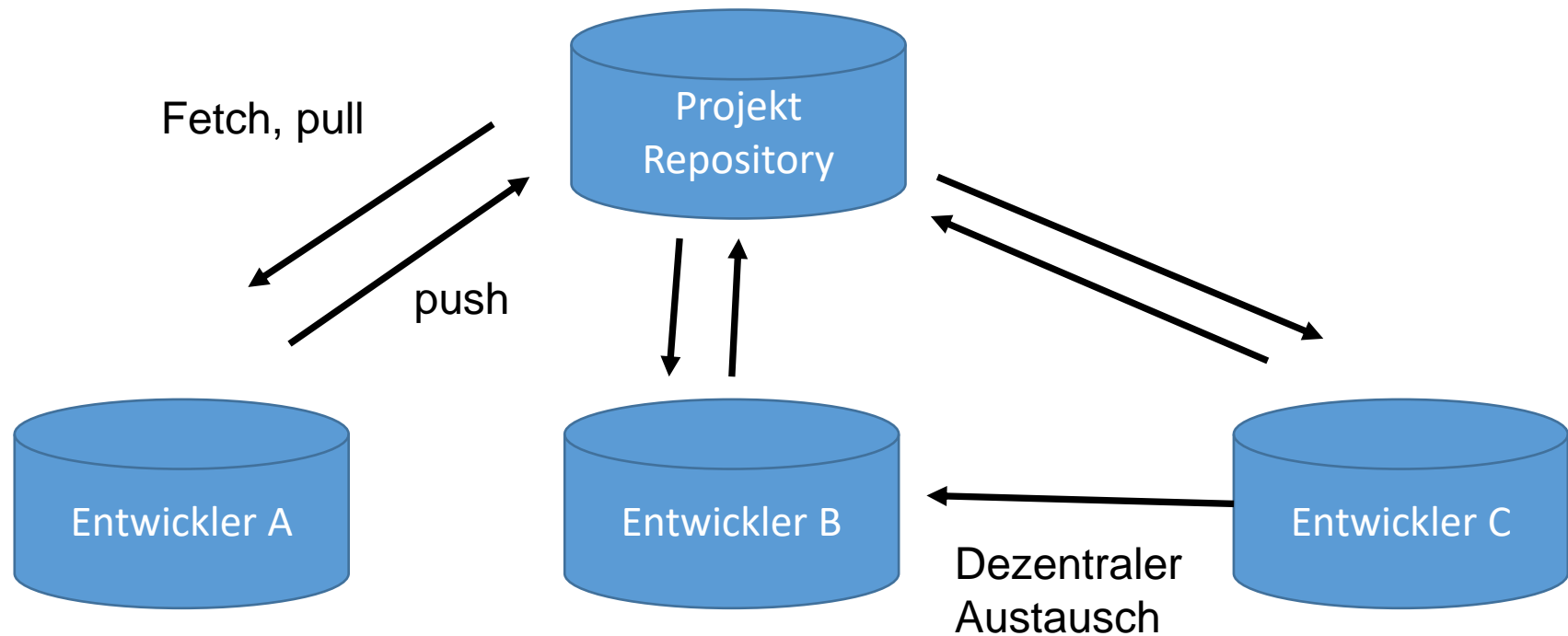
- Anlegen: *git init*
 - Kein Commit damit verbunden!
- Klonen: *git clone*
 - Jeder Entwickler hat einen Klon
 - Multisite-Entwicklung: Jede Site hat einen Hauptklon
 - Bestimmte Workflows: Fork
- Bare Repositories: Repository, kein Workspace

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
- Merging
- Rebase
- Repositories
-  ▪ Austausch zw repositories
- workflow

Austausch zwischen Repositories

Austausch via fetch, pull und push



In Git hat jeder Entwickler seinen lokalen Satz von Branches:

- Eigene Commits,
- Eigene branches
- Branches lokal löschen

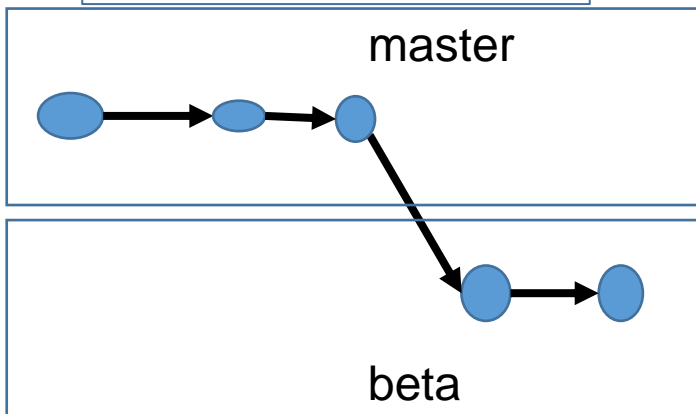
➔ die Klone entwickeln sich auseinander

Lokalen Stand mit dem Stand eines anderen Repositories vergleichen ➔ Remote-Tracking-Branches

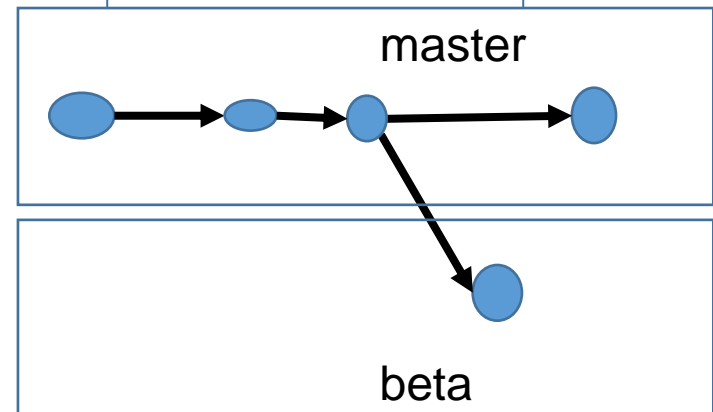
Remote Tracking Branches

Lokale Stellvertreter für Branches aus anderen Repositories

Entwickler-Repository

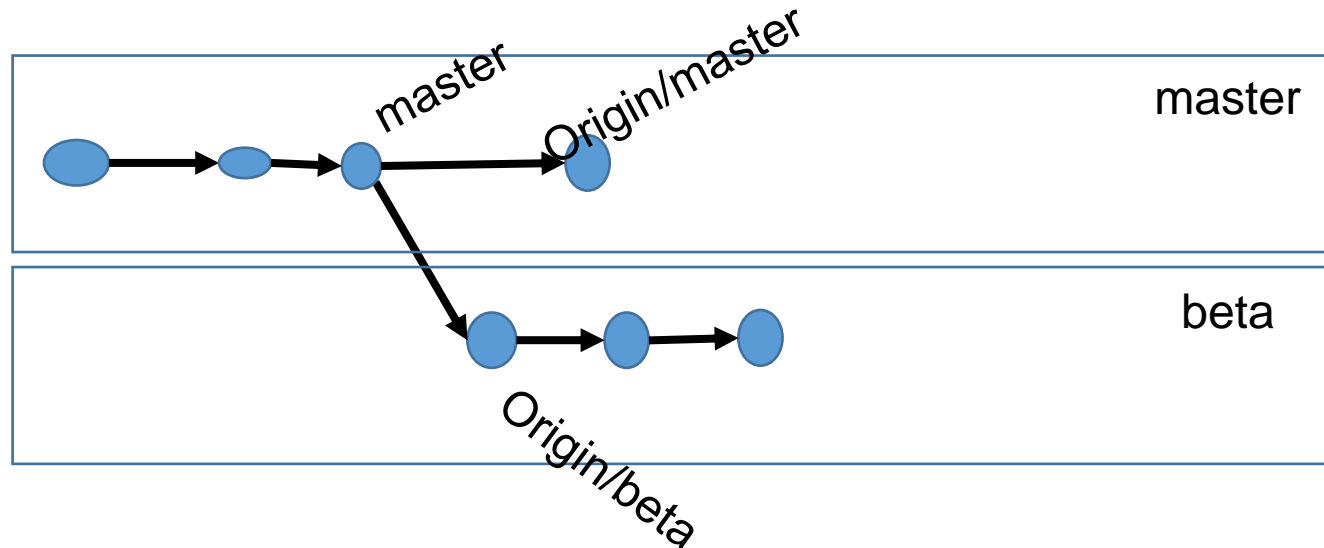


Origin-Repository



Remote tracking Branches

Remote Tracking branches im Entwickler Repository



Die Remote-Tracking-Banches zeigen an, wo die Branches master und beta im origin Repository stehen

Remote-Tracking-Branches

Die Remote-Tracking-Branches werden beim Aufruf von fetch und pull aktualisiert.

Namen der Remote Tracking Branches:

<Herkunfts Repository>/<branch Name>

Ein Remote Tracking Branch zeigt auf das commit, auf das der OriginalBranch im anderen Repository bei der letzten Aktualisierung gezeigt hat.

Nicht auf den Remote Tracking Branches arbeiten!

Begriffe:

- **Branch:** branch im lokalen repository
- **Remote branch:** branch in einem anderen repository
- **Remote tracking branch:** spezieller lokaler branch, der anzeigt, wo ein remote branch zum Zeitpunkt des letzten updates stand
- **Upstream Branch:** Verknüpfung zwischen einem lokalen und einem Remote-Tracking-Branch

Fetch, Pull, Push

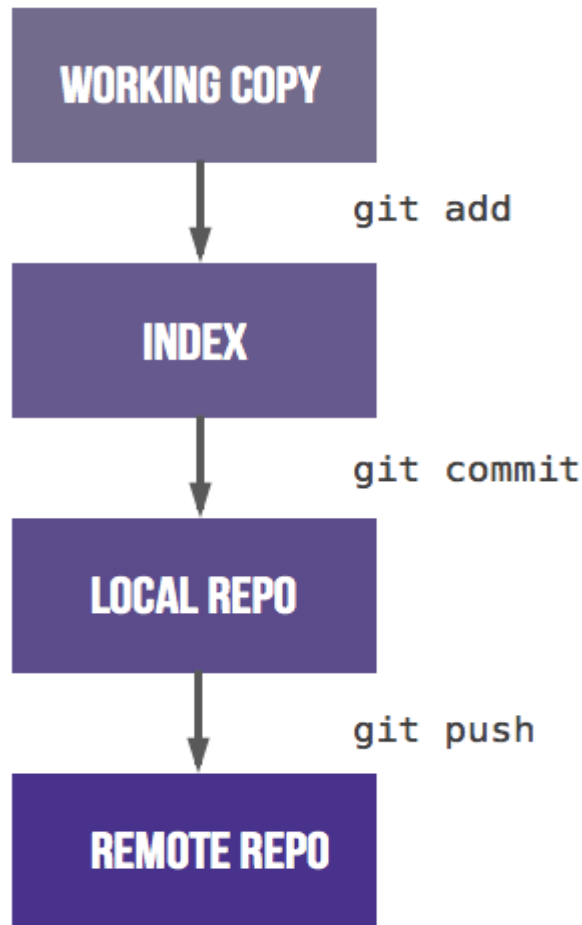
- Fetch: Branches aus einem anderen Repository holen
- Pull: Fetch + Merge
- Push: Änderungen veröffentlichen

Agenda

- Einleitung
- Erste Schritte
- Commits
- Projekthistorie
- Staging
- Branching
- Merging
- Rebase
- Repositories
- Austausch zw repositories
- workflow



Git Flow – Prinzip des Austauschs



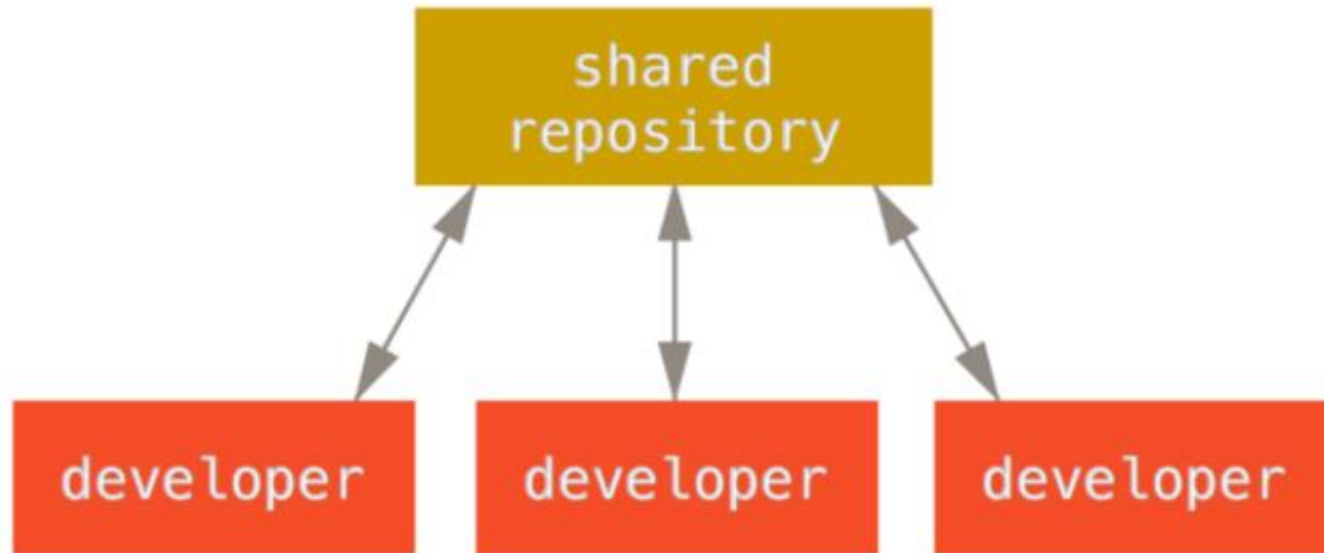
3 Schritte (add, commit, push)

svn: 1 Schritt (commit)

Grafik aus
https://docs.gitlab.com/ee/topics/gitlab_flow.html

Zentralisierter Arbeitsablauf

Analog zu svn



1 zentrales Repository

Alle Developer pushen ihre Änderungen auf das zentrale repository

Ggfs. nachdem sie vorher die zwischenzeitlichen Änderungen bei sich gemergt haben.

Branching

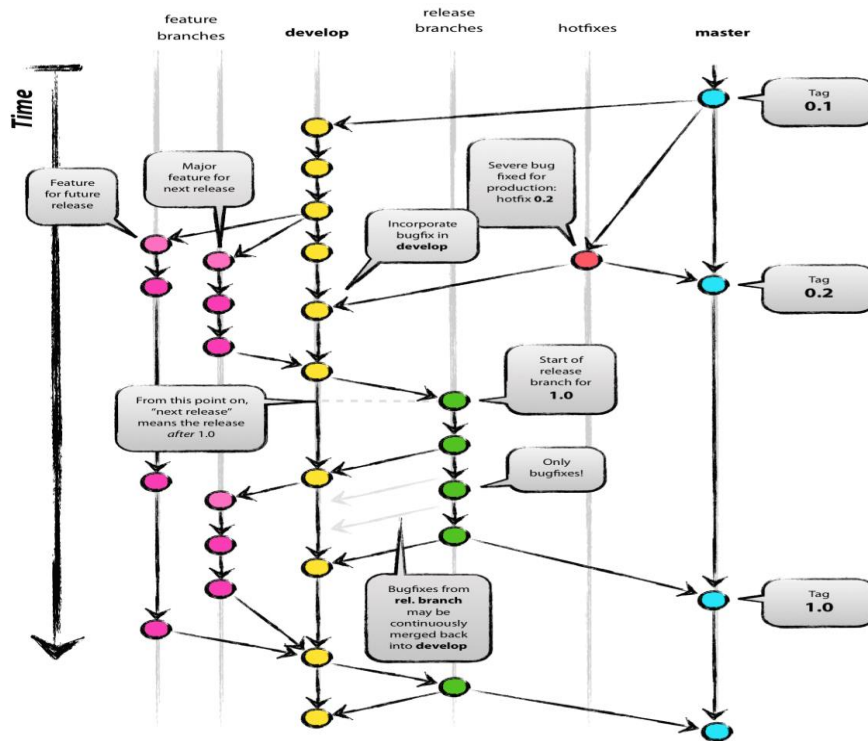
Ohne Konventionen besteht die Gefahr, dass das verwendete Branching Modell unübersichtlich und schwer handhabbar wird.

→ Es gibt einige Standard Pattern für das verwendete Branching Konzept

- Git Flow <https://de.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>
- Github Flow <https://guides.github.com/introduction/flow/index.html>
- Gitlab Flow https://docs.gitlab.com/ee/workflow/gitlab_flow.html

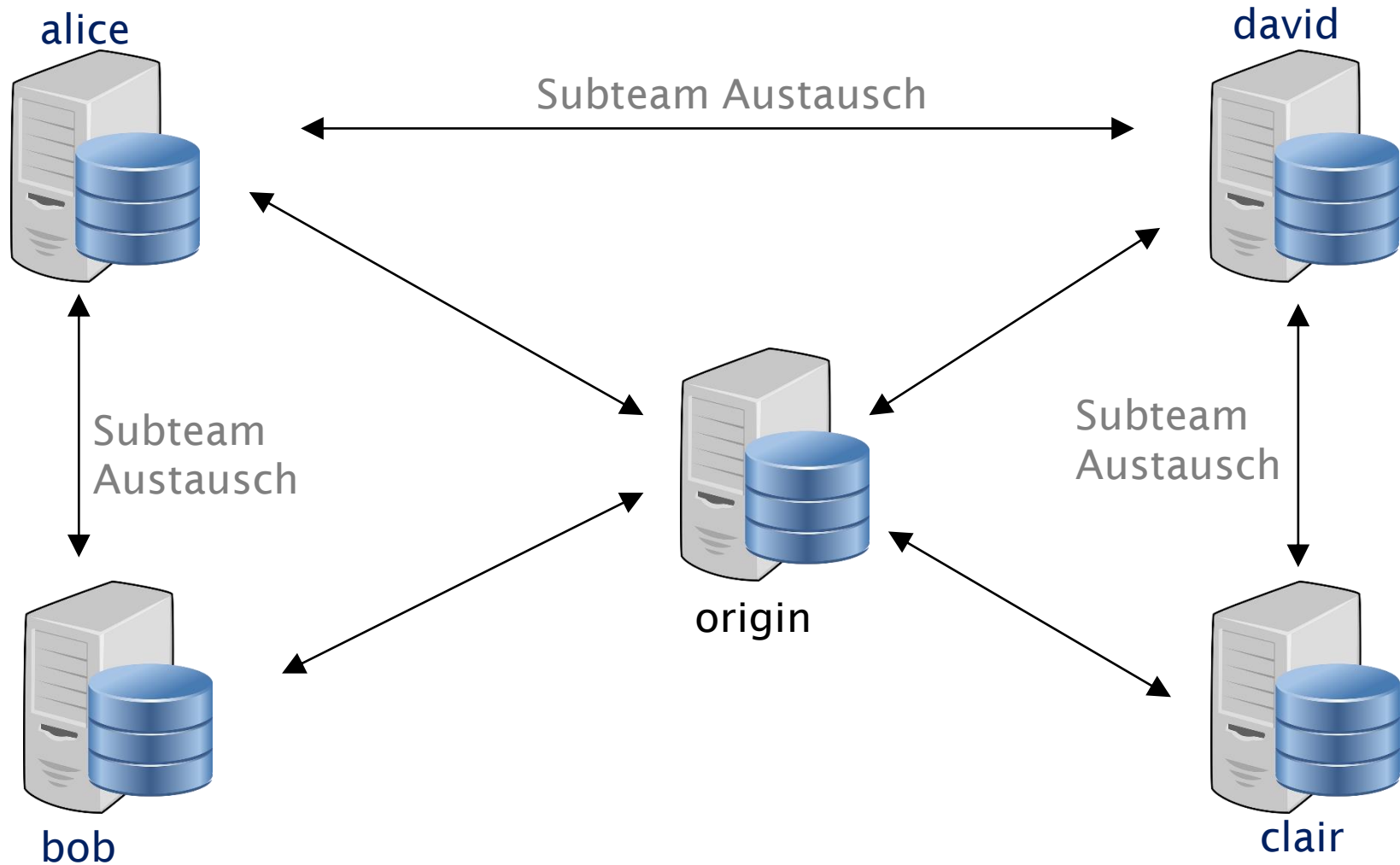
Git Flow

- <https://nvie.com/posts/a-successful-git-branching-model/>
- <https://www.atlassian.com/de/git/tutorials/comparing-workflows/gitflow-workflow>



Grafik aus
<https://nvie.com/posts/a-successful-git-branching-model/>

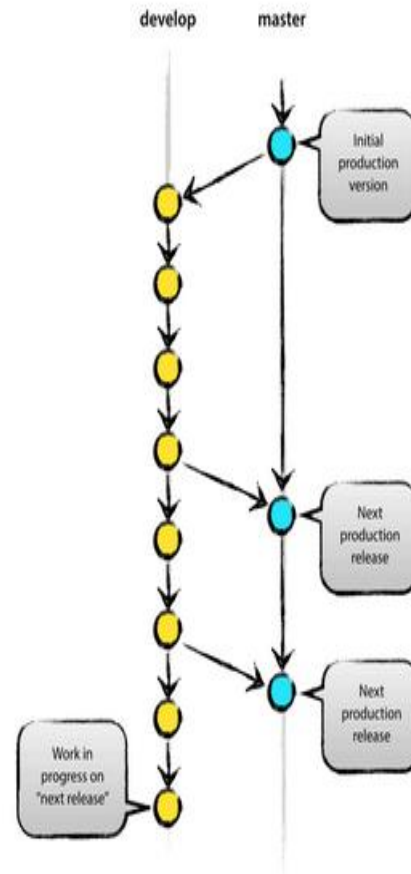
Ein ausgezeichnetes Repository



Branches in zentralen Repository

Im zentralen Repository existieren 2 branches:

- **master:**
Der Quellcode von Head hält produktionsreifen Code.
- **Develop:**
Der Quellcode von Head hält die aktuellsten Lieferungen für das nächste release. Das ist der Integration branch, der für die nächtlichen builds verwendet wird.



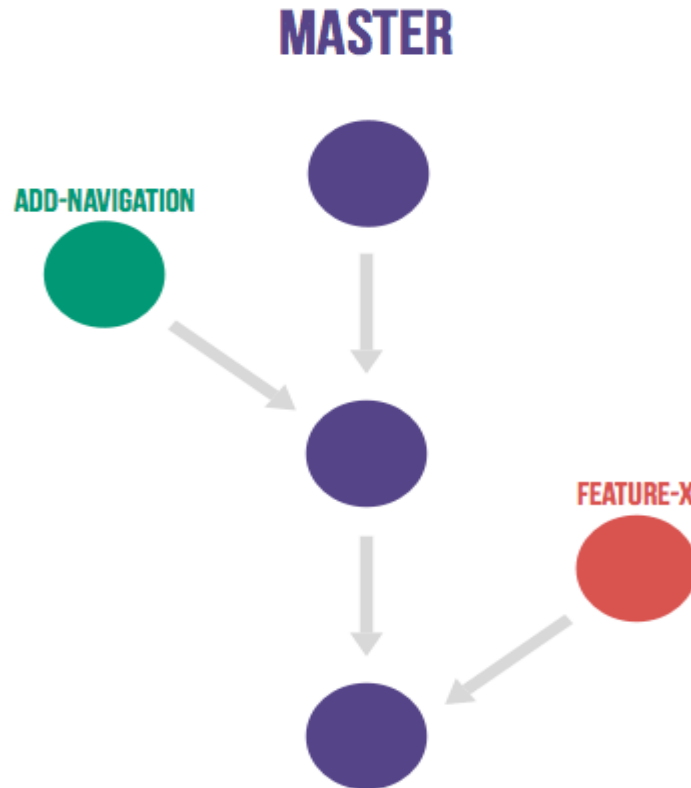
- Feature branches – nur bei den Entwicklern, nicht im origin
 - Quelle für die branches: develop-branch
 - Merge in: develop-branch

- Vorgehen:
 1. Feature branch erzeugen – ausgehend von develop
 2. Entwicklung im feature branch
 3. Feature branch in develop mergen
 4. Feature branch löschen
 5. Develop auf den origin pushen

- Quelle: develop
- Merge in: develop und master
- Vorgehen:
 - Branch von develop erzeugen
 - Im branch die SW releasefähig machen
 - Release branch in den master mergen
 - Release tag setzen
 - Release branch in develop mergen

Github Flow

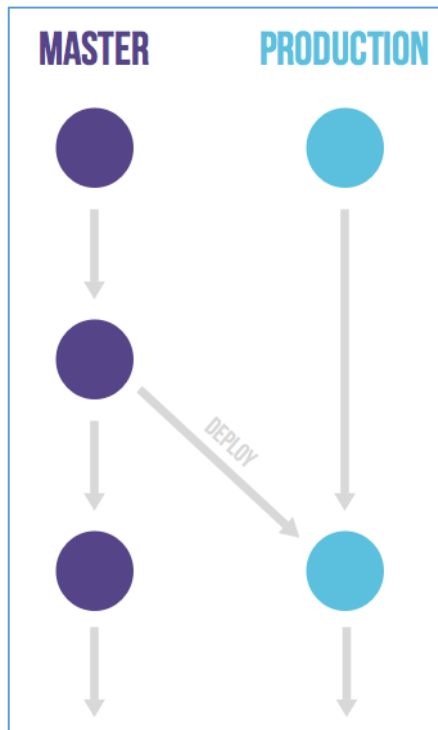
<https://guides.github.com/introduction/flow/index.html>



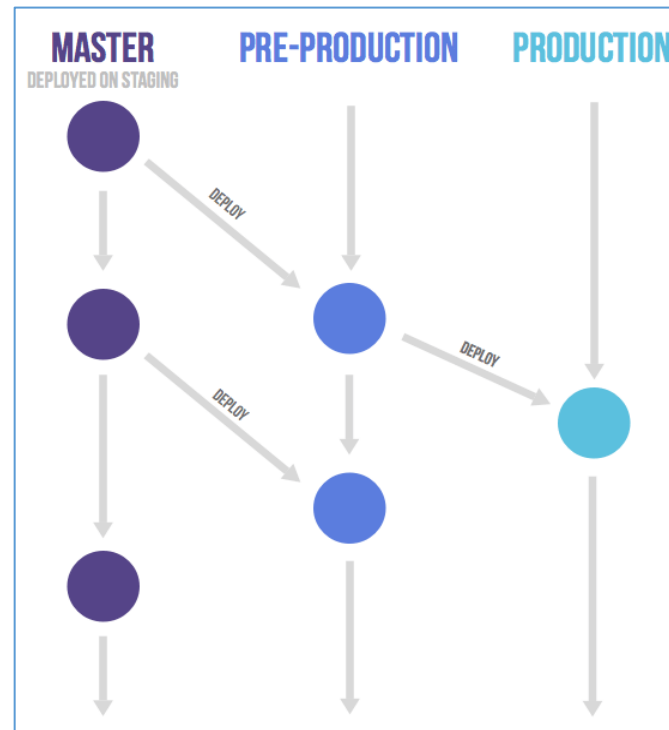
Nur master und feature branches, merge in den master, der master ist für Continuous deployment geeignet.

Gitlab Flow

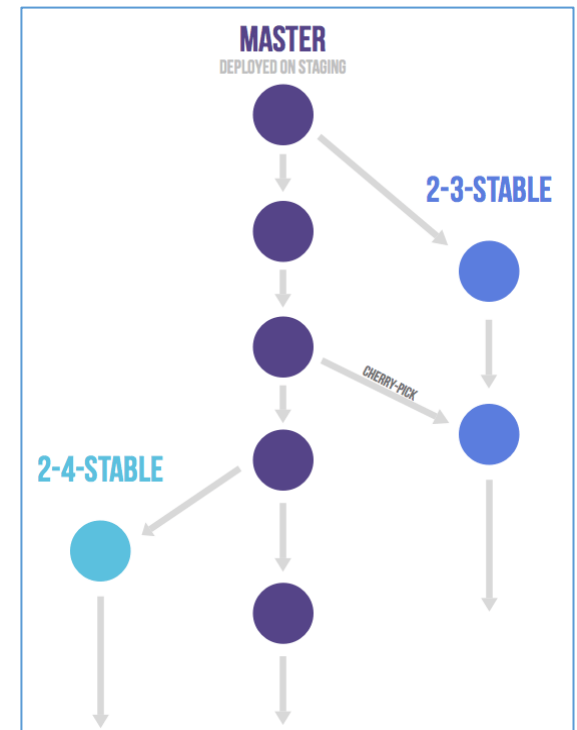
https://docs.gitlab.com/ee/workflow/gitlab_flow.html



Production branch



Environment branches



Release branches

- **Clean:** entfernt nichtversionierten Inhalt aus dem working directory.
- **Checkout:** setzt die Arbeitskopie auf eine spezifizierte Revision
 - Z.B. um einen branch an einer definierten revision starten zu lassen.
- **Revert:** inverses commit zu einem spezifizierten commit.
 - Um einen bereits gepushten commit rückgängig zu machen.
- **Reset:** setzt den aktuellen branch pointer auf eine spezifizierte revision
 - Für lokales Verwerfen. NICHT für bereits gepushte Revisions.