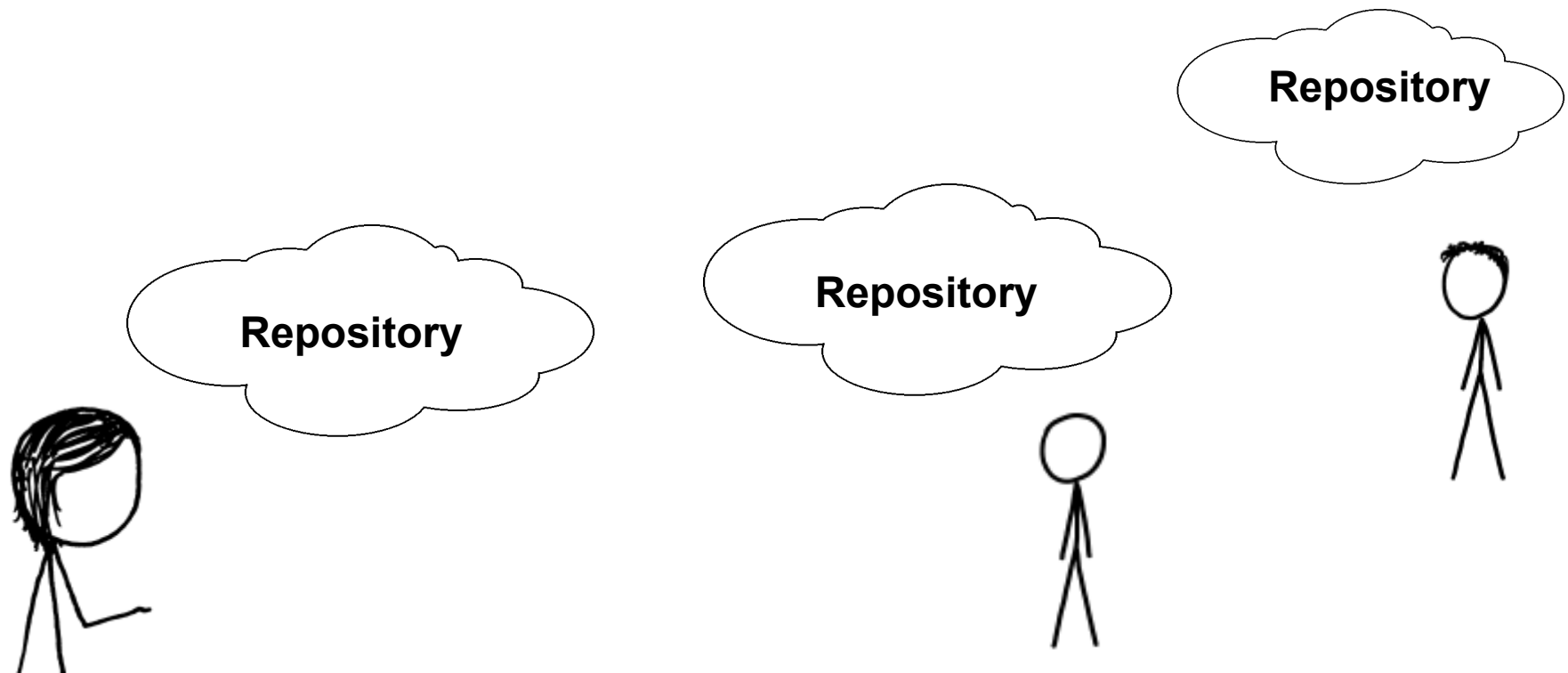


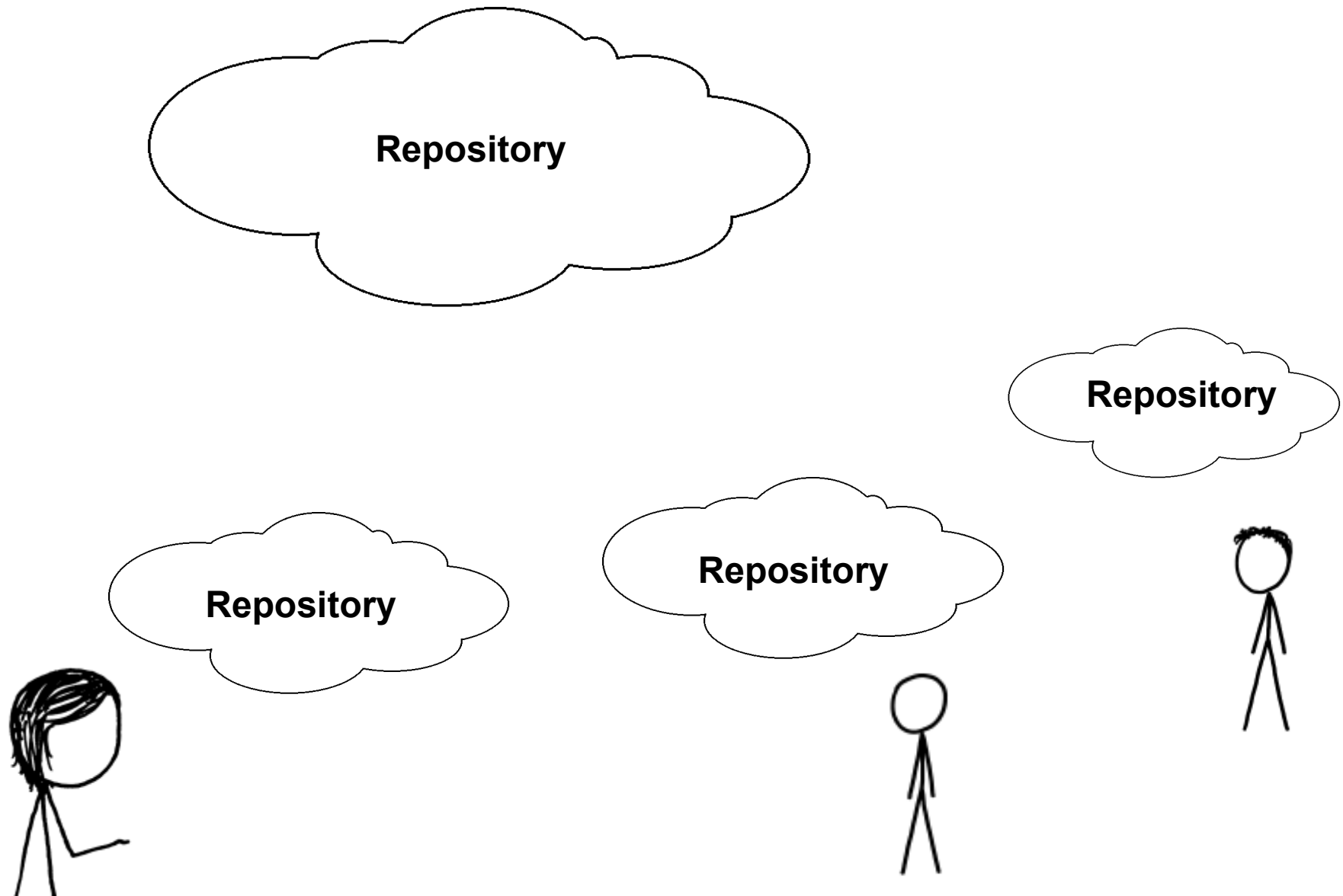
Git und du

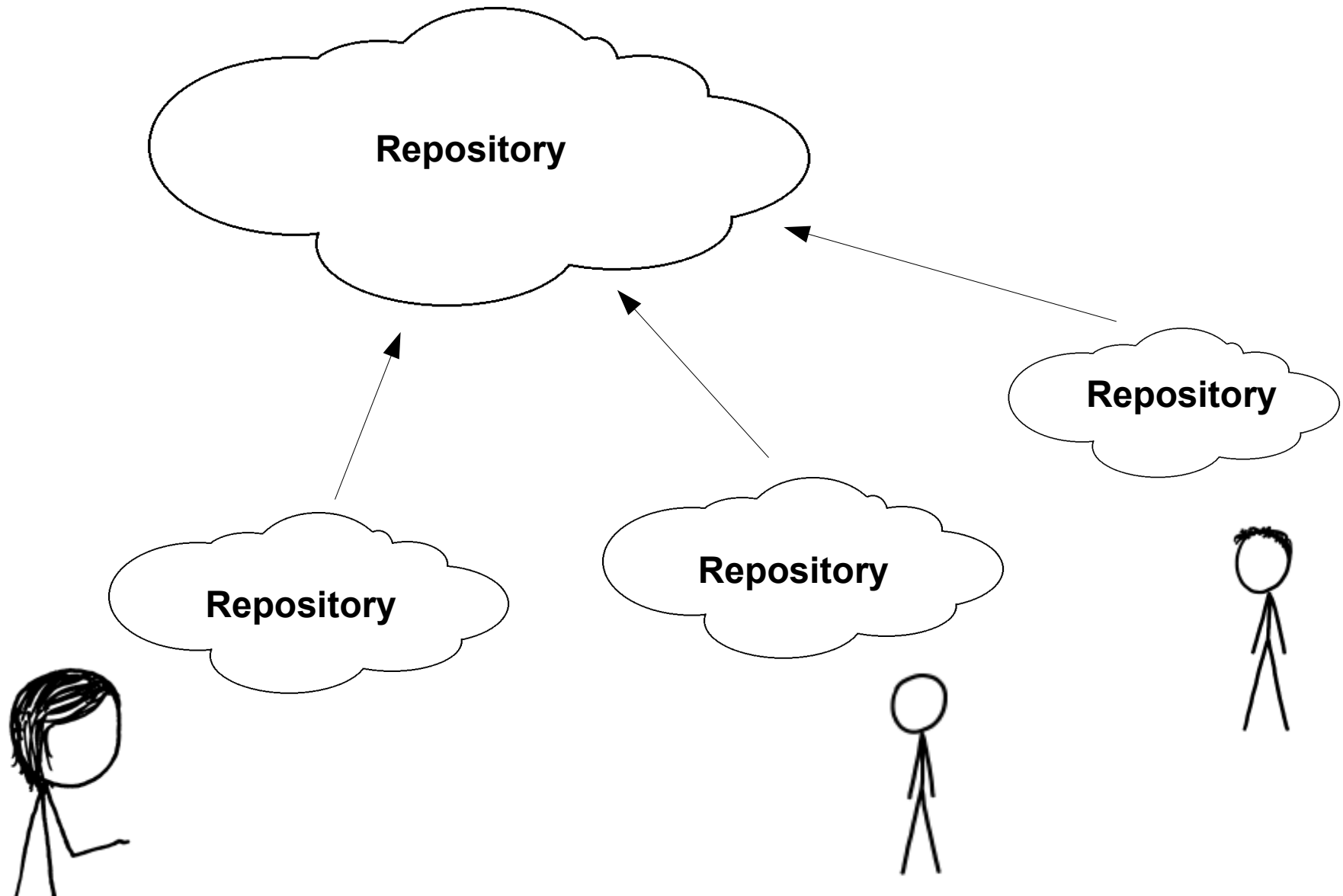
Eine Einführung in Git und Github

Zusammenarbeit?

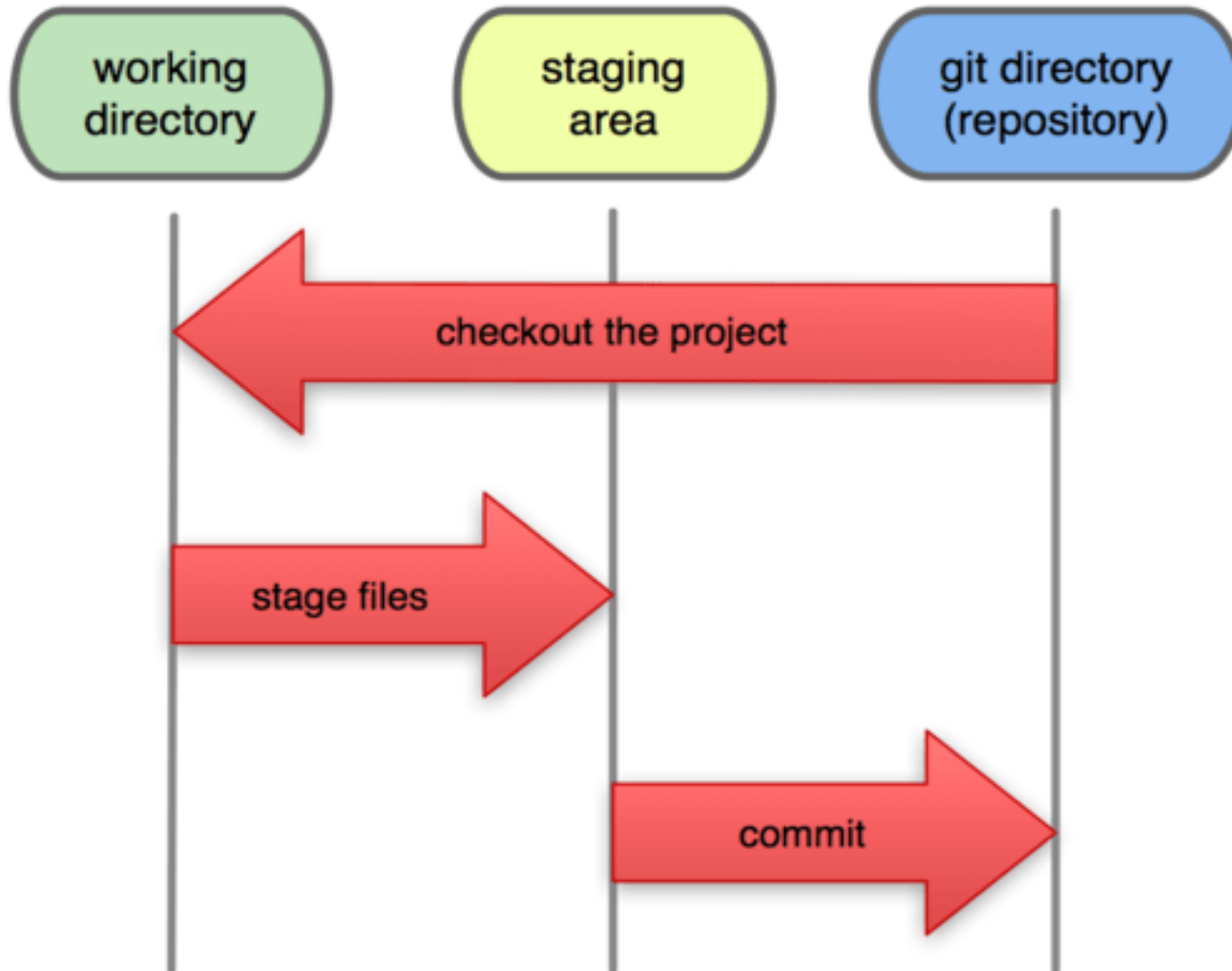








Local Operations



How git works

working directory

- Dateien auf der Festplatte.

Enthält *"modified" files*

staging area (= index)

- Datei im git-Verzeichnis
- Sagt: Was fließt in den nächsten Commit?

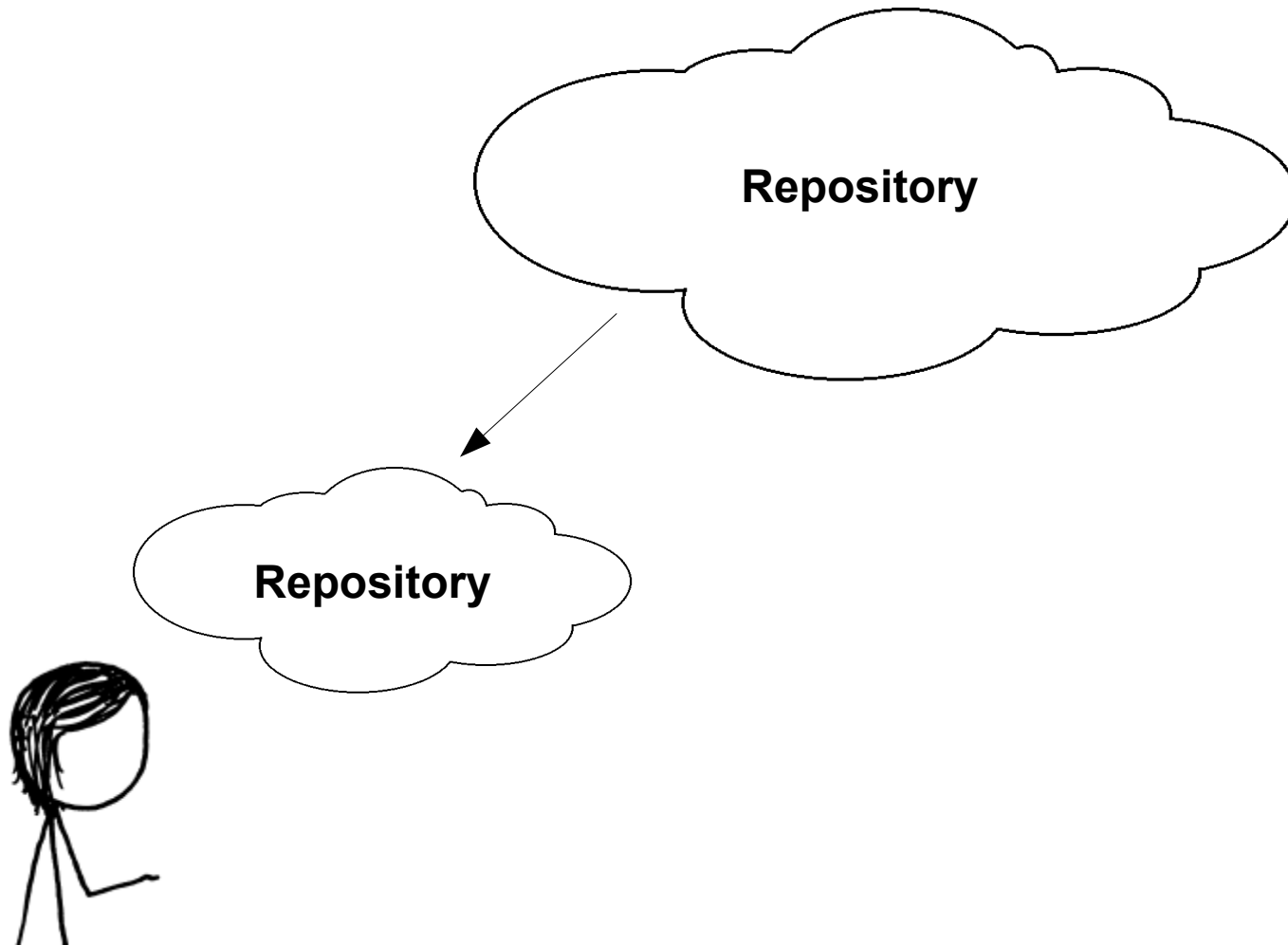
Enthält *"staged" files*

git repository

- im Repo gesicherte Dateien

Enthält *"committed" files*

Dateien holen



Dateien holen

Lokale Kopie eines github repository erzeugen:

```
git clone <https://github.com/Jane333/test1.git>
```

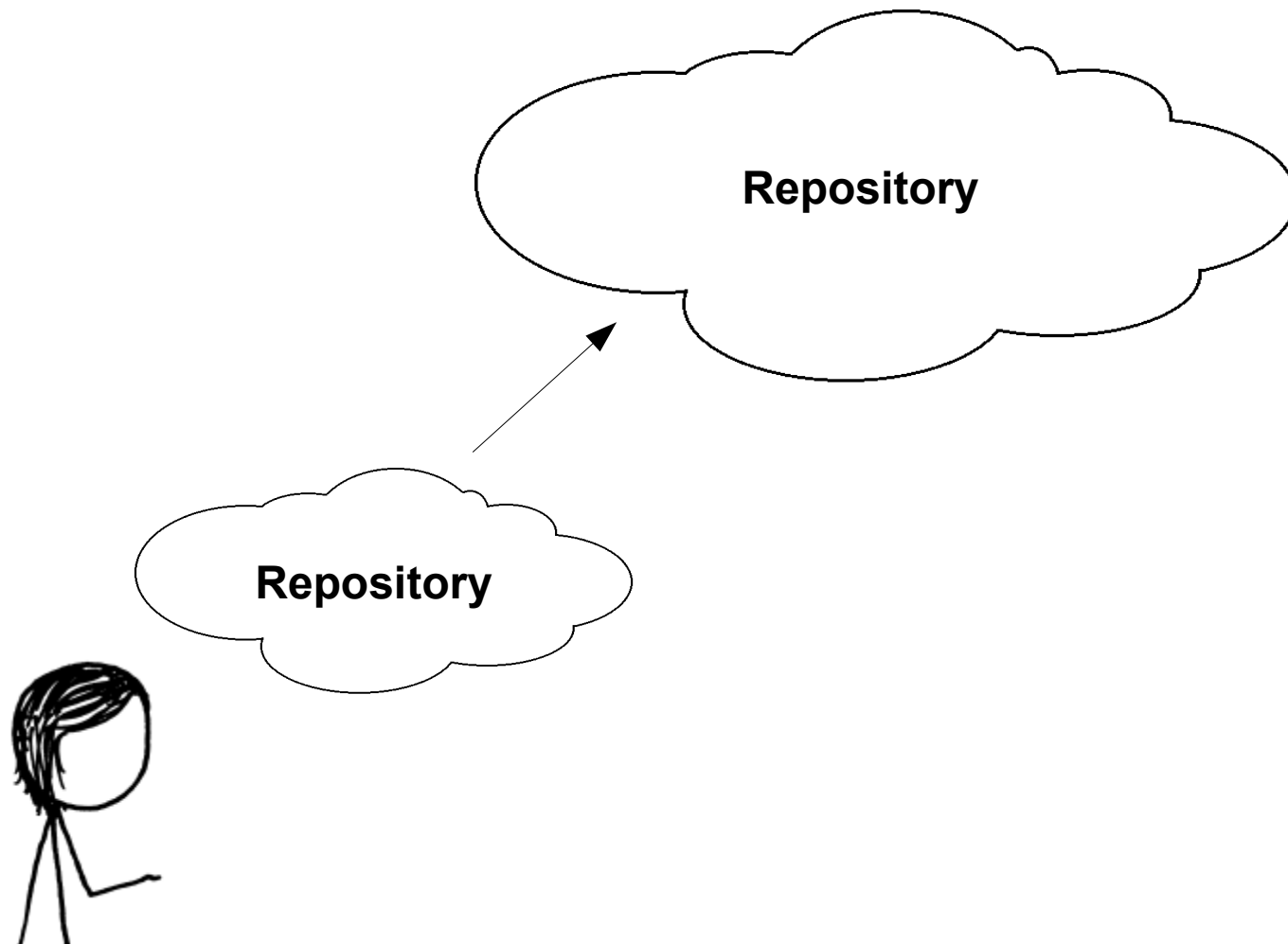
Oder neues, leeres repository anlegen:

```
git init
```

Alle Dateien / Änderungen aus dem remote repository runterladen:

```
git pull
```

Dateien hochladen



Dateien hochladen

Neue Datei zum git repository hinzufügen (in die “staging area”):

```
git add <file1.txt>
```

Datei entfernen:

```
git rm <file1.txt>
```

Datei umbenennen / verschieben:

```
git mv <file1.txt>
```

Dateien hochladen

Alle Veränderungen der “getrackten” Dateien in die “staging area” hinzufügen UND sofort commiten:

```
git commit -am "I just added some files..."
```

Commit nach github hochladen. Tut das Gleiche wie git push origin master:

```
git push
```

Alternativ: ALLE Dateien (rekursiv) in die staging area hinzufügen:

```
git add .
```

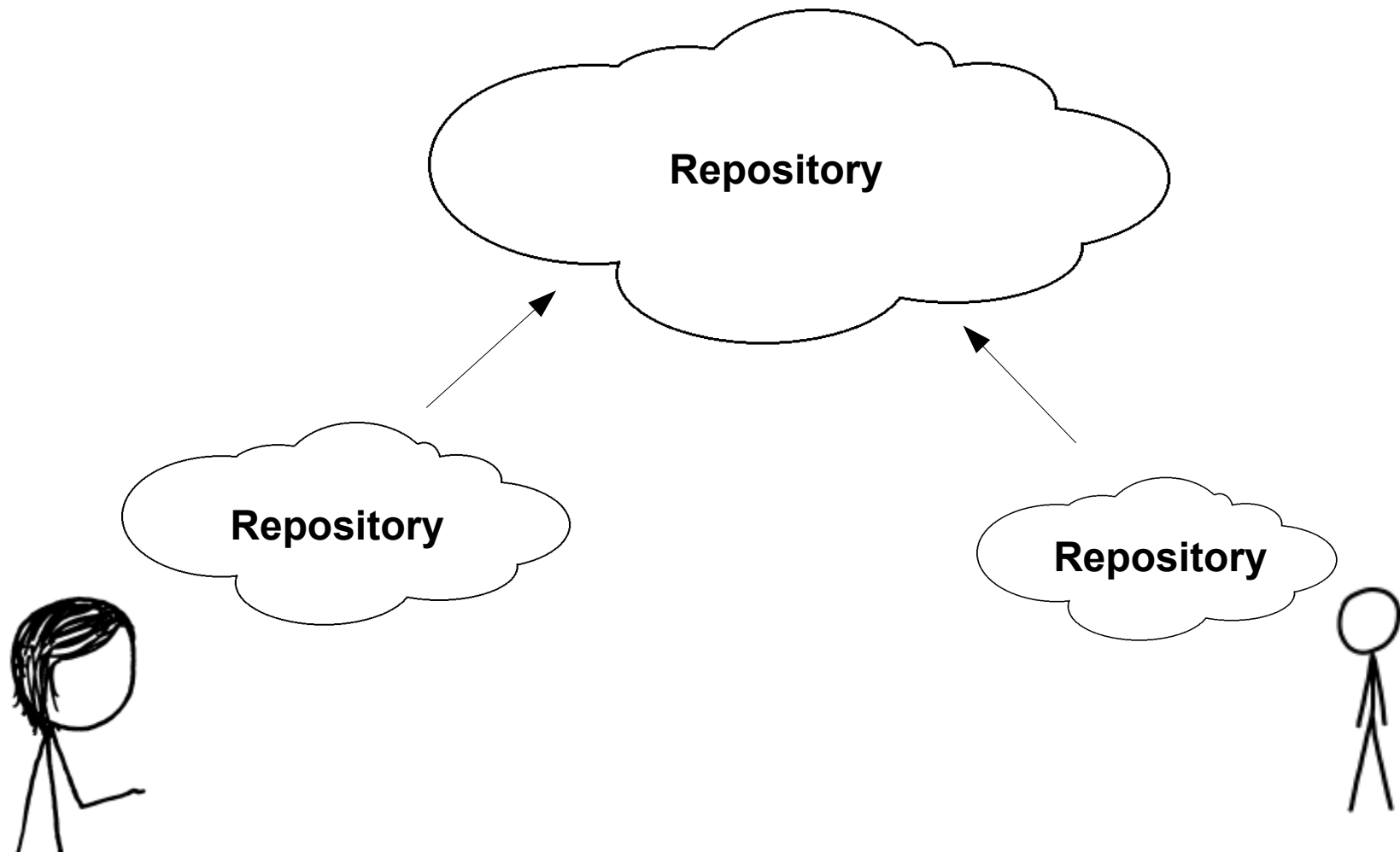
Staged files commiten:

```
git commit -m "I just added some files..."
```

Commit hochladen:

```
git push
```

Konflikte auflösen



Konflikte auflösen

Szenario:

Person A erzeugt commits und pusht sie.

Person B macht aber keinen pull, sondern erzeugt eigene commits (lokal) mit Änderungen in den gleichen Zeilen einer Datei wie Person A, und versucht dann, sie zu pushen.

Person B bekommt nun den error:

"Updates were rejected because the remote contains work that you do not have locally. You may want to first integrate the remote changes (e.g., 'git pull ...') before pushing again."

Also muss Person B einen pull machen... Bei dem pull kommt jedoch dieser Fehler: "CONFLICT (content): Merge conflict in Automatic merge failed; fix conflicts and then commit the result."

Konflikte auflösen

Nun muss man die Konflikte per Hand auflösen. Das geht mit einem mergetool, z.B. kdiff3:

git mergetool --tool=kdiff3

Gewünschte Änderungen manuell auswählen, speichern. Damit ist die Konfliktauflösung abgeschlossen (lokal). Diese konfliktfreie Version möchte man also hochladen:

git commit -am "some message"

git push

Zu alten Commits zurückkehren

git checkout <old_commit_ID> # zum Zustand bei old_commit_ID zurückkehren.
Resultiert im „detached head state“ - man ist nachher auf keinem Branch. Z.B.:

git checkout c9cda6dc4fd35bf4640f534af4a507f1092fcbce # dies führt zu einem
"detached HEAD" state. Lösung: neuen branch erzeugen.

git checkout master # in die Gegenwart zurückkehren

oder:

git reset --hard <old_commit_ID> # Arbeitsverz. UND HEAD zum älteren commit
zurückkehren

git reset --hard <new_commit_ID> # Zurück in die Gegenwart. Vorsicht! Hierzu
braucht man die new_commit_ID, diese wird jedoch von git log nicht mehr angezeigt.
Lösung: aufschreiben oder mittels **git reflog** finden.

Andere nützliche Befehle

git status # Ist alles ok? Auf welchem branch bin ich?

git log # Was sind die letzten commit messages and commit IDs?

git log origin/branch2 # Wie sieht der Log von "branch2" auf github aus?

gitk # ein grafisches git-Tool, welches die Commits sehr schön darstellt

Damit mehrere Menschen in das gleiche github repository pushen können: Eine Person legt das repo an und fügt alle anderen als “collaborators” hinzu.

Branching (Teil 1)

Branch lokal erzeugen:

```
git branch <my_branch>
```

Zum neuen Branch wechseln:

```
git checkout <my_branch>
```

Den lokal erzeugten Branch zu Github pushen:

```
git push origin HEAD
```

Den lokalen Branch mit dem nun remote existierenden fest verbinden (damit git pull funktioniert):

```
git branch --set-upstream-to=origin/<my_branch> <my_branch>
```

Branching (Teil 2)

Alle existierenden Branches auflisten:

```
git branch -a
```

Zurück zum alten Branch "master" wechseln:

```
git checkout master
```

Den remote Branch branch2, der lokal nicht existiert, von Github holen:

```
git checkout -b branch2 origin/branch2
```

Remote Branches, die gelöscht wurden, werden leider weiterhin lokal als existierend angezeigt. Dieses Problem löst man mit:

```
git fetch --prune
```

Nun wird der remote branch, der nicht mehr existiert, bei git branch -a nicht mehr angezeigt. Der mit ihm verbundene lokale Branch jedoch schon. Diesen lokalen Branch löscht man nun mit

```
git branch -D <branch1>
```

Branching (Teil 3)

Branch „branch1“ löschen. Diese Operation wird fehlschlagen, falls branch1 Änderungen enthält, die in keinen anderen Branch gemergt wurden.

git branch -d <branch1>

Löschen von branch1 erzwingen, auch wenn branch1 nicht gemergte Änderungen enthält. Man verwirft einfach diesen Branch, auch wenn dabei Commits verloren gehen können.

git branch -D <branch1>

Mit diesem Befehl mergt man den branch „new_branch_name“ in den Branch, auf welchem man sich gerade befindet (z.B. master). Dies holt alle Commits aus new_branch_name in den aktuellen Branch, lässt aber new_branch_name unverändert.

git merge <new_branch_name>

Stashing

Szenario:

Man hat Änderungen in der working directory gemacht, sie **nicht** committed, und möchte nun einen git pull machen. Diese lokalen Änderungen stehen dem git pull jedoch im Weg. Lösung: Man „stasht“ sie temporär weg:

`git stash`

macht dann den pull von github:

`git pull`

und holt anschließend die weggestashten Änderungen wieder zurück, auf die heruntergeladenen Commits oben drauf:

`git stash pop`

REBASE vs. MERGE

Note: ein fetch gefolgt von einem **merge** ist das, was passiert, wenn man **git pull** ausführt.

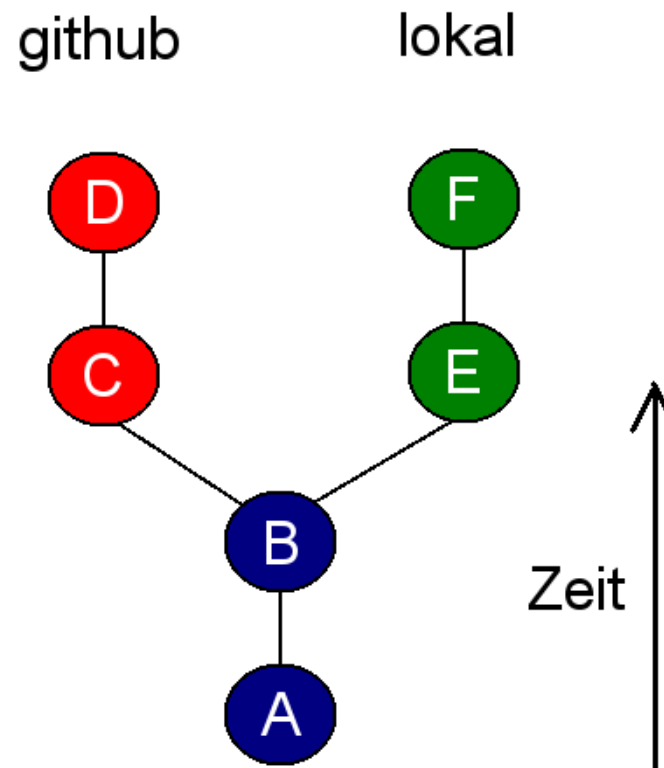
Zweck: Zwei branches (z.B. einen lokalen und einen remote branch) zu mergen, die nicht identisch sind.

Unterschied zw. REBASE und MERGE: rebase hat eine lineare Commit-Historie als Resultat, pull lässt uns sehen, dass die Commits aus zwei unterschiedlichen Quellen kamen.

Ein pull ist in der Regel weniger fehleranfällig als ein merge.

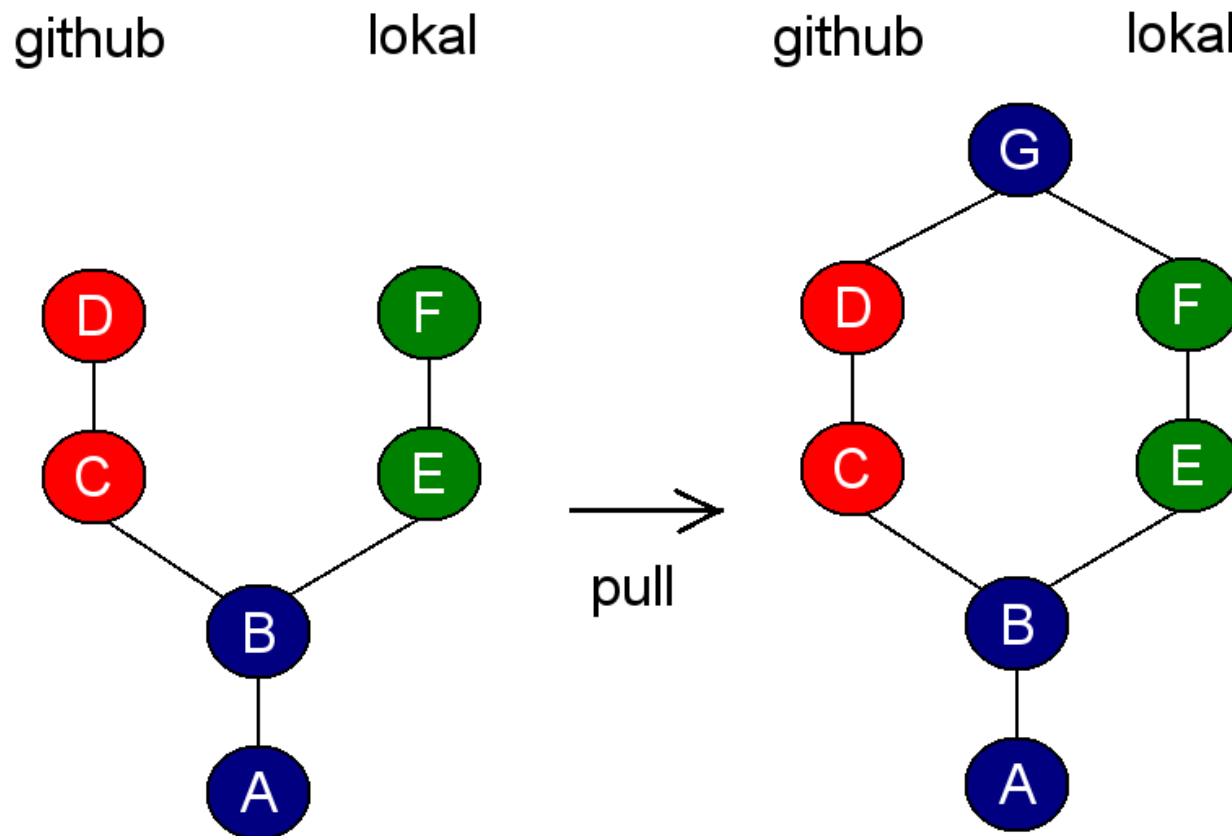
REBASE vs. MERGE

Problem



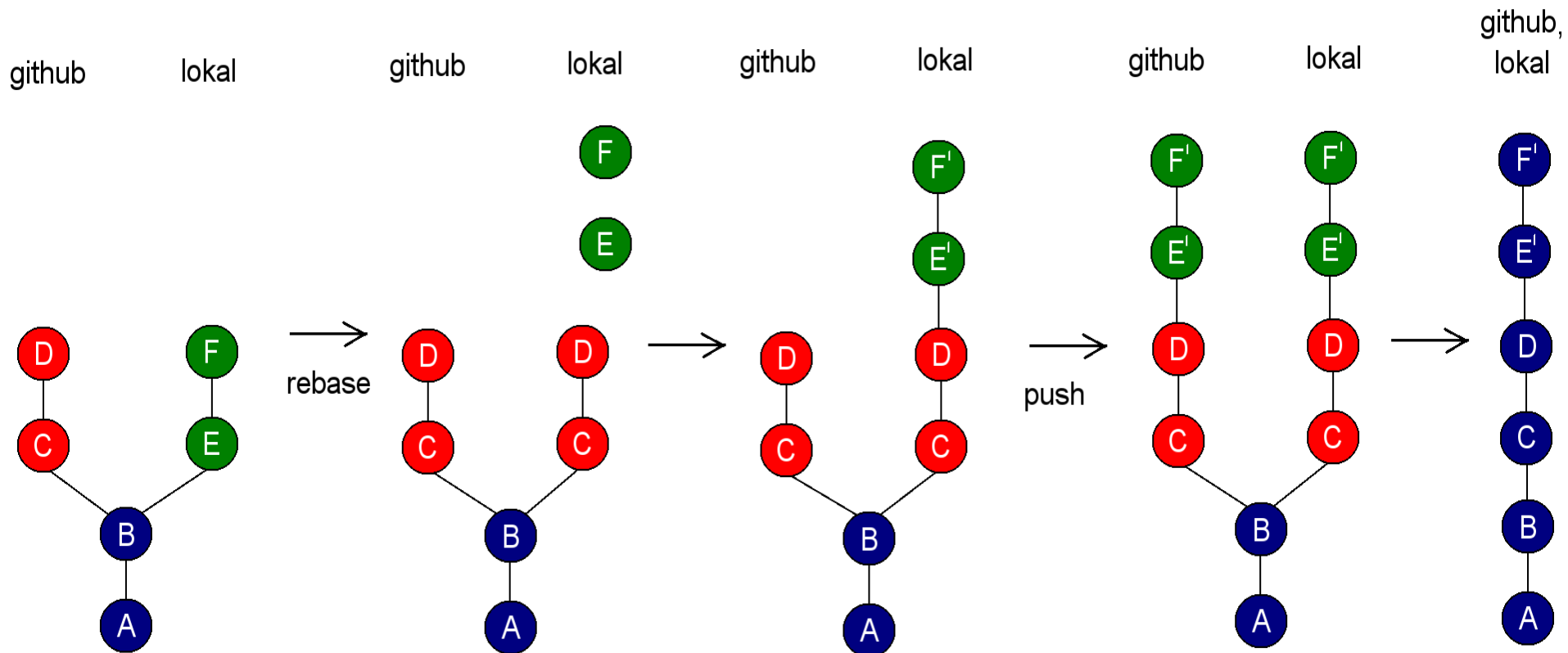
REBASE vs. MERGE

Lösung mit git pull ("merge")



REBASE vs. MERGE

Lösung mit git rebase



MERGE (PULL)

Zunächst muss man alle remote commits zu sich holen und in sein eigenes Repo reinmergen. Falls das remote Repo „origin“ ist und der remote Branch „master“, muss man sie nicht explizit abgeben, ein „git pull“ reicht. Ansonsten:

```
git pull <repo_name> <branch_name>
```

Nun hat man die Commits von github und seine eigenen Commits im lokalen Repo. Bei Github fehlen jedoch noch unsere lokalen Commits. Um beide Repos auf den gleichen Stand zu bringen, muss man seine Commits hochladen:

```
git push
```

REBASE

Alle Änderungen aus dem remote repository holen, die man lokal nicht hat (jedoch erstmal ohne sie mit dem eigenen Repo zu mergen):

git fetch --all

Eigene Commits, die sich im remote repository nicht befinden, weglegen, remote commits runterladen, sie auf eigene commits drauflegen, die das lokale und remote repo gemeinsam haben, und anschließend die weggelegten eigenen commits auf die von github geholten oben drauflegen:

git rebase <repo_name>/<branch_name>

Nun alles, was man bei sich lokal hat, in einer schönen linearen Historie, zu Github hochladen. Ein „git push -f“ (force) ist manchmal nötig, weil ein Rebase die commit IDs verändert. Aber: force ist SEHR gefährlich!

git push

Pull Requests

- Wozu sind sie gut?
 - Ihr arbeitet an einem Software-Projekt. Zu diesem Zweck habt ihr das Repository des Software-Projekts geforkt. Nun ist das Projekt fertig und der Prof möchte eure Arbeit (euren Branch) in den Master-Branch des Software-Projekts reinmergen. Dazu macht ihr einen Pull Request. Der Prof akzeptiert ihn und mergt somit euren Branch in den Master Branch des Projekts.
- Wie macht man sie?
 - Geht auf Github, zu eurem Fork, und sucht nach dem „Pull Request“-Button. Einfach weiterklicken, bis es heißt, dass der Request submittet wurde.

Glossar

HEAD

Eine Art Zeiger auf (meistens) den neuesten Commit des "current branch". Wenn man mit git checkout zu einem anderen Branch wechselt, wird auch HEAD auf den neuesten Commit in diesem neuen Branch zeigen. Man kann überprüfen, auf was HEAD gerade zeigt:

`cat .git/HEAD`

Beispiel-Output:

`ref: refs/heads/master`

Wenn HEAD auf einen Commit zeigt, der nicht der neueste in einem Branch ist, redet man von einem **detached HEAD** state. In diesen Zustand gelangt man nach jedem git checkout, wenn man zu einem Commit geht, der nicht an der Spitze eines Branches ist (also nicht der neueste Commit eines Branches ist).

Glossar

ORIGIN

Origin ist das Original-Remote-Repository, welches geklont wurde, um das lokale Repository zu erzeugen. By default ist es auch das primäre zentrale Repo.

Wenn Repo B als ein Github-Fork von Repo A entsteht, und Repo B auf eine lokale Festplatte geklont wird (wodurch das lokale Repo C entsteht), hat C ein default Remote Namens „**origin**“ - Repo B ist dieses origin.

Um auch die Änderungen in A tracken zu können, muss A als ein zusätzliches Remote hinzugefügt werden.

Glossar

REMOTE

Wenn man ein Repo von Github auf seine Festplatte klonet, ist das geklonte Github-Repo ein „Remote“ des lokalen Repos. Man kann auch weitere Remotes hinzufügen:

```
git remote add <neues_remote>  
<https://github.com/irgendein_github_name/ein_github_repo.git>
```

oder alle existierenden Remotes anzeigen:

```
git remote -v
```

Quellen

Sehr gut und umfangreich:

<https://www.atlassian.com/git/tutorials/setting-up-a-repository/>

Using Branches

<https://www.atlassian.com/git/tutorials/using-branches>

Git – The Simple Guide

<http://rogerdudler.github.io/git-guide/>

Git Newbie Commands

<http://www.javacodegeeks.com/2012/08/git-newbie-commands.html>

<http://stackoverflow.com/questions/5617211/what-is-git-remote-add-and-git-push-origin-master>

Danke!

