# Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

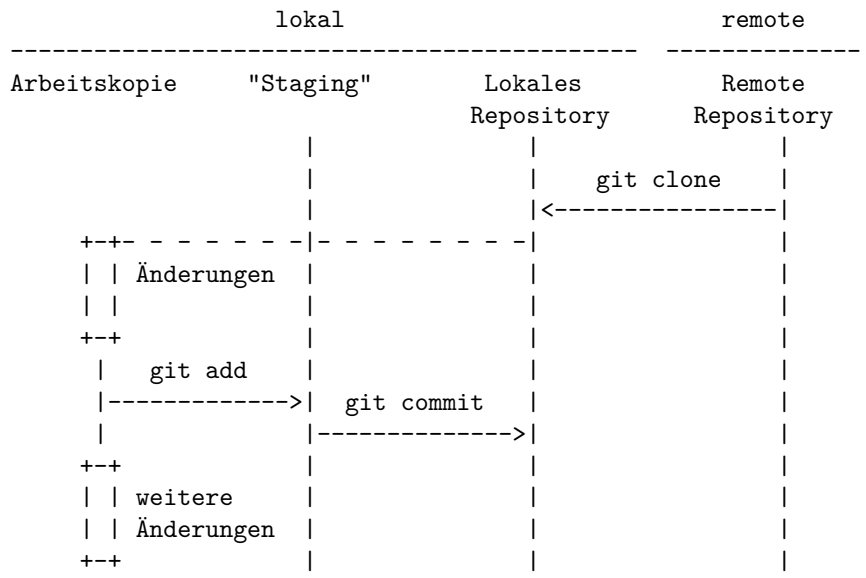
## Versionsverwaltung II

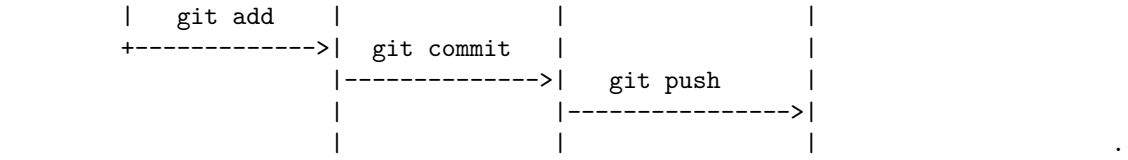
| Parameter | Kursinformationen |
|---|---|
| **Veranstaltung:** | `Vorlesung Softwareentwicklung` |
| **Semester** | `Sommersemester 2021` |
| **Hochschule:** | `Technische Universität Freiberg` |
| **Inhalte:** | `Versionsverwaltung mit Git und GitHub` |
| **Link auf den GitHub:** | https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/12_VersionsverwaltungII.md |
| **Autoren** | @author |

---

### Verteiltes Versionsmanagement

*Einfaches Editieren*: Sie klonen das gesamte Repository, dass sich auf dem "Server-Rechner" befindet. Damit haben Sie eine vollständige Kopie aller Versionen in Ihrem Working Directory. Wenn wir annehmen, dass keine branches im Repository bestehen, dann können Sie direkt auf der Ihrer Arbeitskopie arbeiten und diese verändern. Danach generieren Sie einen Snappshot des Arbeitsstandes *Staging*. Ihre Version ist als relevant markiert und kann im lokalen Repository als neuer Eintrag abgelegt werden. Vielleicht wollen sie Ihren Algorithmus noch weiterentwickeln und speichern zu einem späteren Zeitpunk eine weitere Version. All diese Vorgänge betreffen aber zunächst nur Ihre Kopie, ein anderer Mitstreiter in diesem Repository kann darauf erst zurückgreifen, wenn Sie das Ganze an den Server übermittelt haben.






*Kooperatives Arbeiten:* Nehmen wir nun an, dass Ihr Kollege in dieser Zeit selbst das Remote Repository fortgeschrieben hat. In diesem Fall bekommen Sie bei Ihrem `push` eine Fehlermeldung, die sie auf die neuere Version hinweist. Nun "ziehen" Sie sich den aktuellen Stand in Ihr Repository und kombinieren die Änderungen. Sofern keine Konflikte entstehen, wird daraus ein neuer Commit generiert, den Sie dann mit Ihren Anpassungen an das Remote-Repository senden.


Versuchen wir das ganze noch mal etwas plastischer nachzuvollziehen. Rechts oben sehen Sie unser Remote-Repository auf dem Server. Im mittleren Bereich den Status unseres lokalen Repositories.

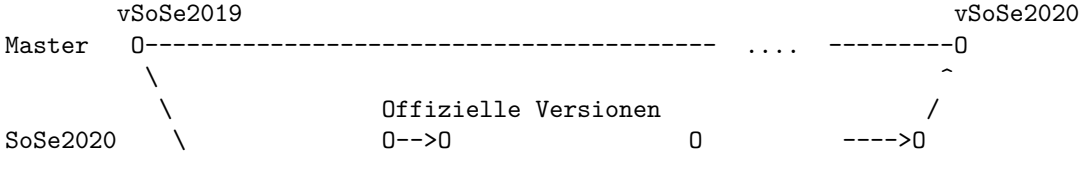
`text @ExplainGit.eval create origin`

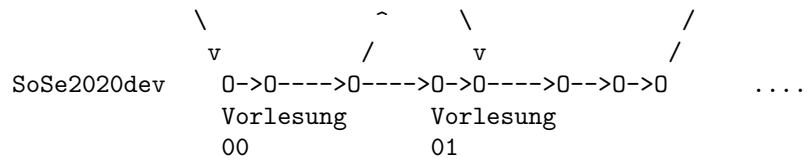
## Arbeiten mit Branches

Die Organisation von Versionen in unterschiedlichen Branches ist ein zentrales Element der Arbeit mit git. Branches sind Verzweigungen des Codes, die bestimmte Entwicklungsziele kapseln.

Der größte Nachteil bei der Arbeit mit nur einem Branch liegt darin, dass bei einem defekten Master(-Branch) die Arbeit sämtlicher Beteiligter unterbrochen wird. Branches schaffen einen eignen (temporären) Raum für die Entwicklung neuer Features, ohne die Stabilität des Gesamtsystems zu gefährden. Gleichzeitig haben die Entwickler den gesamten Verlauf eines Projekts in strukturierter Art zur Hand.

Wie sieht das zum Beispiel für unsere Kursmaterialien aus?






Ein Branch in Git ist einfach ein Zeiger auf einen Commit zeigt. Der zentrale Branch wird zumeist als `master` bezeichnet.

### Generieren und Navigation über Branches

Wie navigieren wir nun konkret über den verschiedenen Entwicklungszweigen.

```
text @ExplainGit.eval git commit -m V1 git commit -m V2 git commit -m V3
```

### Mergoperationen über Branches

Nehmen wir folgendes Scenario an. Sie arbeiten an einem Issue, dafür haben Sie einen separaten Branch (newFeature) erzeugt und haben bereits einige Commits realisiert. Beim Kaffeetrinken denken Sie über den Code von letzter Woche nach und Ihnen fällt ein Bug ein, den Sie noch nicht behoben haben. Jetzt aber schnell!

Legen Sie dafür einen neuen Branch an, commiten Sie eine Version und mergen Sie diese mit dem Master. Kehren Sie dann in den Feature-Branch zurück und beenden Sie die Arbeit. Mergen Sie auch diesen Branch mit dem Master. Worin unterscheiden sich beide Vorgänge?

```
text @ExplainGit.eval git branch newFeature git checkout newFeature git commit -m FeatureV1
git commit -m FeatureV2
```

Mergen ist eine nicht-destruktive Operation. Die bestehenden Branches werden auf keine Weise verändert. Das Ganze “bläht” aber den Entwicklungsbaum auf.

### Rebase mit einem Branch

Zum `merge` existiert auch noch eine alternative Operation. Mit `rebase` werden die Änderungen eins branches in einem Patch zusammengefasst. Dieser wird dann auf head angewandt.

```
text @ExplainGit.eval git branch newFeature git checkout newFeature git commit -m FeatureV1
git commit -m FeatureV2 git checkout master git commit -m V1
```

## Arbeit mit GitHub

### Issues

Issues dienen dem Sammeln von Benutzer-Feedback, dem Melden von Software-Bugs und der Strukturierung von Aufgaben, die Sie erledigen möchten. Dabei sind Issues umittelbar mit Versionen verknüpft, diese können dem Issue zugeordnet werden.

Sie können eine Pull-Anfrage mit einer Ausgabe verknüpfen, um zu zeigen, dass eine Korrektur in Arbeit ist und um die Ausgabe automatisch zu schließen, wenn jemand die Pull-Anfrage zusammenführt.

Um über die neuesten Kommentare in einer Ausgabe auf dem Laufenden zu bleiben, können Sie eine Ausgabe beobachten, um Benachrichtigungen über die neuesten Kommentare zu erhalten.

https://guides.github.com/features/issues/

### Pull requests und Reviews

Natürlich wollen wir nicht, dass “jeder” Änderungen ungesehen in unseren Code einspeist. Entsprechend kapseln wir ja auch unseren Master-Branch. Den Zugriff regeln sowohl die Rechte der einzelnen Mitstreiter als auch die Pull-Request und Review Konfiguration.

| Status des Beitragenden | Einreichen des Codes |
|---|---|
| Collaborator | Auschecken des aktuellen Repositories und Arbeit. Wenn die Arbeit in einem eigenen Branch erfolgt, wird diese mit einem Pull-Request angezeigt und gemerged. |

| Status des Beitragenden | Einreichen des Codes |
|---|---|
| non Collaborator | Keine Möglichkeit seine Änderungen unmittelbar ins Repository zu schreiben. Der Entwickler erzeugtn eine eigene Remote Kopie (*Fork*) des Repositories und dortige Realsierung der Änderungen. Danach werden diese als Pull-Request eingereicht. |

Wird ein Pull Request akzeptiert, so spricht man von einem *Merge*, wird er geschlossen, so spricht man von einem *Close*. Vor dem Merge sollte eine gründliche Prüfung sichergestellt werden, diese kann in Teilen automatisch erfolgen oder durch Reviews Doku

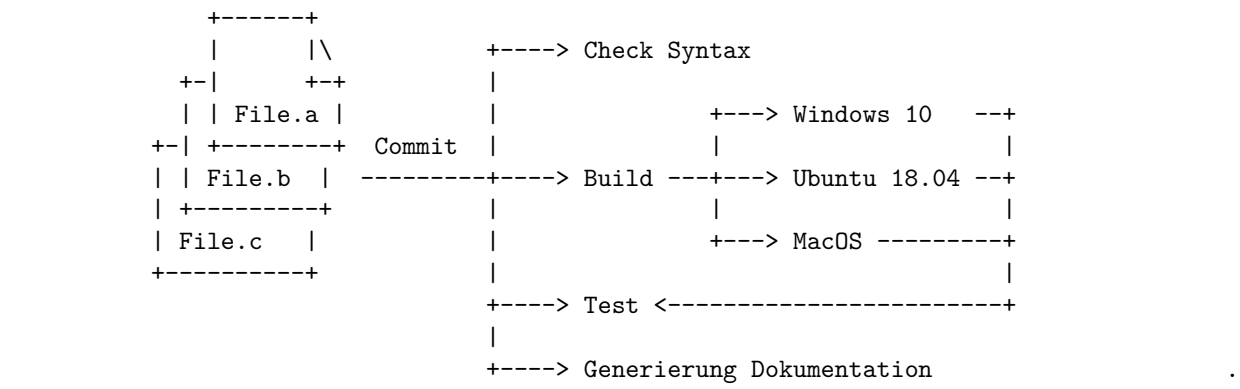
### Automatisierung der Arbeit

Hervorragend! Wir sind nun in der Lage die Entwicklung unseres Codes zu "verwalten". Allerdings sagt noch niemand, dass ein eingereichter Code auch lauffähig ist. Wie können wir aber möglichst schnell realisieren, dass etwas schief geht? Es wäre wünschenswert, dass wir unmittelbar mit den Aktivitäten unserer Entwickler entsprechende Tests durchführen und zum Beispiel deren Commits zurückweisen.

An dieser Stelle wollen wir zunächst die Möglichkeiten des *Continuous Integration* aufzeigen, die differenzierte Diskussion einer Folge von Build und Test-Schritten folgt später. Wir werden diese Möglichkeit im Rahmen der Übungsaufgaben nutzen, um Ihre Lösungen zu testen.

GitHub stellt dafür die sogenannten *Actions* zur Verfügung. Dies sind Verarbeitungsketten, die auf verschiedensten Architekturen, Betriebssystemen, Konfigurationen usw. laufen können. Damit haben wir die Möglichkeit einen Quellcode für verschiedene Plattformen zu bauen und zu testen oder eine Dokumentation zu erstellen.

Ein *Workflow* wird durch vordefinierten *Trigger* ausgelöst. Dies können das Anlegen einer Datei, ein Commit oder ein Pull Request sein. Danach wird das System konfiguriert und die Folge der Verarbeitungsschritte gestartet.




GitHub gliedert diese Punkte in *Workflows* und *Jobs* in einer hierachischen Struktur, die über `yaml` Files beschrieben werden. Eine kurze Einführung zur Syntax findet sich unter Wikipedia.

```yaml
name: Hello World
on: [push]

jobs:
  build-and-run:
    name: Print Hello World
    runs-on: ubuntu-latest
    steps:
      - name: Checkout files (master branch)
        uses: actions/checkout@v2
      - name: Show all files
        run: pwd && whoami && ls -all
```

Spanndend wird die Sache nun dadurch, dass es eine breite Community rund um die Actions gibt. Diese stellen häufig benötigte *Steps* bereits zur Verfügung, fertige Tools für das Bauen und Testen von .NET Code.

Die Dokumentation zu den GitHub-Actions findet sich unter https://github.com/features/actions. Ein umfangreicheres Beispiel finden Sie in unserem Projektordner im aktuellen Branch `SoSe2020`. Hier werden alle

LiaScript-Dateien in ein pdf umgewandelt.

> **Merke** Workflow files müssen unter `.github\workflows\*.yml` abgelegt werden.

## Ein Wort zur Zusammenarbeit

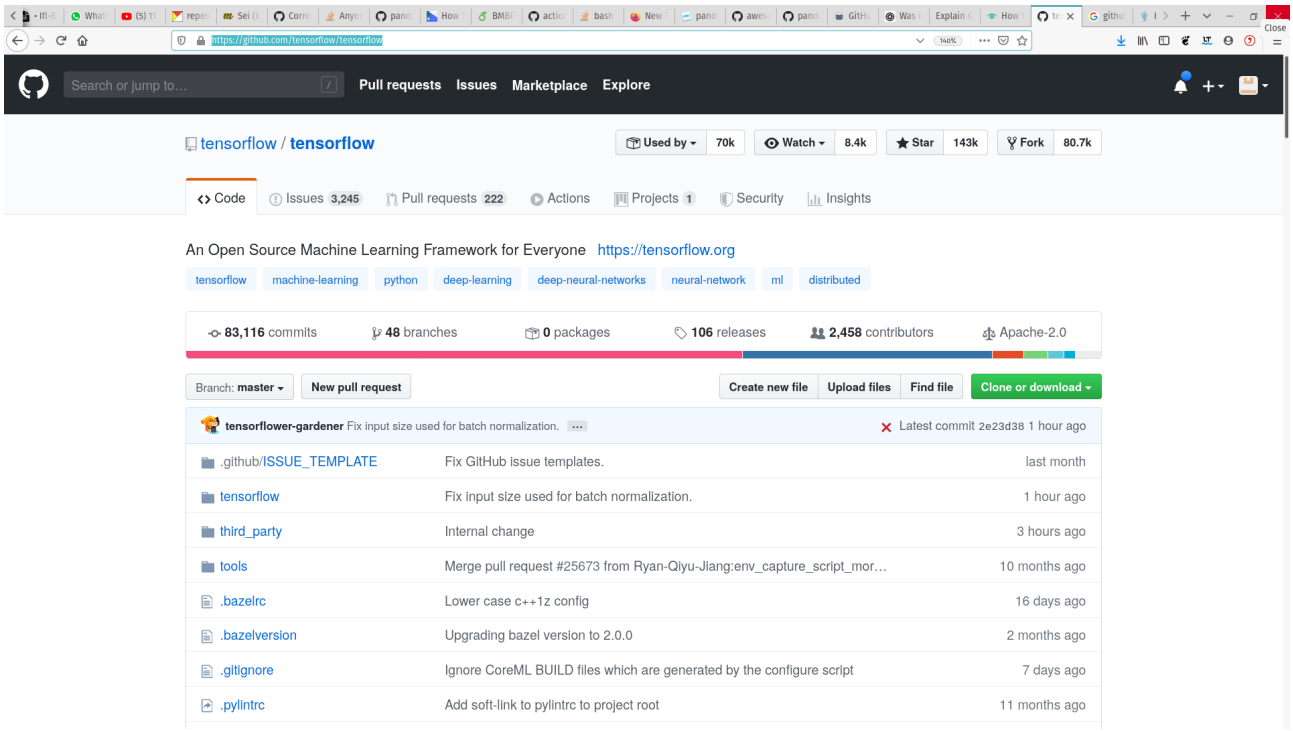
Bitte haben Sie immer den spezifischen Kontext Ihres Projektes vor Augen. Üblicherweise legt man am Anfang, bei einem "kleinen Hack" keinen Wert auf formelle Abläufe und Stukturen. Diese sind aber in großen Projekten unablässig.

Ein neues Feature wird in einem Issue beschrieben, in einem eigenen Branch implementiert, mit Commits beschrieben, auf den master branch abgebildet und das Issue mit Referenz auf den commit geschlossen.

Entsprechend ist die Dokumentation in Form der Issues und Commit-Messages der zentrale Schlüssel für die Interaktion im Softwareentwicklerteam. Entsprechend hoch ist Ihre Bedeutung anzusetzen.

### Commit Messages

Stöbern Sie dafür mal durch anderen Projekte (zum Beispiel GitHub Tensorflow) und informieren Sie sich über deren Policies.




```
Kurze (72 Zeichen oder weniger) Zusammenfassung

Ausführlicherer erklärender Text. Umfassen Sie ihn auf 72 Zeichen. Die Leerzeile
Zeile, die die Zusammenfassung vom Textkörper trennt, ist entscheidend (es sei
denn, Sie lassen den den Textkörper ganz weg).

Schreiben Sie Ihre Commit-Nachricht im Imperativ: "Fix bug" und nicht "Fixed
Fehler" oder "Behebt Fehler". Diese Konvention stimmt mit den Commit-Nachrichten
überein die von Befehlen wie "git merge" und "git revert" erzeugt werden.

Weitere Absätze kommen nach Leerzeilen.

- Aufzählungspunkte sind für eine Liste von Anpassungen in Ordnung.
- ... und noch einer
```

Folgende Regeln sollte man für die Beschreibung eines Commits berücksichtigen:

- Trennen Sie den Betreff durch eine Leerzeile vom folgenden Text
- Beschränkt Sie sich bei der Betreffzeile auf maximal 50 Zeichen
- Beginnen Sie die Betreffzeile mit einem Großbuchstaben

- Schreiben Sie die Betreffzeile im Imperativ
- Brechen Sie den Text der Message 72 Zeichen um

**Merke:** Beschreiben Sie in der Commit-Nachricht das was und warum, aber nicht das wie.

Das *Semantic Versioning* geht einen Schritt weiter und gibt den Commit-Messages eine feste Satzstruktur, vgl. entwickler.de

### Generelles Vorgehen

Lassen Sie uns einen Blick auf das Aufgabenblatt der kommenden Woche werfen. Ihre Aufgabe besteht darin, in einem zweier Team verschiedene Rollen einzunehmen.

```
@startuml
actor Maintainer
actor Developer
== Vorbereitung ==
Maintainer --> Maintainer: Einfügen der Rolleninformation\n und des Fragebogenschlüssels\n in [[https:/
Developer --> Developer: Einfügen der Rolleninformation\n und des Fragebogenschlüssels\n in [[https://g
== Projekt Initialisierung ==
Maintainer --> Maintainer: Konfiguration [[https://docs.github.com/en/organizations/organizing-members-
Maintainer --> Maintainer: Anlegen [[https://guides.github.com/features/issues/{Mastering Issues} Issue
Maintainer --> Developer:  Zuweisung Issue
== Implementierung ==
activate Maintainer
Maintainer --> Maintainer:  Monitoring Projekt
Developer --> Maintainer:  //Issue in progress//
activate Developer
Developer --> Developer:  Anlegen eines Branches
Developer -> Developer:  Implementieren 1
note right
 * Anlegen neue Datei
 * Kopieren der txt Inhalte
 * Formatieren als md
 * Einbau einiger Typos, die
    der Maintainer finden soll
end note
Developer --> Developer:  Commiten
Developer -> Developer:  Implementieren 2
note right
 * Ergänzen eines Hello-World Codebeispiels
end note
Developer -> Developer:  Commiten
== Review ==
Developer --> Developer:   Starte  [[https://docs.github.com/en/github/collaborating-with-issues-and-pu
Developer --> Maintainer : Anforderung [[https://docs.github.com/en/github/collaborating-with-issues-an
Maintainer --> Maintainer:  Code Review / Kommentare
Maintainer --> Developer :  Anforderung Nachbesserungen
note left
Bitte um zusätzliche
 * Bildunterschrift
 * Rechtschreibkorrektur
end note
Developer -> Developer:  Implementieren 3
note right
 * Korrektur der Fehler
 * Einfügen einer Beschreibung der Grafik
end note
Developer --> Maintainer :  Bitte um erneutes Rereview
Maintainer --> Developer :  Review abgeschlossen
Maintainer --> Maintainer:  Abschluss Pull request
deactivate Developer
```

```
== Deploy ==
Maintainer --> Maintainer:  Abschluss des Pullrequests
Maintainer --> Maintainer:  Generierung Release
deactivate Maintainer
@enduml
```

> **Wichtig:** Schließen Sie Ihre Arbeit mit einem Release ab! Damit ist für uns erkennbar, dass Sie die Aufgabe erfolgreich umgesetzt haben.

## Aufgaben

!?alt-text

- [ ] Recherchieren Sie die Methode des "Myers-diff-Algorithmus" https://blog.jcoglan.com/2017/02/12/the-myers-diff-algorithm-part-1/
- [ ] Legen Sie sich ein lokales Repository mit git an und experimentieren Sie damit.
- [ ] Richten Sie sich für Ihren GitHub-Account einen SSH basierten Zugriff ein (erspart einem das fortwährende Eingeben eines Passwortes).
- [ ] Sie erhalten im Laufe der Woche Ihre erste Einladung für einen GitHub Classroom. Ausgehend davon werden Sie aufgefordert sich in Zweierteams zu organisieren und werden dann gemeinsam erste Gehversuche unter git zu unternehmen.