Versionsverwaltung mit git

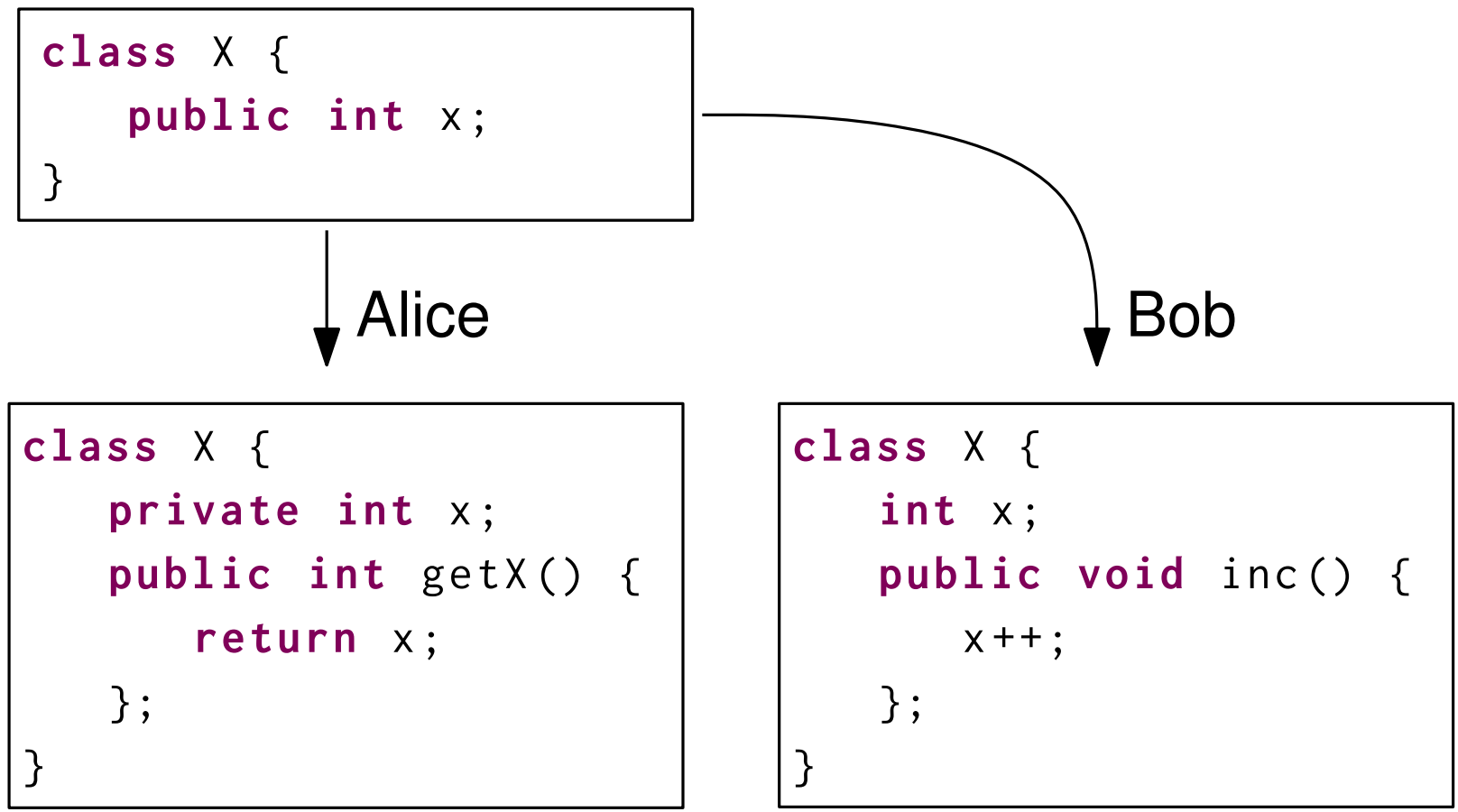
Versionsverwaltung

Versionsverwaltungssysteme verwalten Dateien und zeichnen alle Änderungen an den Dateien im Laufe ihrer Entwicklung auf.

- alte Versionen sind stets verfügbar
- jede Änderung ist dokumentiert
- Entwicklung wird nachvollziehber
- Koordination mehrerer Entwickler
  - gleichzeitiges Arbeiten an allen Teilen des Projekts
  - Zusammenführung von Änderungen an einem Projekt
  - Welche Versionen welcher Dateien passen zusammen?

Versionsverwaltung

Hin und her Kopieren von Dateien mit E-Mail oder Dropbox ist nicht geeignet.



Kopieren der neuesten Version überschreibt eine der Änderungen stillschweigend

Versionsverwaltungssysteme

Software wird mithilfe von Versionsverwaltungssystemen (engl. **Version Control System**) entwickelt.
Beispiele: CVS, SVN, git, Mercurial, ...

- Verwaltung von Datei- und Projektversionen
- Integration von Änderungen verschiedener Entwickler
- Änderungen nachvollziehbar machen
- Erkennen von Konflikten

git

Im Praktikum verwenden wir **git**

- entwickelt von Linus Torvalds u.a. für den Linux Kernel
- weit verbreitet, auch für kleine Projekte

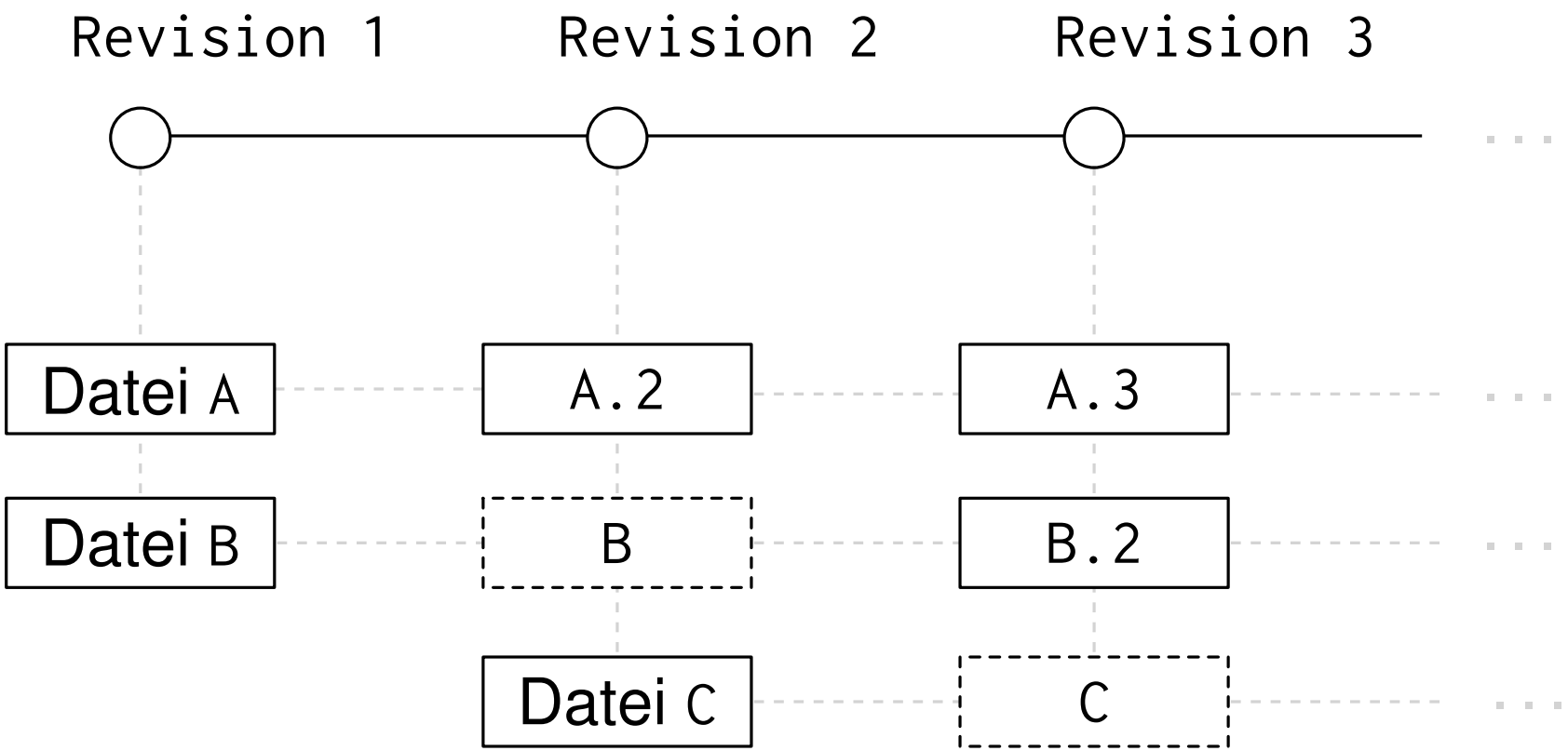
Möglichkeiten zur Benutzung von git:

- Kommandozeile
- integriert in Eclipse, IntelliJ oder Netbeans
- andere GUIs

Die grundlegenden Konzepte sind gleich.

Versionsverwaltung mit git

Ein git-Repository verwaltet verschiedene Revisionen eines Projekts.



Versionsverwaltung mit git

Revisionen werden manuell angelegt.

Arbeitsablauf:

- Dateien ändern und/oder neu angegen/
- Dateien für die nächste Revision vormerken. (**staging**)
- Neue Revision anlegen (**commit**).
  Die neue Revision wird in einem kurzen Text (**commit-message**) kurz beschrieben.
- Revision mit anderen Repositories abgleichen
  - Fremde Revision herunterladen
  - Eigene Revision verschicken

Anlegen eines Repositories

Kommandozeile:

```
git init
```

- macht das aktuelle Verzeichnis zu einem (leeren) git Repository
- eventuell existierende Dateien werden ignoriert

git speichert alle Repository-Daten in einem (versteckten) Unterverzeichnis.

Arbeitsablauf: `git add` und `git commit`

- **Dateien bearbeiten oder anlegen** (wie üblich)
- **Inhalte zur nächsten Revision hinzufügen**

```
git add datei1
git add datei2
...
```

- **Revision fertig stellen**

```
git commit -m "kurze Beschreibung der Revision"
```

Eine commit-message ist notwendig. Ruft man nur

```
git commit
```

auf, so wird ein Editor gestartet, in dem man die Nachricht eingeben kann.

Revisionshistorie anzeigen: `git log`

Kommandozeile:

```
git log
```

Ausgabe:

```
commit b5902ac4d86bdb84e82b453492ff7650028c4567
Author: Ulrich Schoepp <schoepp@tcs.ifi.lmu.de>
Date:   Sun Oct 18 12:13:20 2015 +0200

    Text uebersetzt

commit 6d118f0d2d2f5f3b97f3b032a2d3ccb3c8622e40
Author: Ulrich Schoepp <schoepp@tcs.ifi.lmu.de>
Date:   Sun Oct 18 11:36:35 2015 +0200
...
```

Jede Revision hat einen eindeutigen Namen (z.B. `b5902...`).
Die aktuelle Revision heißt auch `HEAD`.

Revisionen zurückholen: `git checkout`

Kommandozeile:

```
git checkout revisionsname
```

Beispiel:

Gehe zur ersten Revision:

```
git checkout 6d118f0
```

Zurück zur aktuellsten Revision:

```
git checkout master
```

N.B. `master` bezeichnet den Hauptzweig des Repositories. Es gibt auch Nebenzweige, die wir hier nicht betrachten.

Status: `git status`

Anzeigen des Status aller Dateien.

```
git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   src/test/Main.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)

src/test/Point.java

no changes added to commit (use "git add" and/or "git commit -a")
```

Anzeigen von Änderungen: `git diff`

Alle Änderungen seit der letzten Revision:

```
git diff
```

Änderungen an einer Datei seit der letzten Revision:

```
git diff dateiname
```

Änderungen seit einer bestimmten Revisionen:

```
git diff rev1
git diff HEAD@{yesterday}
```

Änderungen zwischen zwei Revisionen:

```
git diff rev1..rev2
```

Verteilte Versionsverwaltung

Daten können zwischen verschiedenen git-Repositories ausgetauscht werden.

- `clone`: duplizieren eines git-Repositories in einem eigenen Verzeichnis
- `pull`: neue Revisionen von einem anderen Repository übernehmen
- `push`: neue Revisionen auf ein anderes Repository übertragen

Bei der Entwicklung hat jeder ein eigenes Repository.
Die Daten werden zwischen den Repositories ausgetauscht.

Remote Repositories

Kopieren eines Repositories:

```
git clone url
```

Lädt den Inhalt eines gesamten git-Repositories von `url`, inklusive der kompletten Versionsgeschichte.

Es wird eine Referenz auf das remote Repository gespeichert. Diese hat oft den Namen `origin`.

Beispiel:

- Dienste wie github oder gitlab bieten die zentrale Speicherung von git-Repositories an.
- Zum Zugriff auf das Repository wird eine `url` für `git clone` angegeben.

Remote Repositories

Normales Arbeiten mit dem git-Repository:
Änderungen, `add`, `commit`, Änderungen, `add`, `commit`, `...`

Neue Revisionen können vom remote Repository in das lokale übernommen werden.

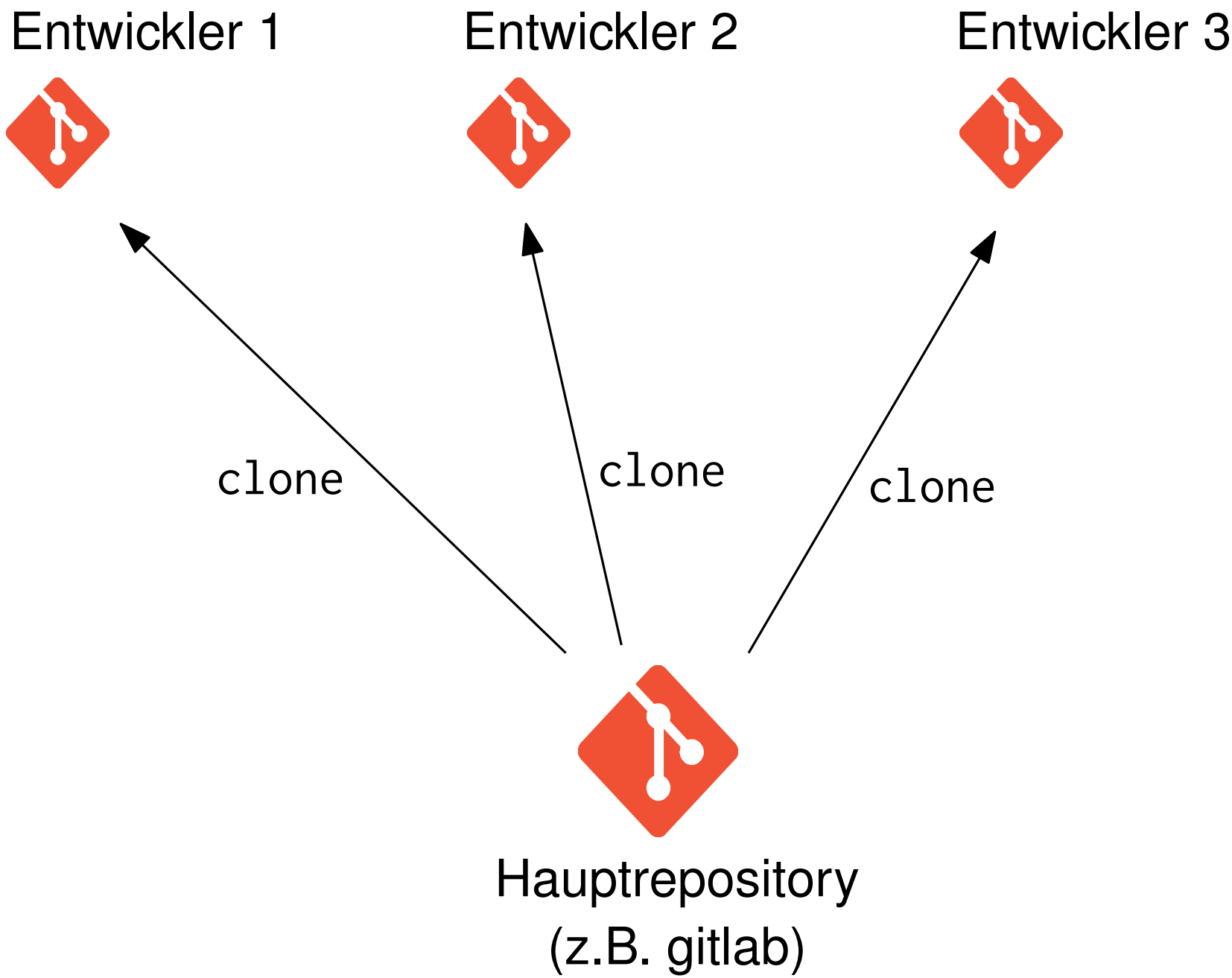
```
git pull
```

Hat man Schreibzugriff auf das remote Repository, so kann man die neuen Revisionen dort hochladen.

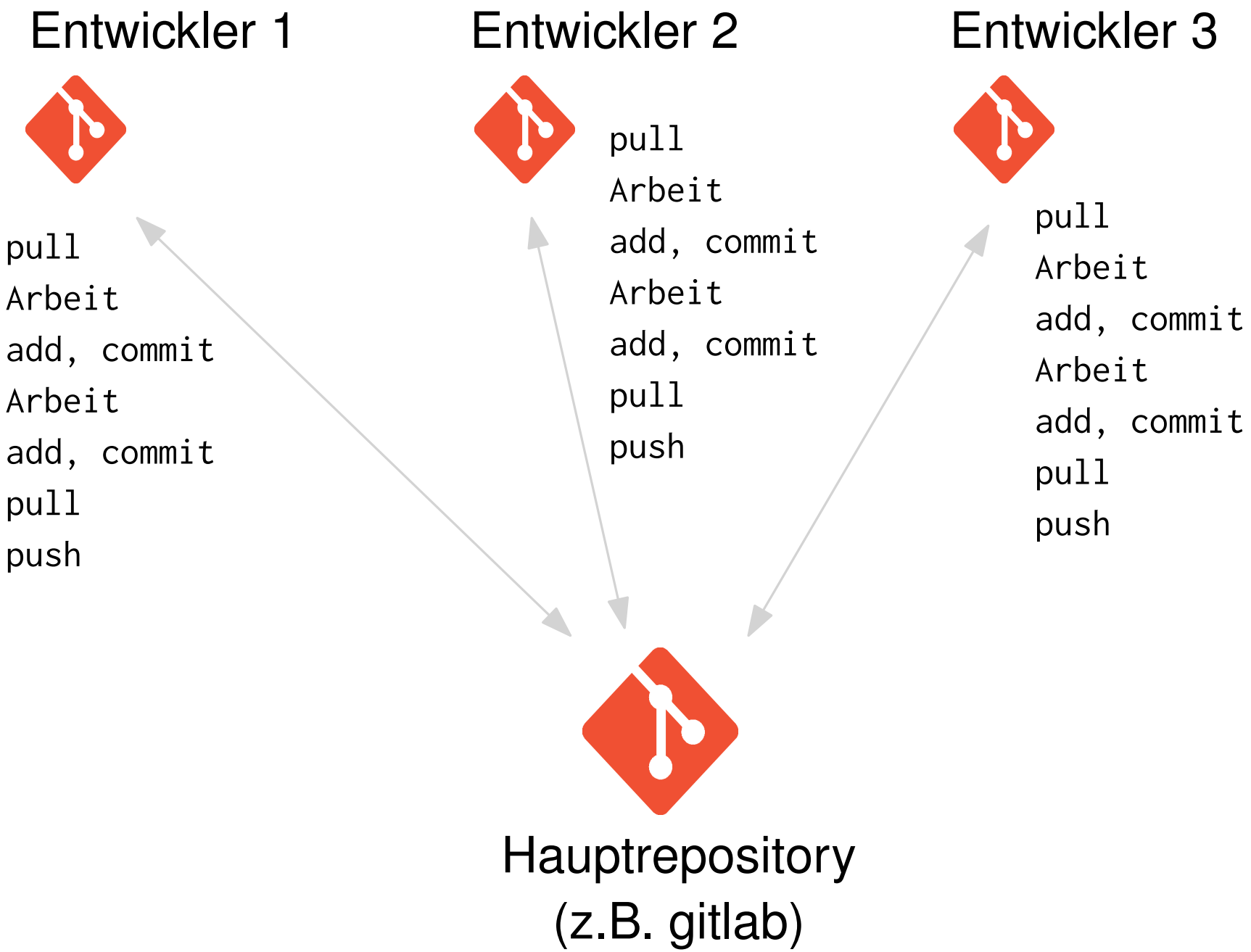
```
git push
```

Dies ist nur möglich, wenn das lokale Repository auf dem neuesten Stand ist (evtl. `pull` nötig).

Möglicher Arbeitsablauf

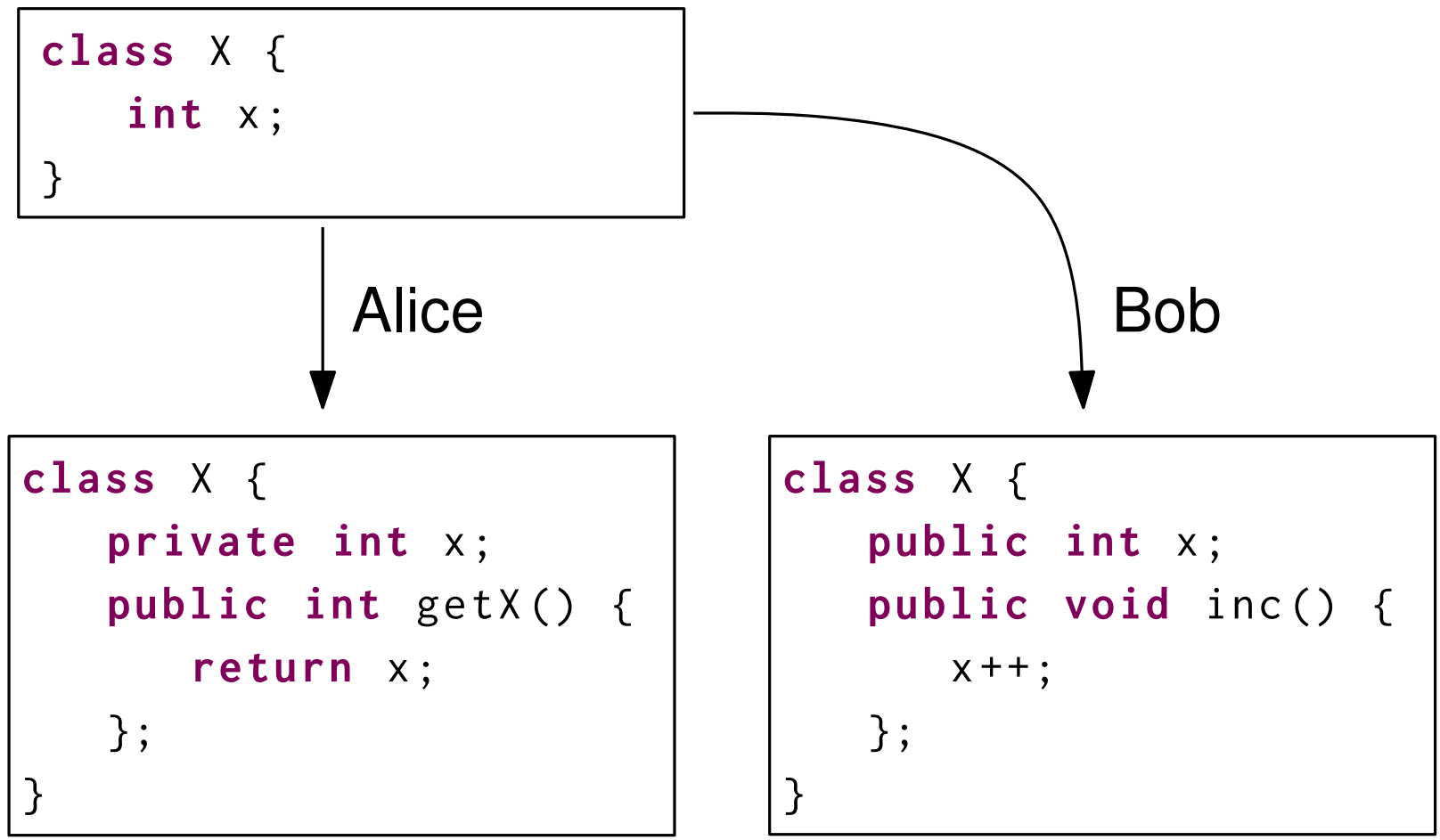


Möglicher Arbeitsablauf

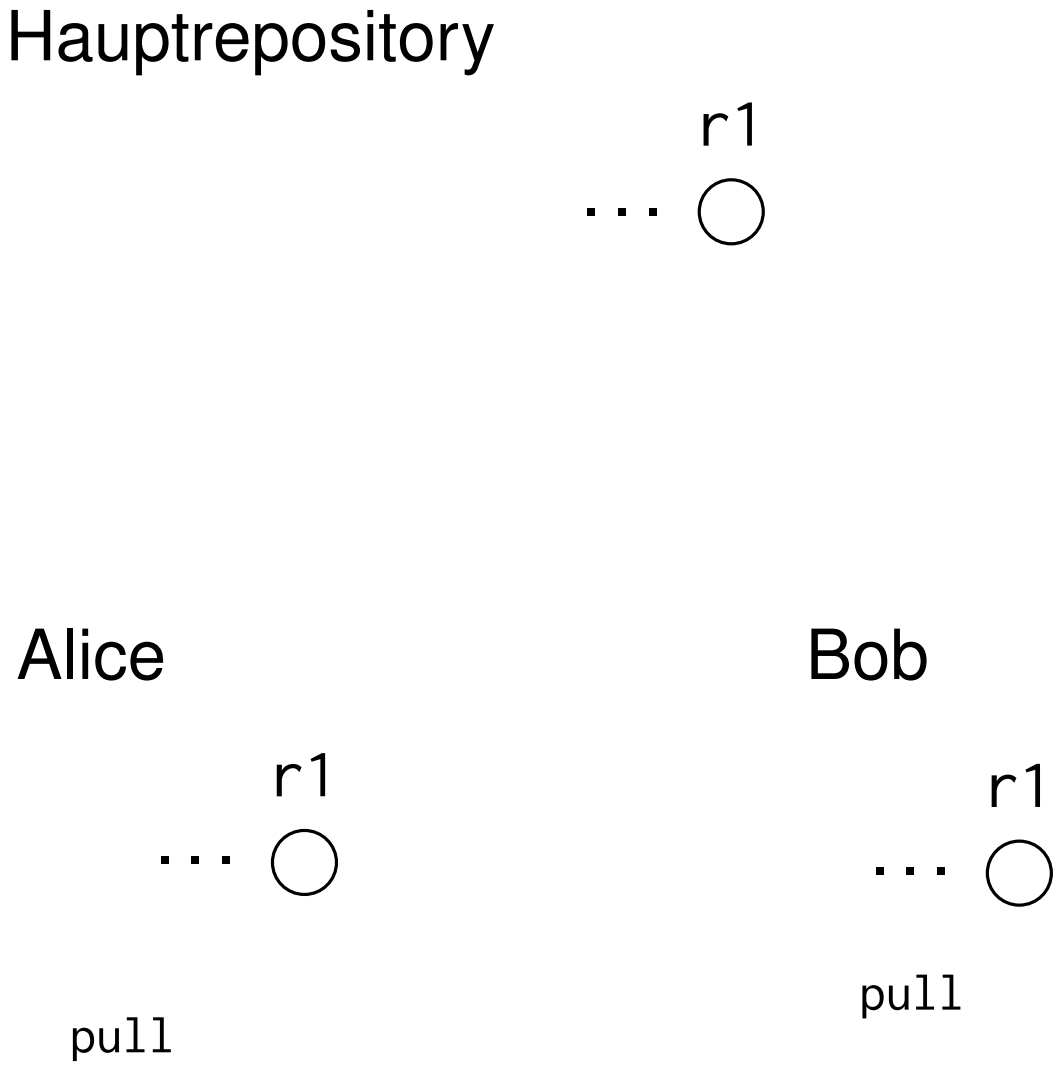


Konflikte

Bei der parallelen Arbeit an einem Projekt kann es zu Konflikten kommen.

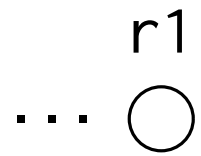


`git pull` führt ein `merge` aus

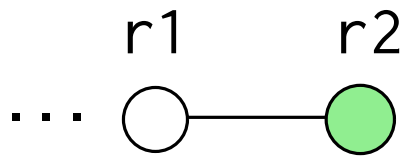


`git pull` führt ein `merge` aus

Hauptrepository

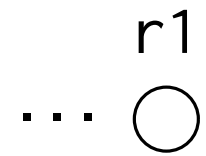


Alice




```
pull
add, commit
```

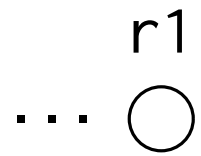
Bob



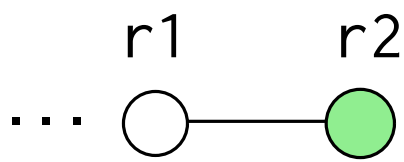

```
pull
```

`git pull` führt ein `merge` aus

Hauptrepository

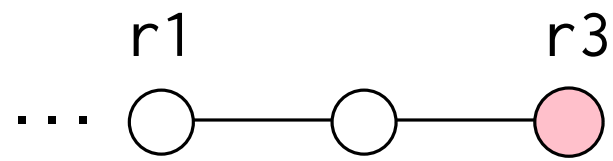


Alice




```
pull
add, commit
```

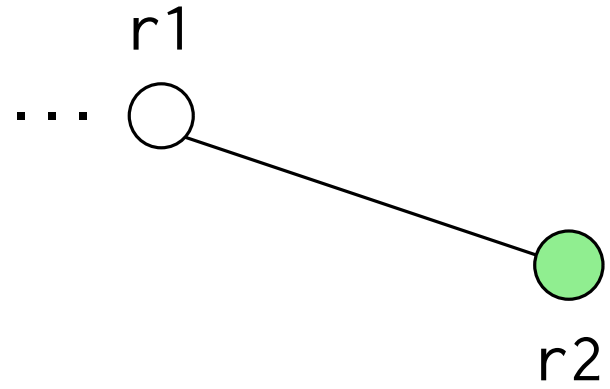
Bob



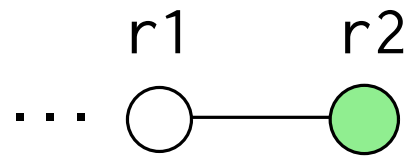

```
pull
add, commit
add, commit
```

`git pull` führt ein `merge` aus

Hauptrepository

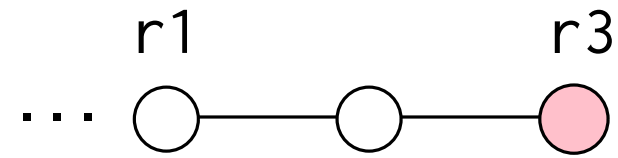


Alice




```
pull
add, commit
push
```

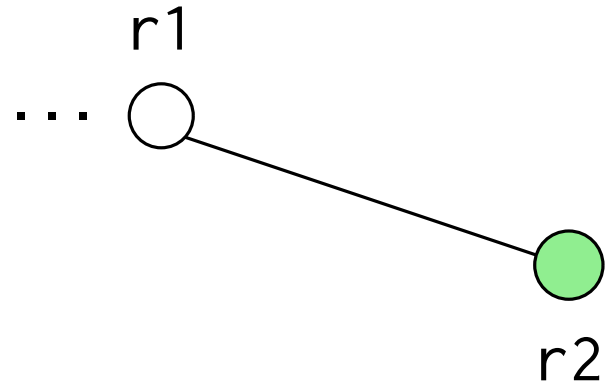
Bob



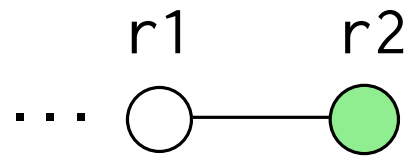

```
pull
add, commit
add, commit
```

`git pull` führt ein `merge` aus

Hauptrepository

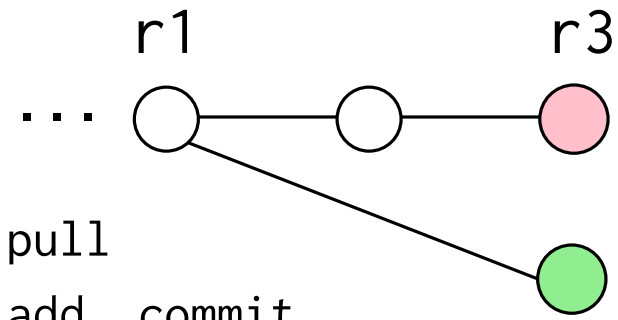


Alice




```
pull
add, commit
push
```

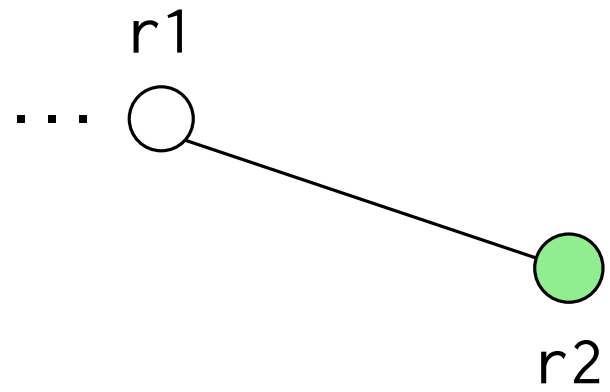
Bob



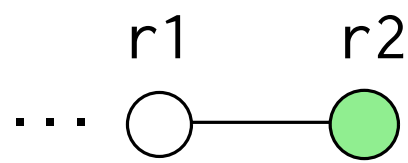

```
pull
add, commit
add, commit
pull
```

`git pull` führt ein `merge` aus

Hauptrepository

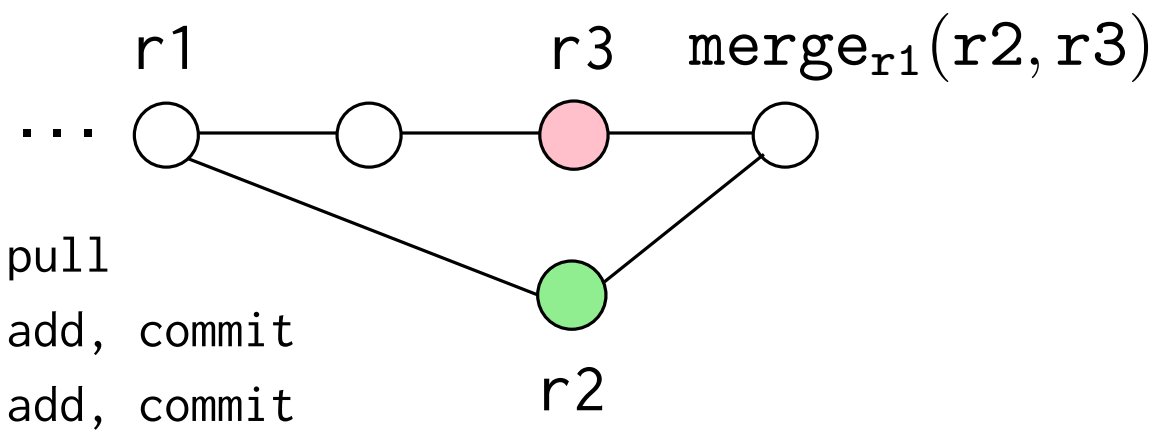


Alice




```
pull
add, commit
push
```

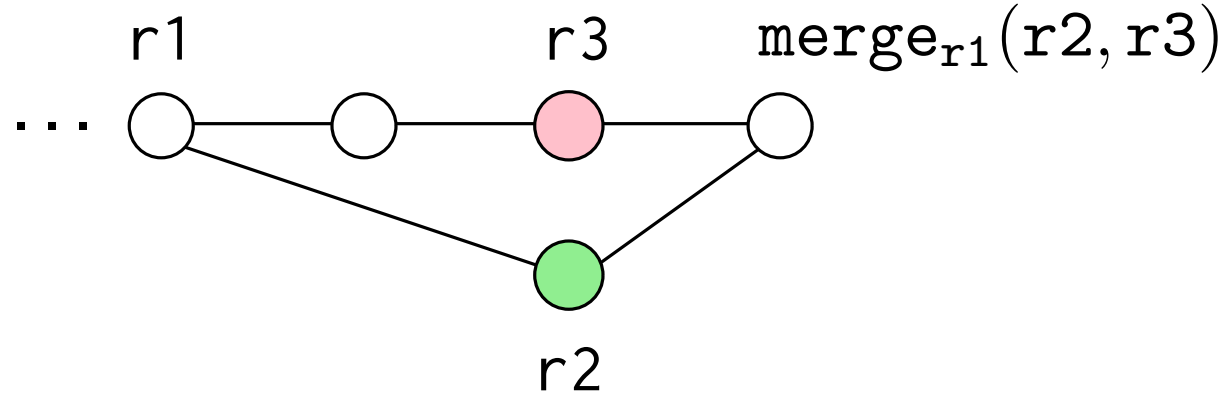
Bob



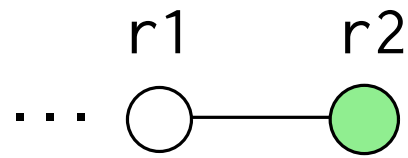

```
pull
add, commit
add, commit
pull
```

`git pull` führt ein `merge` aus

Hauptrepository

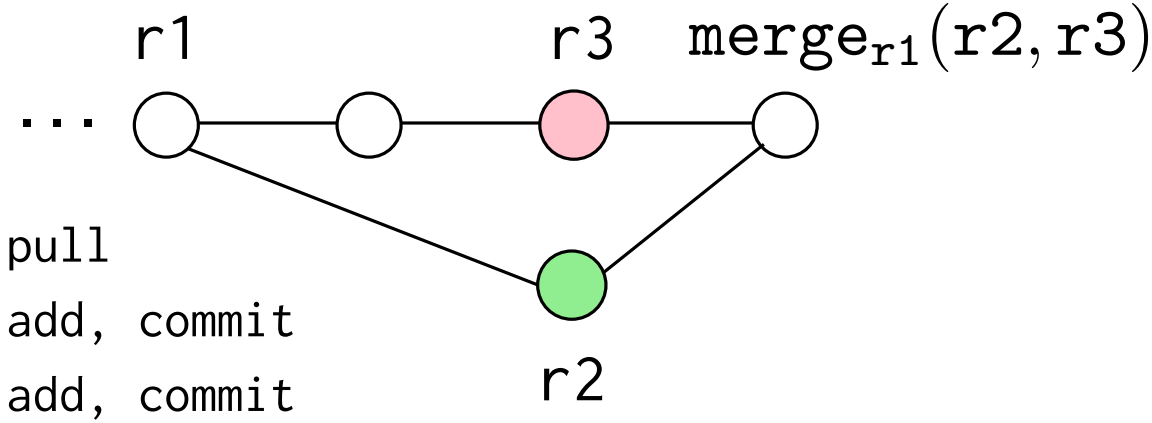


Alice




```
pull
add, commit
push
```

Bob




```
pull
add, commit
add, commit
pull
push
```

Konflikte

Bei einem `merge` können Konflikte auftreten, wenn es in beiden Revisionen Änderungen an der gleichen Stelle gibt.

- git kennzeichnet die Konflikte in der Datei selbst.

```
class X {
<<<<<<< HEAD
   private int x;
   public int getX() { return x; };
=======
   public int x;
   public void inc() { x++; }
>>>>>>> 41bff37efee048db1ef40c779e63a1686028af97
}
```

- Konflikte müssen manuell behoben werden:
  Ersetze alle markierten Teile durch den gewünschten Text.
- Das Merge wird mit `add`, `commit` abgeschlossen.

Konflikte

Bei einem `merge` können Konflikte auftreten, wenn es in beiden Revisionen Änderungen an der gleichen Stelle gibt.

- git kennzeichnet die Konflikte in der Datei selbst.

```
class X {

   private int x;
   public int getX() { return x; };


   public void inc() { x++; }

}
```

- Konflikte müssen manuell behoben werden:
  Ersetze alle markierten Teile durch den gewünschten Text.
- Das Merge wird mit `add`, `commit` abgeschlossen.

Zusammenfassung: git

- Repository anlegen *oder* klonen **(nur einmal)**
  ⇒ `git init` *oder* `git clone` <*url*>
- Dateien für die nächste Revision vormerken
  ⇒ `git add` <*dateiname*>
- Neue Revision anlegen
  ⇒ `git commit`
- Fremde Revision herunterladen
  ⇒ `git git pull`
- Eigene Revision verschicken
  ⇒ `git push`
- Spezielle Revision laden
  ⇒ `git checkout` <*versionsname*>

Allgemeine Hinweise

- Repository regelmäßig mit `pull` aktualisieren
- viele kleine Commits
- `push` nur wenn Projekt in konsistentem Zustand
- aussagekräftige Commit-Nachrichten
- alten Programmcode nicht auskommentieren; gleich löschen

git im Praktikum

- Alle Teilnehmer registrieren sich auf:
  `https://gitlab.cip.ifi.lmu.de`
- Für jedes Projekt wir ein eigenes git-Repository angelegt mit Namen `<gruppenname>-<nachname>-<projektname>`
- Repository freigegeben für:
  Stephan Barth, Steffen Jost und den jeweiligen Gruppentutor. Vollzugriff erforderlich!
  Members > Add Members als Developer

.gitignore

Vorsicht, nicht alle Dateien ins Repository einstellen!

Ins Repository gehören:

- Alle `.java` und `.fxml` Dateien

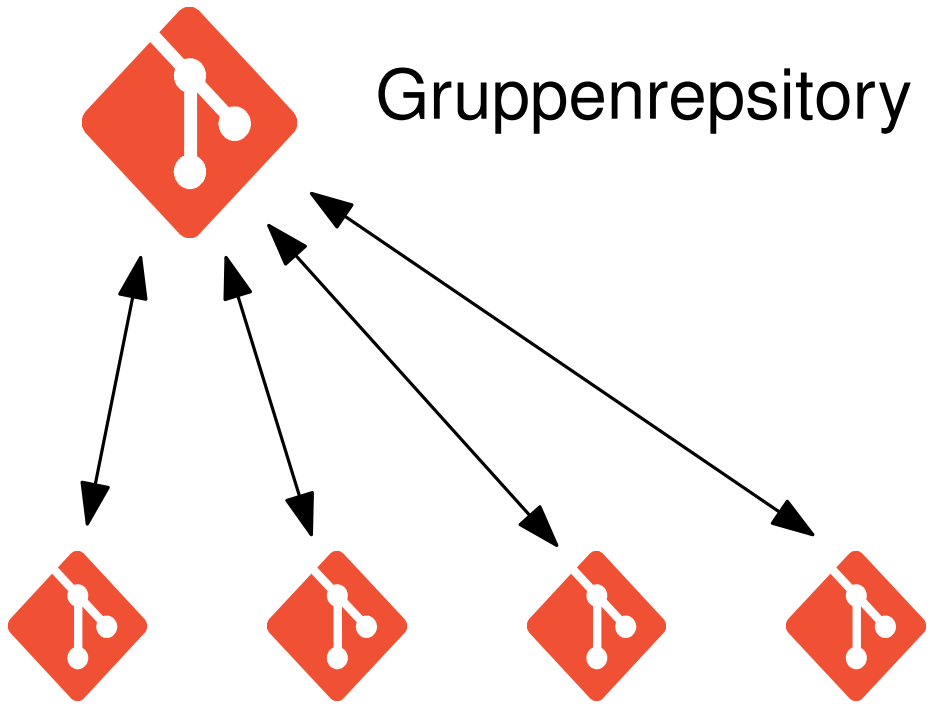
Niemals ins Repository:

- keine IDE-Dateien (z.B. bei intellij nicht `.idea`)
- keine Konfigurationsdateien mit absoluten Pfadangaben
- keine temporären/erzeugten Dateien

In einer Datei `.gitignore` kann angegeben werden, welche Dateien und Pfade von git ignoriert werden sollen.
Vorlage auf SEP Homepage verfügbar.

git für Gruppenprojekte

Für jedes Gruppenprojekt wird jeweils ein frisches Projekt in `gitlab` angelegt: `<gruppenname>-<projektname>`
Ein Gruppenmitglied erstellt es und gewährt anderen Gruppenmitgliedern Schreibzugriff, Tutoren Lesezugriff
Members > Add Members
Gruppenmitglieder: `git clone` *<url des projekts>*



Gruppenrepsitory auf gitlab.cip.ifi.lmu.de

jedes Gruppenmitglied hat ein lokales Repository

git mit intellij oder Eclipse

Anlegen eines gemeinsamen Eclipse-Projekts:
(Details siehe Screencast auf Praktikumshomepage)

Einmal für jede Gruppe:

- In Editor ein Java-Projekt anlegen
- Projektdateien in ein lokales git-Repository einchecken
- `.gitignore` nicht vergessen, ggf. anpassen
- git-Repository auf `gitlab.cip.ifi.lmu.de` anlegen
- Push vom lokalen git-Repository auf das `gitlab`-Repository

Rest der Gruppe:

- Import des Java-Projekts direkt von `gitlab`
- das Hauptrepository wird dabei gecloned

git mit intellij

Durch die Entwicklungsumgebung vereinfachen sich viele Arbeitsschritte: z.B. bietet intellij zwei Knöpfe für pull und push an, wobei letzterer mit Hilfe der Dialoge eine Kombination von `add + commit + push` ist.

Auch Merging im Falle von Konflikten wird bequem grafisch dargestellt.

Status der Dateien ist Farbcodiert:

Rot Datei nicht im Repository

Grün Datei added, aber noch nicht committed

Blau Datei seit letzter Revision geändert, noch nicht committed

Schwarz Datei im Repository und unverändert