

GIT

Anwenderschulung

Ich und Du, Mü...



- Bernd Hegmanns
- Freiber. Berater
 - Agile Methoden
 - Softwareentwicklung
 - Middletier (JAVA-Enterprise)
 - Objektorientierung (DDD)
 - Agile Softwareentwicklung
 - Clean-Code
 - TDD
 - Agiles Testen
 - Projektabwicklung und Schulung/Training

Agenda

TODO

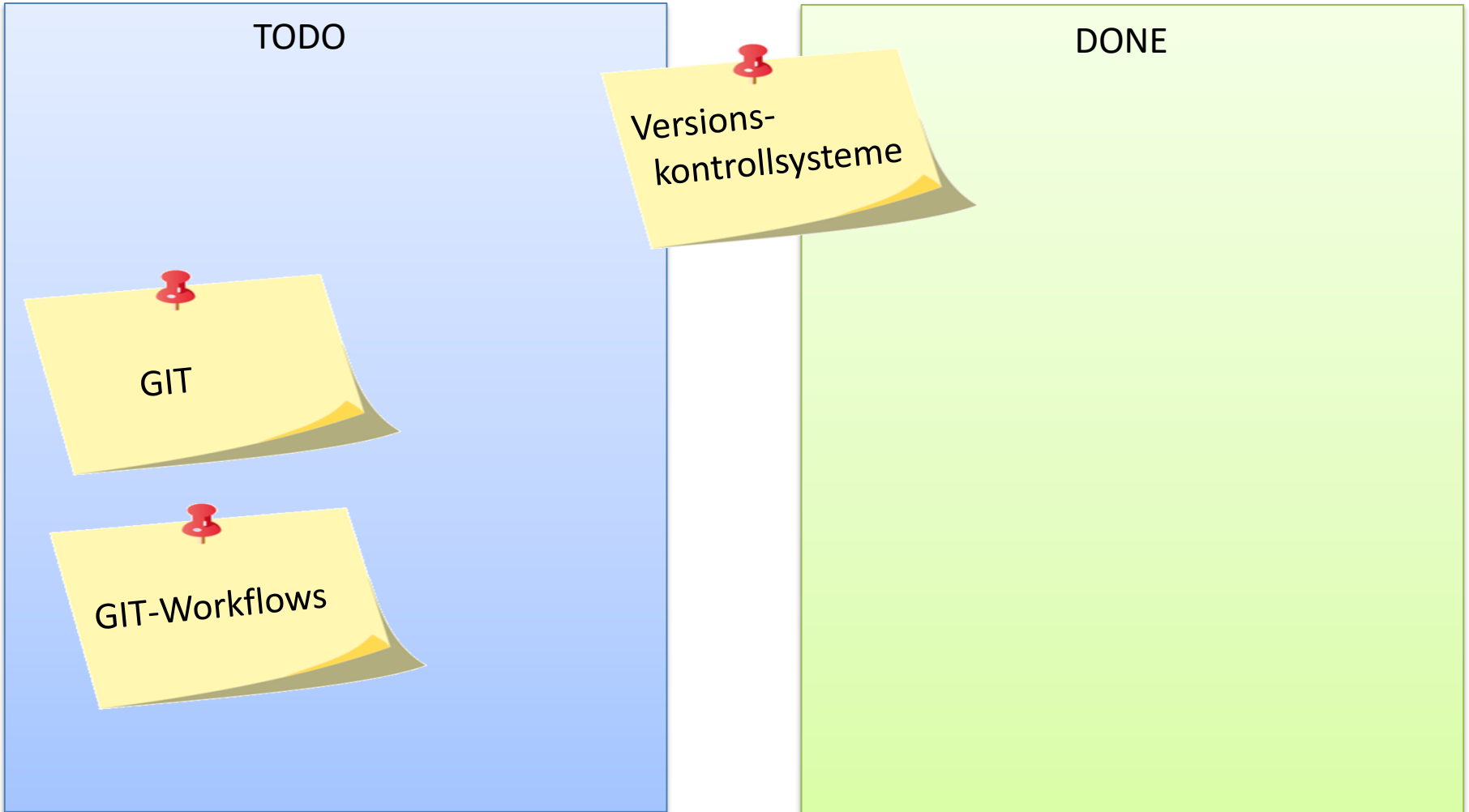
Versions-
kontrollsysteme

GIT

GIT-Workflows



Seminar-Taskboard



VERSIONSKONTROLLSYSTEME

git commit -m "fixed issue with fan"



...ein paar einleitende Worte und „Ist und Sein“

Versionskontrolle ist selbst
für eine Person gut



- Versionskontrollsysteme
beinhalten bereits die Logik
für ...
 - Historie
 - Reihenfolge
 - Beschreibung
- Versionskontrollsysteme
können erhöhte Sicherheit
bieten
 - Speicherung im Server
 - Sicherungskopien

Das kann Versionskontrolle



- Historie
 - Protokoll ...
 - Was wurde geändert
 - Wieso wurde geändert
 - Wer hat geändert
- Wiederherstellung
 - „Rückgängigmachen“ von Änderungen
- Mergen
- Archivierung
- Teamentwicklung
 - Wer darf wann zugreifen?
 - Workflow
 - Parallele Entwicklung



Mergen

Erna

Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche

?

Hochzeitsliste:

1. Musicalbesuch

?

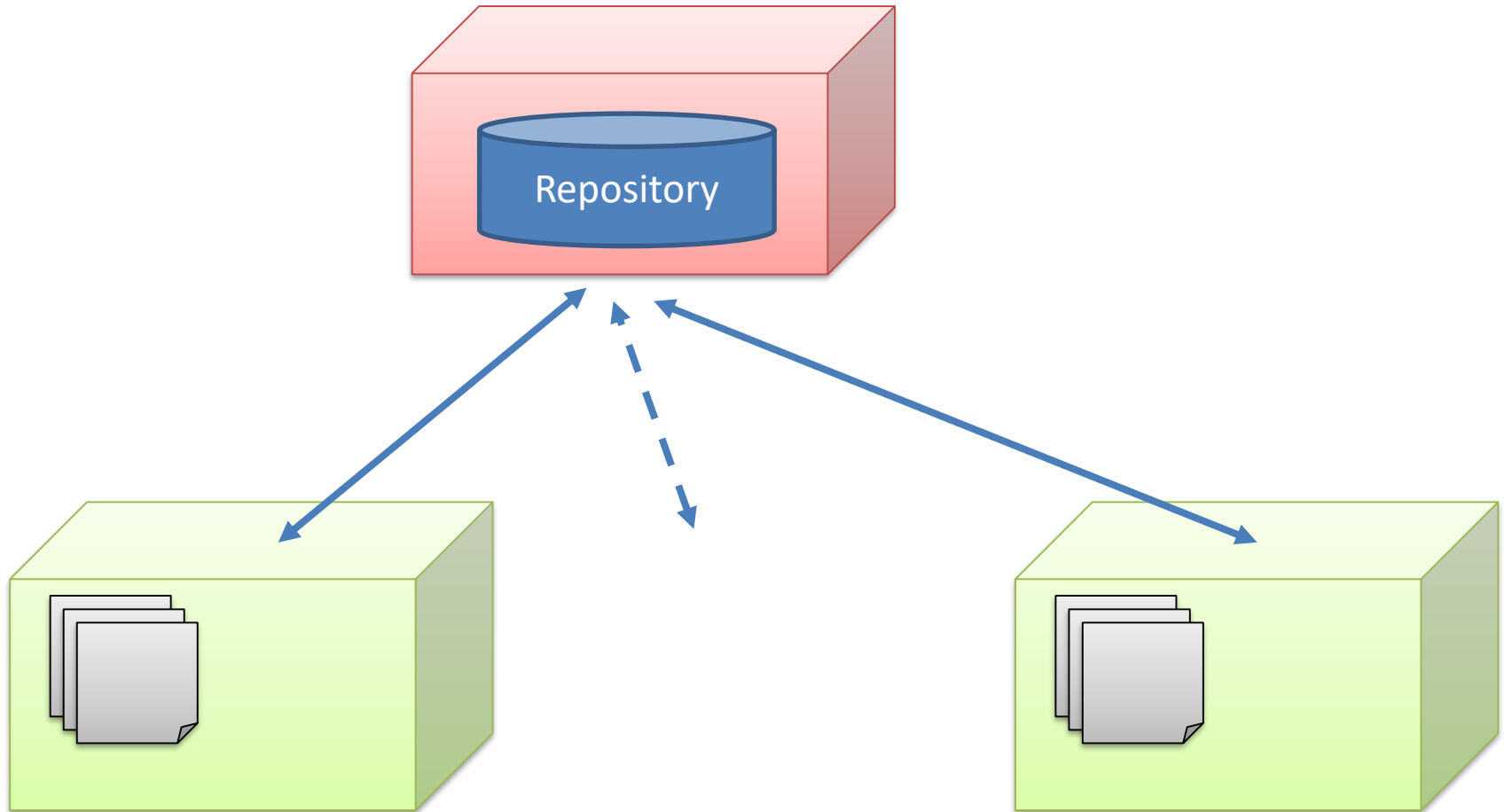
Ernst

Hochzeitsliste:

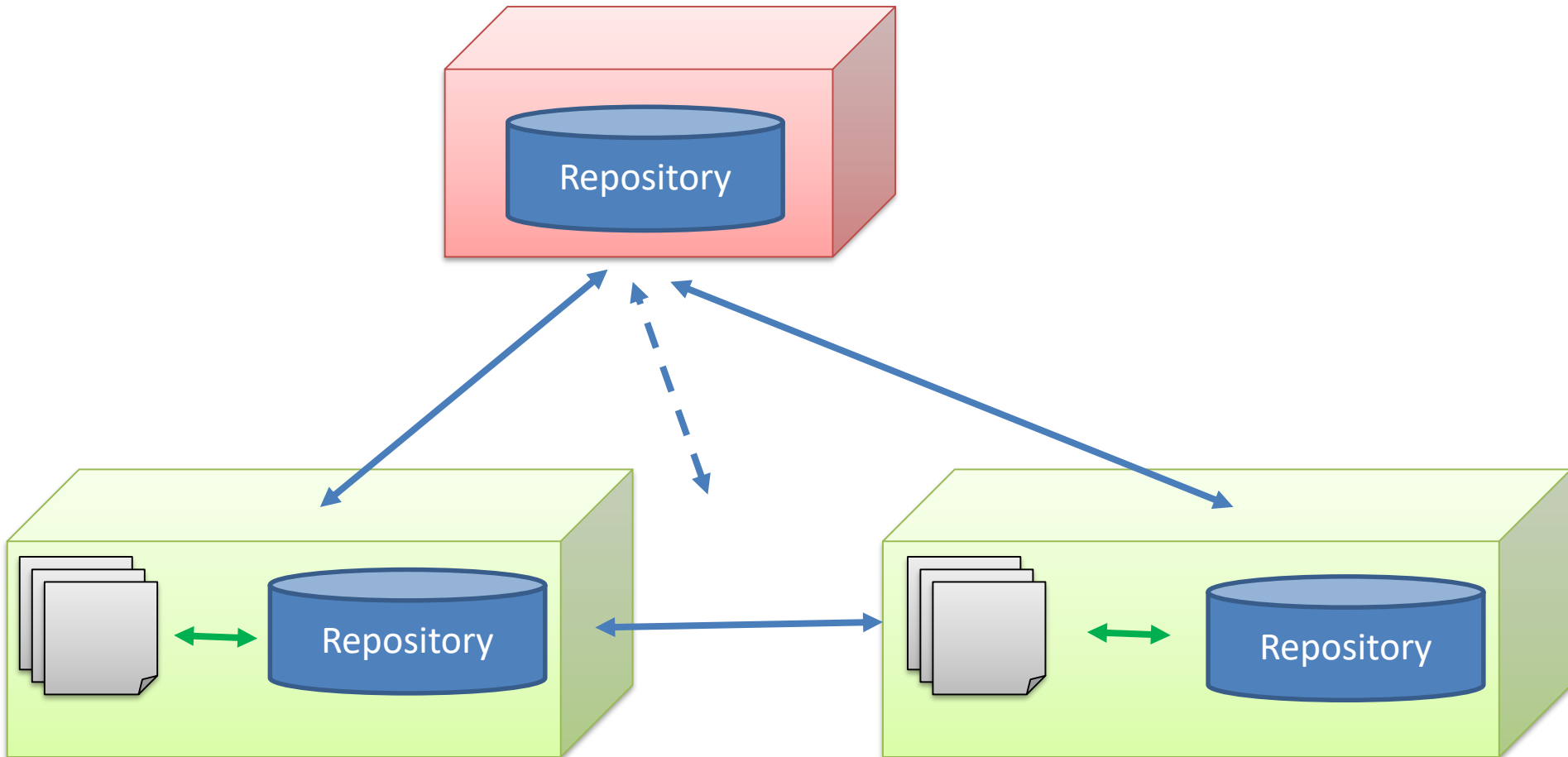
1. Musicalbesuch
2. Weltreise

Arbeitsweise Versionskontrollsysteme

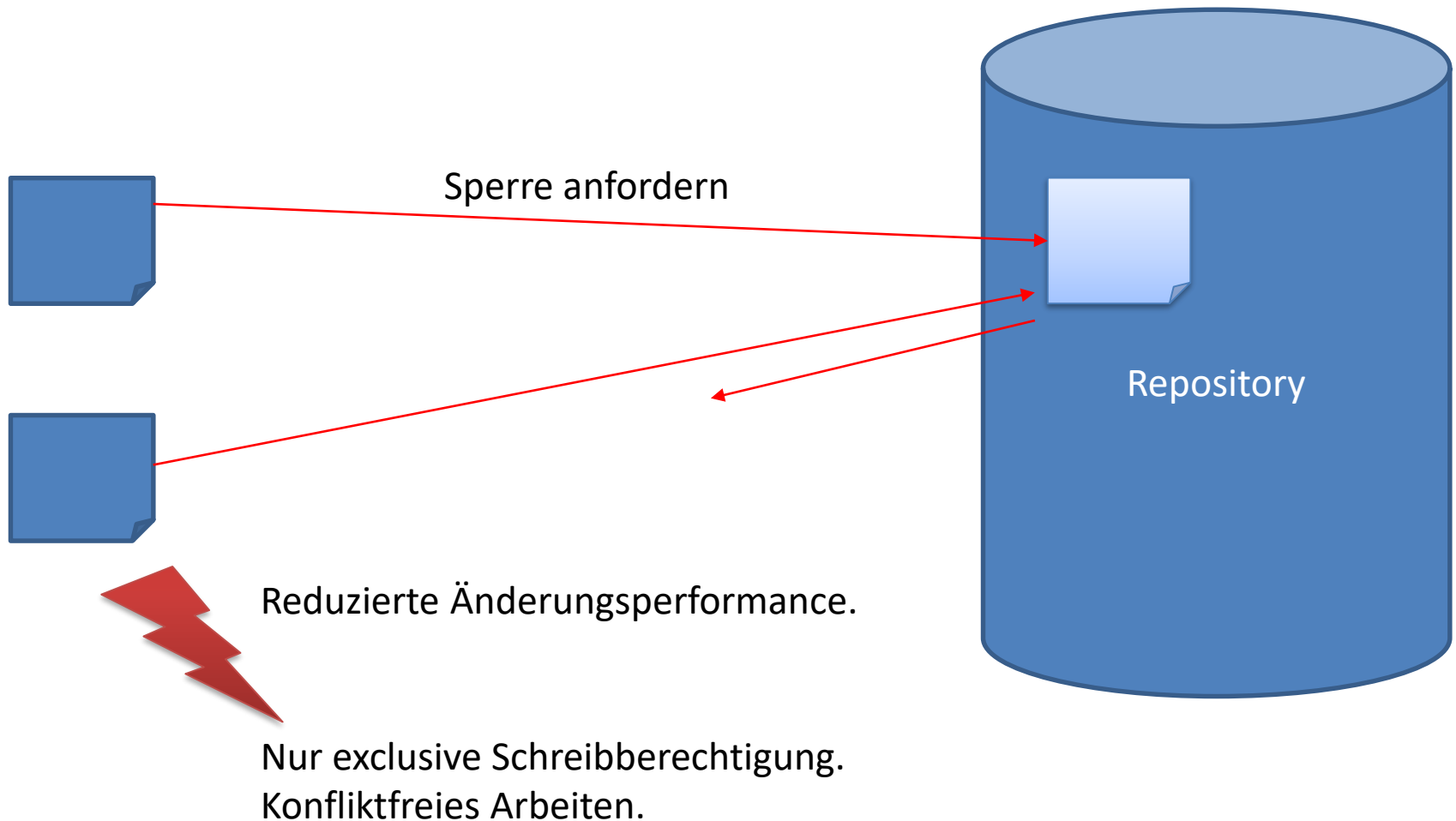
... zentral



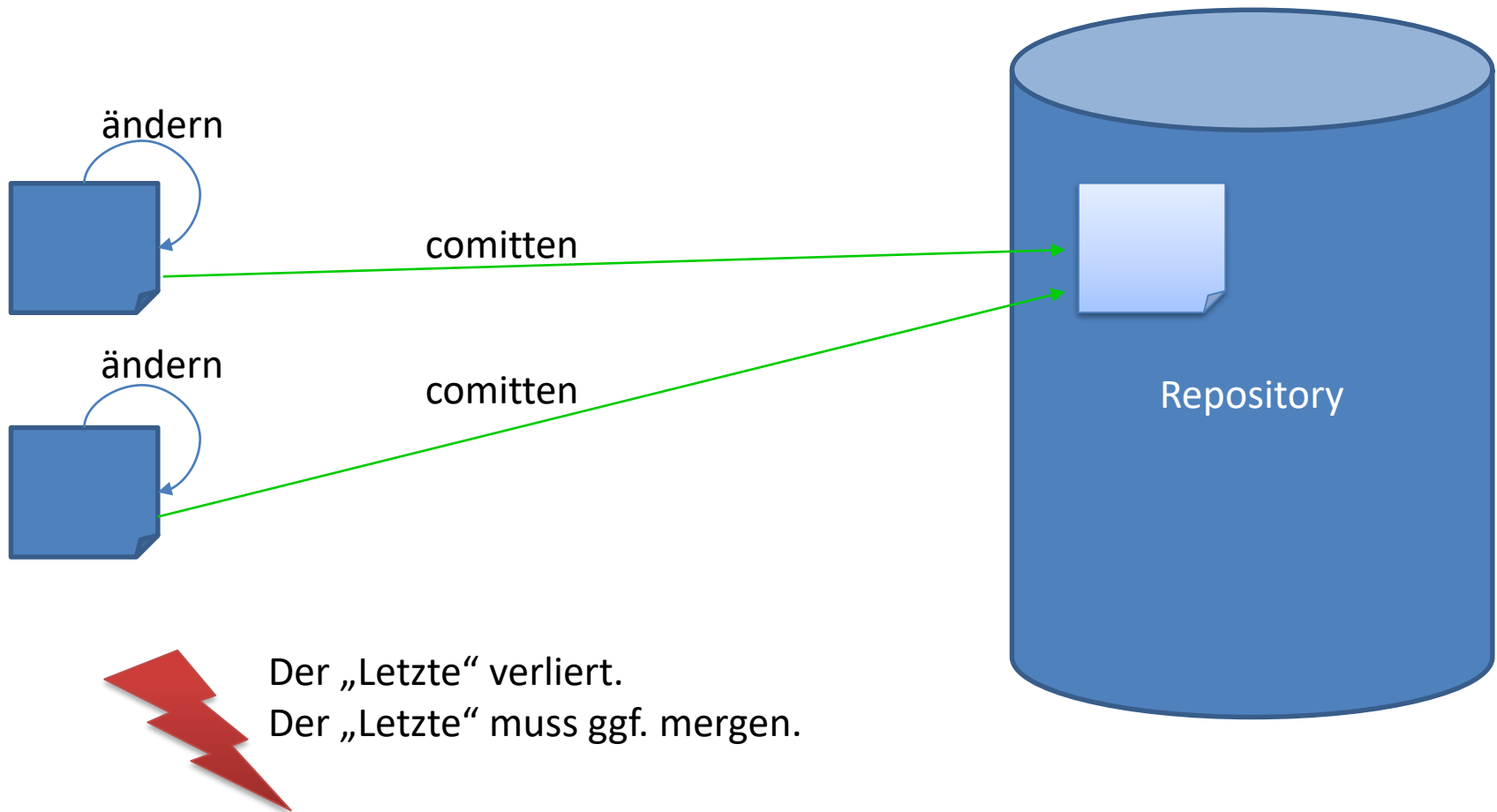
Arbeitsweise Versionskontrollsysteme ... verteilt



Arbeitsweise Versionskontrollsysteme ... mit Sperren



Arbeitsweise Versionskontrollsysteme ... ohne Sperren



Versionskontrollsysteme

... grundsätzliche Arten

	Sperren	Keine Sperren
zentral	ClearCase Visual SourceSafe	Subversion CVS
verteilt		GIT Mercurial


Schlagworte im Umgang mit dezentralen Repositories

- Performance
- Effizienz
- Offline
- Flexibilität
- Backup
- Wartbarkeit

Repositories im Umgang mit dezentralen Versionsverwaltungssystemen

- Blessed Repository
- Shared Repository
- Workflow Repository
- Fork Repository

Für Interessierte

- Tech Talk von Linus Torvalds
 - <https://www.youtube.com/watch?v=4XpnKHJAok8>
- Ein DevOps-Buch über GIT
 - 
- GIT vs. SVN (scheinbar neutral)
 - <https://svnvsgit.com/>

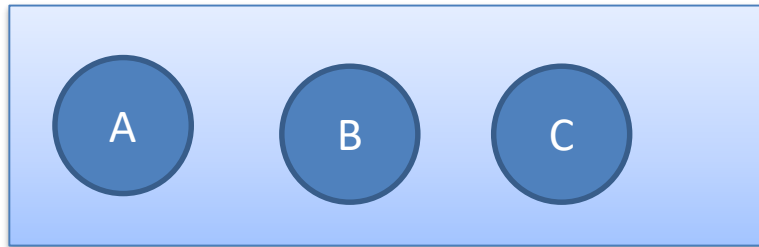
Wichtige Workflows

- Zentralisierter Workflow
- Feature Branch Workflow
- Forking Workflow

Zentralisierter Workflow

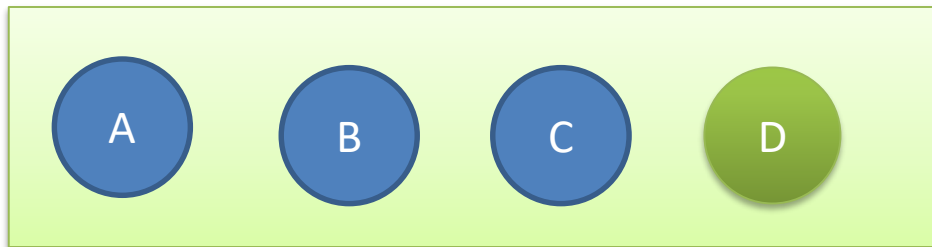
- Das gesamte Team arbeitet auf dem gleichen (zentralen) Stand
 - Änderungen werden zum (zentralen) Repository commitet (gepushed) und werden jeweils ans letzte Commit gehängt
 - In Konfliktsituationen ist ein Merge nötig, ein so genanntes Rebasing
- Erfordert schnelle Synchronisation
 - Je größer das Team desto schnellere Synchronisation ist nötig

Zentralisierter Workflow: Beispiel



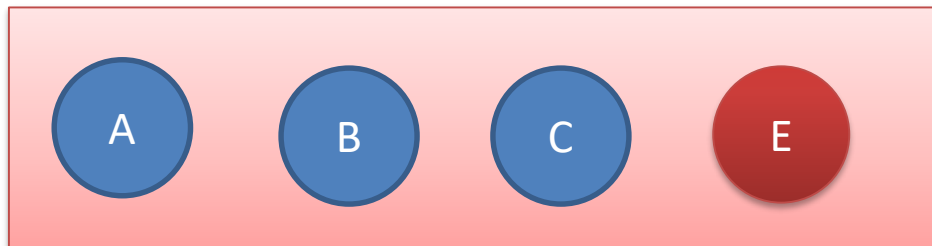
Hochzeitsliste:

1. Musicalbesuch



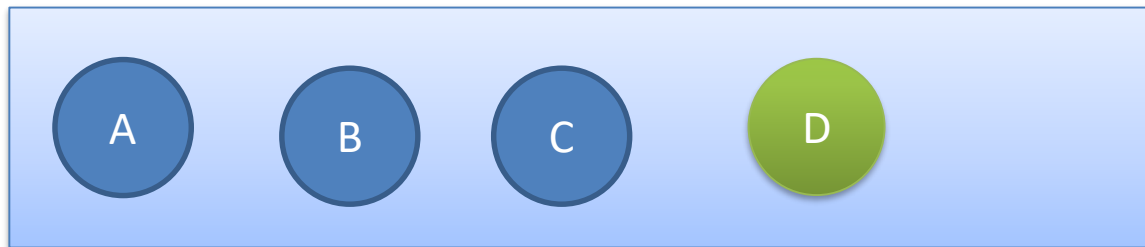
Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche



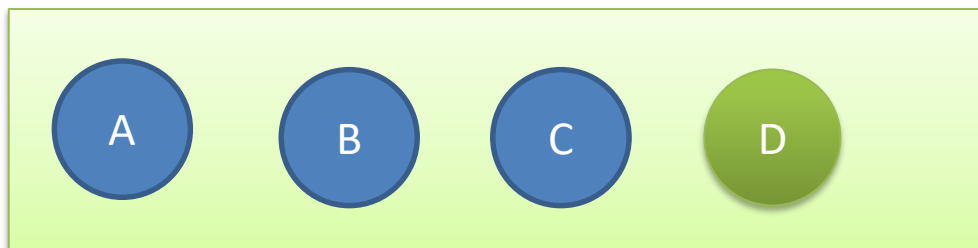
Hochzeitsliste:

1. Musicalbesuch
2. Weltreise



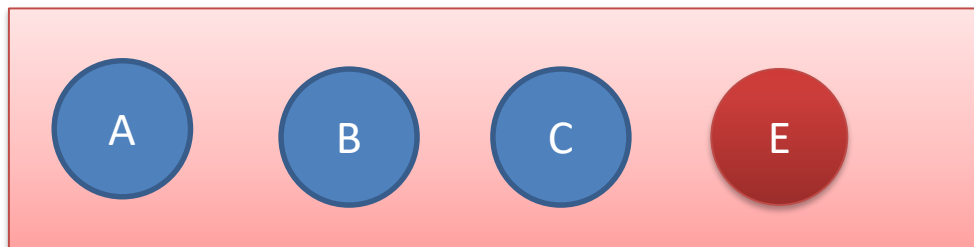
Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche



Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche



Hochzeitsliste:

1. Musicalbesuch
2. Weltreise



Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche



Hochzeitsliste:

1. Musicalbesuch
2. Weltreise

Ernst muss nun mergen
und über den neuen
Zustand entscheiden ...

A

B

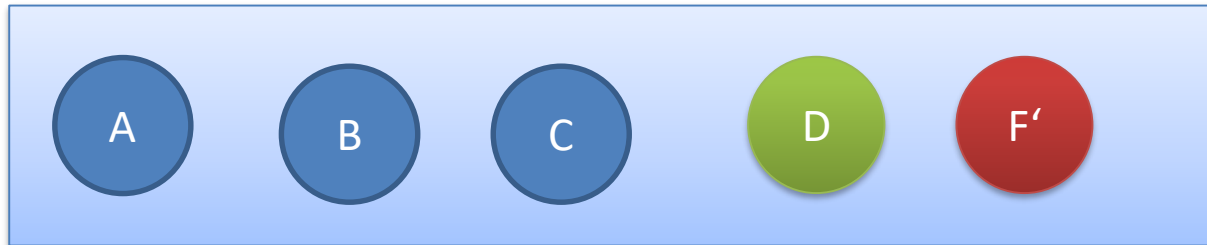
C

D

F'

Hochzeitsliste:

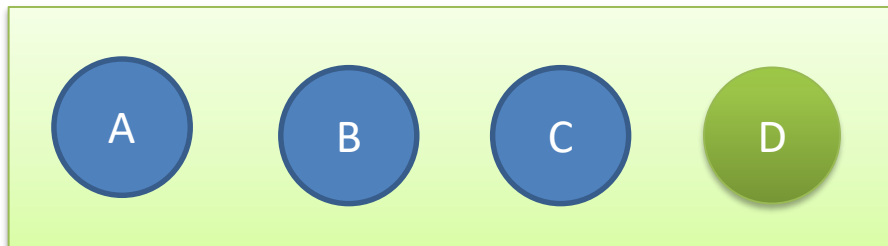
1. Musicalbesuch
2. Babywäsche
3. Weltreise



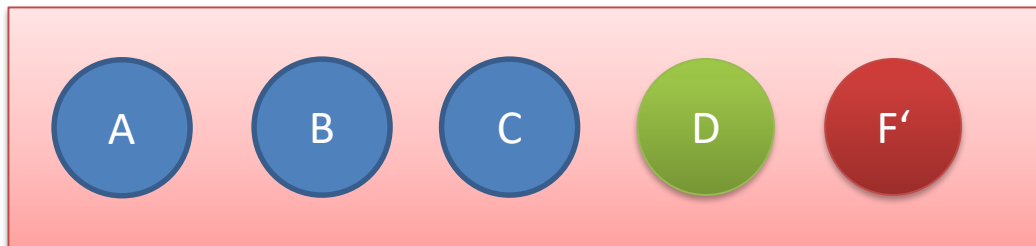
Hochzeitsliste:

1. Musicalbesuch
2. Babywäsche
3. Weltreise

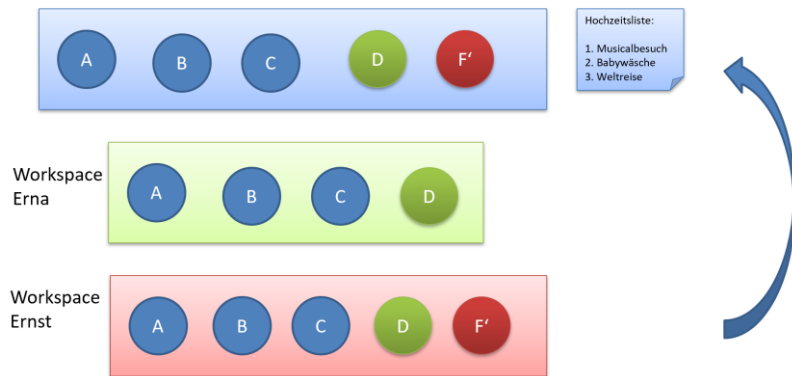
Workspace
Erna



Workspace
Ernst



- Was ist mit der Version „F“ (von Ernst) ?



- Interessiert sie noch?
- Geht sie verloren?

Auf ein Wort ...

... Sperren

- Sperren verhindern parallele Entwicklung
 - Verminderte Geschwindigkeit
 - Häufig automatisierte Merge möglich
 - Manuelle Merge durch beteiligte Entwickler häufig sehr einfach
- Es gibt wenige Situationen, in denen Sperren sinnvoll sein können
 - Binaries
 - Bestimmte Dokumente

Auf ein Wort ...

... Binaries

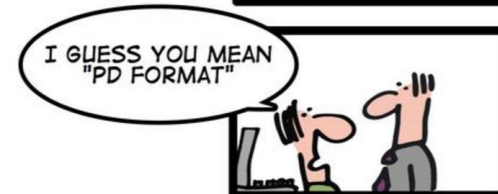
- Hier beginnt spätestens das Thema „Sperren“
- Grundsätzlich nicht möglich
- GIT kennt Text-Dateien
 - Es gib GIT-Zusätze für Sperren
- ABER:
 - Benötigt man wirklich Binaries/Sperren?
 - Ist es ggf. eine Alternative, die Quelle der Binaries zu sichern? (ähnlich wie Quellcode)

... ein paar Vokabeln/Abkürzungen

VCS (version control system)	
SCCS (Source Code Control System)	1972
RCS (revision control system)	1982
CVS (concurrent version system)	1986
SVN (subversion)	2001

PEDANTIC GEEKS

ONLY GEEKS KNOW THAT...



GIT-Kurzgeschichte

- 2005 von Linus Torwald
- Eigenes Kommandozeilentool
- Verteilte Versionsverwaltung
- Besondere Eigenschaften
 - Rebase, merge

Cloud-Hosting-Anbieter

- GitHUB
- Gitlab
- Bitbucket

- Riouxsvn (riouxsvn.com)
- Beanstalk
- Assembla

Cloud-Hosting

- Hosting von Repositories
 - Privat und öffentlich
- Workflow
- Sicherheit
- Forking
- Datensicherung

Okay ...

Gitlab-Panne bestätigt Murphys Law

01.02.2017

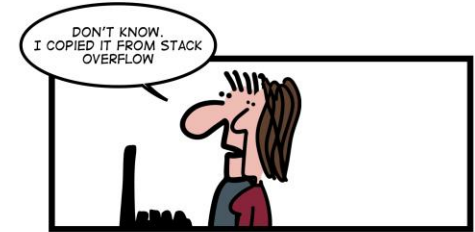
Gitlab ist derzeit nicht erreichbar, weil ein Admin versehentlich Daten gelöscht hat und sich Murphys Law beim Versuch den Fehler zu beheben, voll bestätigt.

Die Panne bei Gitlab ist ein Lehrstück dafür, wie wieder einmal alles schiefgegangen ist, was schiefgehen konnte. Auslöser ist das Versehen eines Admins, der einen leeren Ordner löschen wollte, weil er ihn für PostgreSQL-Replikationsprobleme verantwortlich wählte. Beim Löschen allerdings war der Admin -- schon das ist ein Klassiker -- auf einem anderen Rechner eingeloggt, als er annahm. Dadurch löschte er auch keinen leeren Ordner, sondern die produktive Datenbank. Das wurde ihm nach wenigen Sekunden klar, aber da war es zu spät. Der größte Teil der Daten war weg.

Natürlich gab es gleich mehrere Backupmechanismen, aber die meisten hatten unbemerkt versagt. Das PostgreSQL-Backup war ausgefallen, weil es mit Binaries gestartet wurde, die nicht mehr kompatibel zur eingesetzten Version waren. Snapshots in Azure existierten, aber nur für die NFS-Server, nicht für die Datenbank. Die Backups auf der Grundlage von Amazons Speicherdienst S3 hatten ebenfalls nicht funktioniert. Ein Monitoring der Backups gab es nicht. Offensichtlich war in letzter Zeit auch das Recovery nicht getestet worden.

Die Wiederherstellung aus älteren LVM-Snapshots wird nun noch einige Zeit in Anspruch nehmen. Immerhin kann man daraus lernen: Backups müssen unbedingt überwacht und das Recovery regelmäßig getestet werden.

... das mit der Datensicherheit kann auch mal schief gehen ...



GOOD QUESTIONS

- 

Google Trends

Suchbegriffe rund um Versionskontrolle

● svn
Suchbegriff

⋮

● git
Suchbegriff

⋮

● Concurrent Versio...
Software

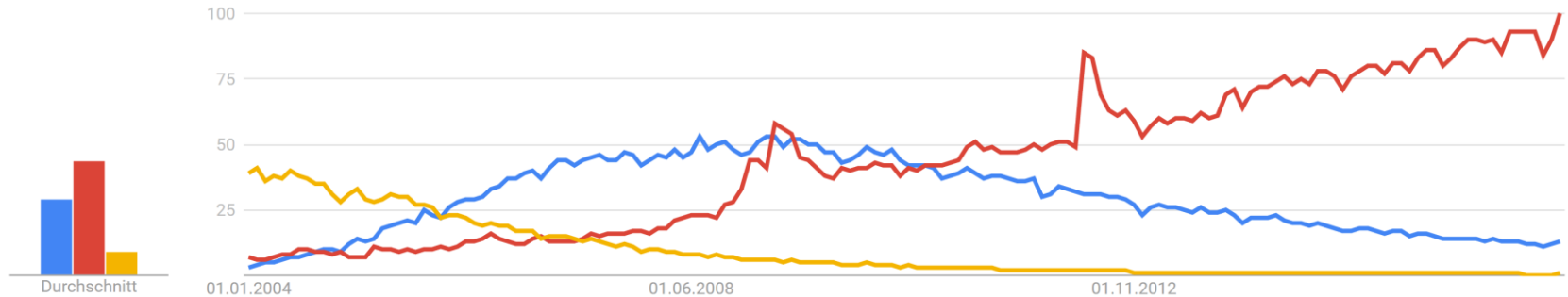
⋮

+ Vergleich hinzufügen

Weltweit ▼ 2004 - heute ▼ Alle Kategorien ▼ Websuche ▼

"Suchbegriffe" entsprechen bestimmten Wörtern, während es sich bei "Themen" um Konzepte handelt, die ähnlichen Begriffen in einer Sprache entsprechen. [Weitere Informationen](#)

Interesse im zeitlichen Verlauf ?



GIT - SOFTWARE

...

GIT Distribution

- GIT-Bash
- GIT-Kommandozeilentools
- DIFFS
- Basierend auf Linux

GIT: Andere Clients

- Spezielle Clients der IDE
- Tortoise GIT
- Smartgit
- ...

Typischerweise nutzen die Clients die
GIT-Basisinstallation

SmartGIT

- Windows-Tool
- Abbildung aller typischer Arbeiten
- Einblick im Repository
 - Lokal
 - Remote

Tortoise GIT

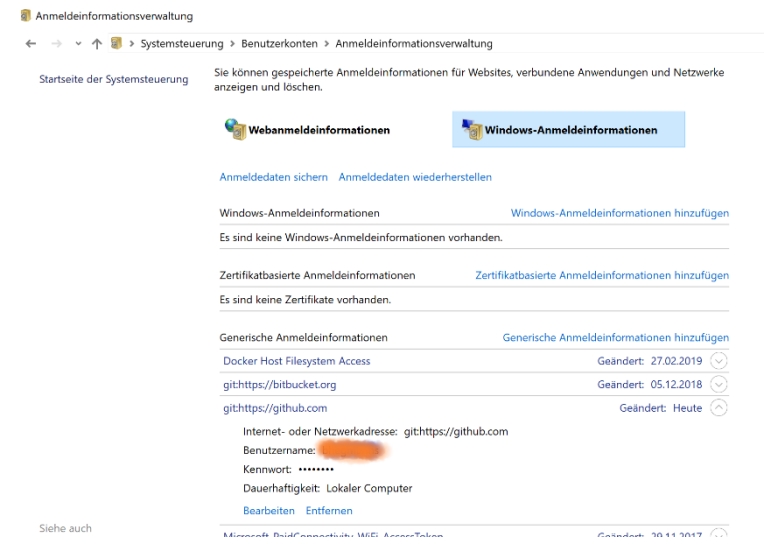
- Arbeitet über den Explorer
- Ableitung aller typischer Arbeiten im Repository

GIT in IDEs

- Eclipse-Unterstützung
- IntelliJ-Unterstützung

Windows Anmeldeinformationen löschen

- Systemsteuerung
 - >>> Anmeldeinformationsverwaltung (CredentialManager)
- Windows-Anmeldeinformationen

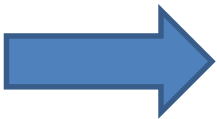


LINUX UND GIT

... Zwei, die zusammen gehören ...

GIT ist ein LINUX-Tool

- Ein paar LINUX-Kenntnisse sind sinnvoll
- GIT-Bash ist eigentlich eine Linux-Shell
 - Alle nötigen Umgebungsvariablen gesetzt
- Grobe Verzeichnisstruktur
- Grobes Arbeiten mit der Shell/Bash
- Grobes Arbeiten mit VI



Kleine Hilfe in der Bash:

https://github.com/bhegmanns/gitschulung_material/blob/master/bash_befehle.txt

Seminar-Taskboard

TODO



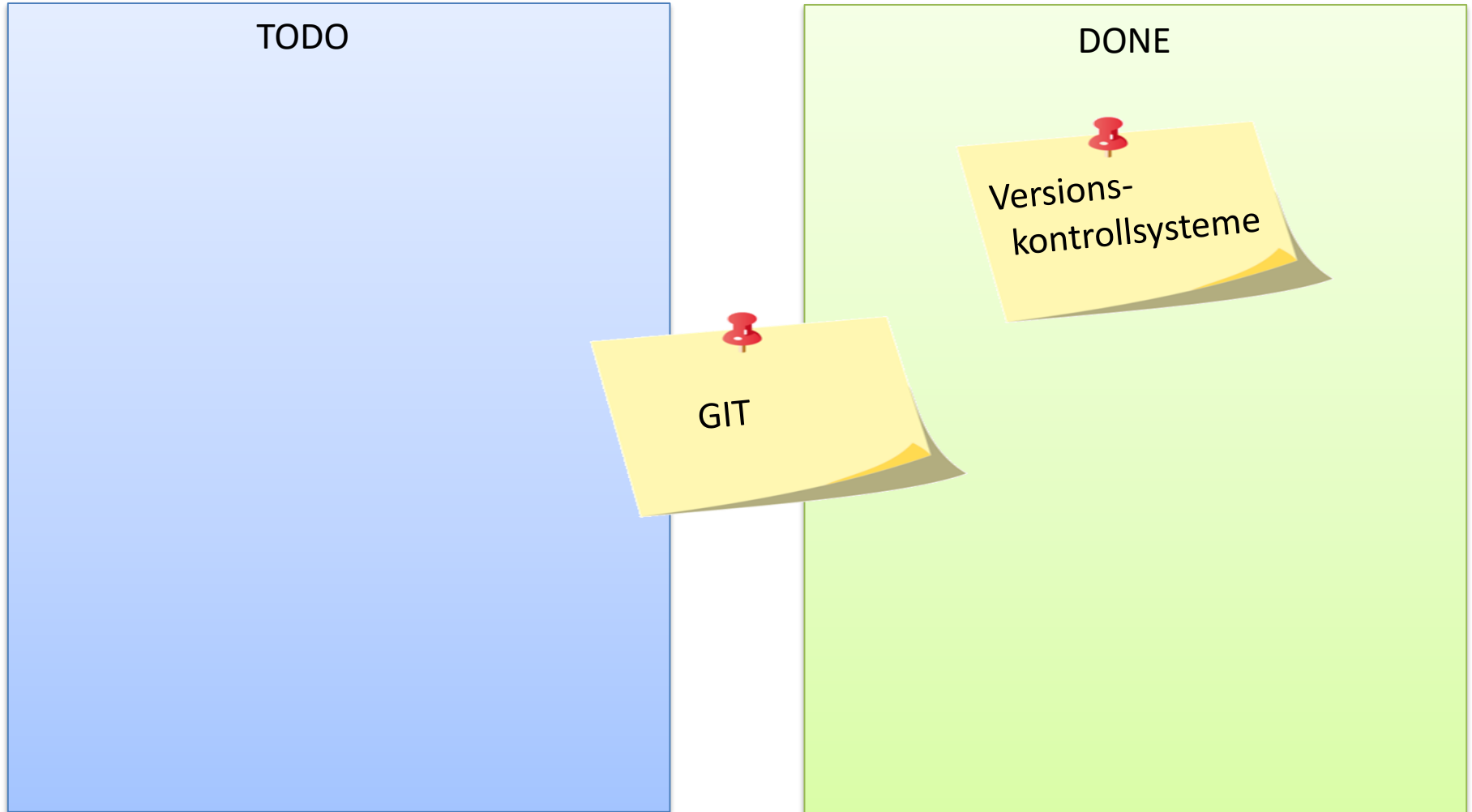
GIT

DONE



Versions-
kontrollsysteme

Seminar-Taskboard



GIT Bash: Shortcuts bzw. Alias

- Legen Sie für häufige Aufgaben Abkürzungen an

– ALIAS

```
$ alias  
alias ll='ls -l'  
alias ls='ls -F --color=auto --show-control-chars'
```

GIT-Hilfesystem

- „git help“
- „git help - -a“
- „git help <command>“
 - Bsp „git help commit“

GIT Befehl

- GIT-Befehle
 - ... für Workspace
 - ... für lokales Repository
 - ... für remote Repository

Allgemeine Struktur: „git <command> <attribut(e)> --option1 --option2 --option2

```
$ git init rep001 --bare
```

Initialized empty Git repository in C:/gituebung/test/rep001/

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-
path]
          [-p | --paginate | --no-pager] [--no-replace-objects] [--
bare]
          [--git-dir=<path>] [--work-tree=<path>] [--
namespace=<name>]
          <command> [<args>]
```

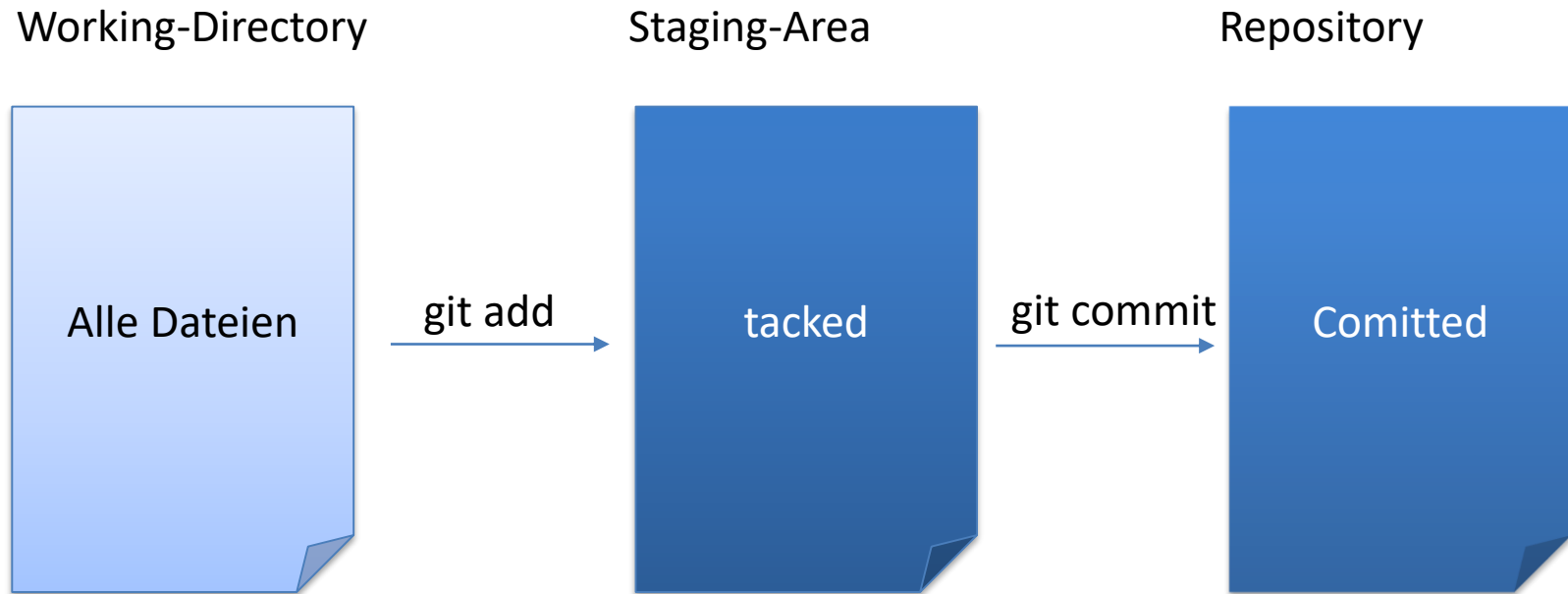
Kurz- und Langoptionen

- Lang-Optionen mit „- -“
- Kurz-Optionen mit „-“

```
$ git commit --message "eine Datei"  
[master (root-commit) 76f8878] eine Datei  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 text001
```

```
$ git commit -m "eine Datei"  
[master 70a3876] eine Datei  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 text002
```

Von der Änderung zum Commit

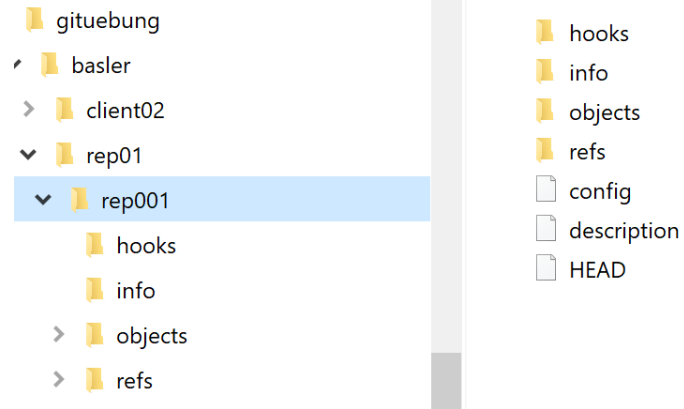


START

GIT Repository erstellen

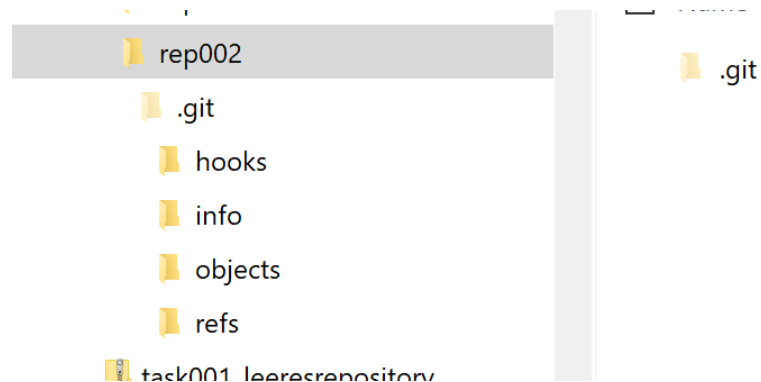
`git init --bare`

Ohne Arbeitsverzeichnis
Typischerweise das
Remote-Repository



`git init`

Ohne Arbeitsverzeichnis
Ohne Anbindung zum remote



GIT Repository kopieren

git clone

Es wird ein lokales Repository erzeugt.
Das lokale Repository kennt durchs clonen sein Remote

```
[remote "origin"]
  url = C:/gituebung/basler/rep01/rep001
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Repository clonen

```
git clone <url, pfad> <ggf. Name>
```

Anbinden an ein Repository

```
git remote add <name> <url>
```

Das Repository wird an ein entferntes Repository geknüpft; mit einem symbolischen Namen.

Typischer Name: „origin“

```
git remote add origin ...
```

```
git pull origin master
```

Den „master“-Branch des Remote-Repositories „origin“ in den Arbeitsbereich laden.
„git pull –u origin master“

Ansicht des remote-Repositories: `git remote -v`

Ersteinrichtung im Repository

A light blue rectangular box with a thin black border and a small folded-corner effect at the bottom right. It contains the text "git config" in a black sans-serif font.

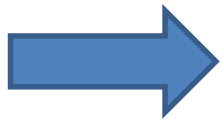
git config

```
git config user.name „client01“  
git config user.email „client01@web.de“
```

Ersteinrichtung für Installation

```
git config --global
```

```
git config - -global user.name „client01“  
git config - - user.email „client01@web.de“
```



„--global“ kann auch weg gelassen werden.
Dann gelten die Einstellungen für das aktuelle Repository.

COMMITTS, REVISIONEN

GIT-Objekte

- Blobs
 - Version einer Datei
- Bäume
 - Verzeichnisinformation
- Commits
 - Informationen einer Änderung am Repository
- Tags
 - Alias für ein Objekt (im Allgemeinen ein Commit)

GIT-Objekte:

Blobs

- Dateien
 - Aber nicht der Dateiname
- Referenzierung durch eine Prüfsumme



Mehrere Dateien gleichen Inhalts werden von GIT erkannt und nur einmal gespeichert.

GIT-Objekte:

Trees

- Pfad im Verzeichnis
 - Blob-Identifizier
 - Pfadnamen
 - Noch weitere Informationen
- Trees können sich gegenseitig referenzieren

GIT-Objekte:

Commits

- Daten für die Veränderung eines Repositories
 - Autor, Committer, Datum, Nachricht
- Ein Commit referenziert eines Verzeichnisobjekt
- Mit Ausnahme des ersten Commits:
 - Jeder Commit als ein Eltern-Commit

GIT-Objekte:

Tag

- Lesbarer Name für ein Objekt
 - Typischerweise für ein Commit

GIT-Bereiche



The diagram consists of four light blue cylinders stacked vertically, representing the different areas of a Git repository. From top to bottom, they are labeled: Remote-Repository, Lokale Repository, Stage / Index, and Working-Area.

Remote-Repository

Das Basis-Repository

(Zu einem lok. Repository können auch mehrere zentrale Repositories existieren)

Lokale Repository

Das Repository, mit Versionen, ...

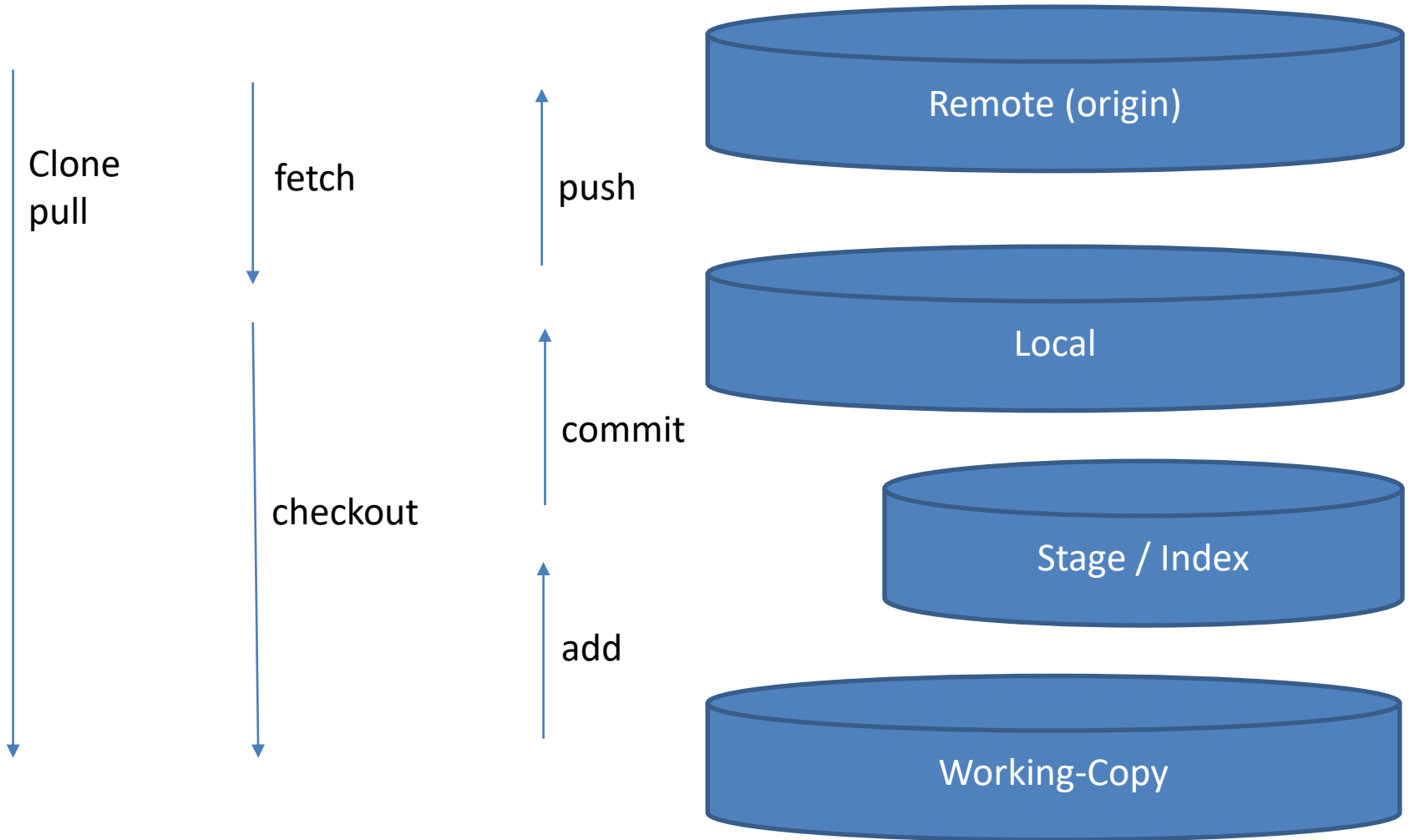
Stage / Index

Ein Sicherungsbereich

Working-Area

Der eigentliche Arbeitsbereich

GIT Aktionen



Index, Stage

- Snapshot des Arbeitsverzeichnis
- Dieser Zustand kann durch ein commit ins (lokale) Repository gebracht werden
- Änderung möglich
 - Nicht das gesamte Arbeitsverzeichnis liegt im Stage

Objektspeicher, Überblick

- Repository nach init
- Repository nach einem ersten commit
 - „touch info.txt“
 - „git add info.txt“
 - „git commit -m „neue Datei““

Status des lokalen Arbeitsbereiches



git status

- Unbekannte Dateien
- Änderungen an getrackte Dateien
- Etwas zu committen?
- Unterschied lokales Repo zum Remote Repo

Unterschiede anschauen



git diff

Schaut Unterschiede

```
git diff --cached --numstat  
git diff origin
```

History anschauen

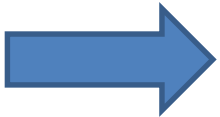


git log

Commit

```
git commit -m „Message>
```

Schiebt den Inhalt des Index in das (lokale) Repository



Ohne Message (--m <message>) öffnet sich VI

Push



git push

Schiebt den Inhalt des
lokalen Repositories ins remote Repository

git push origin



Ohne Message (--m <message>) öffnet sich VI

Pull



git pull

Holt Inhalt/Änderungen ins lokale Repository
bzw. den Arbeitsbereich

git pull origin master

Wenn das Repository bekannt ist bzw. eingerichtet
ist, reicht auch „git pull“

Schnellbremse

Änderung zurück nehmen

- Ungestaged (nicht mit add aufgenommen)

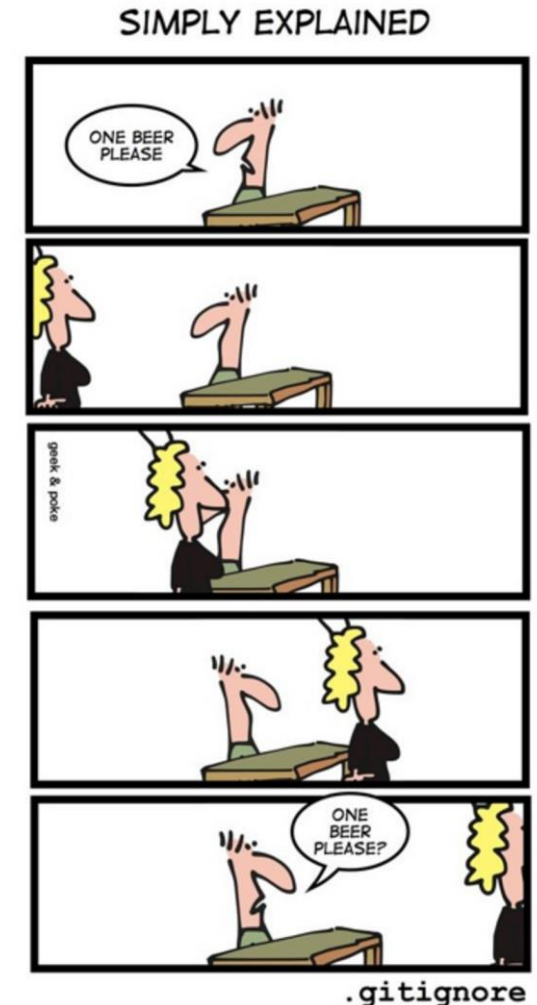
`git checkout -- <datei>`

- Gestaged (mit add aufgenommen)

`git reset HEAD <datei>`

Und nun ... ein paar Fingerübungen

- Repository erzeugen
- GIT minimal konfigurieren
- Erstes commit
- Commits anschauen
- Diff
- Ein paar kleinere Operationen im Inhalt
- Clonen
- Weitere Konfigurationen



GIT ... ist verteilt

Revisionsnummern

- Numerisch auf folgende Revisionsnummern funktionieren nur in zentralen Repositories
- Für verteilte Repositories:
 - Anhand der Revisionsnummern lässt sich nicht die Reihenfolge ableiten
- Es wird ein Hash (SHA1) als Revisionsnummer gebildet
 - Okay, es werden mehrere Hashes gebildet

GIT Standard-Arbeit

Änderungen erstellen
Änderungen stagen

```
git add a.txt b.txt  
git add .
```

Änderungen prüfen

```
git status
```

Änderungen hinzufügen

```
git commit -m „ein commit“
```

Änderungen hochladen

```
git push
```

Übung 01.a

- Erstellen Sie ein bare-Repository und clonen es
- Erstellen Sie eine Datei „bundeslaender.txt“ als leere Datei
- Nach jedem folgenden Schritt stagen und commiten
 - Fügen Sie jeweils nach einander ein beliebiges Bundesland in die Datei in einer neuen Zeile ein
- Pushen Sie ganz zum Schluss

Übung 01.b

- Erstellen Sie ein Repository und clonen es
- Erstellen Sie eine Datei „bundeslaender.txt“ als leere Datei
- Nach jedem folgenden Schritt stagen und commiten UND pushen
 - Fügen Sie jeweils nach einander ein beliebiges Bundesland in die Datei in einer neuen Zeile ein

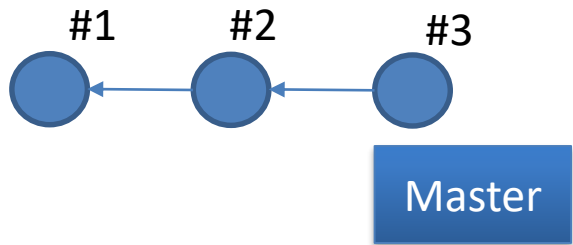
Übung 01.c

- Schauen Sie sich die Histories (lokal und remote) der beiden Dateien an
 - Gibt es Unterschiede?

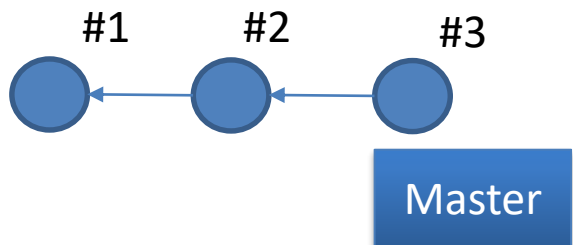
BRANCHEN

Branch:

Start von einem beliebigen commit



`git checkout -b Feature #2`

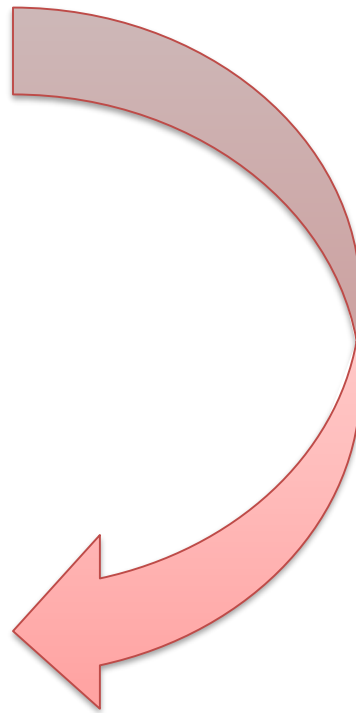
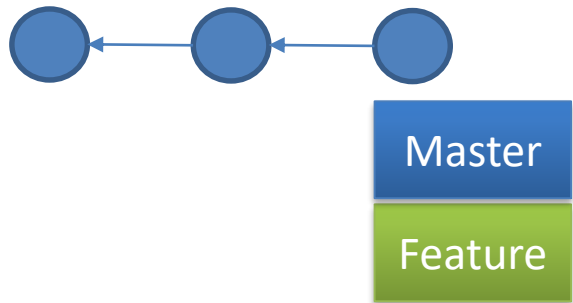
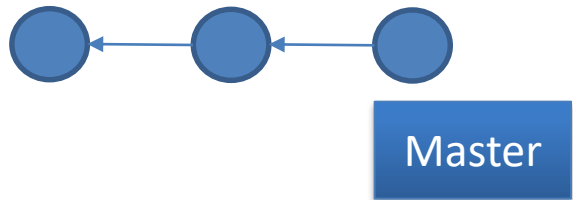


`git branch Feature #2`
`git checkout Feature`

Feature

Branch:

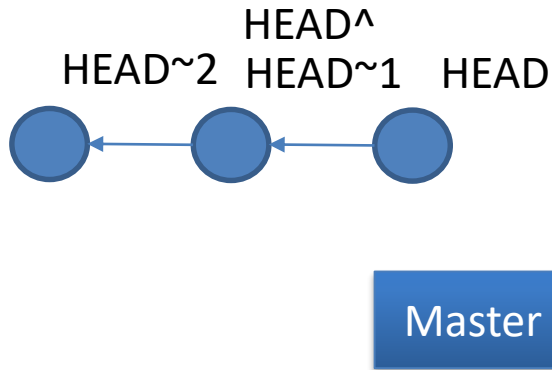
Erst mal nur ein Flag



`git checkout -b Feature`

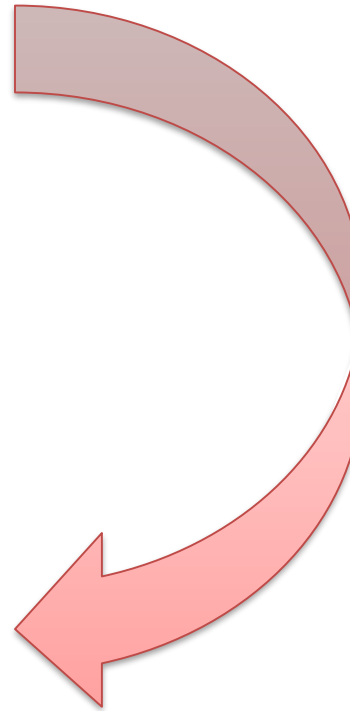
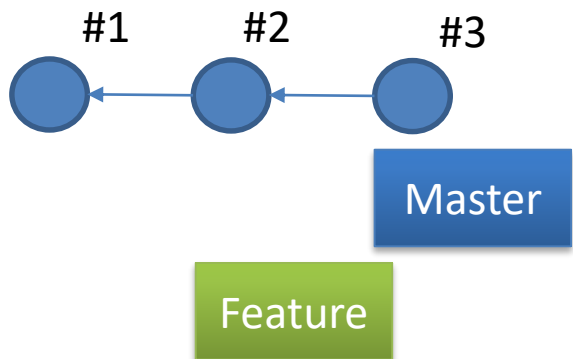
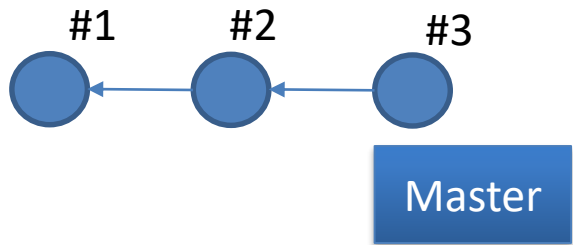
`git branch Feature`
`git checkout Feature`

Ein wenig Commit-Nomenklatur



- HEAD
 - Letztes Commit
- HEAD[^], HEAD~1
 - Vorletztes Commit
- HEAD~x
 - x. Commit vor dem Letzten

Branch: mit HEAD-Zeiger



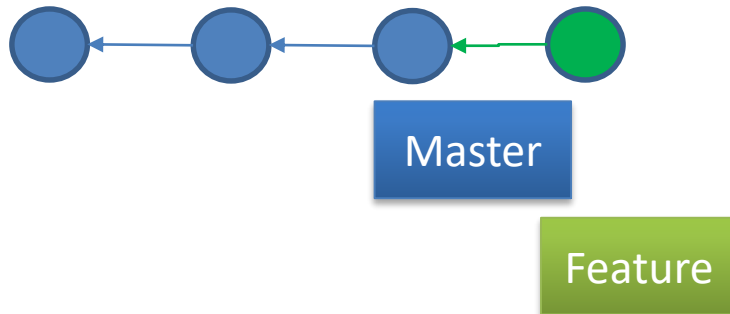
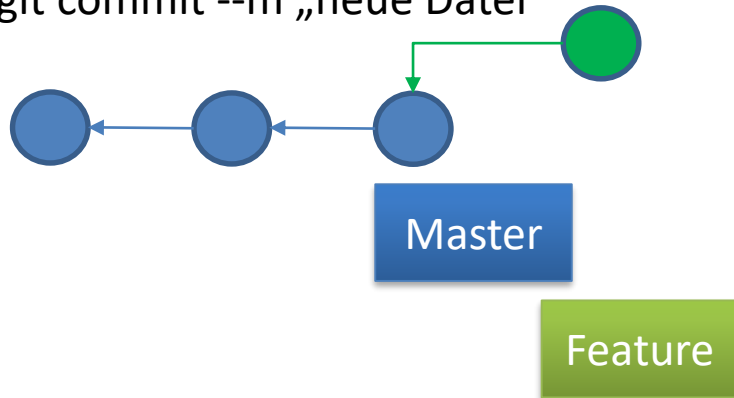
git branch Feature HEAD~1
git checkout Feature

git branch Feature HEAD^
git checkout Feature

Branch: nach einem Commit ..

.. Nur erkennbar an Zeigern

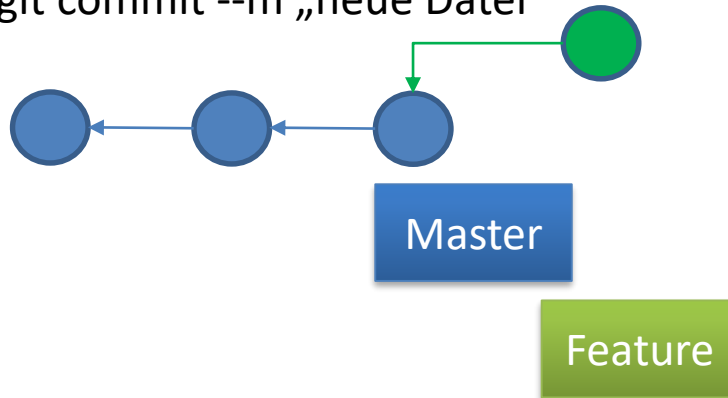
```
git checkout -b Feature  
touch featuredatei.txt  
git add featuredatei.txt  
git commit --m „neue Datei“
```



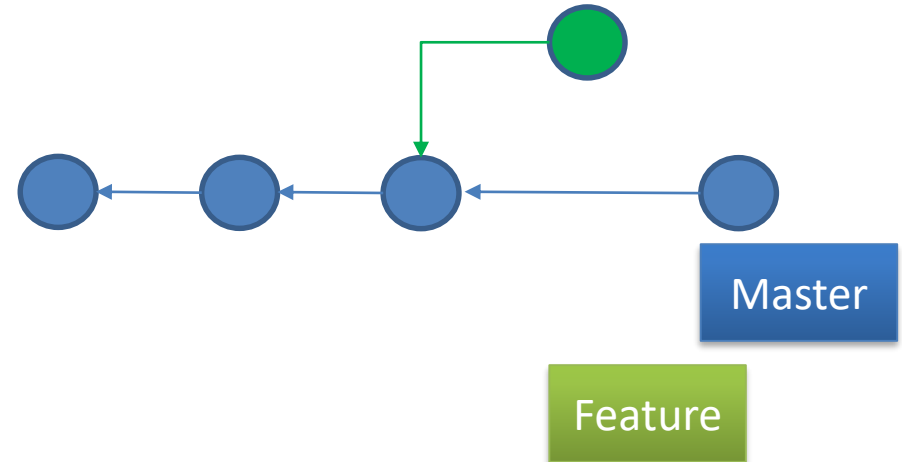
Branche: nach Commits in jedem Branch..

.. Zwei Zweige auch visuell sichtbar

```
git checkout -b Feature  
touch featuredatei.txt  
git add featuredatei.txt  
git commit --m „neue Datei“
```

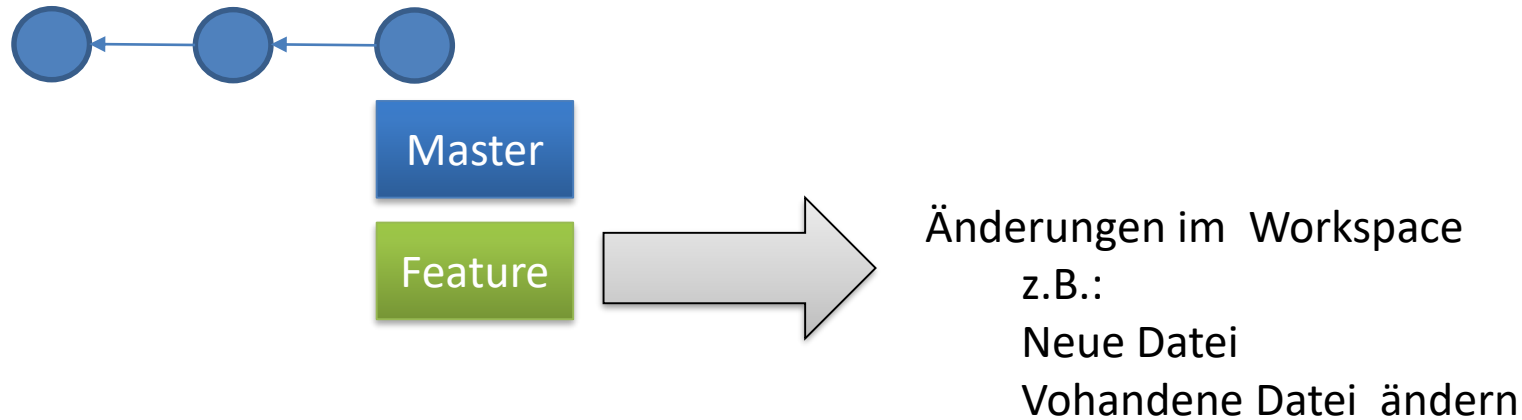


```
git checkout master  
touch masterdatei.txt  
git add masterdatei.txt  
git commit --m „neue Datei“
```



Branch wechseln ...

... mit nicht commiteten Änderungen

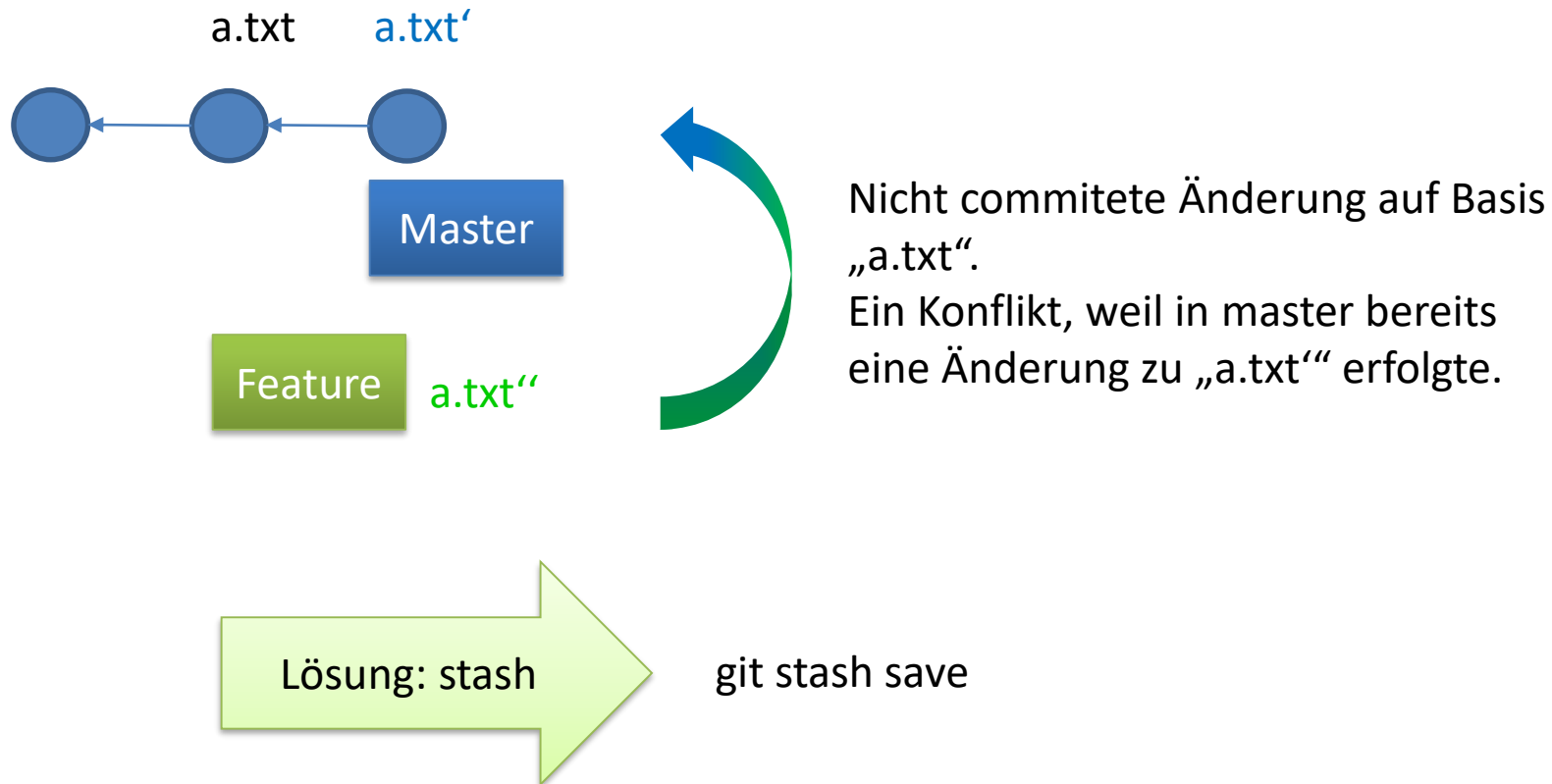


Änderungen werden mit genommen.
Ggf. im Index

Branch wechseln mit uncommiteteten Änderungen: Problemsituationen

- Konfliktmöglichkeiten
 - Änderung einer Datei, die im Zielbranch noch gar nicht vorhanden ist
 - Änderung einer Datei, die im Zielbranch bereits geändert wurde (kann durch durch eine Änderung im Remote-Repository auftreten)

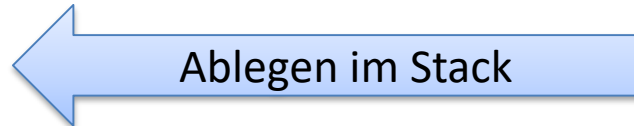
Beispiel



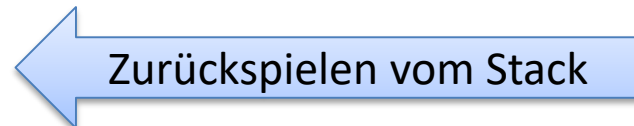
Stash

- Ein Stack (Speicher)
 - Zwischenspeichern von nicht committeten Änderungen

git stash



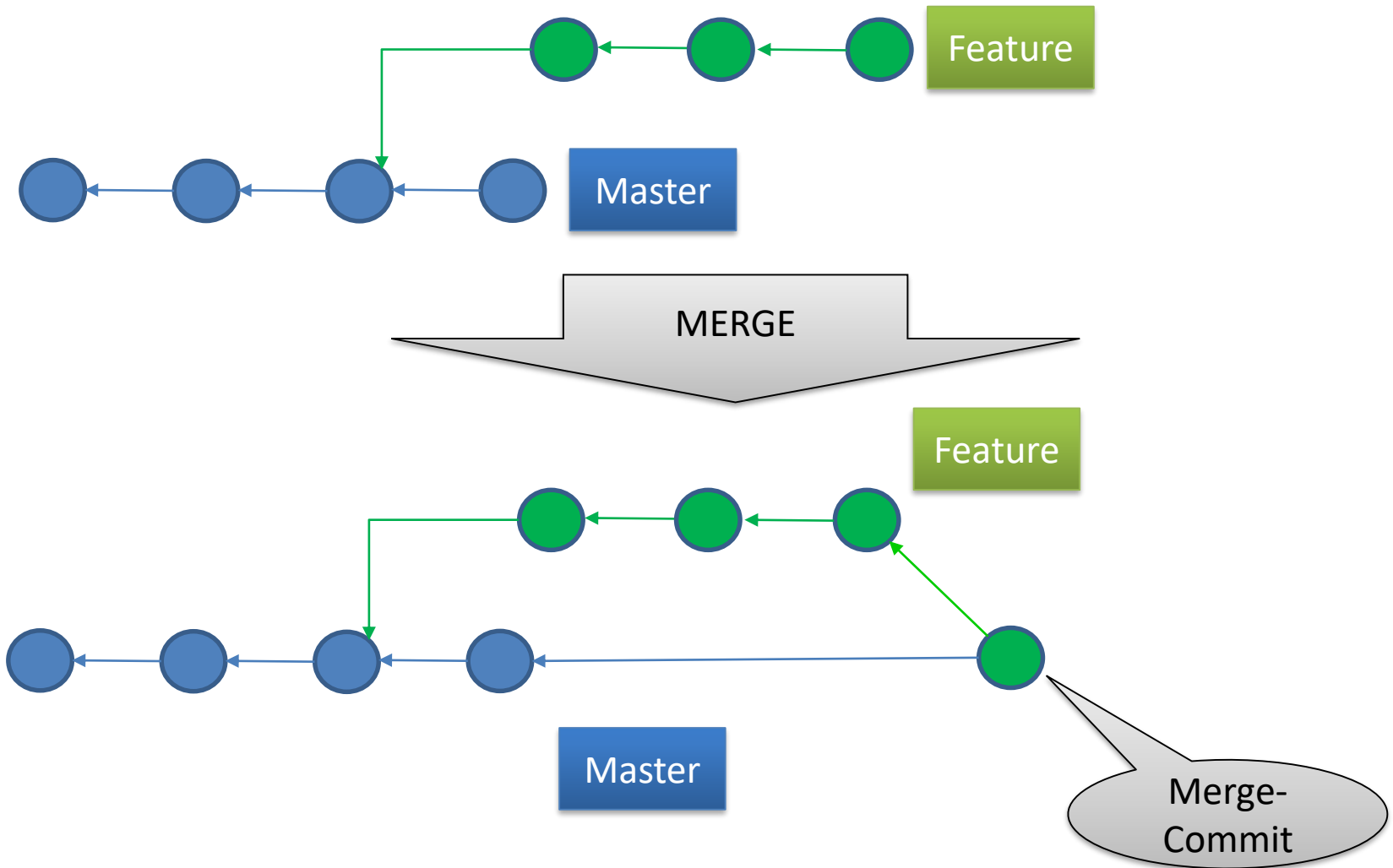
git stash apply
git stash pop



BRANCHE ZUSAMMEN FÜHREN

MERGEN

Merge

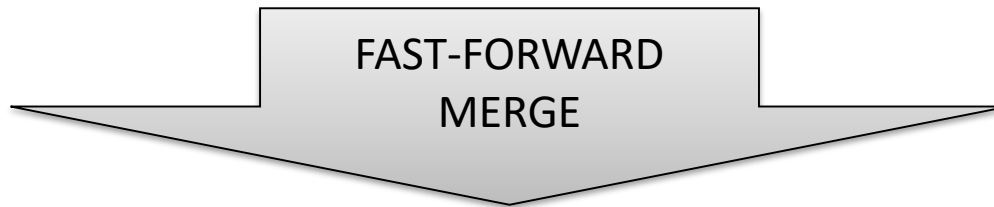
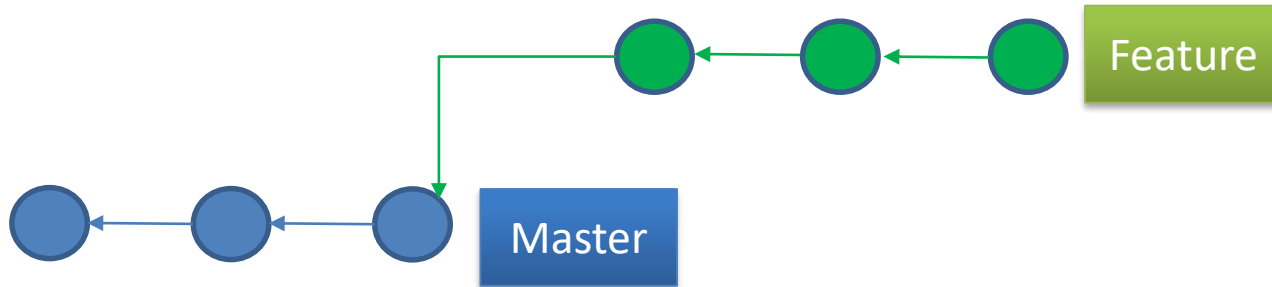


Merge durchführen

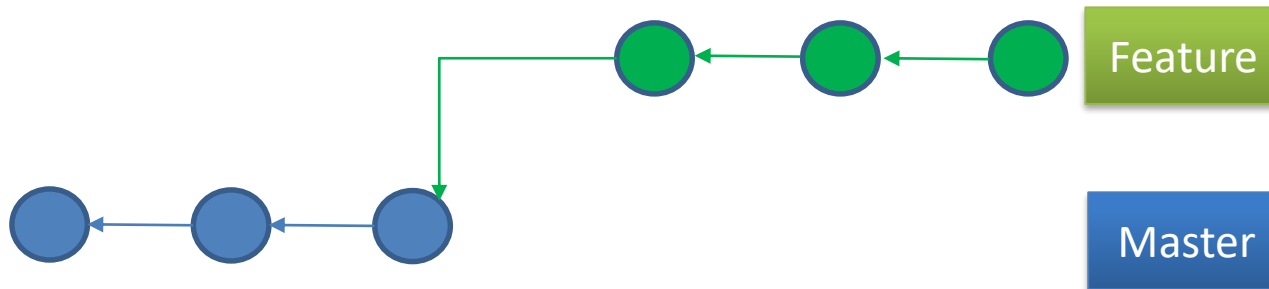
- „git pull“
 - Merge zwischen Remote- und Lokal-Repository im gleichen Branch
- „git merge“
 - Merge zwischen verschiedenen Branchen

Einfache Merge-Strategien:

Fast Forward Merge



Erzwingbar mit der Option
„--ff-only“



Merge mit Konflikt

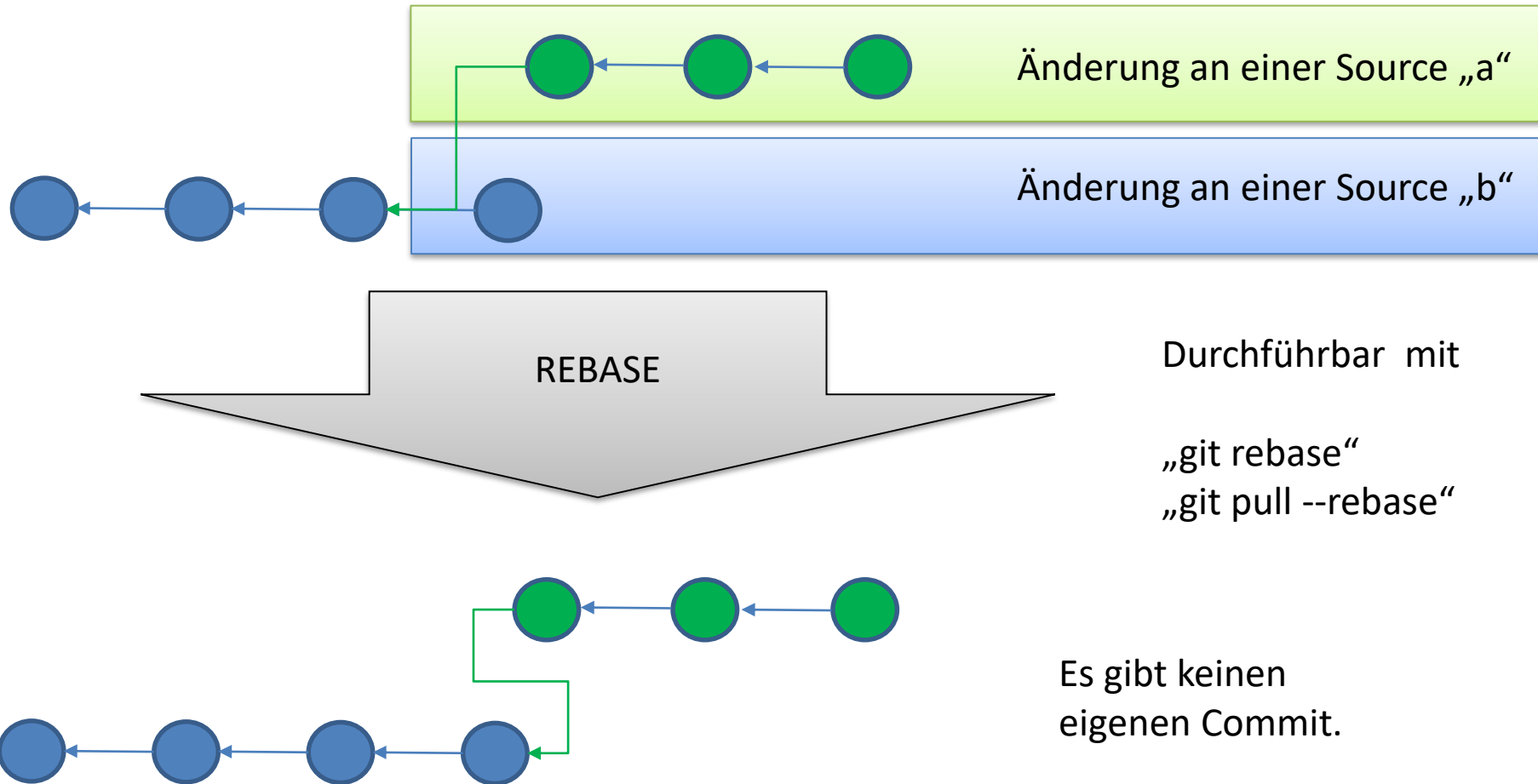
Konflikt auflösen!

Geänderte Datei mit aufnehmen: „git add <DATEI>“

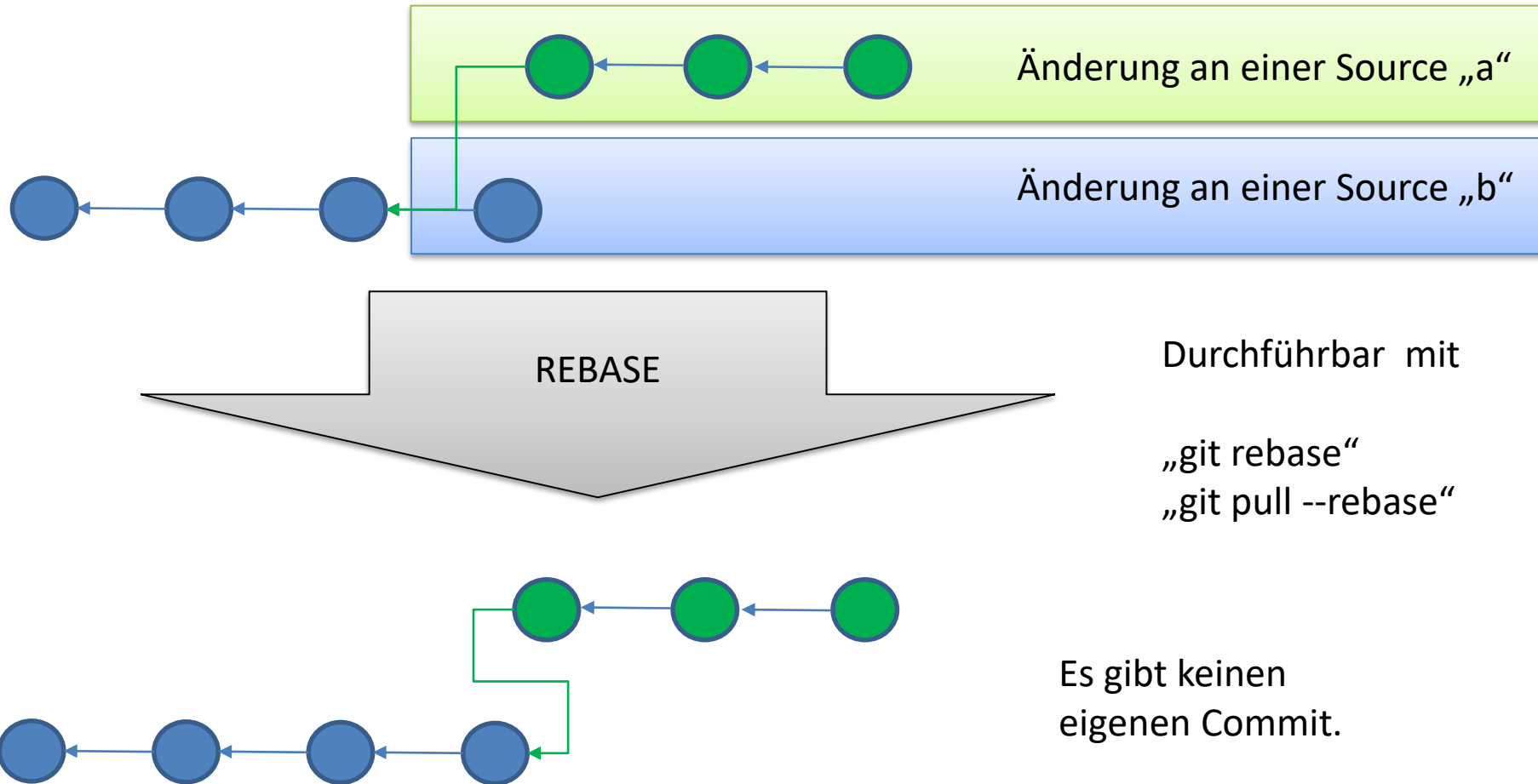
Commit durchführen: „git commit -m „....““

REBASING

Konfliktfreie Parallelität



Konfliktfreie Parallelität



Rebase mit Konflikt

Konflikt auflösen!

Geänderte Datei mit aufnehmen: „git add <DATEI>“

Rebase fortsetzen: „git rebase --continue“

- Beenden und ignorieren
 - „git rebase --abort“
 - Abbruch des Rebasing
 - „git rebase --skip“
 - Überspringen des Konflikt

BRANCH

Branch-Befehle

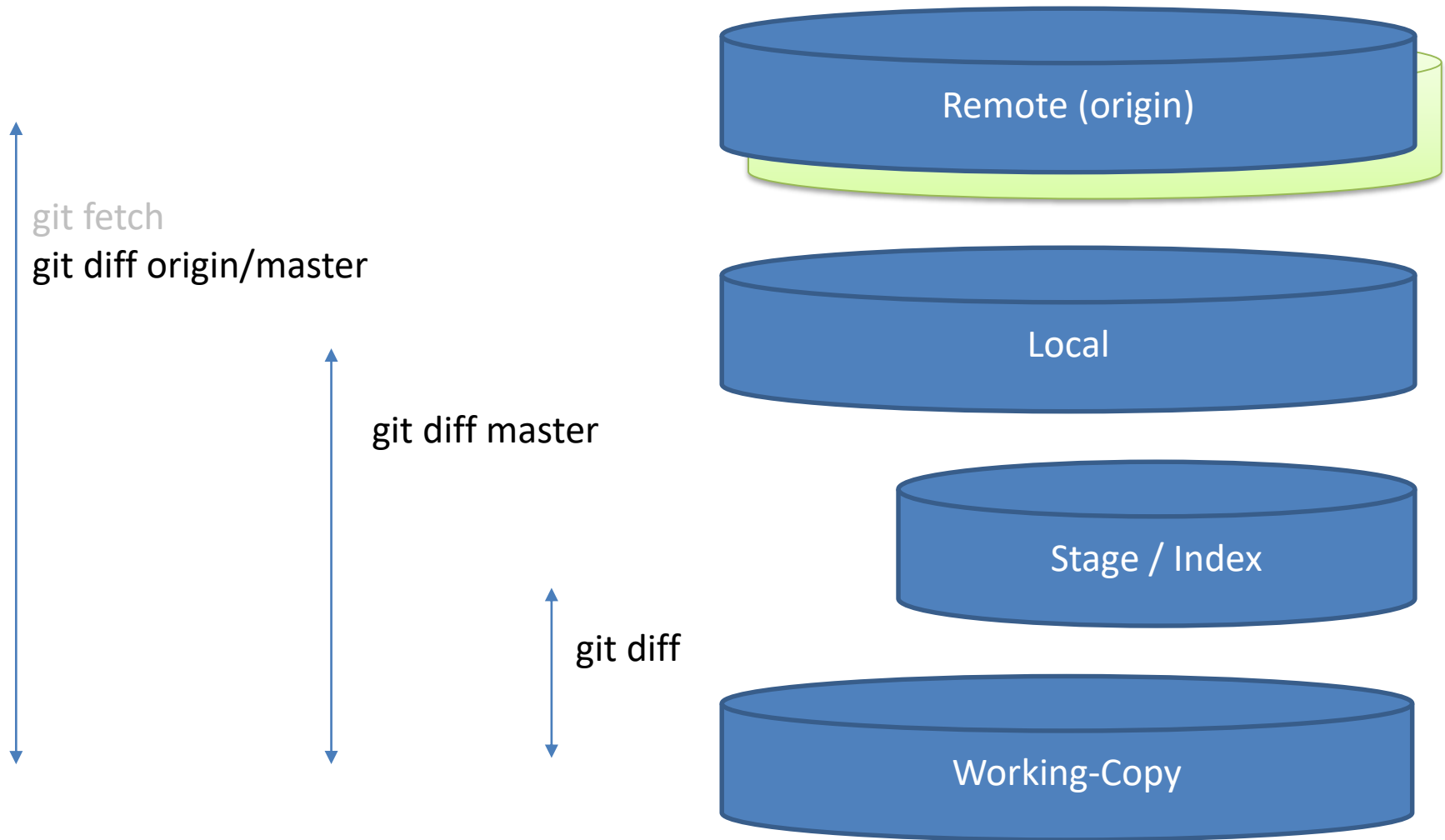
- „git branch <name>“
 - Ein Branch wird erstellt
- „git checkout <name>“
 - In Branch wechseln
- „git checkout –b <name>“
 - Kombination: erstellen und wechseln
- „git branch --d <name>“
 - Branch löschen
- „git branch --D <name>“
 - Löschen eines noch nicht zurück geführten Branch

DIFFS

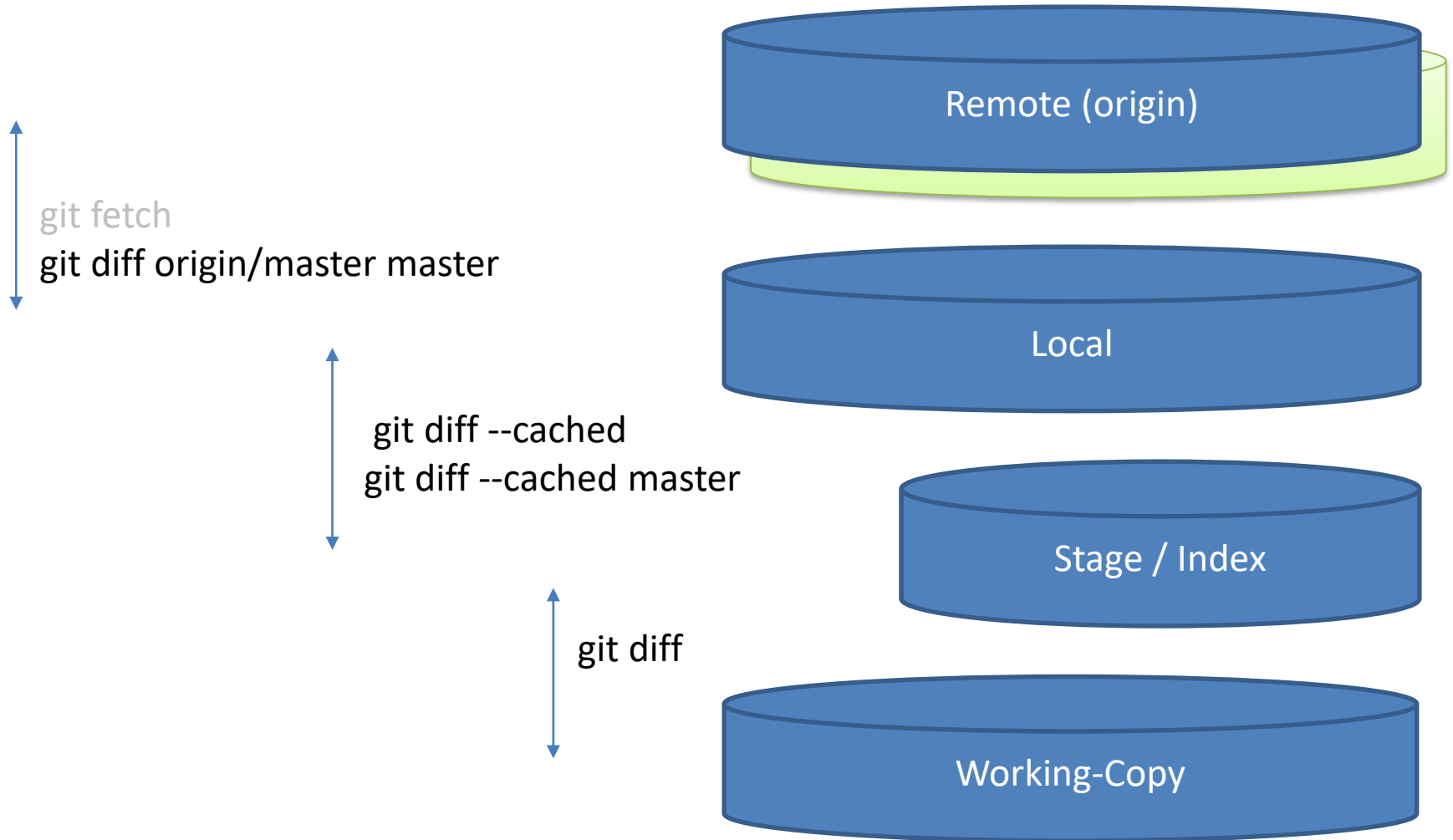
Allgemein

- „git diff“
 - Kann zwischen Repositories verwendet werden
 - Kann zwischen Branches verwendet werden
 - Kann zwischen Commits verwendet werden

Diffs von Working-Copy

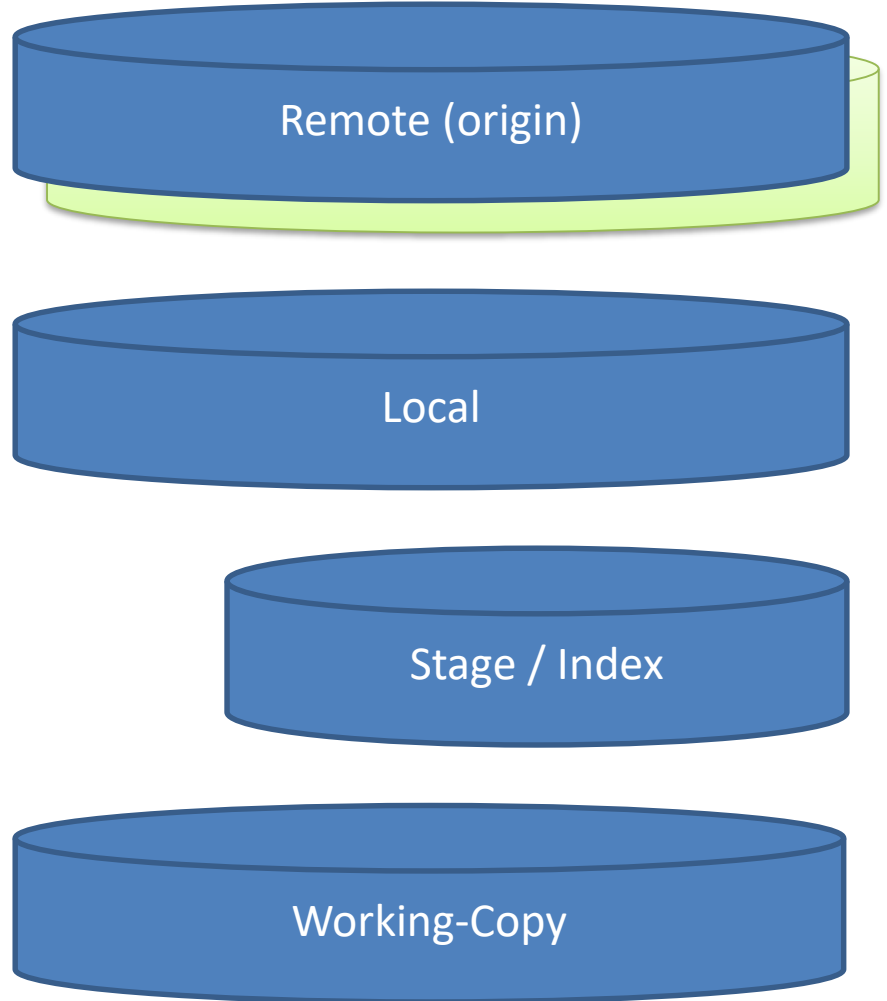


Diffs zwischen den Bereichen



Diffs zwischen den Bereichen

git fetch
git diff --cached origin/master




COMMITTS VERÄNDERN

git reset

	HEAD	INDEX	WORKING	
--soft	JA	-	-	Macht nur Sinn mit einem commit „git reset --soft HEAD~2“
--mixed	JA	JA	-	
--hard	JA	JA	JA	


Reset - Befehl



„git reset --hard <...>

HEAD wird auf einen bestimmten Commit gesetzt.

Workspace und Index werden auch auf diesen Stand gesetzt.

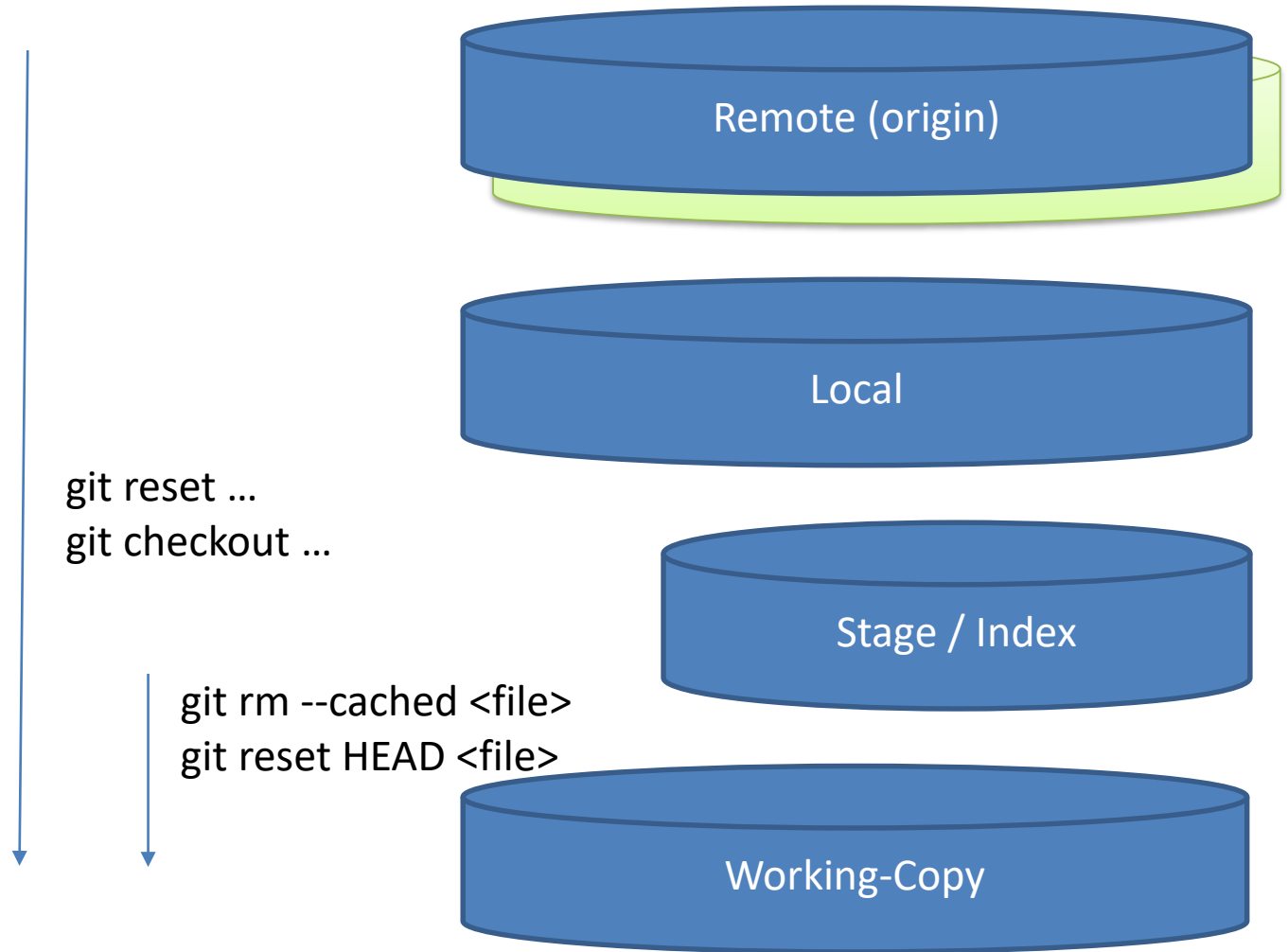


„git reset --soft <...>

HEAD wird auf einen bestimmten Commit gesetzt.

Workspace bleiben bestehen

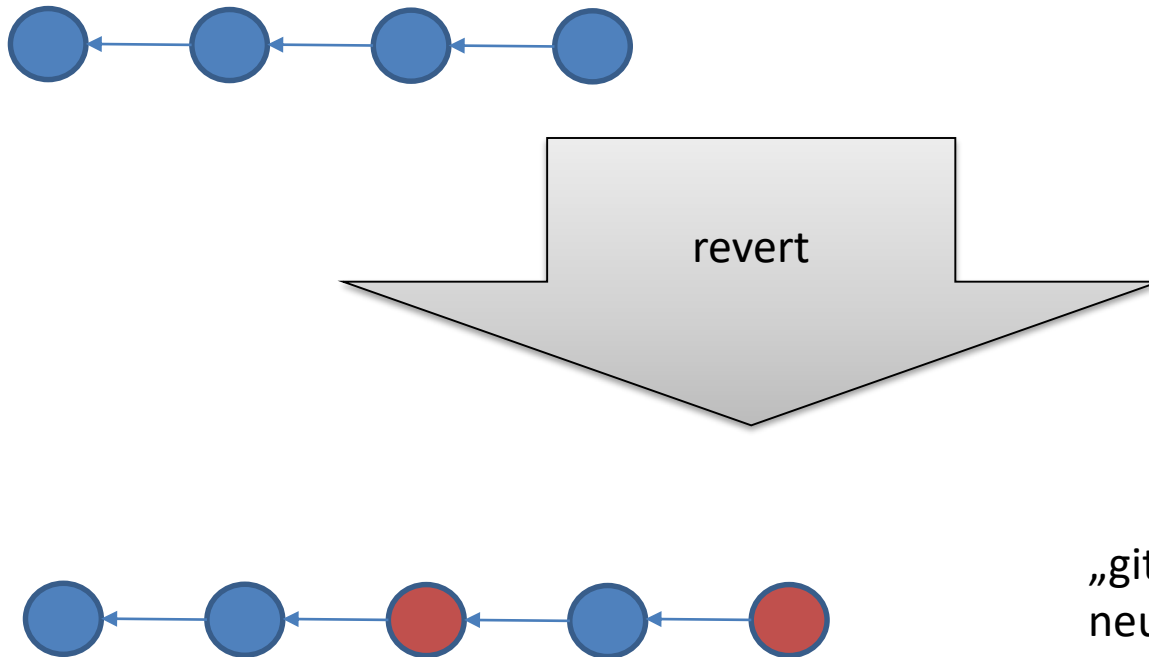
Ein paar Undo's



Letzte Commit-Message nachbearbeiten

- „git commit --amend“
 - Editor (Vi) wird geöffnet
- „git commit--amend --m „neue Nachricht““
 - Letztes Commit wird geändert


Rückgängig durch neues commit



„git revert“ erzeugt ein
neues Commit

Interaktives Rebasing

„git rebase --i HEAD~3“



```
r abcdef Kommentar1  
s abcde2 Kommentar 2  
s abcde5 Kommentar 3
```

„p“ >>> übernehmen

„r“ >>> neue Commit-Message

„e“ >>> neue Commit-Message im edit-Modus (vi)

„s“ >>> squash (ins vorherige Commit)

„f“ >>> fixup (wie „s“, lässt Commit-Message fallen)

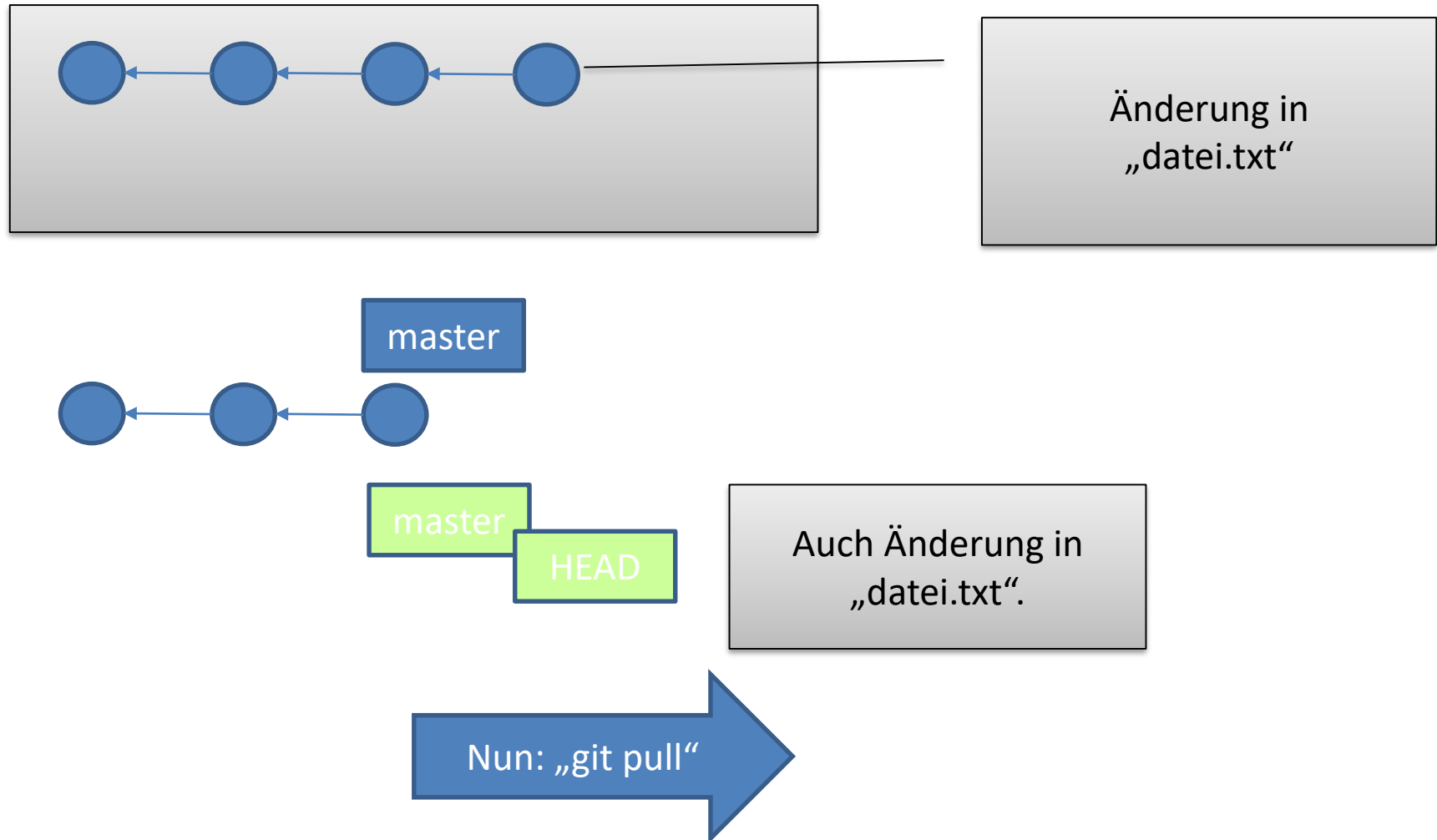
„x“ >>> exit

ZWISCHENSPEICHERN

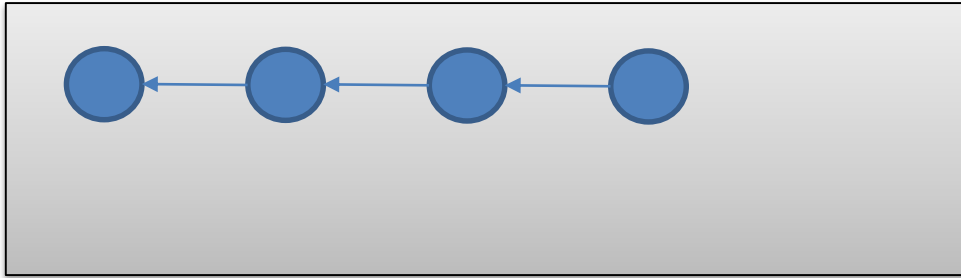
Situationen

- Aktuelle Änderungen sollen beiseite gelegt werden
- Branchwechsel mit Änderung, aber hierdurch würde ein Konflikt auftreten

Beispiel

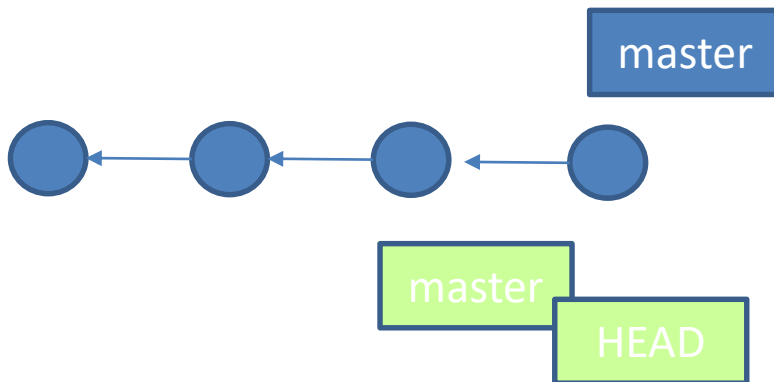


Beispiel



NUN:

Die Änderung in „datei.txt“ ist durch „git pull“ im master angekommen.

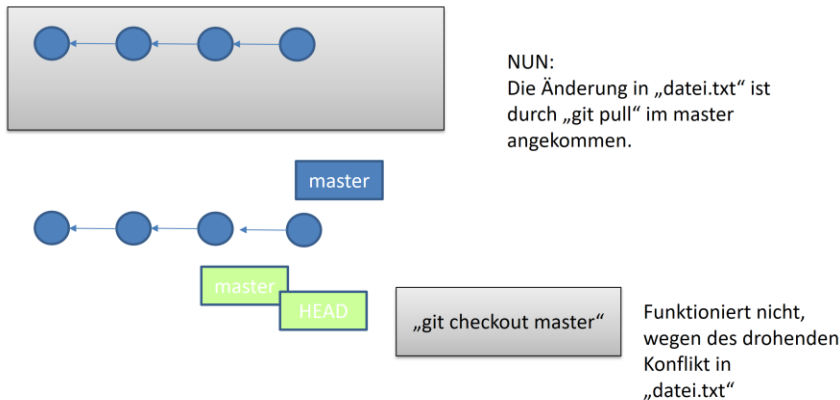


„git checkout master“

Funktioniert nicht, wegen des drohenden Konflikt in „datei.txt“

Möglichkeiten aus dem Konflikt

- Zunächst Änderung in einen neuen Branch sichern
- Änderung stashen



Stash

- Ein einfacher Speicher in Form eines „Stack“
 - Dient zum schnellen bereinigen nicht gesicherter (commiteter) Änderungen
 - Die nicht in den aktuellen Branch genommen werden sollen
 - Die nicht in einen Branch mit genommen werden sollen
 - Die nicht in einen Branch mit genommen werden können

git stash save

git stash list

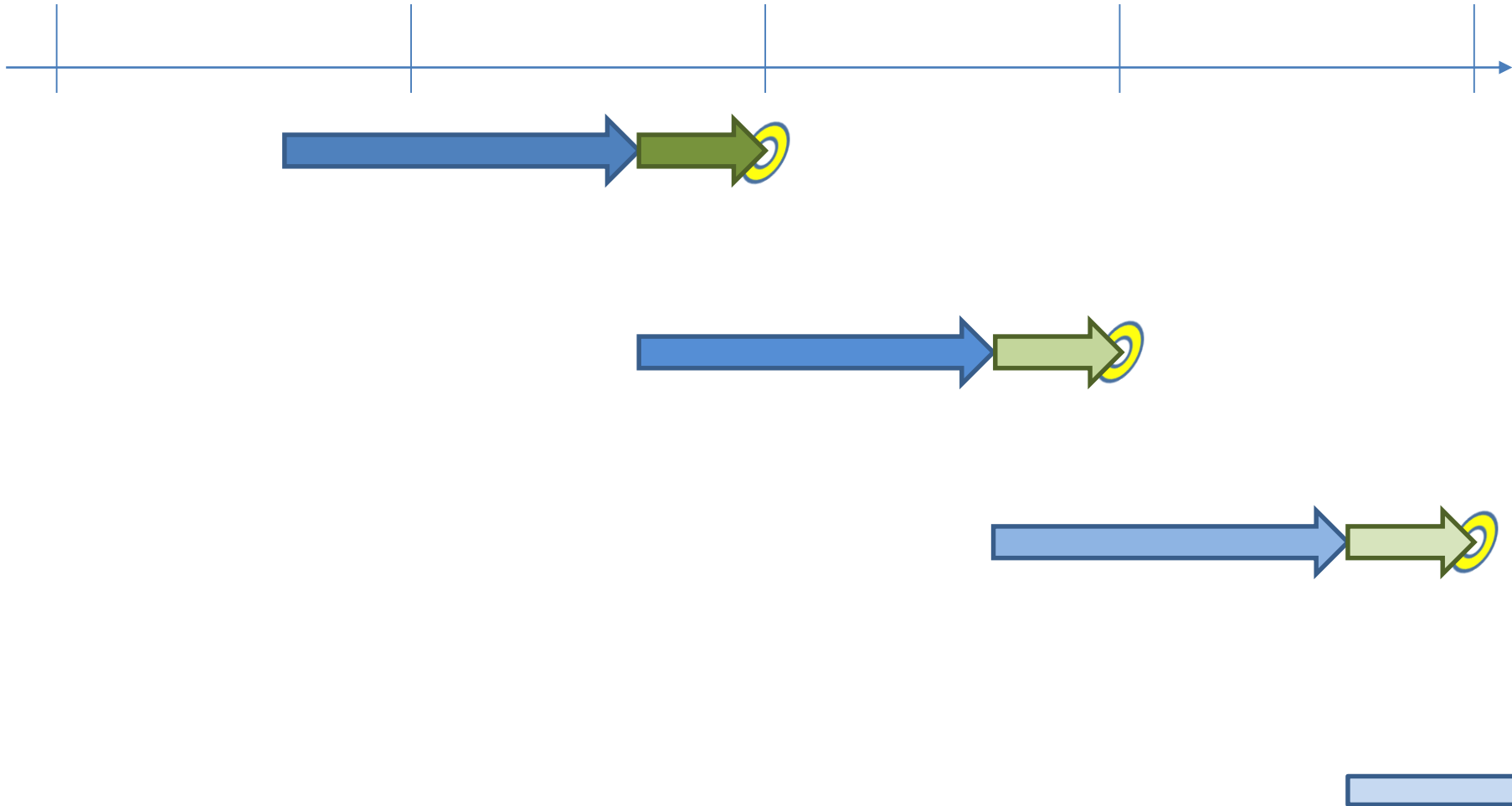
git stash apply save

WORKFLOWS MIT GIT

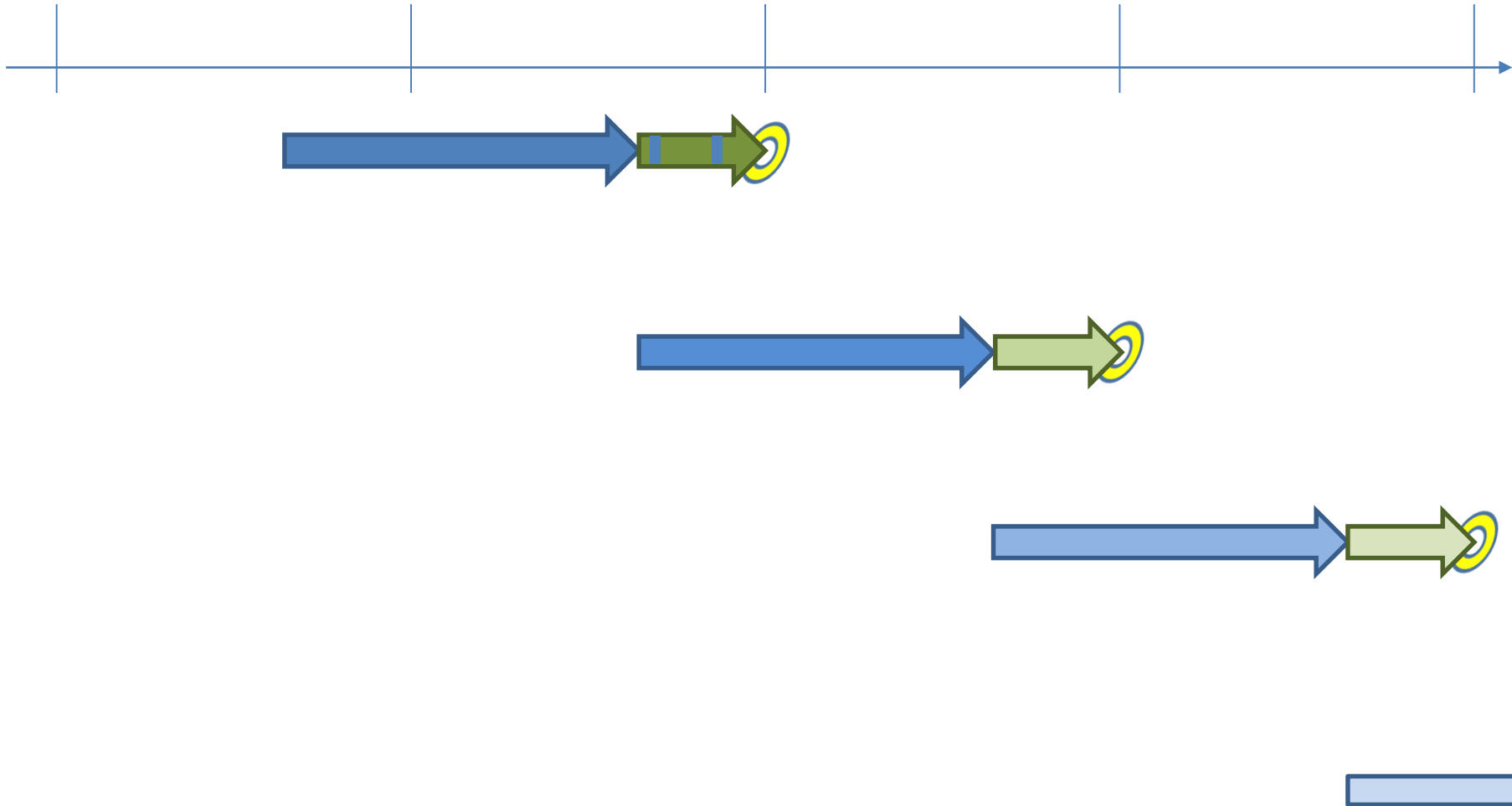
Worum geht es bei den Workflows?

- Wie gelangen Entwicklungsartefakte in Produktion?
 - Normale eingeplante Entwicklungen/Features
 - Kurzfristige Features
 - Bugfixes
- Wie gelangen und bleiben Entwicklungsartefakte in Produktion?

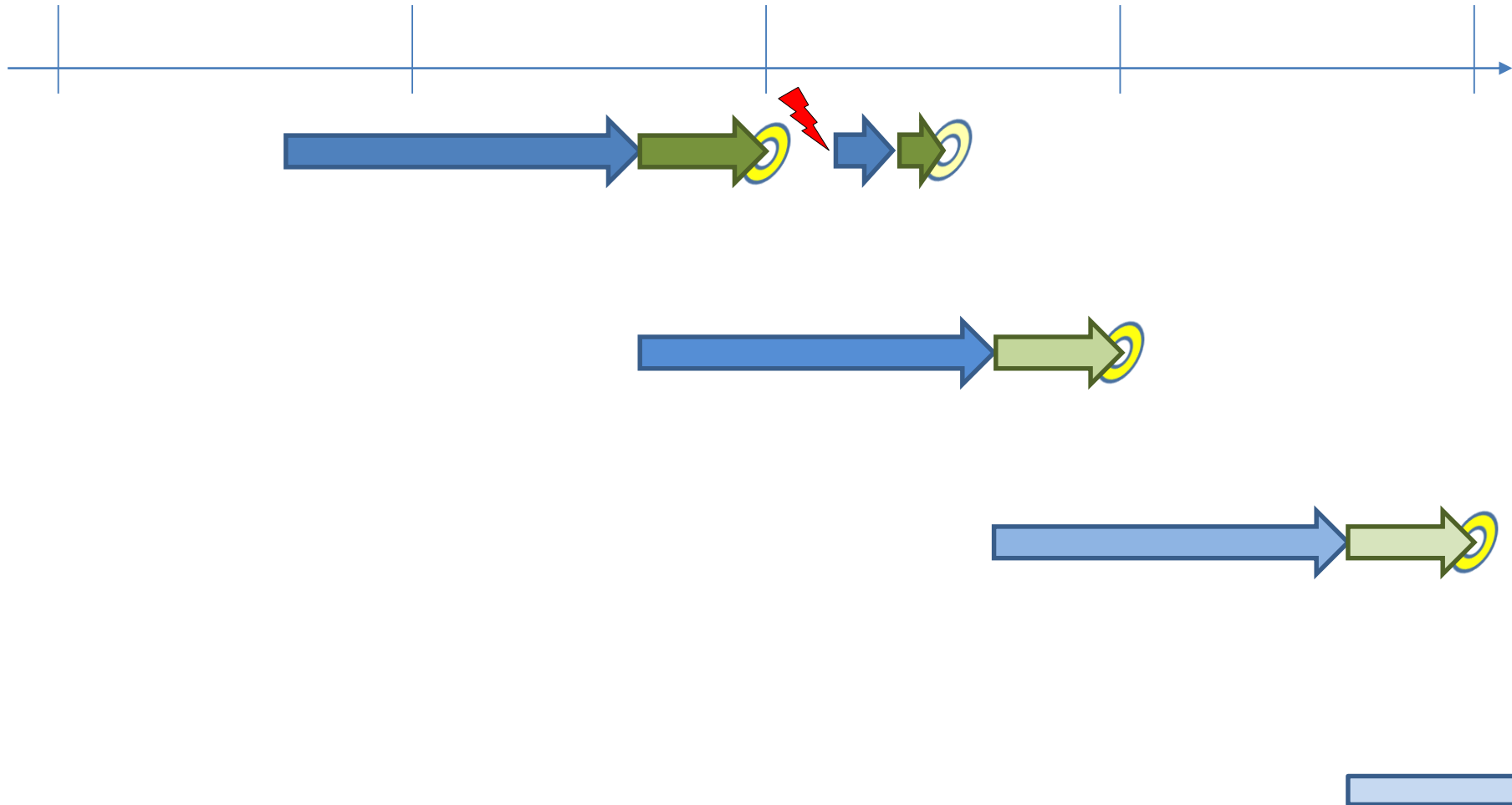
Beispiel: Entwicklungszyklus



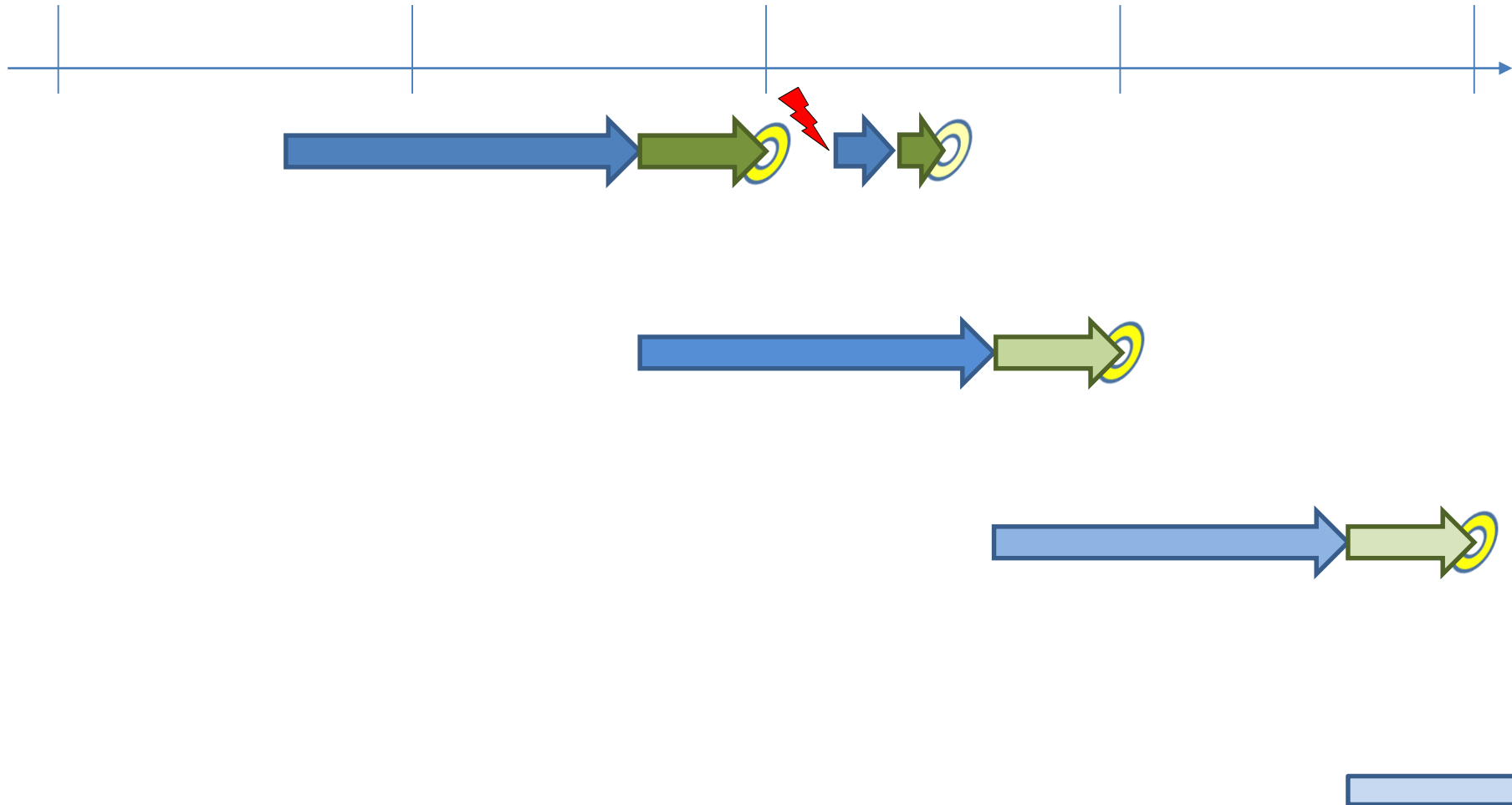
Beispiel: Entwicklungszyklus mit Bugfix



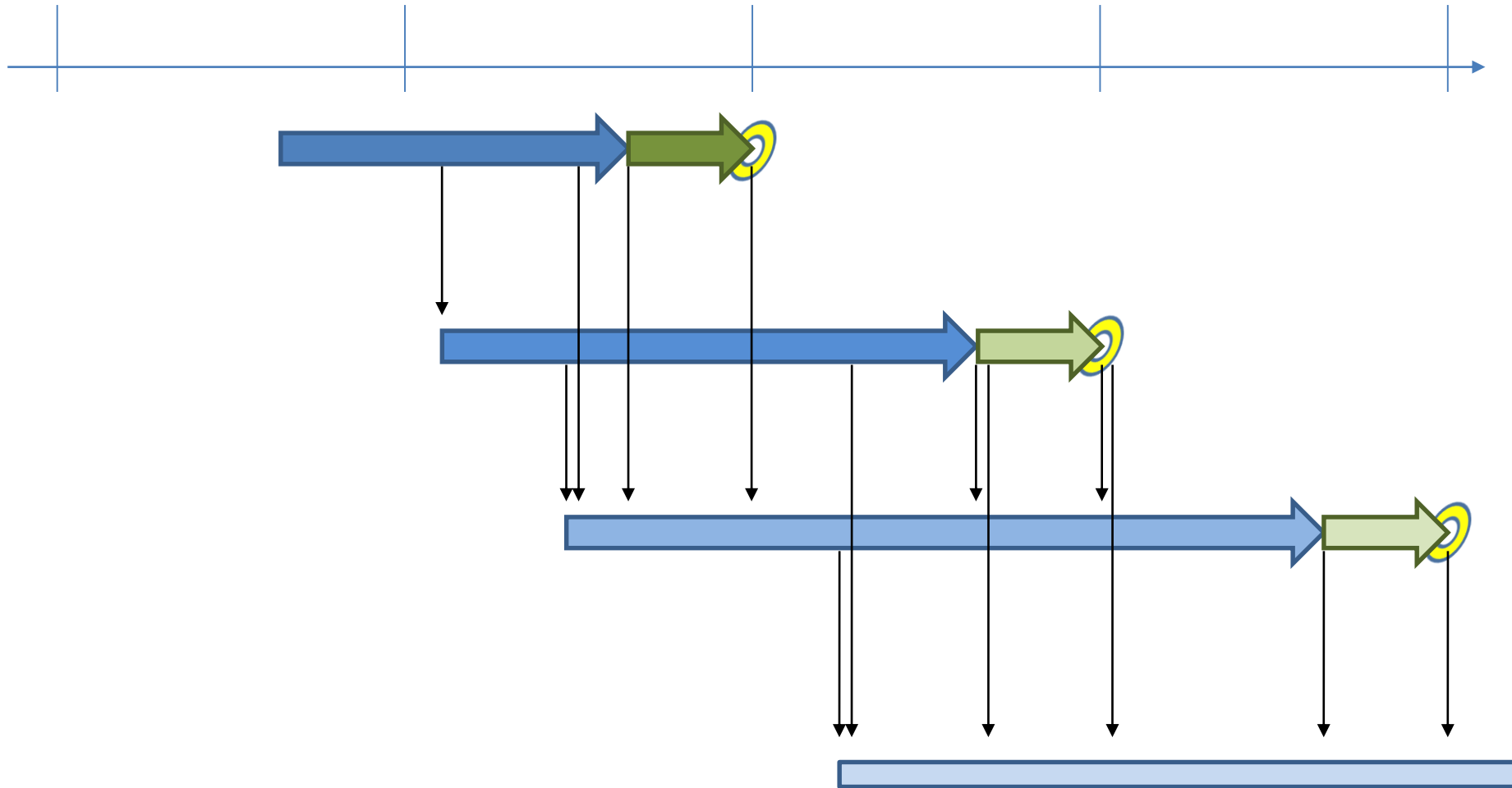
Beispiel: Entwicklungszyklus mit Bugfix



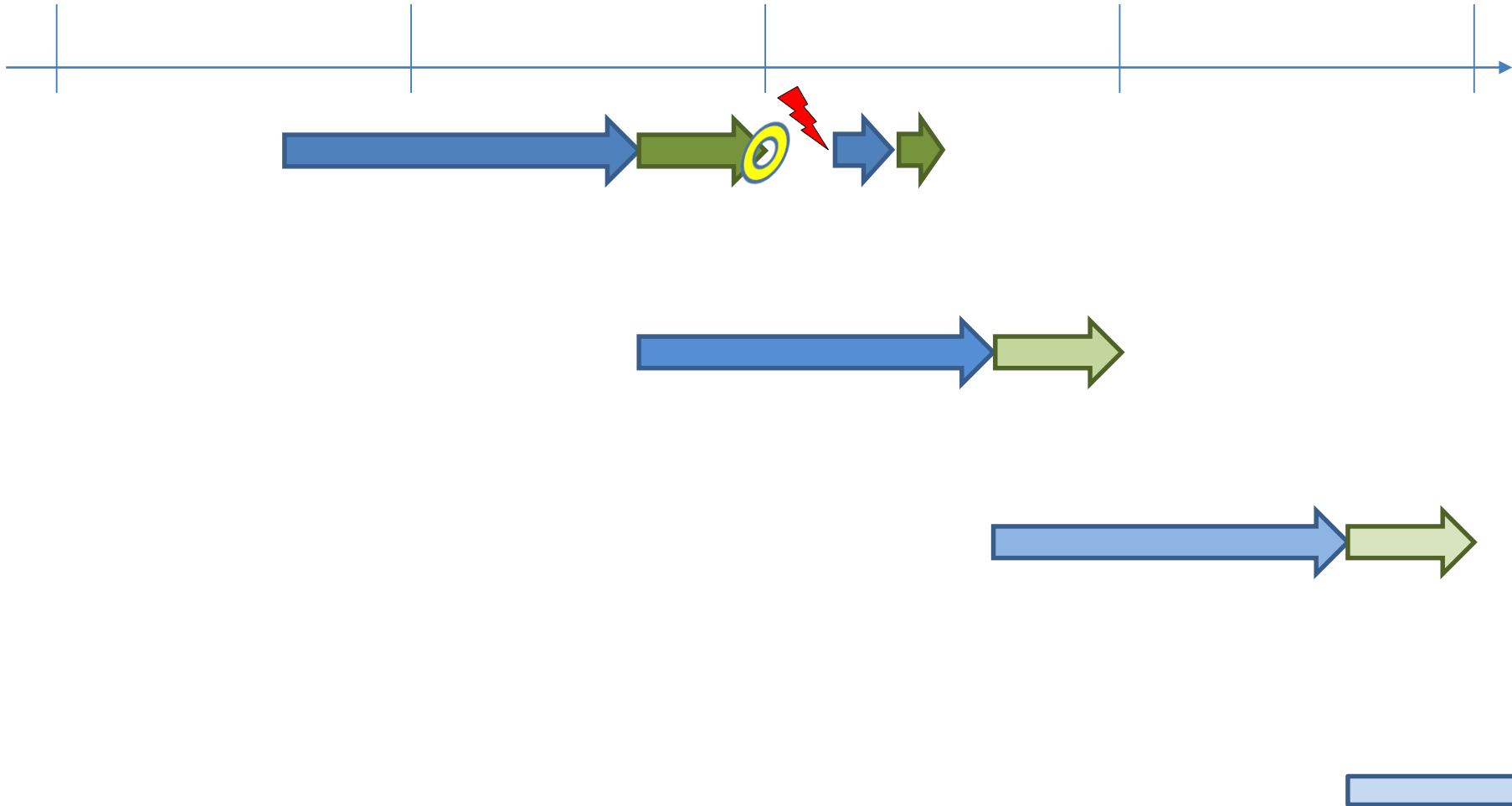
Beispiel: Entwicklungszyklus mit Bugfix



Beispiel: Entwicklungszyklus mit geplanten Features



Beispiel: Entwicklungszyklus mit BugFix



Typische Fragestellungen

- Wie erzeugen wir Artefakte ...
 - Für die Teams und Projekte
 - Für den Test
 - Für das Release
- Wie erstellen wir ein Bugfix zu einer Auslieferung
 - Und bringen des Bugfix wieder in die anderen Teams und Projekte

Workflows

- Zentralisierter Workflow
- Feature Branch Workflow
- Gitflow Workflow
- Forking Workflow

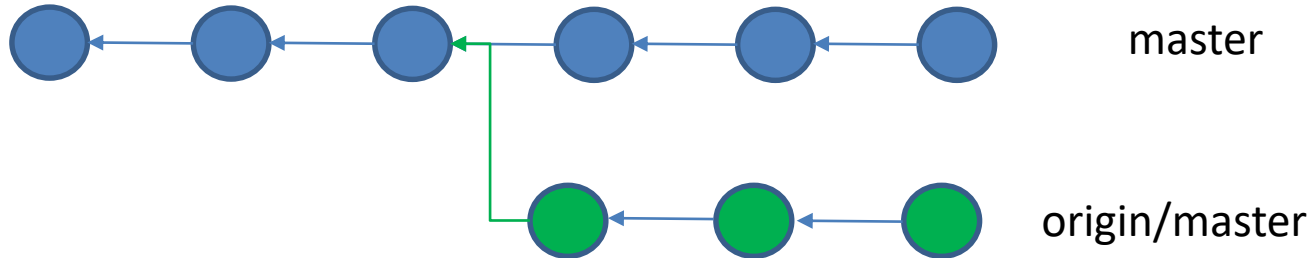
Zentralisierter Workflow

- Ähnlichkeit mit Subversion-Workflow
- Daher:
 - Einfacher Umstieg für Subversion-Nutzer
- ABER:
 - Man nutzt nicht die vollen Eigenschaften von GIT
- Vorteile gegenüber Subversion
 - Alle Sourcen liegen beim Nutzer
 - Nutzung der guten Merge-Eigenschaften von GIT

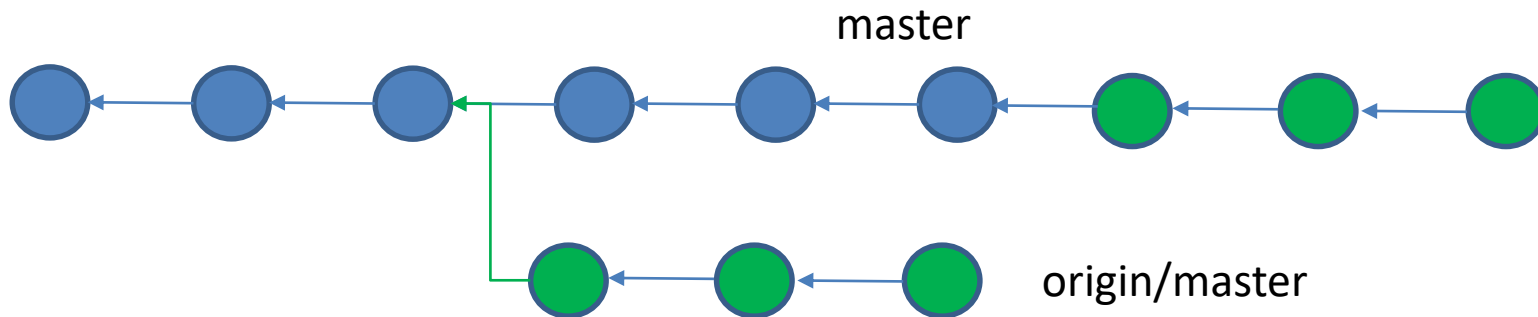
Zentralisierter Workflow

- Ein zentrales Repository mit einem zentralen Branch (master)
- Alle Änderungen werden in master committet
- Nach Abschluss:
 - Änderungen werden in das zentrale Repository im master-Branch gepushed
 - Entspricht quasi „svn commit“

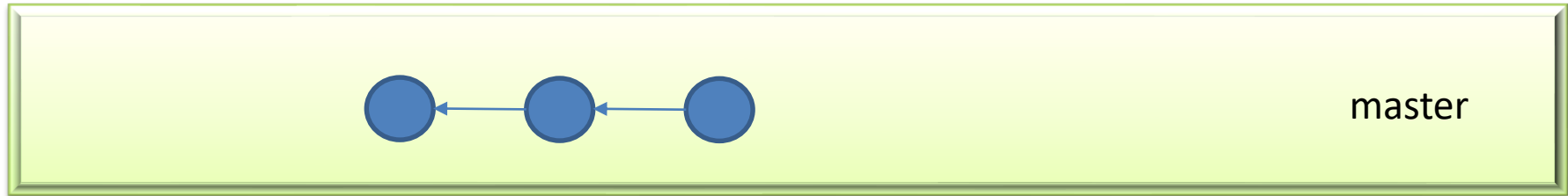
Zentralisierter Workflow: Konfliktbewältigung



`git pull --rebase origin master`



Zentralisiert, ohne Konflikte



1 : git clone ...



master

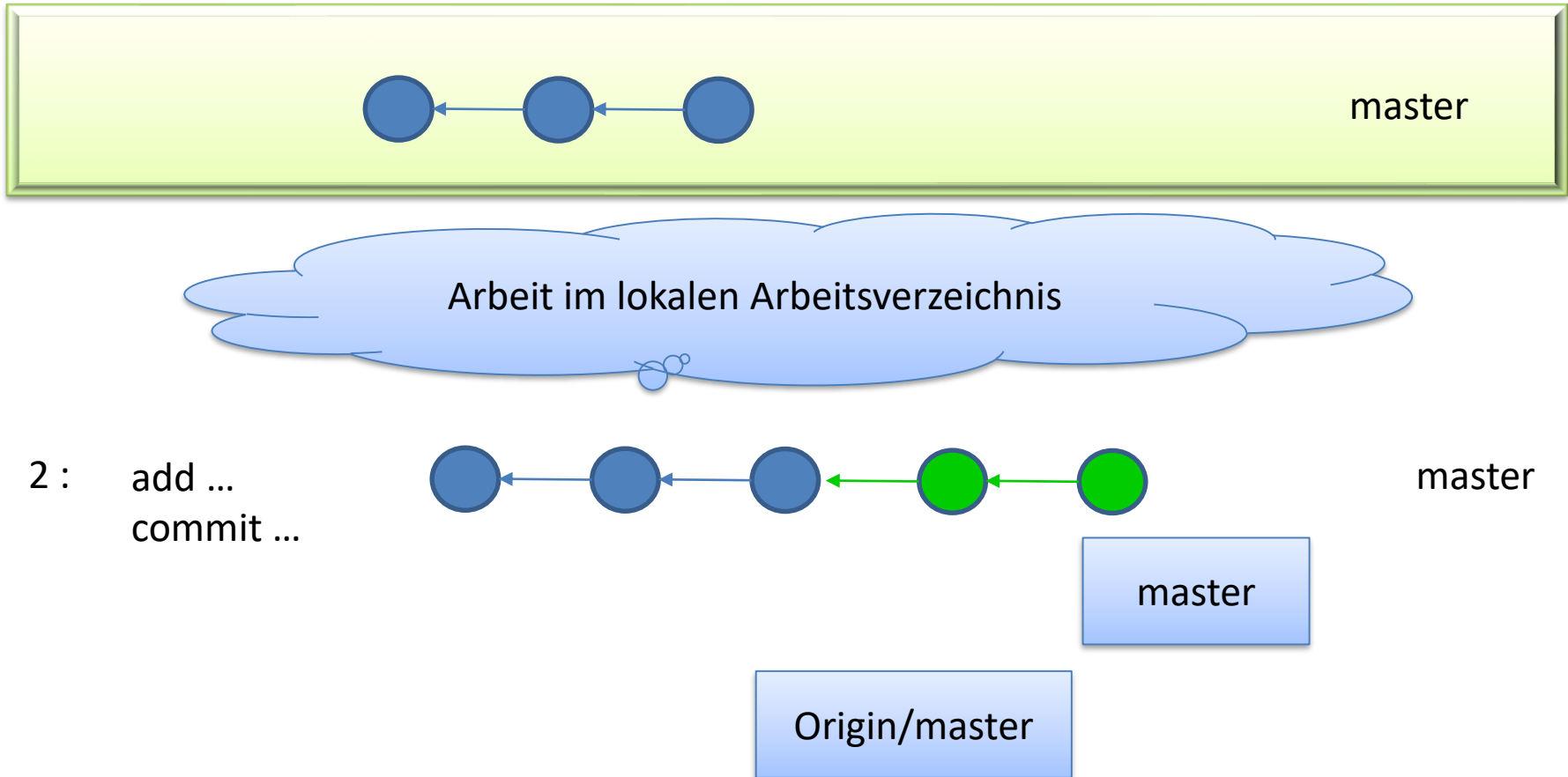
master

Origin/master

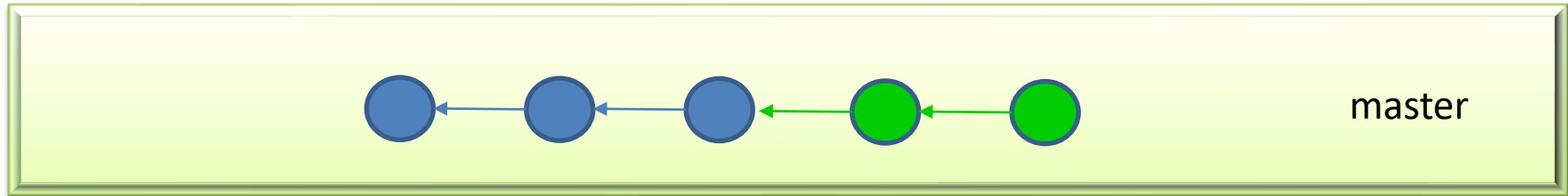
2 :

Arbeit im lokalen Arbeitsverzeichnis

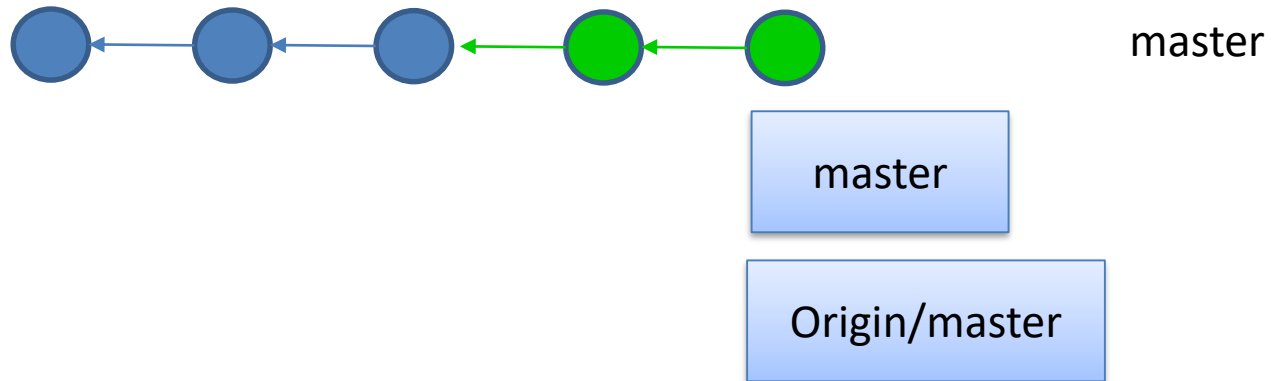
Zentralisiert, ohne Konflikte



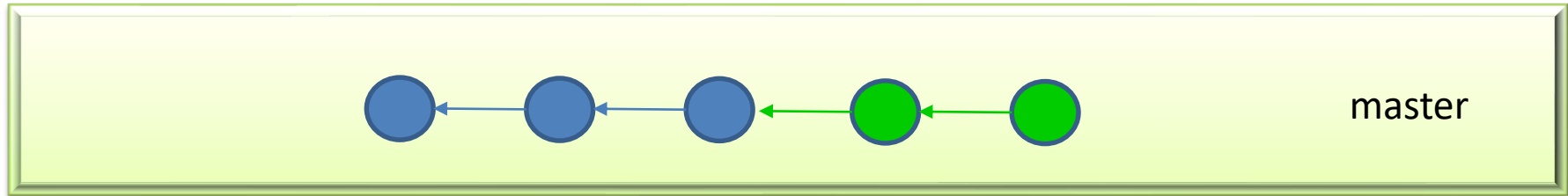
Zentralisiert, ohne Konflikte



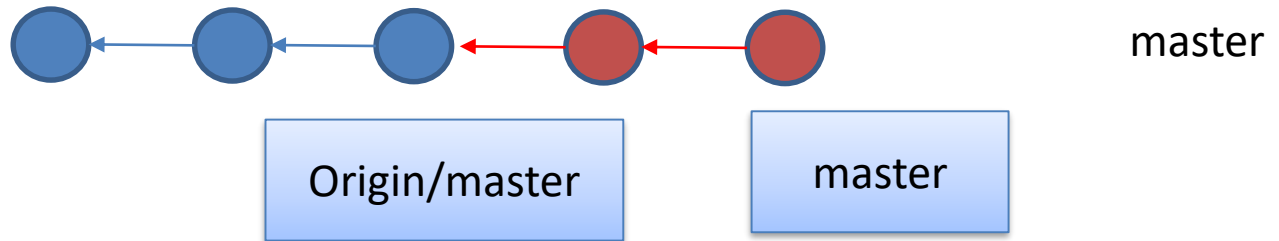
2 : git push
origin master



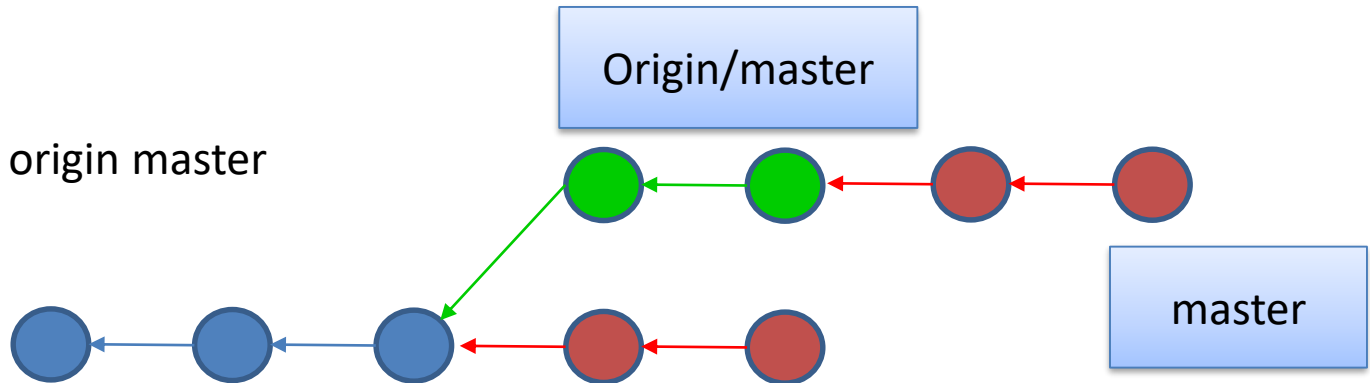
Zentralisiert, mit Konflikte Konflikte



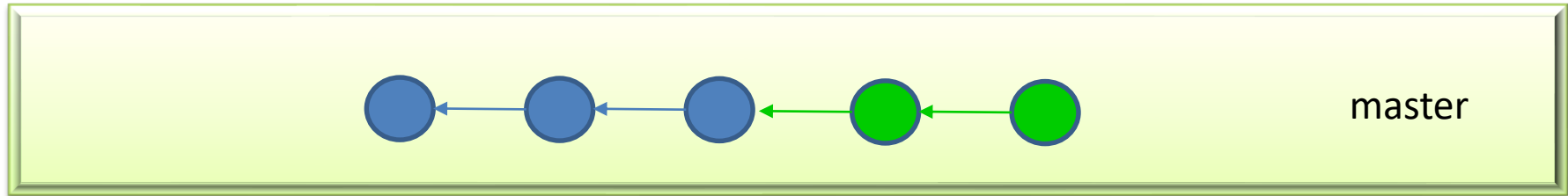
1 : git push
origin master



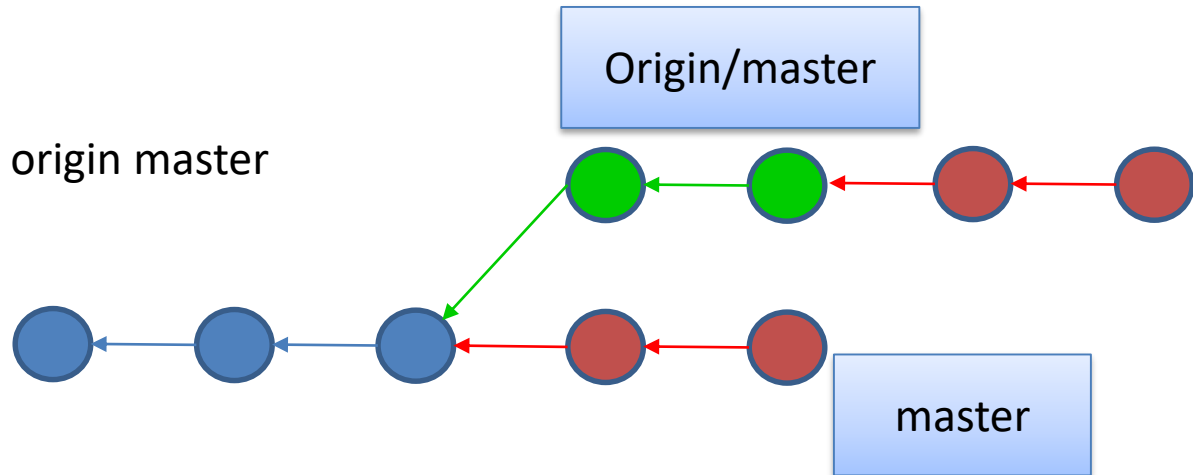
2 : git pull -rebase origin master



Zentralisiert, mit Konflikte Konflikte



2 : `git pull --rebase origin master`



3 : Bei Problemen, wenn das Rebase nicht funktioniert:

<Änderungen durchführen>

`git add ...`

`git rebase --continue`

Zentralisiert, mit Konflikte Konflikte



4 : `git pull -rebase origin master`

