

# Softwaretechnik

## Einführung in Git

Prof. Dr. Bodo Kraft

Prof. Dr. Andreas Terstegge

Hendrik Karwanni

Moritz Mathieu

Johanna Roussel

Oliver Schmidts

# Motivation

## Versionsverwaltung mit Git

---

### **... irgendwann während der Entwicklung ...**

- Da war doch ...?
- Das hatte ich doch gestern geändert?
- Wer hat denn ...?
- War das nicht schon immer so?
- Was stand da, als das noch funktioniert hat?
- Wie, Du hast Deinen Quelltext heute dahin kopiert?



### **... irgendwann während der Entwicklung ...**

- Gestern hat das funktioniert.  
Was ist seitdem passiert?
- Bei mir tut's das, aber warum?
- Warum sind meine Änderungen nicht im  
kompilierten Programm?
- Wo ist mein Bugfix von letzter Woche geblieben?
- Ist der Bugfix jetzt auch in meinen Dateien?



# Funktionalität von Versionsverwaltungen

## Versionsverwaltung mit Git

---

### Basisfunktionalität

- Alte Versionen wiederherstellen
- Urheber von Änderungen feststellen
- Änderungen kommentieren
- Versionen vergleichen
- Verteiltes Arbeiten am gleichen Code

### Fortgeschrittene Features

- Versionsstände „einfrieren“ (Softwarerelease festsetzen)
- Branching und Merging

### Eigenschaften

- freie Software zur verteilten Versionsverwaltung von Dateien
- wurde durch Linus Torvalds initiiert
- **dezentralisierter** Ansatz → beliebige Synchronisation mehrerer Repositories

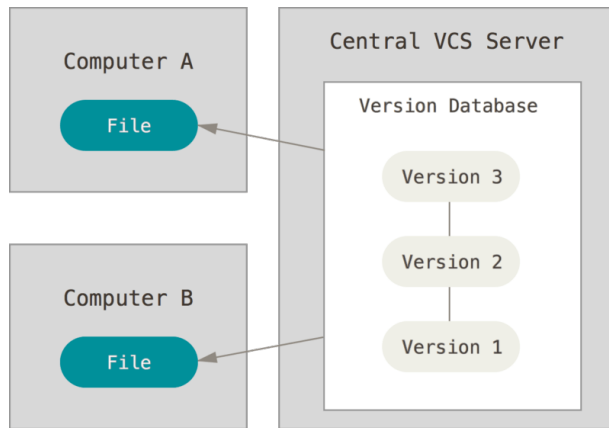
### Alternativen

- Vorgänger CVS, diverse Nachteile:
  - u.a. keine DB
  - kein Verschieben
  - Umgang mit Binärdateien schwierig
- Seit 2004: erste stabile SVN Version, Nachteile
  - Keine lokale Versionierung möglich
  - Keine Verbindung zwischen mehreren Repositories
  - Kaum Unterstützung beim Branching und Merging
- Seit 2008: Beginn der Verbreitung von dezentralen VCS (**V**ersion **C**ontrol **S**ystem)
  - Beispiele: Mercurial, BitKeeper, ...

# Zentrale vs. dezentrale VCS

## Versionsverwaltung mit Git

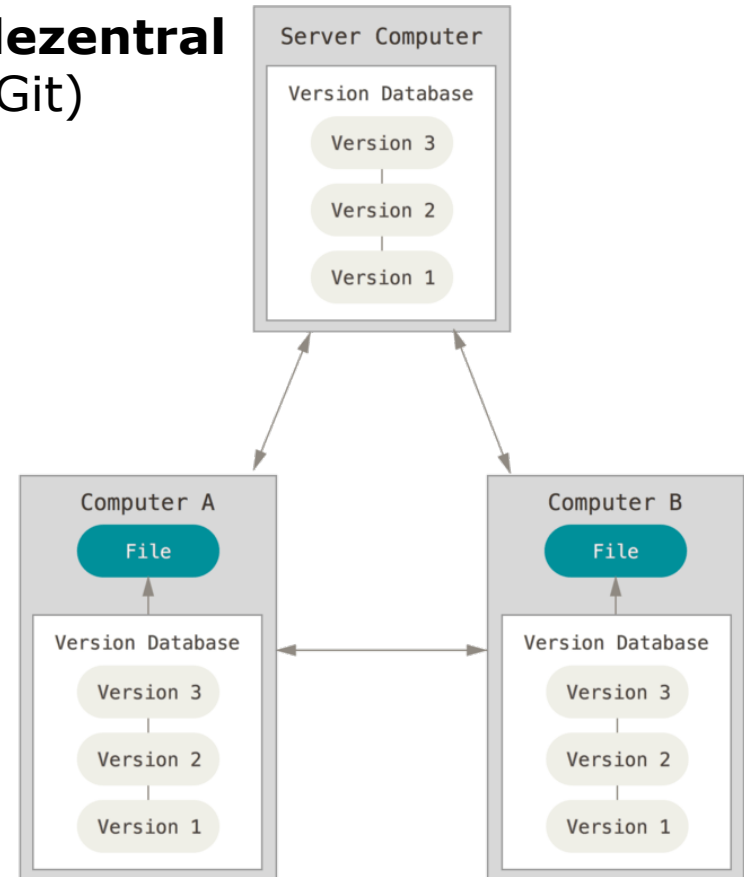
### zentral



Nachteil:

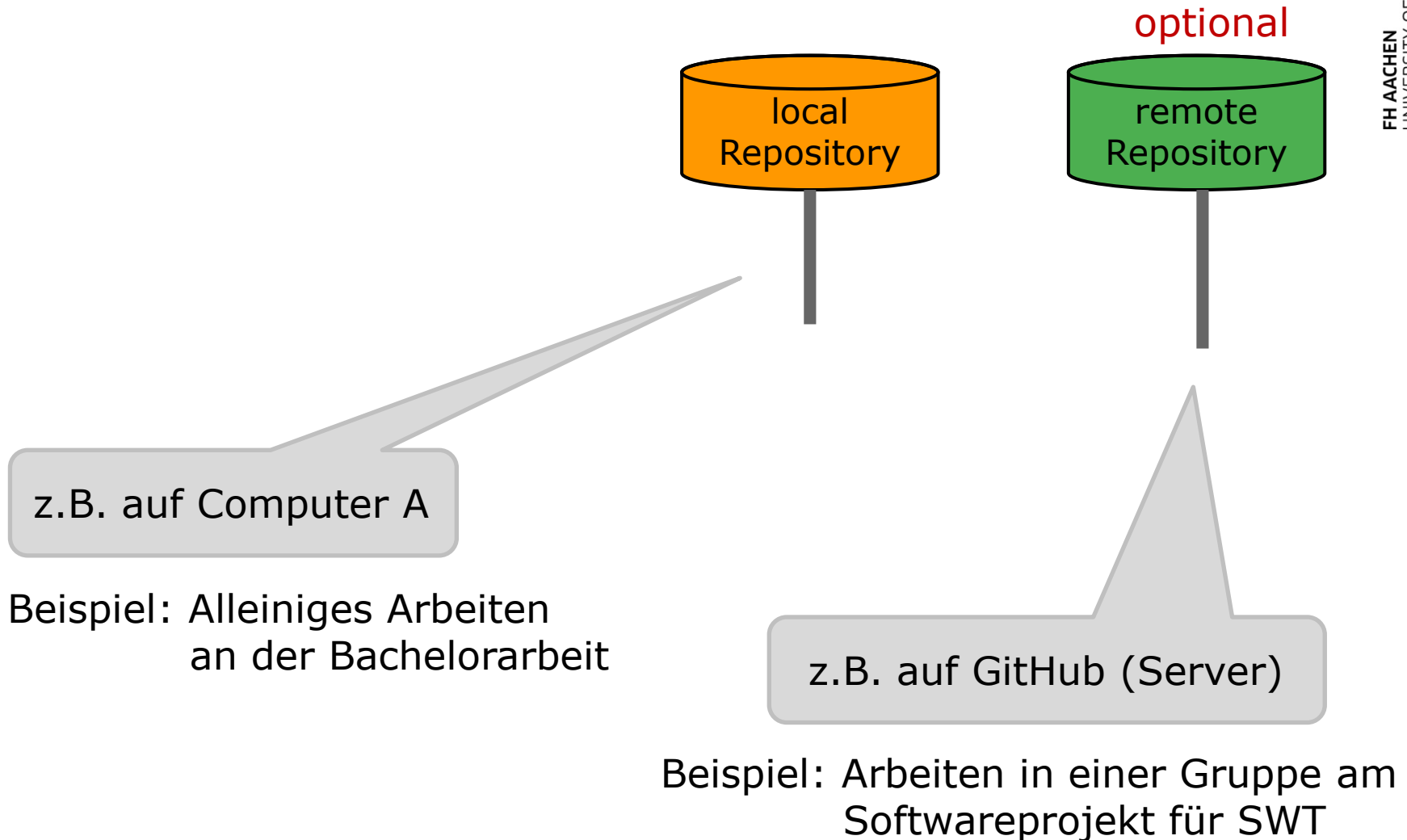
- single point of failure

### dezentral (Git)



# Lokales und entferntes Repository

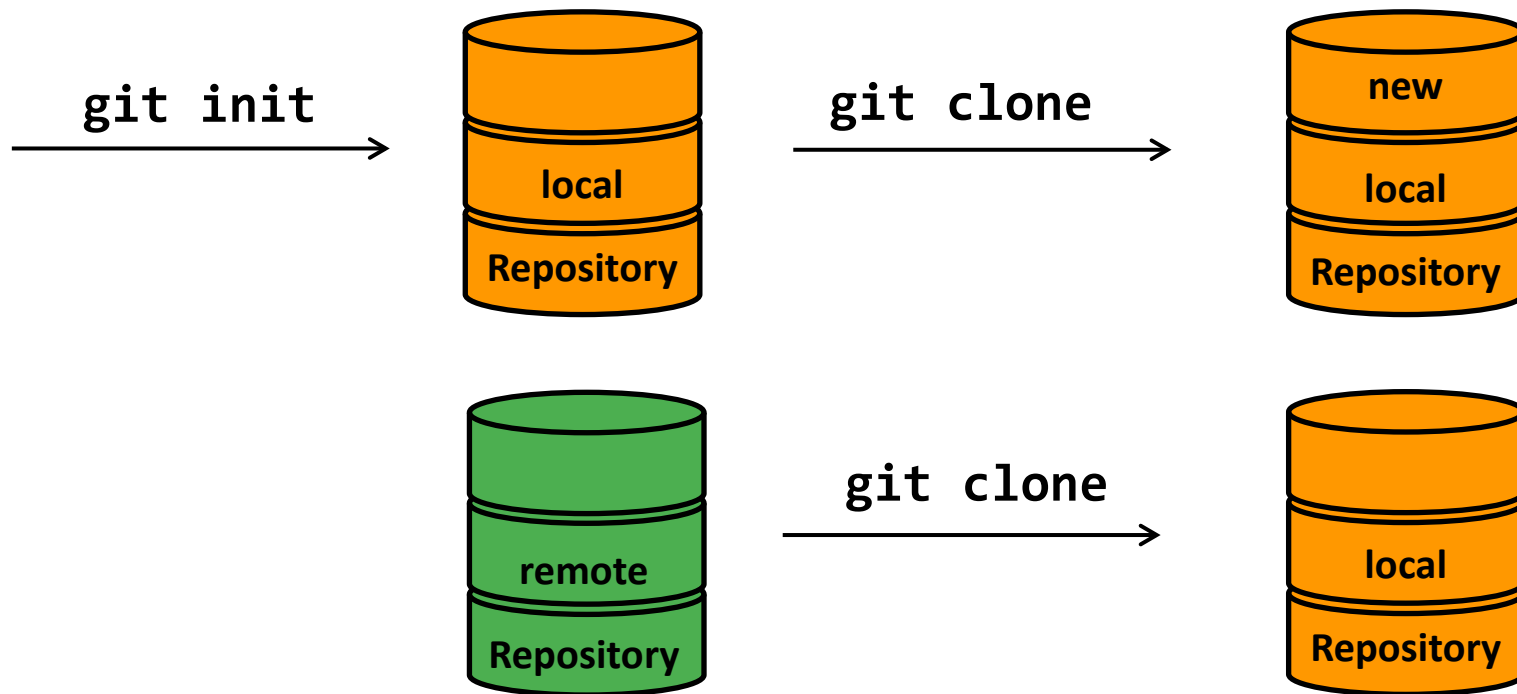
## Versionsverwaltung mit Git



# Erzeugen eines lokalen Git-Repositories

## Versionsverwaltung mit Git

- primär ist Git ein Kommandozeilen-Tool
- Erzeugen eines neuen lokalen Repositories: **git init**
- Erzeugen einer Kopie eines anderen (lokalen oder entfernten) Repositories: **git clone**

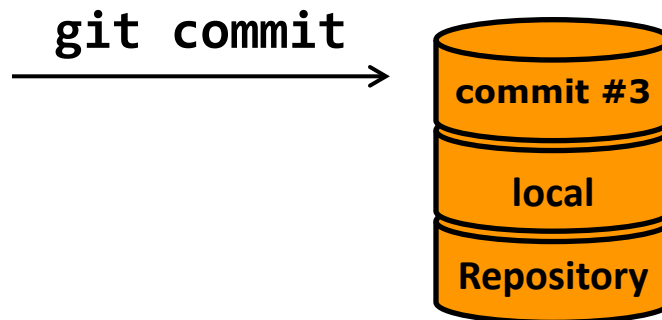




# Zustandsänderungen: Commit

## Versionsverwaltung mit Git

- die Idee von Git ist es, **Änderungen** in einem Projekt zu **tracken**
- eine Änderung von einem Zustand in einen anderen nennt man bei Git **Commit**
- es werden also „Stapel“ von Änderungen gebaut
- Beispiel:
  - hello.txt enthält den String „Hallo Horst“ (Zustand A). Die Datei ist bereits per Git versioniert
  - nun wird der String in „Hallo Welt“ geändert (Zustand B).
  - diese Änderung wird per **git commit** in das lokale Repository übertragen
  - beide Zustände können abgerufen werden



# Commit-Message

## Versionsverwaltung mit Git

- Neben der Änderung an sich wird auch eine Beschreibung dieser (sog. **Commit-Message**) in das Repository übertragen

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	1 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	5 HOURS AGO
○	ADKFJSLKDETSOKLFS	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

7 Regeln für eine gute Commit-Message  
<https://chris.beams.io/posts/git-commit/>

# „Stapel“ von semantischen Änderungen

## Versionsverwaltung mit Git

- **Ziel:** aussagekräftige Beschreibung von semantischen Code-Einheiten, sodass später Grund und Herkunft einer Änderung nachvollzogen werden können
  - Wer hat die Änderung gemacht, wann und warum?



Commits on Oct 5, 2018

Reenable ProcessTests for uap (#32648)



[ViktorHofer](#) authored and [danmosemsft](#) committed 2 days ago



79b2713



Skip some Http auth tests on UAP (#32646) ...



[davidsh](#) committed 2 days ago



e7e60c5



Treat the 'algorithm' parameter in Digest HTTP authentication as case... ...



[filipnavara](#) authored and [davidsh](#) committed 2 days ago



1d73791



Use natural language in Microsoft.CSharp error messages (#32567) ...



[sylveon](#) authored and [333fred](#) committed 2 days ago



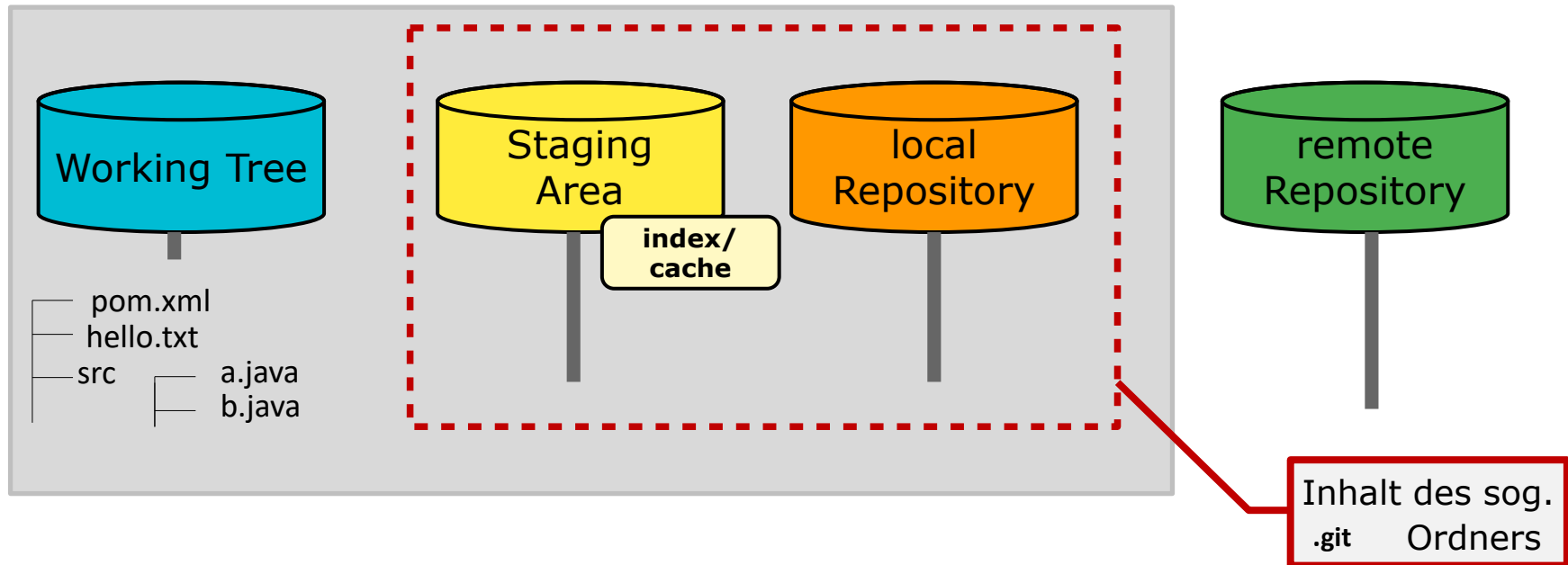
1cd890a



Wie kommt eine solche  
Änderung in mein lokales  
Repository?

# Hauptkomponenten von Git

## Versionsverwaltung mit Git

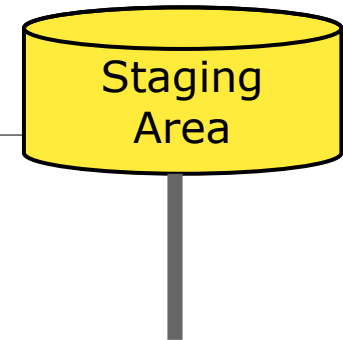


Die **lokale** Arbeitsumgebung besteht in der Regel aus drei Komponenten:

- **Working Tree:** Lokale Arbeitskopie (Dateien im Dateisystem), aus der alle Änderungen hervorgehen und in die Änderungen hineinfließen
- **Staging Area (index/cache):** Liste von potentiellen Kandidaten, die in das lokale Repository per **commit** übernommen werden sollen
- **local Repository:** Speicherung aller Versionen, die per **commit** übertragen wurden.

# Staging Area

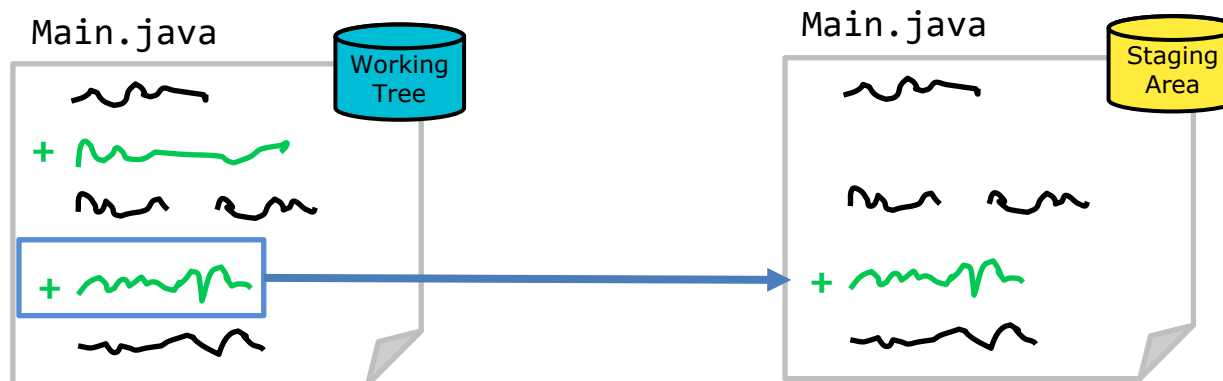
## Versionsverwaltung mit Git



## Wofür wird die Staging Area benötigt?

- **Logische Strukturierung der Commits**

- ermöglicht es, nur Teile einer oder mehrere Datei(en), die logisch zusammengehören in einem Commit zusammenzufassen



# Staging Area

## Versionsverwaltung mit Git

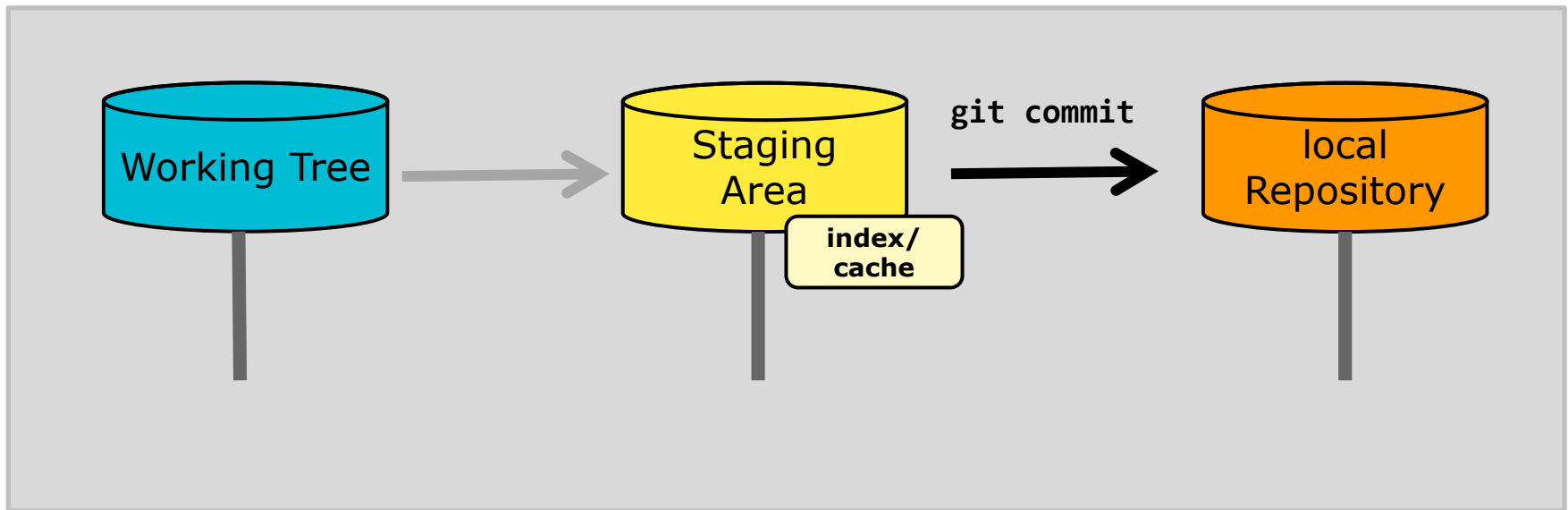


## Was bringt mir die Staging Area zusätzlich?

- die Staging Area als Zwischenschicht ermöglicht es, dass man dort eine Übersicht aller zu commitenden Dateien erhält
- Dateien, die versehentlich geadded wurden, können über **git reset HEAD -- <file>** wieder rausgenommen werden, bevor sie dann ins tatsächliche Repository wandern
- **„Sicherheitsfeature“**
  - es kann passieren, dass Dateien ins Repository wandern, die dort eigentlich nichts zu suchen haben
  - Beispiel: log-files, Config-Files von IDEs, Passwörter, TLS-Keys...

# Weg der Änderungen in Git

## Versionsverwaltung mit Git

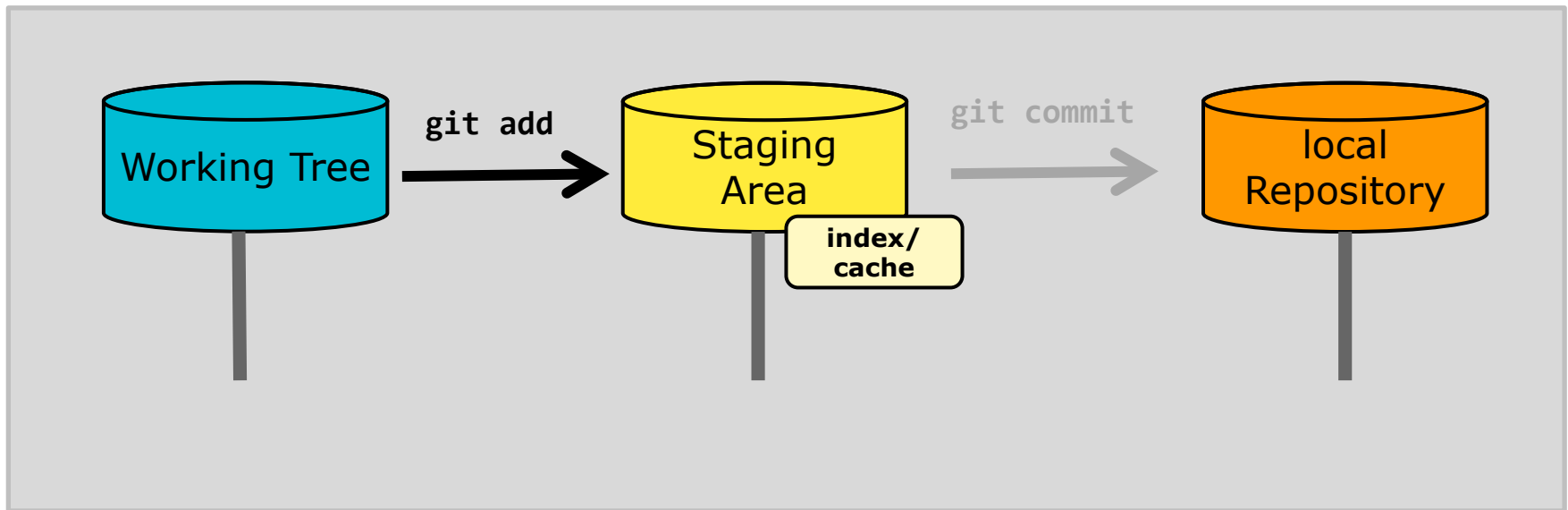




Wie kommt eine Änderung  
in die Staging Area?

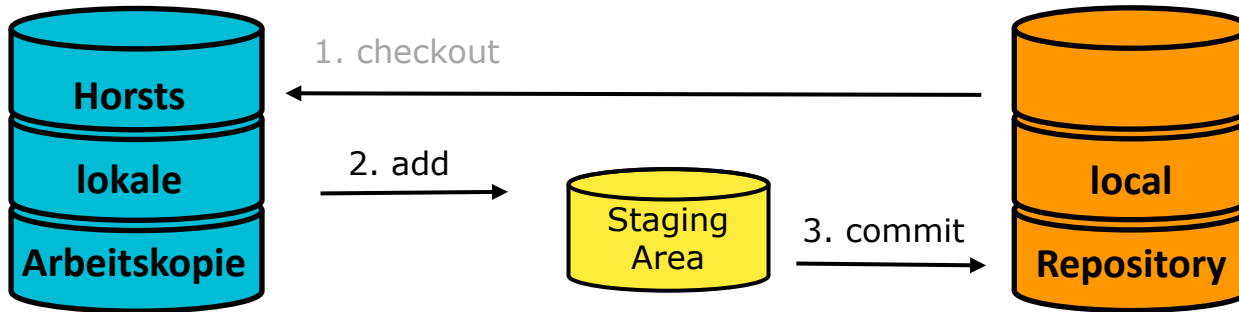
# Weg der Änderungen in Git

## Versionsverwaltung mit Git



# Grundlegende Arbeitsweise

## Versionsverwaltung mit Git



Versionierte  
Softwareobjekte

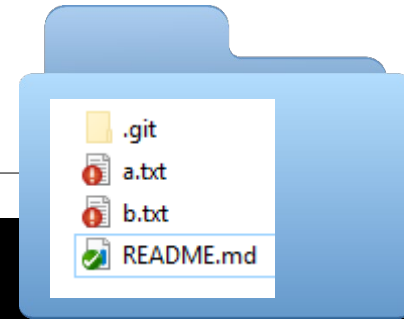


Programmierer Horst

1. lokale Arbeitskopie erstellen  
`git checkout <BRANCH>`
2. lokale Arbeitskopie verändern und  
Änderungen prüfen  
`git status`
3. Änderungen in Repository übertragen  
`git add .`  
`git commit -m „Kommentar“`

# Git status

## Versionsverwaltung mit Git



```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

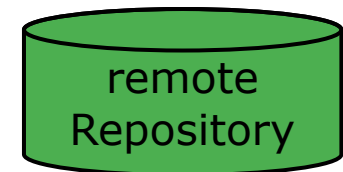
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   a.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   b.txt
```

Welche Datei liegt in welchem Bereich von Git?



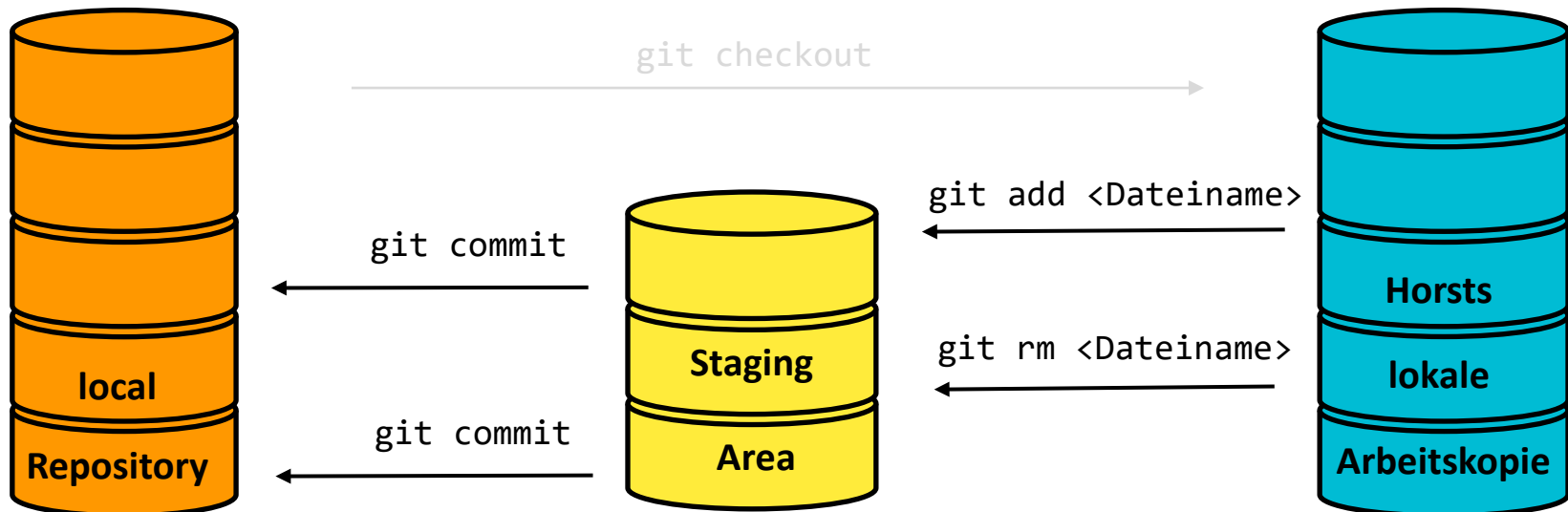
# Grundlegende Arbeitsweise; Löschen von Dateien

## Versionsverwaltung mit Git

- Dateien in der Arbeitskopie prüft Git auf Änderungen.
  - Änderungen müssen bestätigt werden
  - Neue Dateien und Verzeichnisse müssen explizit hinzugefügt werden.
- Löschen von Dateien mit dem Befehl `git rm`



**Programmierer  
Horst**



# Unterstützung des kollaborativen Arbeitens

## Versionsverwaltung mit Git

---



**Programmierer  
Johann**

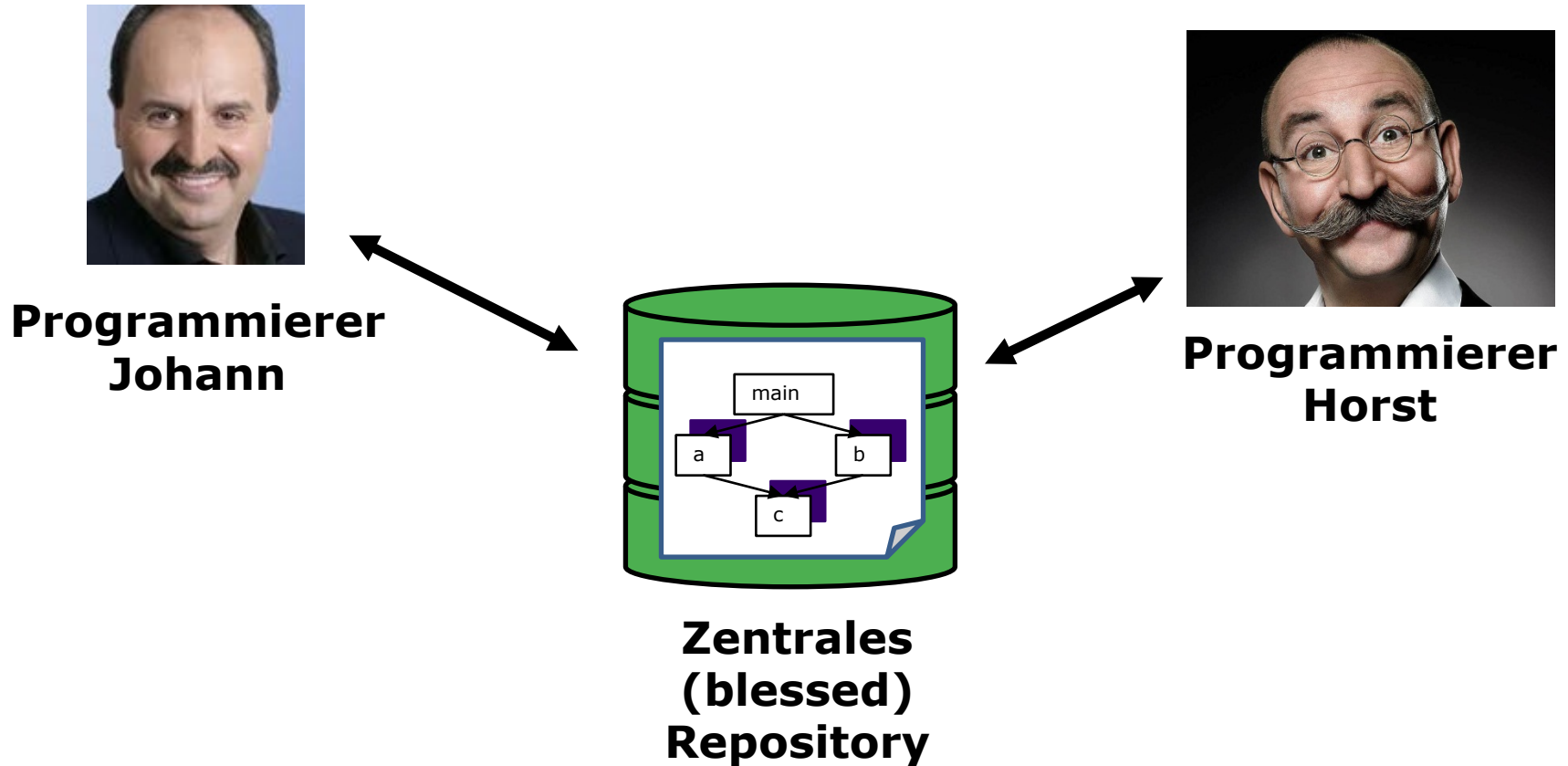


**Programmierer  
Horst**

Wie können Johann und Horst nun zusammen arbeiten?

# Unterstützung des kollaborativen Arbeitens

## Versionsverwaltung mit Git



# Unterstützung des kollaborativen Arbeitens

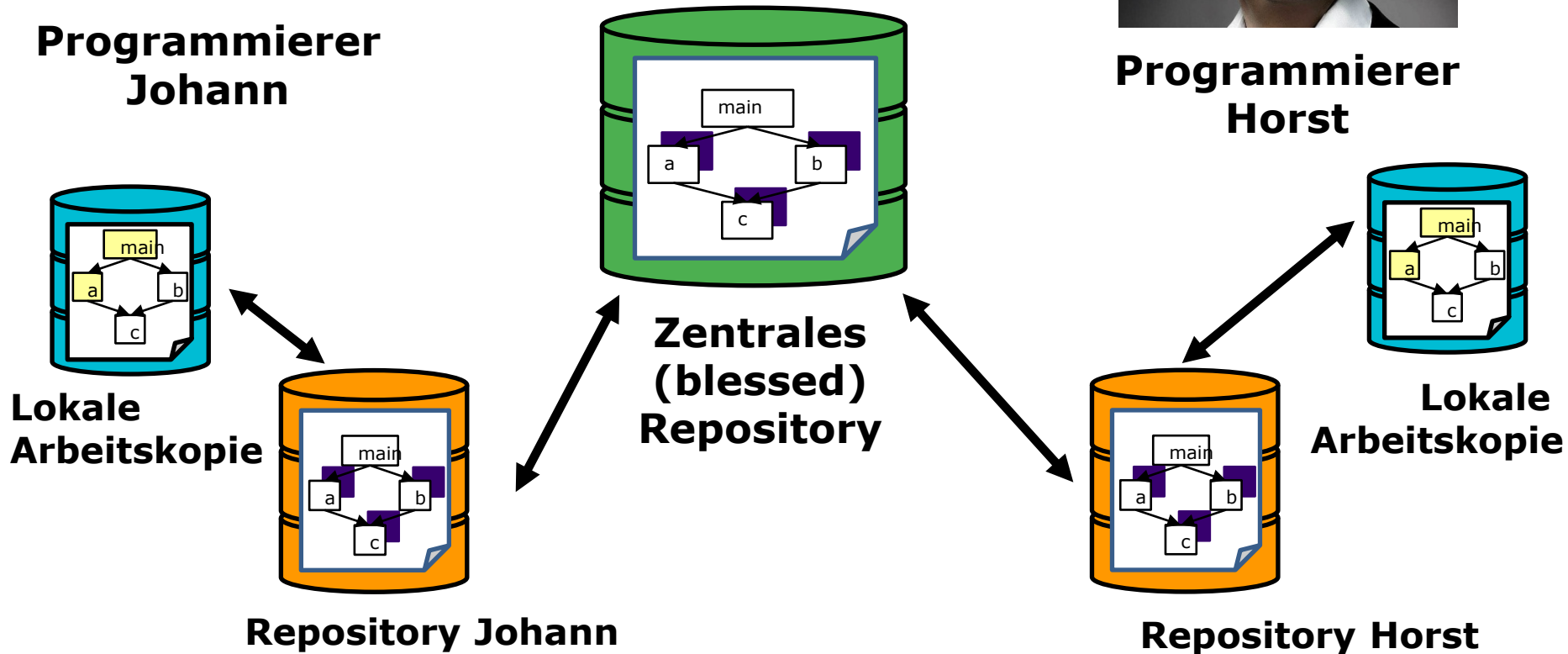
## Versionsverwaltung mit Git



**Programmierer  
Johann**



**Programmierer  
Horst**





# Unterstützung des kollaborativen Arbeitens

## Versionsverwaltung mit Git

---

1. Entferntes Repository verbinden und auschecken

```
git clone https://repository.zentral
```

2. lokale Arbeitskopie verändern

3. Änderungen prüfen

```
git status
```

4. Änderungen in lokales Repository übertragen

```
git add main.java
```

```
git commit -m „Kommentar“
```

5. Änderungen zurück ins entfernte Repository übertragen

```
git push
```

# Unterstützung des kollaborativen Arbeitens

## Versionsverwaltung mit Git

*Pause --- danach Fortsetzung der Arbeit*

6. Lokale Arbeitskopie aktualisieren

**git pull**

7. lokale Arbeitskopie verändern

8. Änderungen prüfen

**git status**

9. Änderungen in lokales Repository übertragen

**git add main.java**

**git commit -m „Kommentar“**

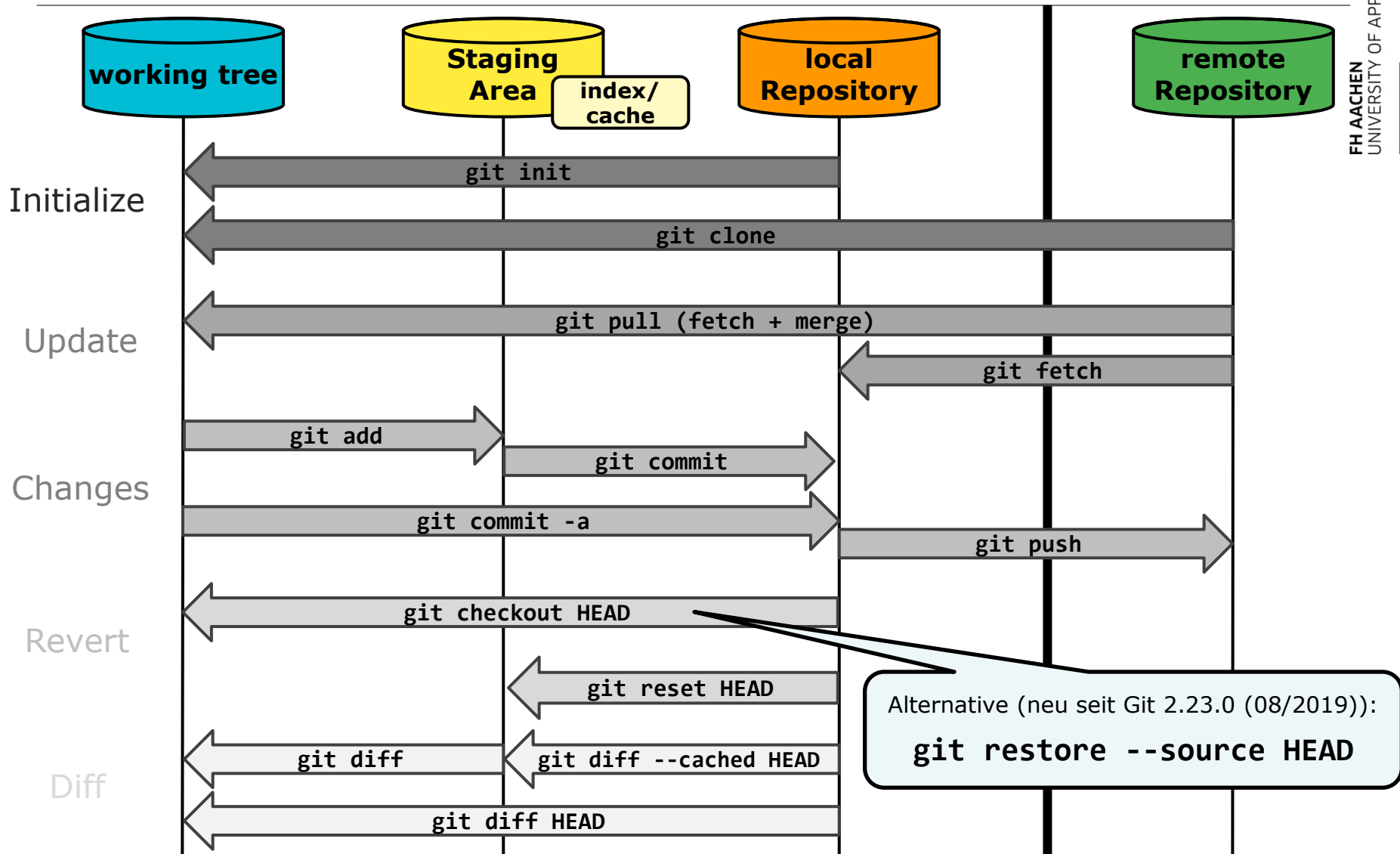
10. Änderungen zurück ins entfernte  
Repository übertragen

**git push**



# Unterstützung des kollaborativen Arbeitens

## Versionsverwaltung mit Git



# Konventionen

## Versionsverwaltung mit Git

- Lokale Kopie regelmäßig aktualisieren
- Änderungen regelmäßig einchecken
- aussagekräftige Commit-Messages schreiben
- Eingetackter Code **MUSS** übersetzbar sein
- keine abgeleiteten Dateien einchecken



Hello.java

Java Compiler

Hello.class



Javadoc Compiler

Hello.html



– .gitignore

– Nützliches Tool: [gitignore.io](https://gitignore.io)

# Konflikte

## Versionsverwaltung Git

Konflikte in Versionsverwaltungssystemen entstehen, wenn mehrere Programmierer die selben Zeilen parallel editieren

### Beispiel

```
public static void main(String[] args) {  
    System.out.println("Hallo Welt");  
}
```

**Horst möchte die Begrüßung** und **Johann den Namen** parametrisieren.

```
public static void main(String[] args) {  
    System.out.println(args[0] + " Welt");  
}
```



git push

```
public static void main(String[] args) {  
    System.out.println("Hallo " + args[0]);  
}
```



git push



**Out of date!**

# Auflösung von Konflikten

## Versionsverwaltung mit Git

- Johanns Version ist veraltet, dies erzeugt einen Konflikt
- Git meldet beim Aufruf von `git pull`: **Merge conflict** in `Hello.java`
- Konflikte werden in der Datei wie folgt markiert:

```
public static void main(String[] args) {  
    <<<<<<< HEAD  
        System.out.println("Hallo " + args[0]);  
    =====  
        System.out.println(args[0] + " Welt");  
    >>>>>> ac573e15f0ef0b2ccdcbf83a98be21ca414fdbea  
}
```



- Johann muss jetzt Konflikte lösen:

```
public static void main(String[] args) {  
    System.out.println(args[0] + " " + args[1]);  
}
```

- Anschließend Git mitteilen, dass die Konflikte gelöst sind

- `git add Datei.java`
- `git commit -m „Konflikt gelöst“`
- `git push`

# git pull vs git pull --rebase

## Versionsverwaltung mit Git

- Ausgangssituation: Horst hat lokal eine Änderung an Hello.java vorgenommen und diese committet sowie erfolgreich gepusht (commit E):

```
A---B---C---E   master, origin/master
```

- Johann hat die Änderung von Horst noch nicht aber auch lokal bei sich Hello.java geändert und committet (commit D):

```
      D   master  
      /  
A---B---C---E   origin/master
```

- Johanns git push wird nun fehlschlagen, weil das remote Repository (origin/master) Änderungen enthält, die er lokal noch nicht hat

# git pull vs git pull --rebase

## Versionsverwaltung mit Git

Nun gibt es zwei Möglichkeiten:

1. Johann macht ein `git pull`. Dies hat folgende Auswirkung auf die Historie:

```
      D
     / \
A---B---C---E---F  master, origin/master
```

es entsteht ein „unnötiger“ Merge-Commit F in der Historie 👎

2. Johann macht ein `git pull --rebase`:

```
A---B---C---E---D'  master, origin/master
```

rebase wendet die Änderung D auf den origin/master-branch an

→ **rebase stellt eine lineare Historie sicher** 👍



# Fortgeschrittene Funktionalität: Branching

## Versionsverwaltung mit Git

### Branching & Merging

- Abspaltung einer existierenden Version
- Parallele Entwicklung einer Software in mehreren Varianten/Versionen
  - Entwicklungszweige für Releasezustände
  - Entwicklung von Features
- Zusammenführung von Branches möglichst einfach

**Beispiel:** Firefox nutzt Branching intensiv



[ Quelle: [Mozilla-Blog](#) ]

### Branching und Merging sind Kernfeatures von Git

→ Gute Hilfsmittel wie GitLab zur Visualisierung

# Branches

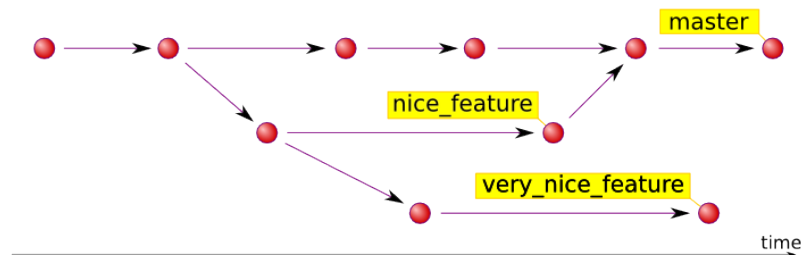
## Versionsverwaltung mit Git

### Standard für die Organisation des Projektarchivs



### Branches sind Seitenzweige

- Features im Alpha/Betastadium
- instabile oder Testversionen



Branch  
Commit

- ein Commit kennt seinen Vorgänger
- ein Branch ist nur ein Zeiger auf einen Commit-Hash

# Commit-Hash

## Versionsverwaltung mit Git

- Commits sind eindeutig über Hashes identifizierbar

SHA-1 - 40-character string: **79b2713**ee31e44d2a9c99a0c5ef33697d2f5b316



Commits on Oct 5, 2018

Reenable ProcessTests for uap (#32648)



[ViktorHofer](#) authored and danmosemsft committed 2 days ago



79b2713



Skip some Http auth tests on UAP (#32646) ...



davidsh committed 2 days ago



e7e60c5



Treat the 'algorithm' parameter in Digest HTTP authentication as case... ...



filipnavara authored and davidsh committed 2 days ago



1d73791



Use natural language in Microsoft.CSharp error messages (#32567) ...



sylveon authored and 333fred committed 2 days ago



1cd890a



# Branching

## Versionsverwaltung mit Git

0. Neuen Branch erzeugen  
`git branch nice_feature`
1. Auf neuen Branch wechseln  
`git checkout nice_feature`
2. lokale Arbeitskopie verändern und Änderungen prüfen  
`git status`
3. Änderungen ins Repository übertragen  
`git add .; git commit -m „Kommentar“`
4. Branch ins entfernte Repository zurückschreiben  
`git push origin nice_feature`

Alternative: **git switch**  
(neu seit Git 2.23.0 (08/2019))

# Merging

## Versionsverwaltung mit Git

5. Auf den Ziel-Branch wechseln

```
git checkout master
```

6. Ziel-Branch aktualisieren

```
git pull
```

7. Quell-Branch in den Ziel-Branch mergen

```
git merge nice_feature
```

Tritt **kein** Konflikt auf,  
wird automatisch ein  
Merge-Commit angelegt.

8. Bei Konflikten: Konflikte beheben und als gelöst  
markieren

```
git add .
```

```
git commit -m „Merge mit nice_feature“
```

9. Ziel-Branch ins entfernte Repository schreiben

```
git push
```

# Fortgeschrittene Funktionalität: Tagging

## Versionsverwaltung mit Git

### Markierung eines Standes im Repository

- Zustand der Software zu bestimmten Zeitpunkt festhalten
  - Auslieferung beim Kunden
  - Demonstration
  - Übergabe an Testabteilung

### Beispiele

Version 1.0

REL\_01\_2013

1. Existierende *Tags* anzeigen

```
git tag
```

2. Versionsstand mit einem *Tag* kennzeichnen

```
git tag -a v1.4 -m „my version“
```

3. Tag ins entfernte Repository übertragen

```
git push origin v1.4
```

4. Details zu einem *Tag* anzeigen

```
git show v1.4
```

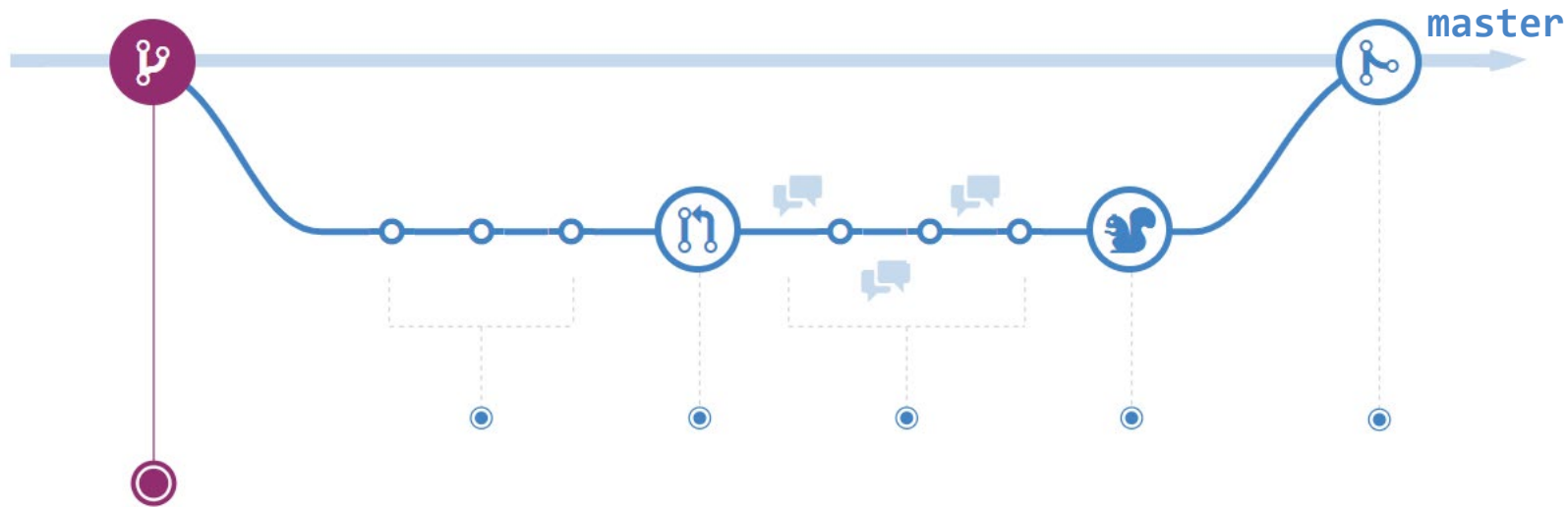
5. Version mit *Tag* anschauen

```
git checkout v1.4
```

- **Definition**
  - Rezept bzw. Empfehlung, wie man produktiv (mit mehreren) an einem Projekt entwickelt
- Es gibt verschiedene Workflows, die ihre Vor- und Nachteile haben
  - Centralized workflow
  - **Feature Branch Workflow**
  - Git-Flow-Workflow
  - Forking-Workflow

# Git Feature Branch Workflow

## Versionsverwaltung mit Git



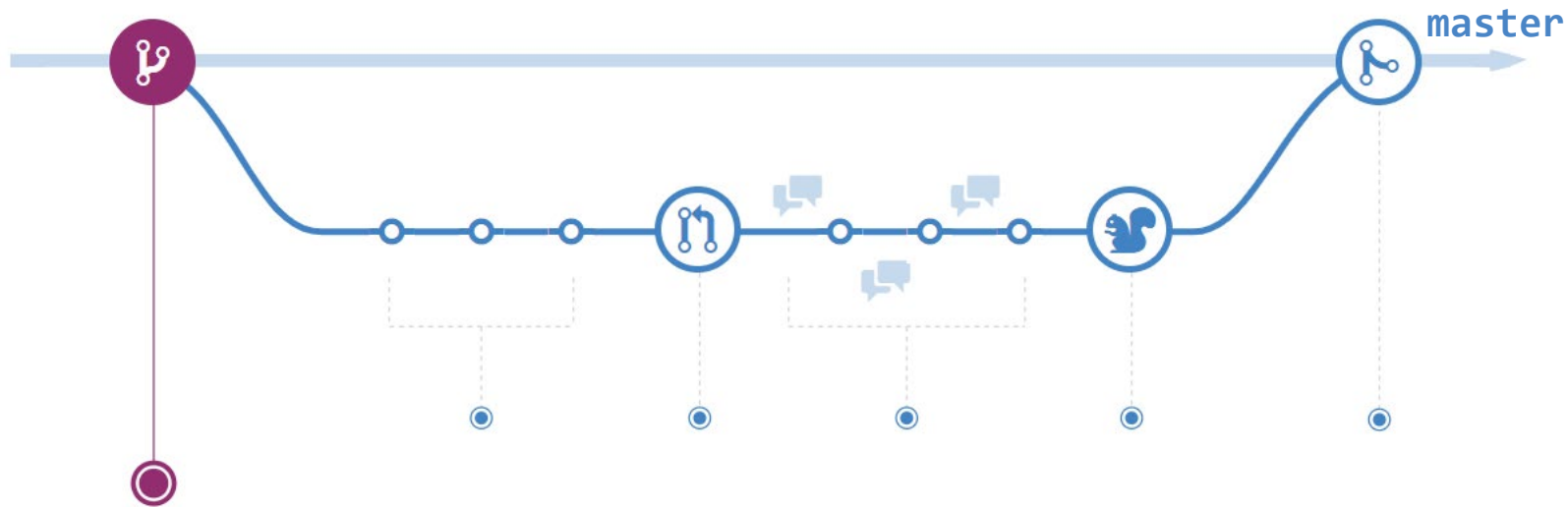
Quelle:  
<https://guides.github.com/introduction/flow/>

- Dieser Workflow basiert stark auf dem Kernkonzept des Branchings:
  - Für jedes/n Feature/Fix wird ein neuer Branch angelegt
  - Der Name des Branches sollte selbsterklärend sein; z.B. refactor-authentication
  - Durch Review-Prozess kann ein fertig entwickelter Branch in den **master** Branch gemerged werden
  - Hauptregel: **Der master Branch bleibt immer kompilierbar!**



# Git Feature Branch Workflow

## Versionsverwaltung mit Git

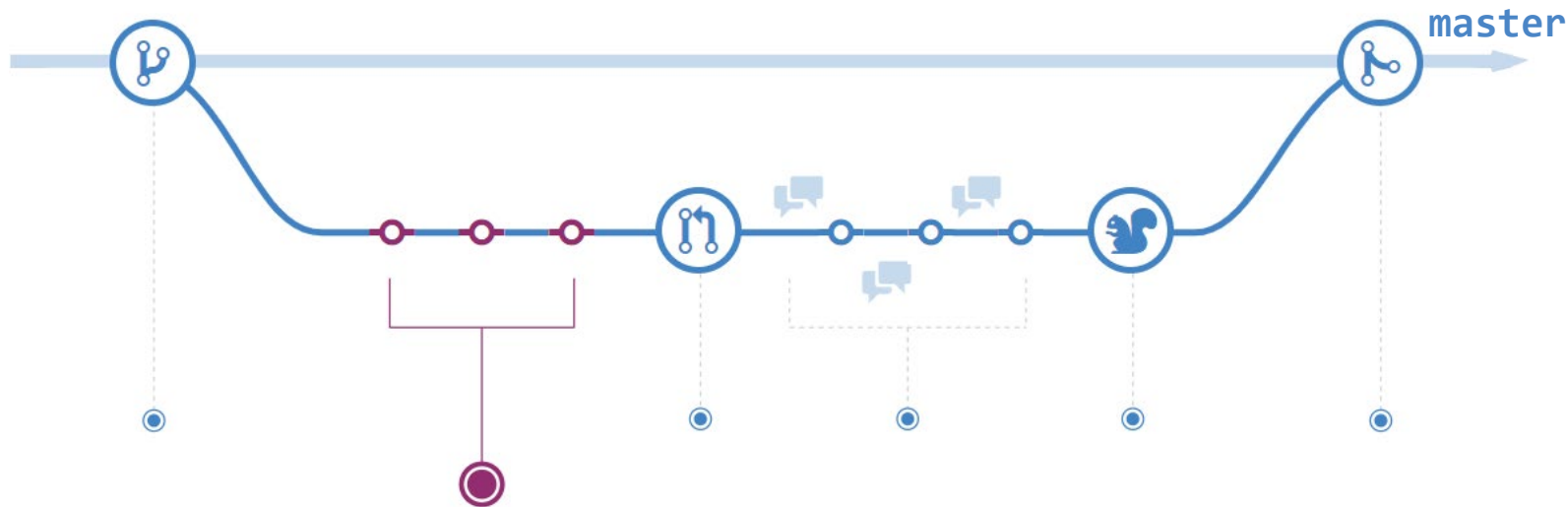


Quelle:  
<https://guides.github.com/introduction/flow/>

- Praxis:
  - **Neuen Branch anlegen und zu diesem wechseln:**  
`git checkout/switch master`  
`git branch new-feature`  
`git checkout/switch new-feature`

# Git Feature Branch Workflow

## Versionsverwaltung mit Git



Quelle:  
<https://guides.github.com/introduction/flow/>

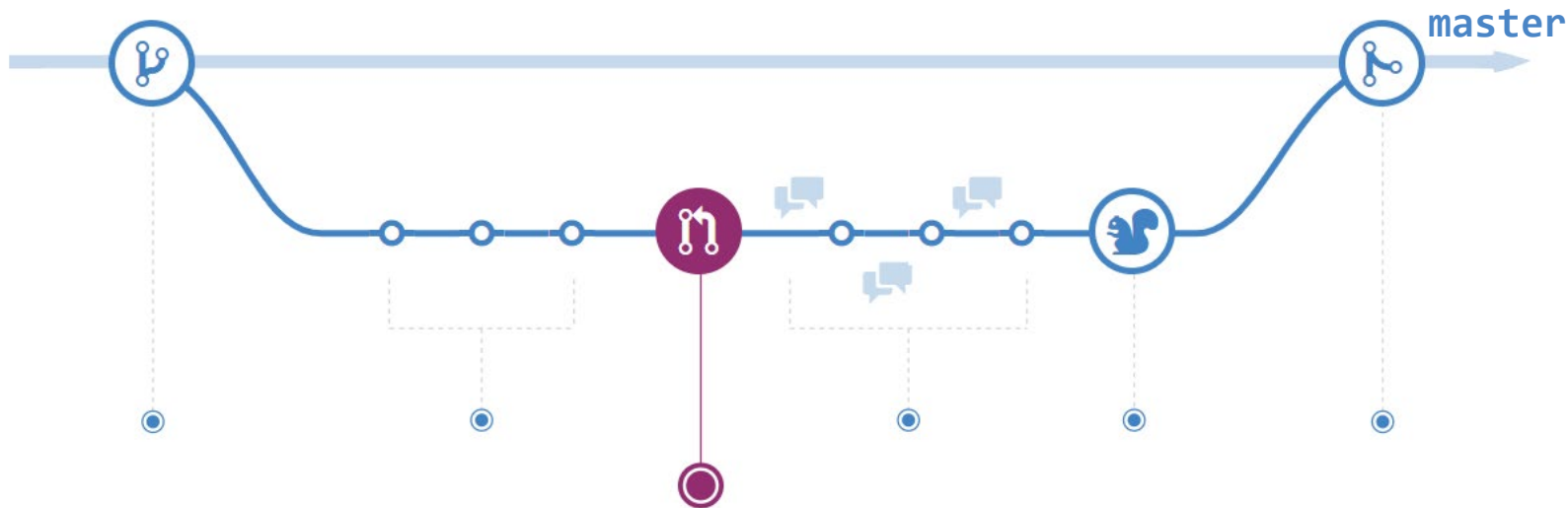
- Arbeite ganz normal auf dem Feature-Branch mit `add` und `commit`
- Achte dabei auf aussagekräftige Commit-Messages, damit andere Team-Mitglieder verstehen, was du in deinem Branch gemacht hast
- Praxis:

```
git add <datei>  
git commit -m "new feature..."  
git push origin new-feature
```

`git push -u origin new-feature`  
beim ersten Mal, damit zukünftig nur  
`git push` eingeben muss

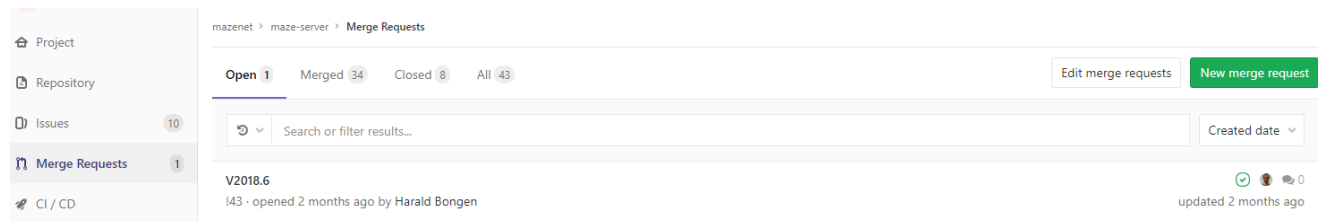
# Git Feature Branch Workflow

## Versionsverwaltung mit Git



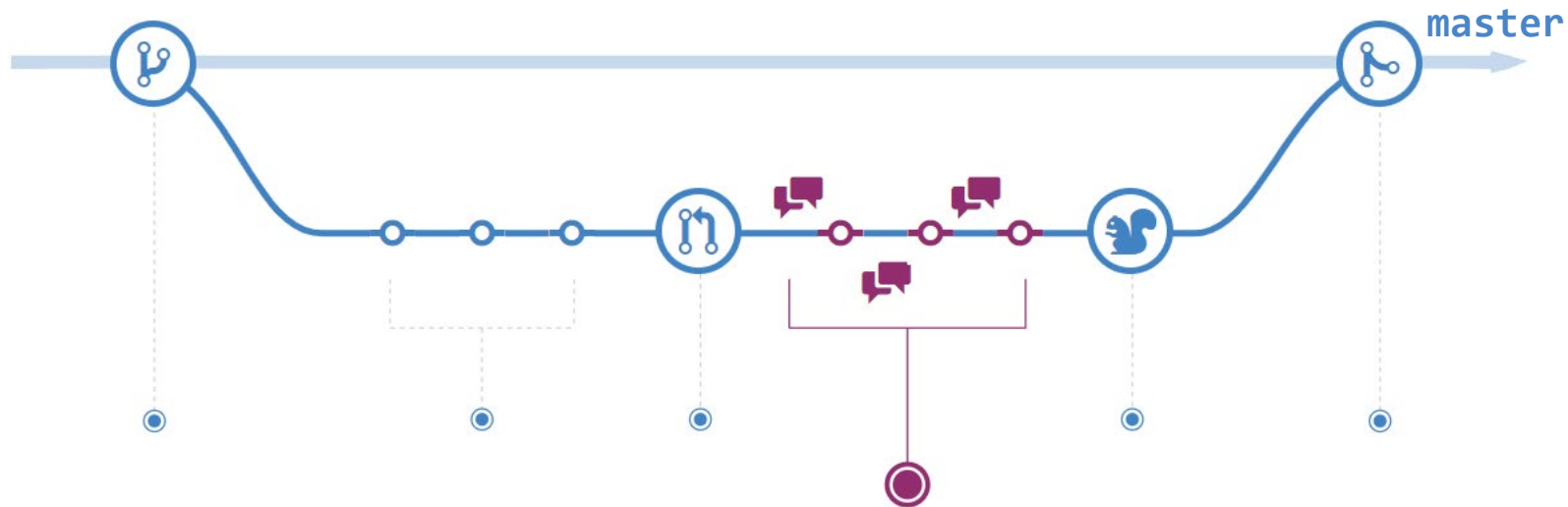
Quelle:  
<https://guides.github.com/introduction/flow/>

- Code-Review
  - Hole Meinungen von anderen Entwicklern ein (z.B. wenn du Hilfe benötigst oder der Code von einem anderen Entwickler begutachtet werden soll)
  - Erstelle dafür mit einer GUI sog. Pull Requests (z.B. im GitLab mit Merge Requests)



# Git Feature Branch Workflow

## Versionsverwaltung mit Git




Quelle:  
<https://guides.github.com/introduction/flow/>

- Diskutiere deine Arbeit im branch mit anderen Team-Mitgliedern


# Git Feature Branch Workflow

## Versionsverwaltung mit Git


 **stephentoub** reviewed 23 hours ago [View changes](#)

```
src/System.IO.FileSystem/tests/File/Delete.cs
```

...	...	@@ -70,6 +70,7 @@	public void PositiveTest()
70	70		[Fact]
71	71		public void NonExistentFile()
72	72		{
	73	+	Delete(Path.Combine(Path.GetPathRoot(TestDirectory), Path.GetRandomF

**stephentoub** 23 hours ago Member

Why GetPathRoot?

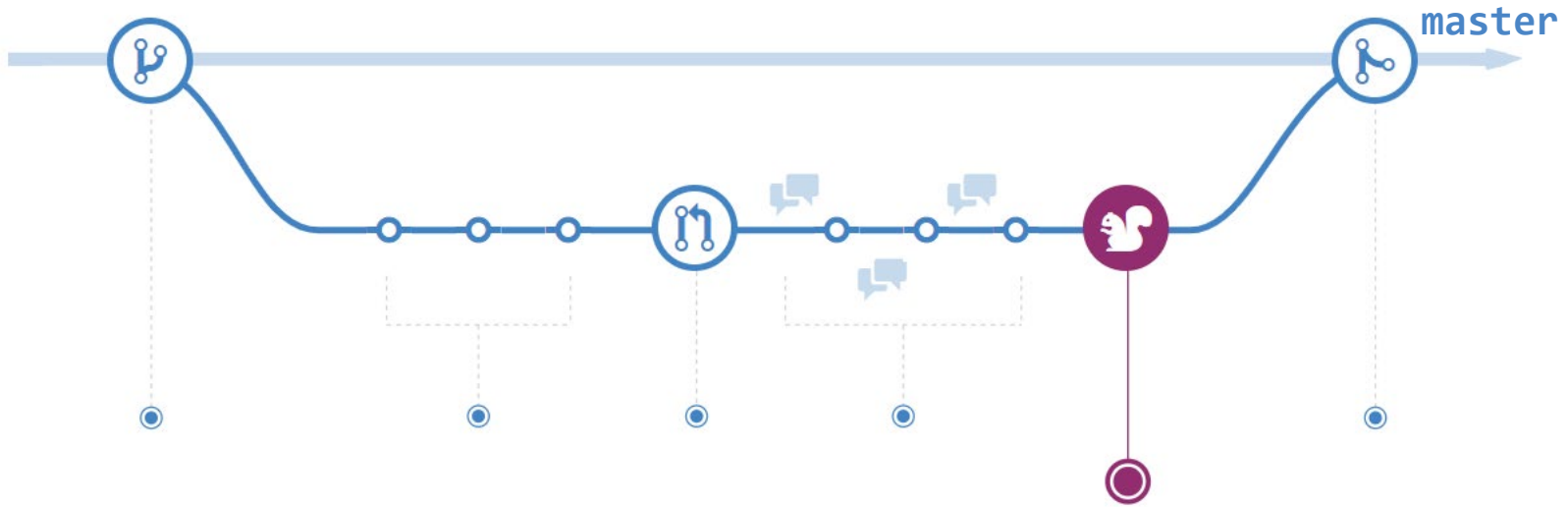
**MarcoRossignoli** 23 hours ago Collaborator

My idea is to add test similar to <https://github.com/dotnet/corefx/pull/32660/files#diff-0fe3419b22b4609b424cd8730bf45893R108>  
Case with "no directory"...do you think it's unuseful?

Quelle: <https://github.com/dotnet/corefx/pull/32660>

# Git Feature Branch Workflow

## Versionsverwaltung mit Git



Quelle:  
<https://guides.github.com/introduction/flow/>

**FH AACHEN**  
UNIVERSITY

- Integrationstests
  - finalisiere deine Änderungen
  - stelle sicher, dass dein Feature-Branch lauffähig ist

**FH AACHEN**  
UNIVERSITY OF APPLIED SCIENCES



- Fachhochschule Aachen | Prof. Dr. Bodo Kraft | Softwaretechnik | Einstieg Versionsverwaltung

# Git Feature Branch Workflow

## Versionsverwaltung mit Git

- Praxis Merging:
  - Damit der Maintainer des **master** Branches so wenig Arbeit wie möglich hat, synchronisieren wir unseren aktuellen Branch erstmal mit dem **master** Branch, in dem wir den **master** Branch in unseren Feature-Branch mergen:

```
git switch master  
git pull  
git switch feature-branch  
git merge master  
git push
```

evtl. Konflikte lösen!

- Maintainer merged nun den Feature-Branch in den **master** Branch:  
Voraussetzung: er hat neueste Version vom master + feature-branch! (git pull)

```
git switch master  
git merge feature-branch  
git push
```



# Weitere Informationen

## Versionsverwaltung mit Git

---

### Git-Clients

- Kommandozeile in der Shell
- In Windows integriert: TortoiseGit
- In Entwicklungsumgebung integriert (Eclipse, IntelliJ, ...)
- u.a. ...

### Informationen im Internet

- Kurz: <http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf>
- Mittel: <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>
- Lang: <http://www.vogella.com/tutorials/Git/article.html>
- Git Game: <https://learngitbranching.js.org/>

# Git

## Versionsverwaltung mit Git



Quelle: <http://xkcd.com/1597/>



;-)