Universität Bremen

Praktische Informatik 1
Versionsverwaltung mit git 2

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für
Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

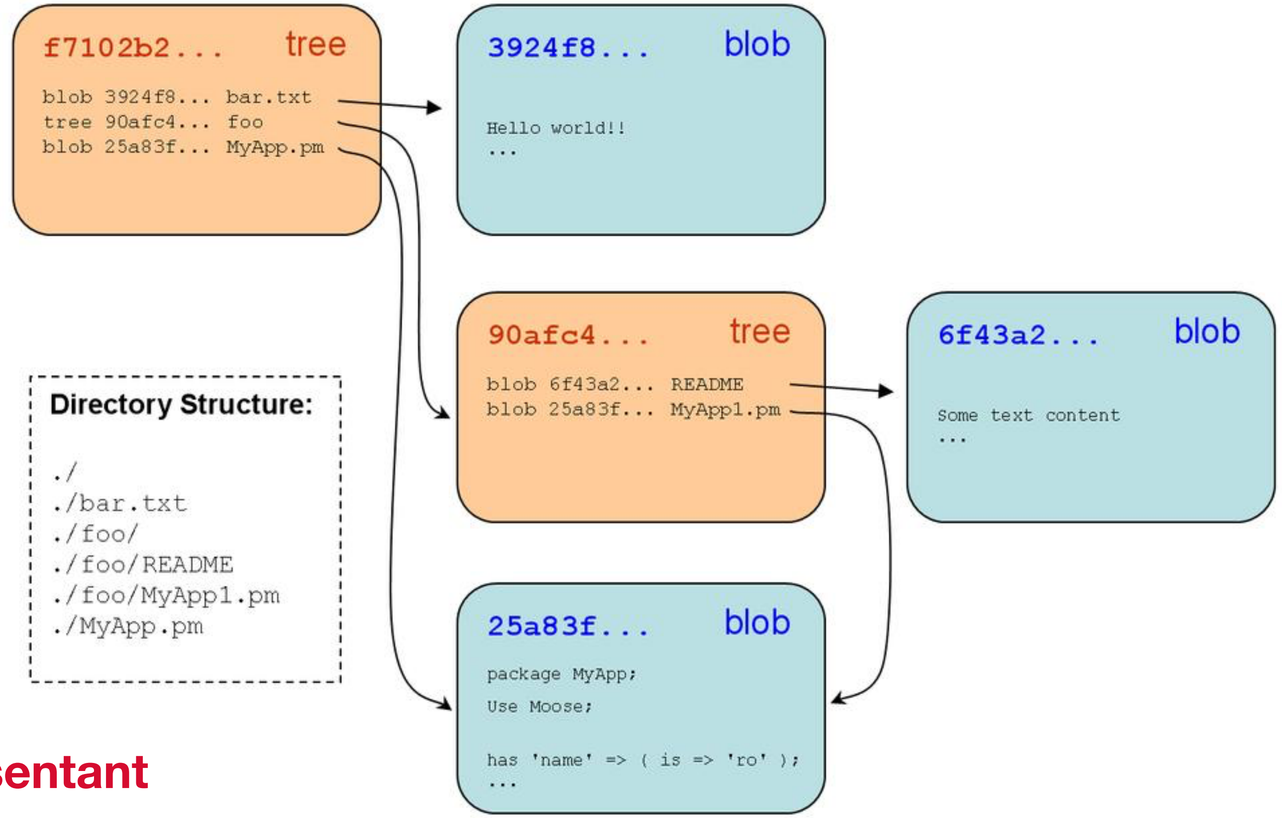
Universität Bremen

SHA-1

- Ein git-Repository ist im Prinzip ein **HashSet<String>** (mit direkter Suche nach dem Hash-Wert), wobei die Zeichenketten Dateiinhalte, Verzeichnisse, Commits usw. sind
- Die Hash-Werte werden mit dem **Secure Hash Algorithm** (**SHA-1**) aus den Daten selbst bestimmt
  - Sind eigentlich immer eindeutig
  - Hash-Werte haben 160 Bit und werden als 40-stellige Hexadezimalzahl geschrieben (oft sind deutlich kürzere Ziffernfolgen schon eindeutig)
- **git** benutzt die Hash-Werte zum Verweisen auf beliebige Informationen
- Durch **SHA-1** ist die Zuordnung von Hash-Werten zu Daten in jedem **git**-Repository gleich

Verweisstruktur in git

https://nnc3.com/mags/LJ_1994-2014/LJ/208/10971.html

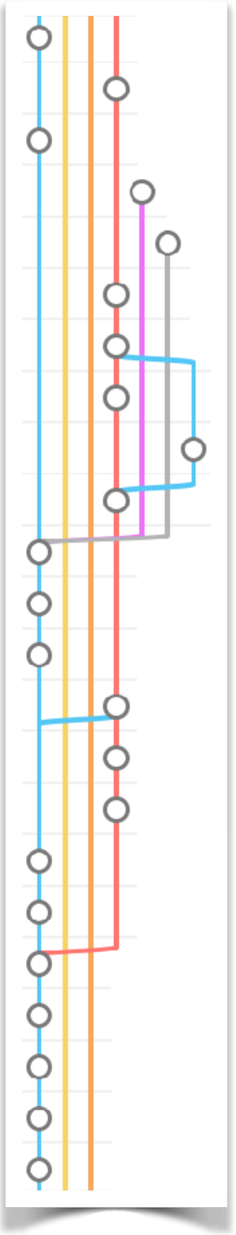


- Eine Datei wird durch ihren **SHA-1**-Hash adressiert
- Ein Verzeichnis ist eine Datei, die Paare von Namen und den dazugehörigen **SHA-1**-Hashes der Unterverzeichnisse bzw. Dateien enthält (und mehr)
  - Als Datei hat es auch einen **SHA-1**-Hash
- Der **SHA-1**-Hash des Wurzelverzeichnisses ist **Repräsentant** des Zustands des gesamten Verzeichnisbaums
  - In ihn gehen rekursiv die **SHA-1**-Hashes aller Dateien im Baum ein
- Ein Commit enthält die **SHA-1**-Hashes des Wurzelverzeichnisses und der Vorgänger-Commits
  - Hat somit auch einen **SHA-1**-Hash, über den Commits in **git** eindeutig identifiziert werden

Universität Bremen

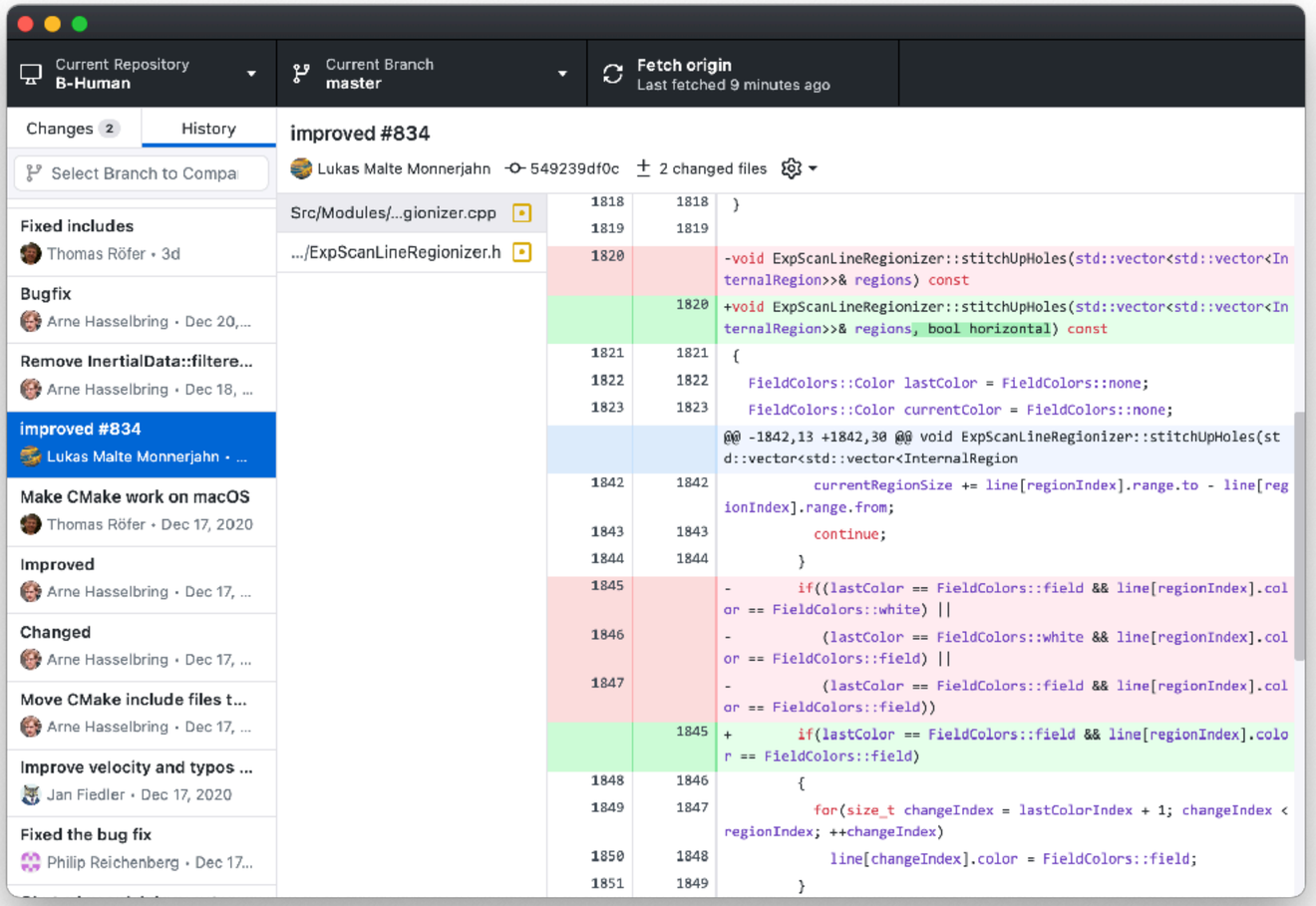
Begriffe

- **Commit**: Zustand aller unter Versionsverwaltung stehender Dateien zu einem bestimmten Zeitpunkt, der explizit festgehalten wurde
  - Enthält **SHA-1**-Hash des Wurzelverzeichnisses (und Nachricht, Autor, Committer)
  - Jeder Commit kennt **SHA-1**-Hash seiner Vorgänger-Commits → Historie
  - Wird natürlich auch durch **SHA-1**-Hash repräsentiert
- **Branch**: Ein benannter Commit (Zuordnung von Name zu **SHA-1**-Hash)
- **Repository**: Eine Ansammlung von Commits mit der Markierung, welcher lokale Branch der aktuelle ist (**HEAD**)
- **Index** (**Staging Area**): Explizit zusammengestellter Zustand von Dateien, aus dem der nächste Commit erzeugt werden kann



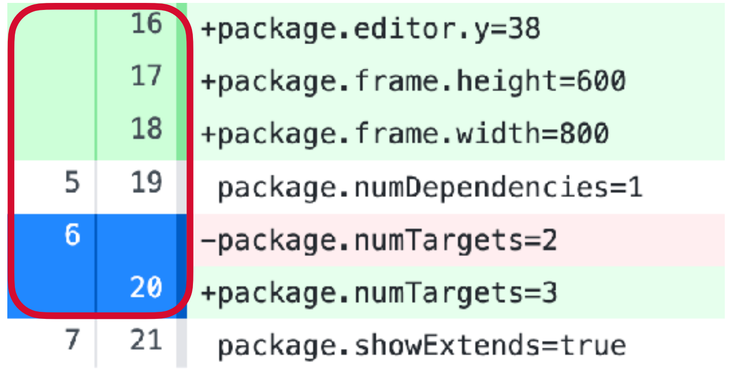
Universität Bremen

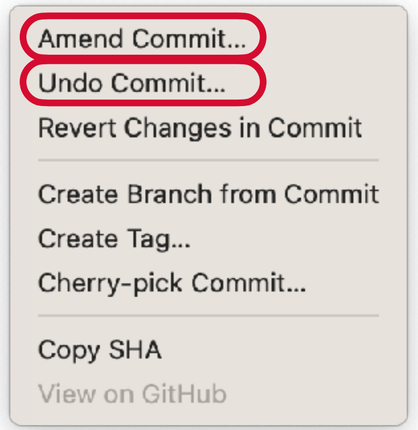
Besser Commit-ten: Demo

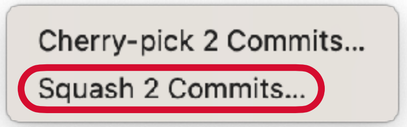


Besser Commit-ten

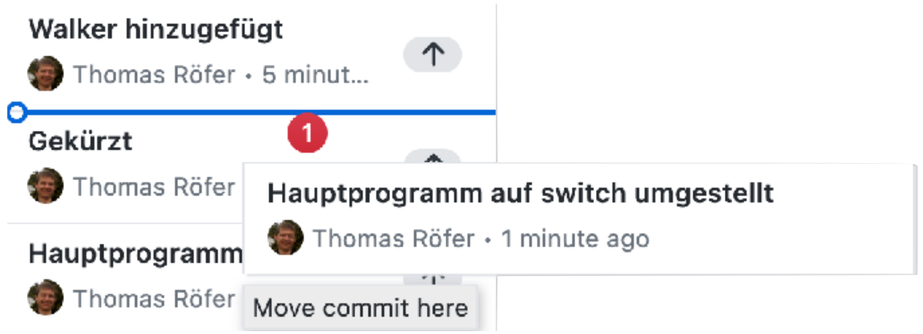
- **Blockweises Staging**: Nur Änderungen in einen Commit aufnehmen, die zu bestimmtem Thema gehören
  - Nachteil: In der Form wurde der Code nie getestet
- **Amend Commit…**: Commit erweitern (**git commit --amend**)
- **Undo Commit…**: Letzten Commit aus Historie entfernen (Änderungen bleiben in Arbeitskopie)
  - Nur bei Commits, die nicht ge-push-t wurden
- **Squash Commits…**: Fasst mehrere Commits zu einem zusammen







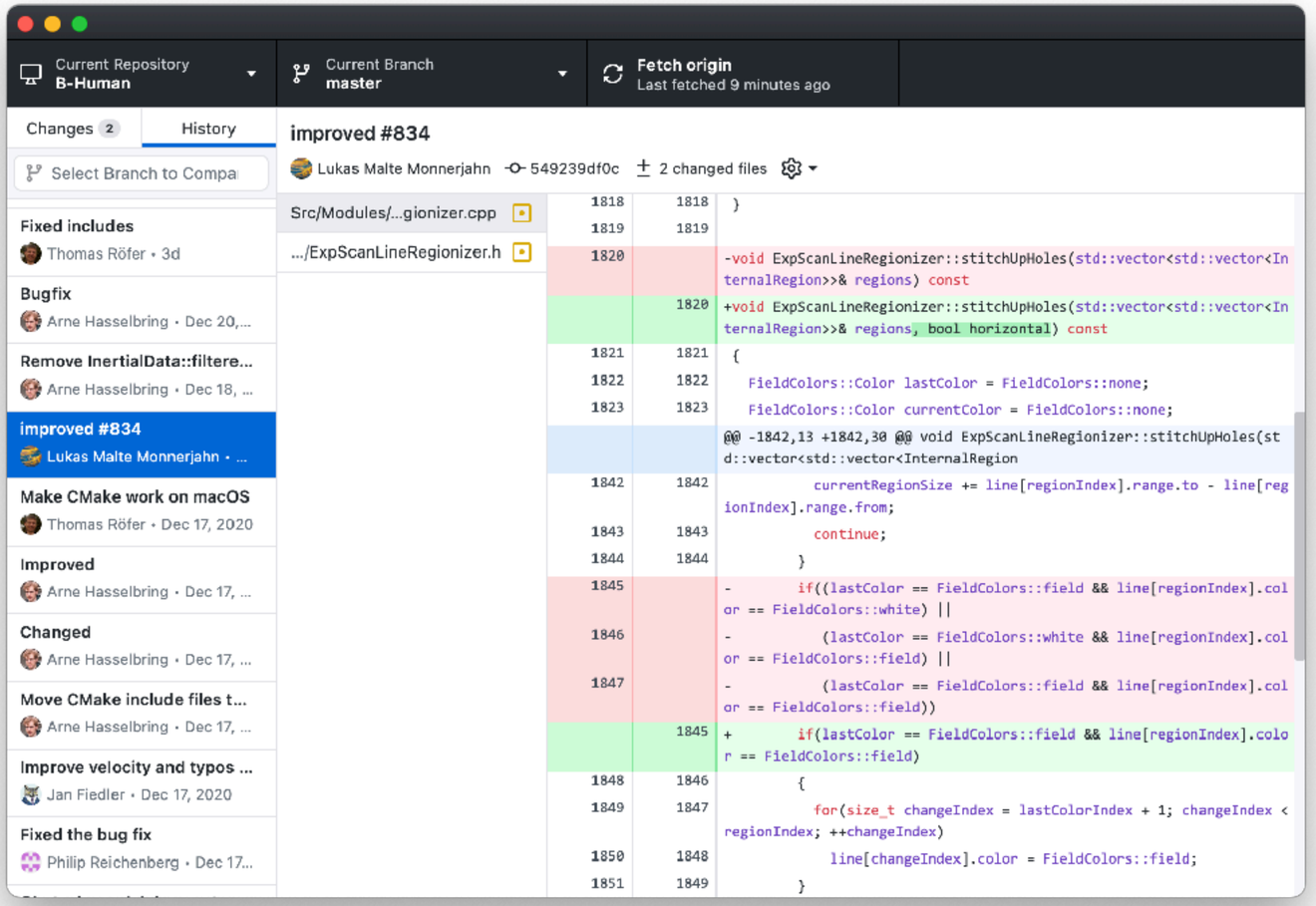
Besser Commit-ten



- **Historie umsortieren**: Per Drag & Drop Commits neu anordnen (**git rebase -i**)
  - Commit zwischen zwei andere Commits ziehen, um Reihenfolge zu ändern (Konflikt möglich)
  - Commit auf einen anderen Commit ziehen, um beide zu kombinieren (wie **Squash Commit…**)
- Eine veränderte Historie kann nur mit **forciertem Push** auf Server übertragen werden (**git push --force**)
  - Datenverlust möglich, da Commits auf dem Server überschrieben werden!
  - Wird von GitLab für den Branch **main** normalerweise abgelehnt

Universität Bremen

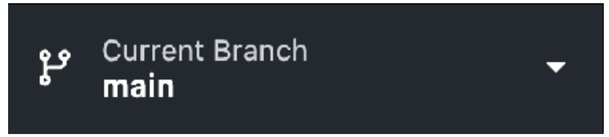
Branches: Demo



Universität Bremen

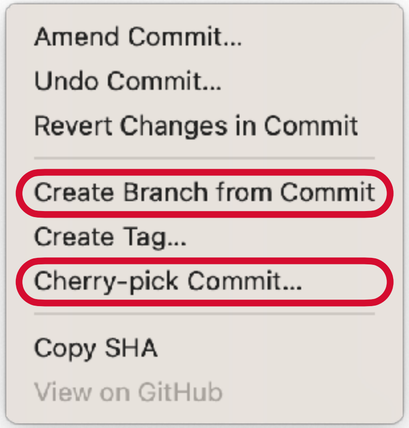
Branches

- **Current Branch**: Wählt den aktuellen Branch, d.h. den **HEAD**

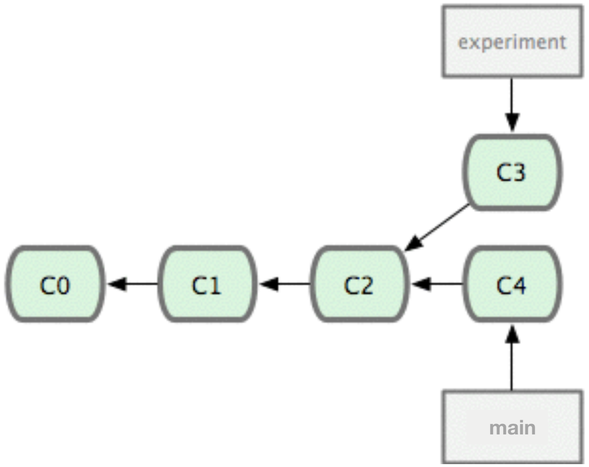


  - Änderung check-t aktuellen Stand des Branches in die Arbeitskopie aus
  - **git** selbst und andere Werkzeuge (**GitHub Desktop** nicht) erlauben es, beliebige Commits auszu-check-en, wodurch **HEAD** nicht mehr auf einen Branch zeigt (**Detached Head**)

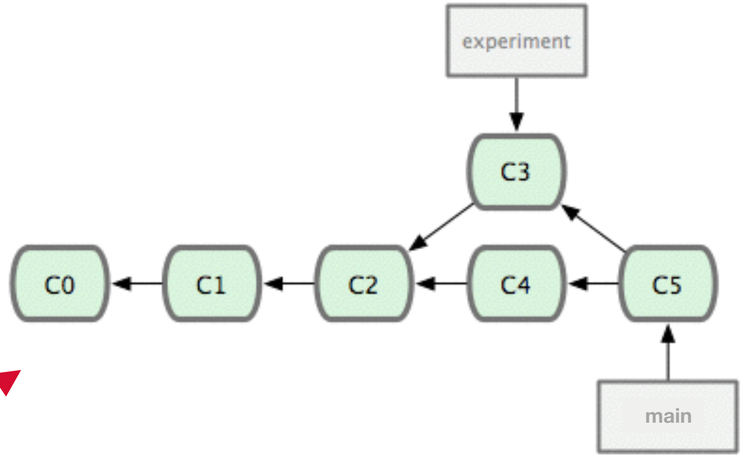
- **Create Branch from Commit**: Erzeugt neuen Branch, der auf den ausgewählten Commit zeigt (**git checkout -b**)
- **Cherry-pick Commit(s)**: Commit(s) zu anderem Branch hinzufügen (**git cherry-pick** ist etwas anders: Anderen Commit zum **HEAD** hinzufügen)
- **Merge**/**Rebase**: Branches zusammenführen



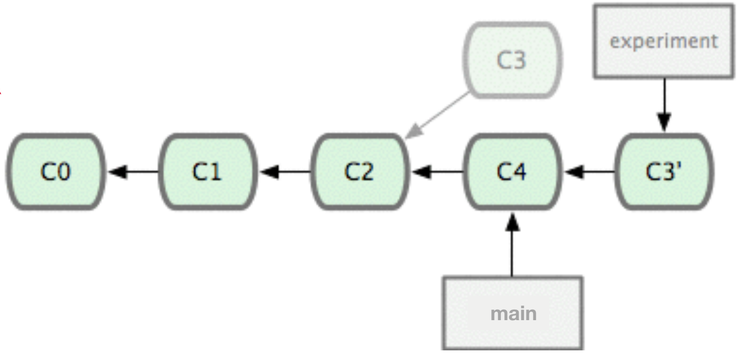
Merge und Rebase



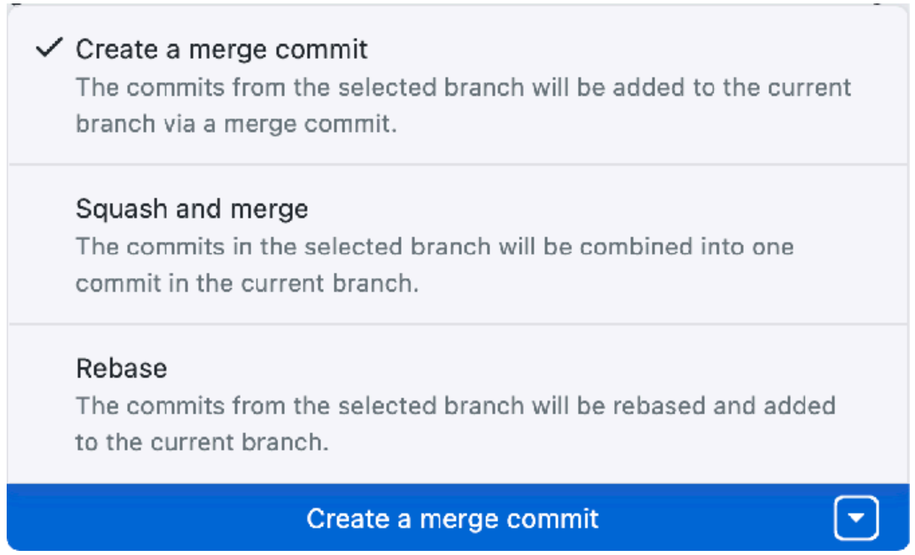
Merge **experiment** in **main**



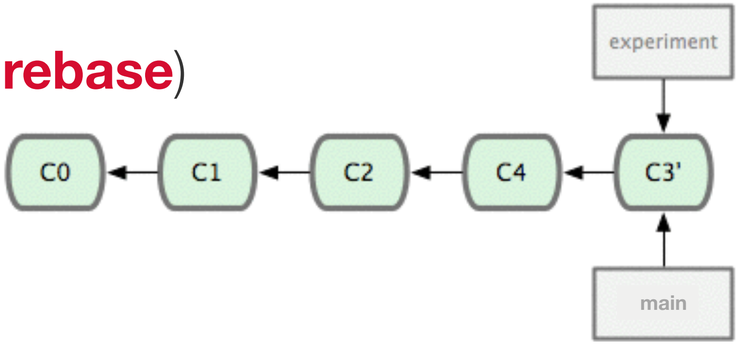
Rebase **experiment** auf **main**



Branches zusammenführen



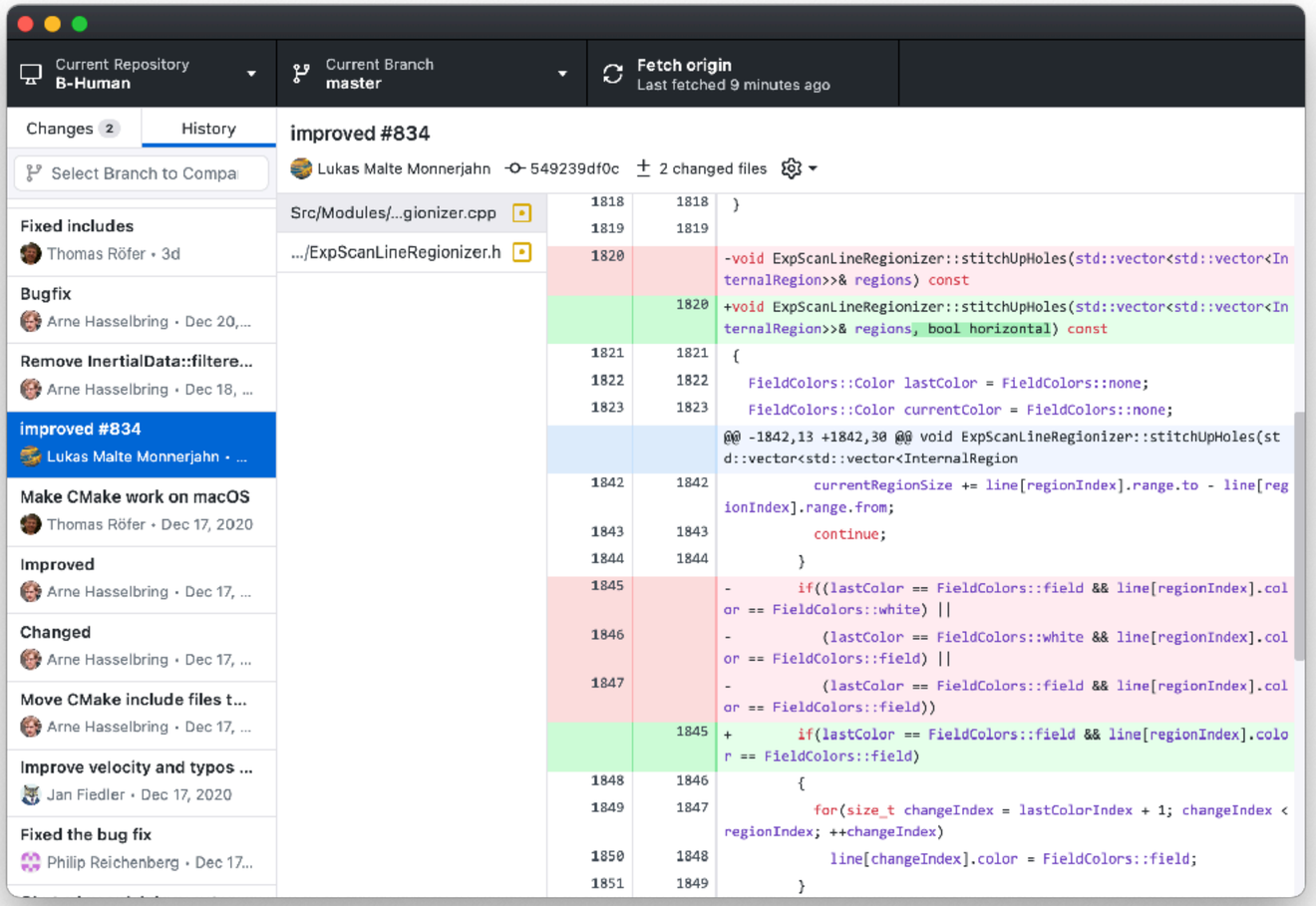
- In GitLab Desktop über **Current Branch** → **Choose a branch to merge**
- **Create a merge commit**: Alle Commits eines anderen Branches werden durch Merge Commit hinzugefügt (**git merge**)
  - Sonderfall **Fast Forward**: Anderem Branch fehlen keine Commits aus **HEAD**
- **Rebase**: **HEAD** nutzt Commits des anderen Branches als Basis (**git rebase**)
  - Z.B. erst Rebase **experiment** auf **main**, dann Merge von **experiment** in **main** (ist dann ein **Fast Forward**)



- **Squash and merge**: Alle Änderungen zu einzigem Commit zusammenfassen (**git merge --squash**)

Universität Bremen

Rebase: Demo



Universität Bremen

Rebase

- Commits, die im anderen Branch fehlen, werden aus **HEAD** entfernt
- Commits aus anderem Branch, die im **HEAD** fehlen, werden eingespielt (**Fast Forward**)
- Entfernte Commits werden wieder eingespielt (Konflikte möglich)
- Dadurch ändert sich die Historie der zeitweilig entfernten Commits!
  - Vorgänger-Commits sind nun andere
  - Konflikte werden direkt in den wieder eingespielten Commits aufgelöst
  - Waren zeitweilig entfernte Commit bereits ge**push**t, ist nur noch **forcierter Push** möglich

Universität Bremen

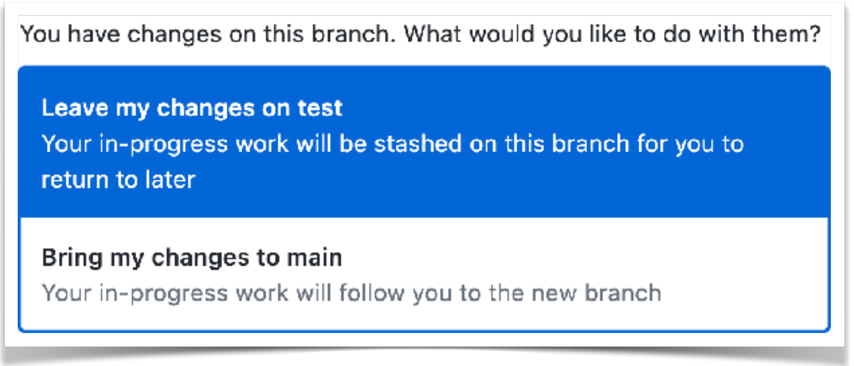
Konfliktbehebung während Rebase

- Wiedereinspielen der Commits kann Konflikte erzeugen (pro Commit!)
  - Müssen wie bei **Merge** aufgelöst werden
- Rebasing ist Modus, in dem sich das Repository befinden kann
  - **Continue Rebase**: Rebase nach Konfliktauflösung mit nächstem Commit fortsetzen
  - **Abort Rebase**: Gesamten Rebase-Vorgang abbrechen und Zustand vor Rebase wieder herstellen
- Nach Abschluss des Rebase sind die ursprünglichen Commits verloren! (nicht ganz: **git reflog**)
  - Zur Sicherheit kann vor dem Rebase weiterer Branch für Originalzustand erzeugt werden

Universität Bremen

Stash

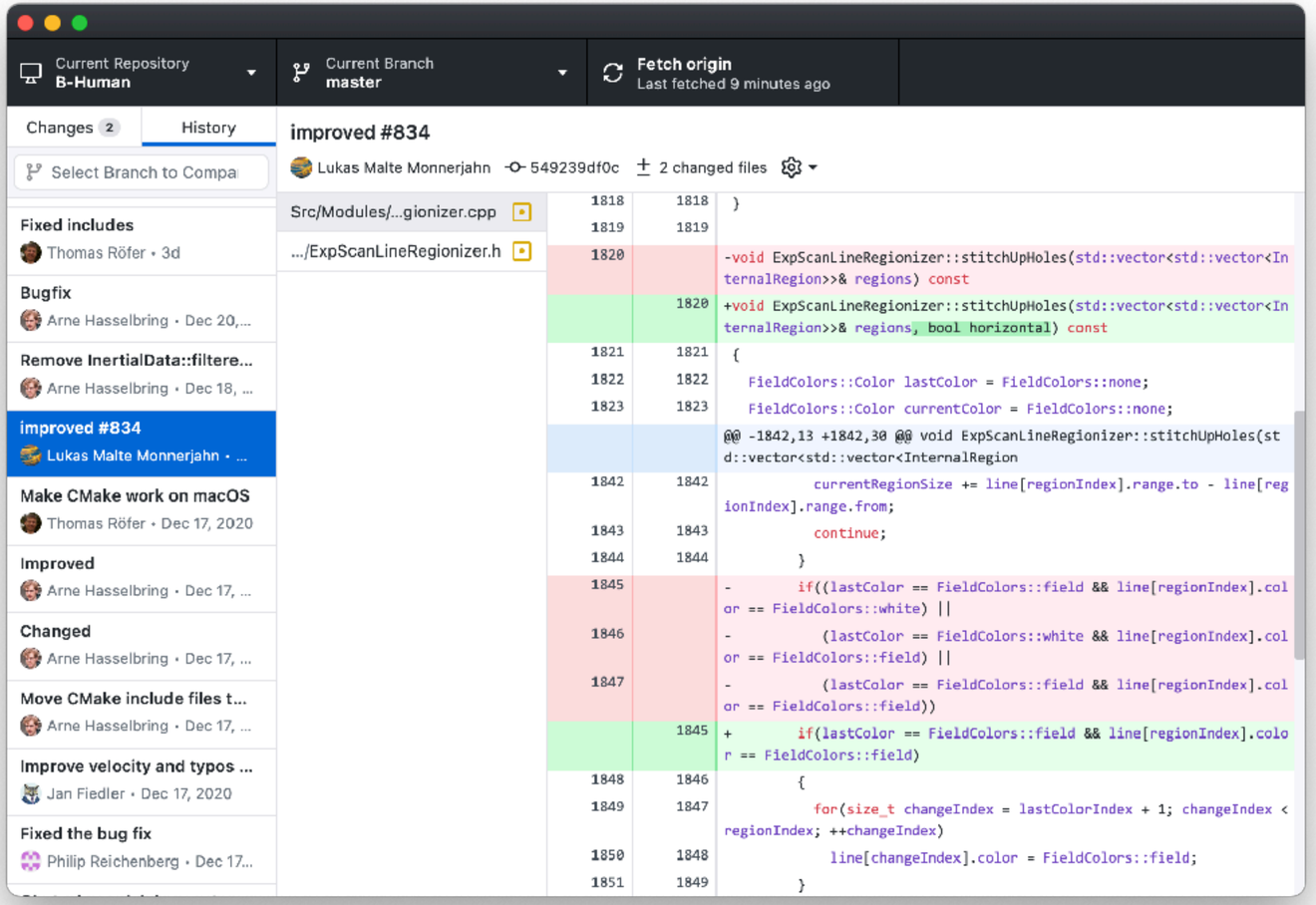
- Viele **git**-Operationen erfordern eine **saubere Arbeitskopie**, d.h. keine Änderungen an Dateien unter Versionskontrolle gegenüber dem Repository
- Der **Stash** erlaubt das Erstellen temporärer Commits (**git stash push**)
- GitHub Desktop setzt **Stash** nur bei bestimmten Operationen ein:
  - **Pull**: Sichert Änderungen auf Nachfrage
  - **Branch-Wechsel**: Fragt, ob Änderungen gesichert oder mitgenommen werden sollen



- **Stash** kann mit **Stashed Changes** wieder hergestellt (**Restore**) oder gelöscht (**Discard**) werden (**git stash pop** / **git stash drop**)

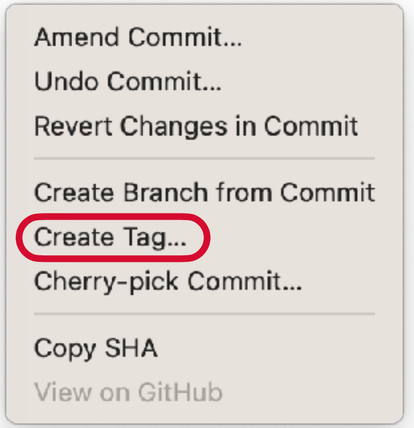
Universität Bremen

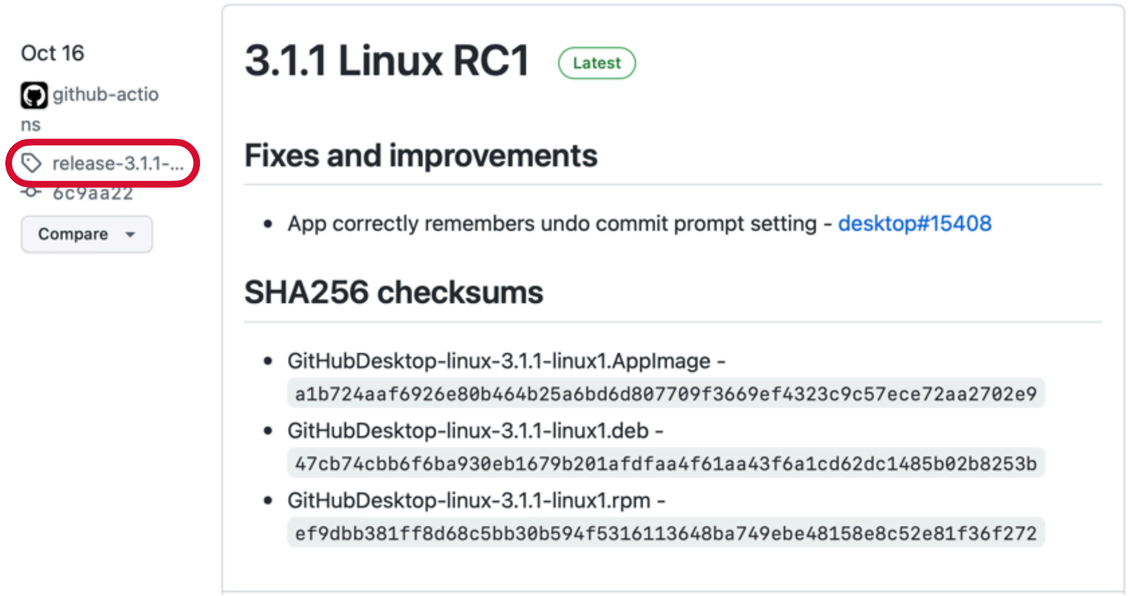
Tags: Demo



Tags

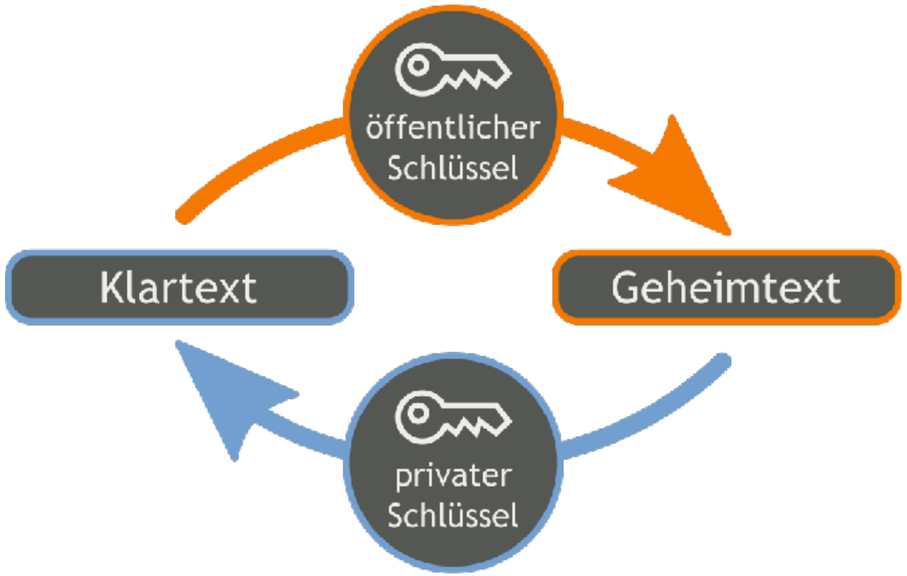
- Tags sind benannte Commits
- Können im Gegensatz zu Branches kein **HEAD** sein
  - Auschecken eines Tags erzeugt einen **Detached Head**
- Werden von Web-Frontends für git für **Releases** verwendet





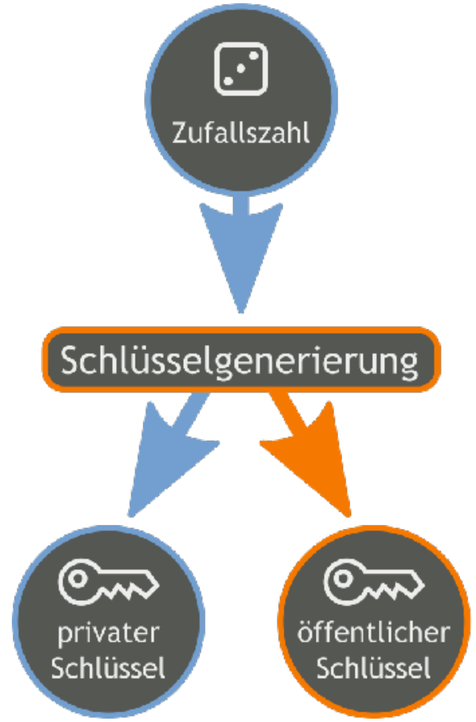
Asymmetrische Verschlüsselung

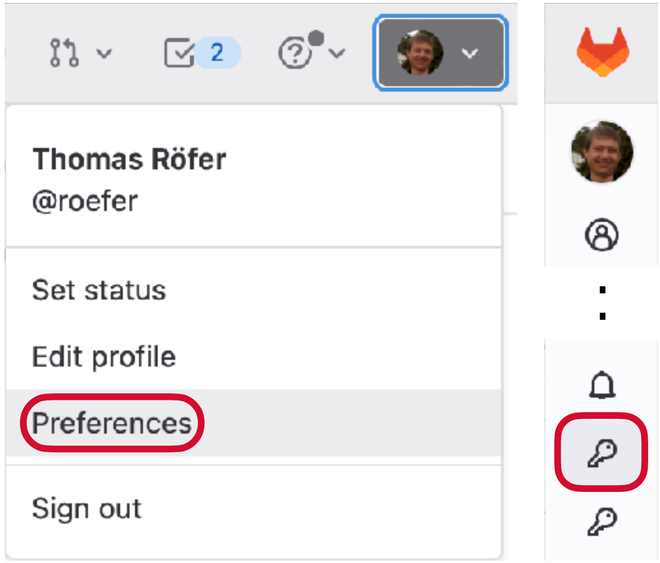
- Zur asymmetrischen Verschlüsselung gehören zwei Schlüssel
  - **Public Key**: Zum Verschlüsseln von Nachrichten (Schlüssel darf jeder kennen)
  - **Private Key**: Zum Entschlüsseln dieser Nachrichten (Schlüssel dürft nur ihr kennen)
- Dient auch zur **Authentifizierung**: Eine Nachricht, die mit einem Public Key verschlüsselt wurde, kann nur die BesitzerIn des Private Keys entschlüsseln



Schlüsselpaar erzeugen

- **ssh-keygen**
  - **Terminal** (Linux/macOS), Schlüssel landen in **~/.ssh** (Versteckter Ordner!)
  - **git-bash** (Windows im Praktikumsbereich), Schlüssel landen in **C:\Users\<account>\.ssh** (**C:\Benutzer\…** im deutschen Explorer)
- Generierte Dateien
  - **id_rsa**: Privater Schlüssel
  - **id_rsa.pub**: Öffentlicher Schlüssel
- Inhalt von **id_rsa.pub** hochladen auf **gitlab.informatik.uni-bremen.de**





Universität Bremen

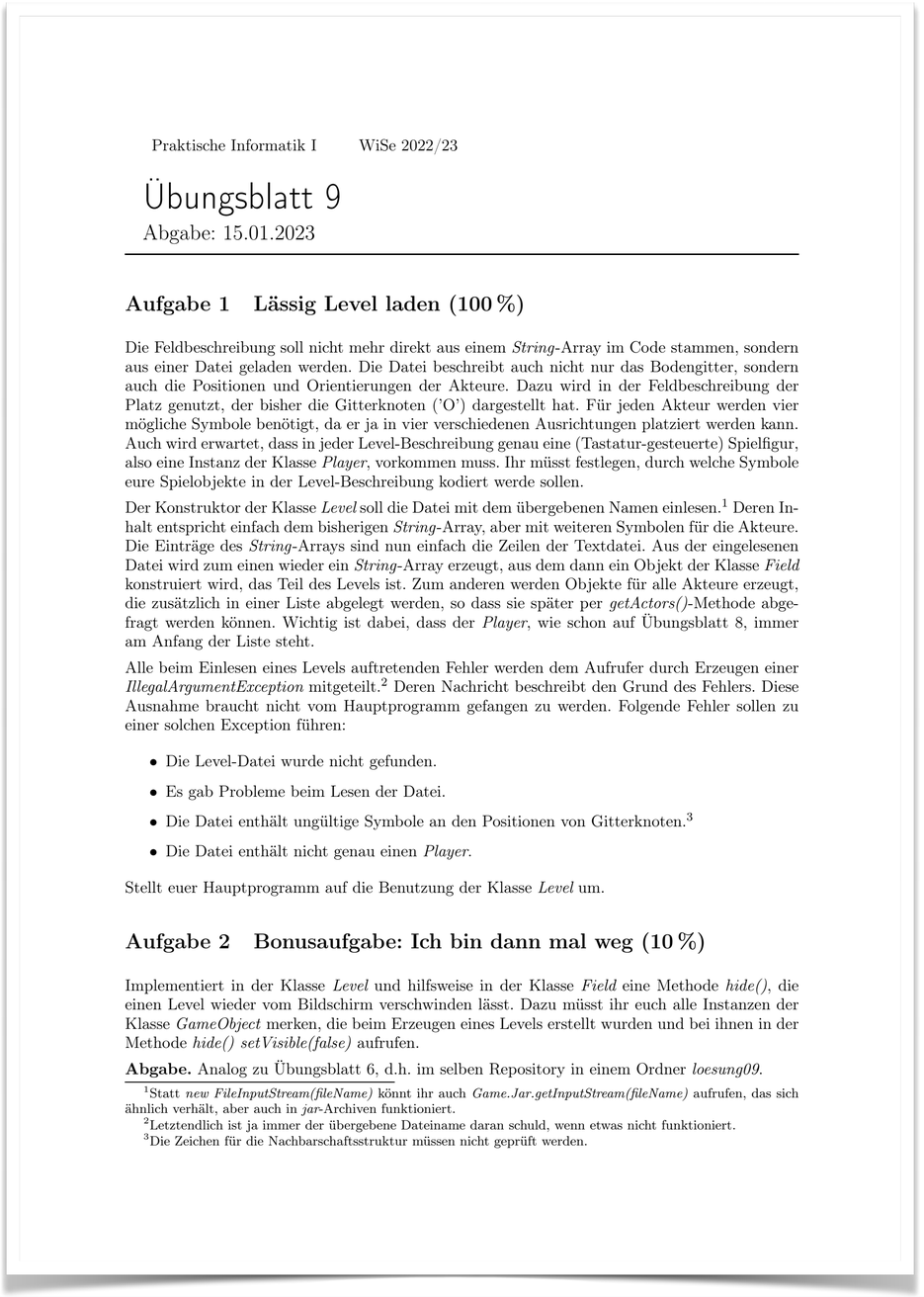
Zusammenfassung der Konzepte

- **SHA-1**
- **Branch**, **Tag** und **Stash**
- **Cherry-Picking**, **Merge** und **Rebase**
- **Asymmetrische Verschlüsselung**

Übungsblatt 9

- Aufgabe 1: Level aus Textdatei einlesen
  - Exceptions erzeugen und behandeln
  - Akteure erzeugen und sammeln
- Bonusaufgabe: Level wieder verschwinden lassen
- Achtung: Abgabe ab jetzt sonntags!

```
O-d-O-G
      |
l-O-O-O
      |
O-O-O-z-O
      |
p-O-O
```



Praktische Informatik I WiSe 2022/23

Übungsblatt 9

Abgabe: 15.01.2023

Aufgabe 1 Lässig Level laden (100 %)

Die Feldbeschreibung soll nicht mehr direkt aus einem *String*-Array im Code stammen, sondern aus einer Datei geladen werden. Die Datei beschreibt auch nicht nur das Bodengitter, sondern auch die Positionen und Orientierungen der Akteure. Dazu wird in der Feldbeschreibung der Platz genutzt, der bisher die Gitterknoten ('O') dargestellt hat. Für jeden Akteur werden vier mögliche Symbole benötigt, da er ja in vier verschiedenen Ausrichtungen platziert werden kann. Auch wird erwartet, dass in jeder Level-Beschreibung genau eine (Tastatur-gesteuerte) Spielfigur, also eine Instanz der Klasse *Player*, vorkommen muss. Ihr müsst festlegen, durch welche Symbole eure Spielobjekte in der Level-Beschreibung kodiert werde sollen.

Der Konstruktor der Klasse *Level* soll die Datei mit dem übergebenen Namen einlesen.[1] Deren Inhalt entspricht einfach dem bisherigen *String*-Array, aber mit weiteren Symbolen für die Akteure. Die Einträge des *String*-Arrays sind nun einfach die Zeilen der Textdatei. Aus der eingelesenen Datei wird zum einen wieder ein *String*-Array erzeugt, aus dem dann ein Objekt der Klasse *Field* konstruiert wird, das Teil des Levels ist. Zum anderen werden Objekte für alle Akteure erzeugt, die zusätzlich in einer Liste abgelegt werden, so dass sie später per *getActors()*-Methode abgefragt werden können. Wichtig ist dabei, dass der *Player*, wie schon auf Übungsblatt 8, immer am Anfang der Liste steht.

Alle beim Einlesen eines Levels auftretenden Fehler werden dem Aufrufer durch Erzeugen einer *IllegalArgumentException* mitgeteilt.[2] Deren Nachricht beschreibt den Grund des Fehlers. Diese Ausnahme braucht nicht vom Hauptprogramm gefangen zu werden. Folgende Fehler sollen zu einer solchen Exception führen:

- Die Level-Datei wurde nicht gefunden.
- Es gab Probleme beim Lesen der Datei.
- Die Datei enthält ungültige Symbole an den Positionen von Gitterknoten.[3]
- Die Datei enthält nicht genau einen *Player*.

Stellt euer Hauptprogramm auf die Benutzung der Klasse *Level* um.

Aufgabe 2 Bonusaufgabe: Ich bin dann mal weg (10 %)

Implementiert in der Klasse *Level* und hilfsweise in der Klasse *Field* eine Methode *hide()*, die einen Level wieder vom Bildschirm verschwinden lässt. Dazu müsst ihr euch alle Instanzen der Klasse *GameObject* merken, die beim Erzeugen eines Levels erstellt wurden und bei ihnen in der Methode *hide() setVisible(false)* aufrufen.

Abgabe. Analog zu Übungsblatt 6, d.h. im selben Repository in einem Ordner *loesung09*.

[1] Statt *new FileInputStream(fileName)* könnt ihr auch *Game.Jar.getInputStream(fileName)* aufrufen, das sich ähnlich verhält, aber auch in *jar*-Archiven funktioniert.
[2] Letztendlich ist ja immer der übergebene Dateiname daran schuld, wenn etwas nicht funktioniert.
[3] Die Zeichen für die Nachbarschaftsstruktur müssen nicht geprüft werden.