Moderne Softwareentwicklung

📘 Java-Documentation ansehen

📄 Aktuelle README als PDF herunterladen

vorgelegt von:

1. Matthias Lindner (matthias.lindner@th-brandenburg.de)
2. Fenna Haan (fenna.haan@stud.hs-emden-leer.de)
3. Janne Surborg (ja.surborg@ostfalia.de)
4. Constantin Moye (wgtz1919@bht-berlin.de)
5. Cornelia Demes (cornelia.demes@stud.hs-emden-leer.de)

vorgelegt am: 13.10.2025

Inhaltsverzeichnis

Aufgabe 1 - Git

1.1 Was ist ein Git und warum sollte es verwendet werden?

Git ist ein Versionskontrollsystem, das Änderungen an Dateien verfolgt und speichert. Es erstellt regelmäßig Snapshots von Dateien, sodass man jederzeit den Zustand des Projekts wiederherstellen kann.

Git kann komplette Verzeichnisbäume verwalten und ermöglicht es lokal zu arbeiten, auch ohne Internet. Git ist ein verteiltes System, das mutiges Programmieren fördert. Durch die Arbeit in Branches können neue Funktionen getestet und Änderungen später problemlos übernommen oder verworfen werden.

Durch einen Commit werden Änderungen im lokalen Arbeitsverzeichnis gespeichert, mit einer Nachricht versehen und optional auf den Server übertragen. So bleiben alle Änderungen und Anmerkungen nachvollziehbar. Zur Sicherheit und Integrität berechnet Git für jeden Commit einen SHA-1-Hash. Dadurch kann Git erkennen, wenn ein Commit verändert wurde und jeder Commit lässt sich eindeutig identifizieren.

GitHub ist ein Webdienst, der Git-Repositories (=Versionsverwaltungsspeicher) hostet und die Zusammenarbeit erleichtert. Viele Plattformen wie GitHub oder GitLab bieten kostenlosen Speicherplatz und ermöglichen es, Projekte direkt zu teilen. Viele Unternehmen, u.a. Facebook, LinkedIn und Microsoft, nutzen GitHub und Entwickler:innen können dort Reputation durch Commits aufbauen.

Git und GitHub dienen somit der effizienten Versionskontrolle, Zusammenarbeit, Nachverfolgbarkeit von Änderungen und fördern gleichzeitig eine sichere, mutige und strukturierte Entwicklungsweise.

1.2. Grundlegende Git-Befehle

Übersicht der grundlegenden Git-Befehle

Befehl	Beschreibung
`git init`	Erstellt ein neues lokales Git-Repository im aktuellen Ordner.
`git clone <repository-url>`	Klont ein bestehendes Remote-Repository (z. B. von GitHub) auf den lokalen Rechner.
`git status`	Zeigt den aktuellen Status der Arbeitskopie (z. B. geänderte, neue oder unversionierte Dateien).
`git add <datei>`	Fügt eine bestimmte Datei zur Staging-Area hinzu.
`git add .`	Fügt alle Änderungen im aktuellen Verzeichnis zur Staging-Area hinzu.
`git commit -m "Nachricht"`	Speichert alle Änderungen aus der Staging-Area dauerhaft im lokalen Repository.
`git log`	Zeigt die Commit-Historie (Zeit, Autor, Nachricht, Hash).
`git diff`	Zeigt Unterschiede zwischen Arbeitsverzeichnis, Staging-Area und Repository.
`git branch`	Listet alle lokalen Branches auf.
`git branch <name>`	Erstellt einen neuen Branch mit dem angegebenen Namen.
`git switch <name>`	Wechselt zu einem bestehenden Branch (neuerer, benutzerfreundlicher Befehl).
`git switch -c <name>`	Erstellt und wechselt gleichzeitig zu einem neuen Branch.
`git checkout <name>`	Wechselt zu einem Branch oder Commit (älterer, aber weit verbreiteter Befehl).
`git checkout -b <name>`	Erstellt und wechselt gleichzeitig zu einem neuen Branch (ältere Variante von `git switch -c`).
`git merge <branch>`	Führt die Änderungen eines Branches in den aktuellen Branch zusammen.
`git remote add origin <url>`	Verknüpft das lokale Repository mit einem Remote-Repository (z. B. GitHub).

`git push -u origin <branch>`	Lädt lokale Commits auf das Remote-Repository hoch.
`git pull`	Holt Änderungen vom Remote-Repository und integriert sie lokal.

Häufige Git-Workflows

Neues Repository erstellen

```
cd /VERZEICHNIS/IN/DEM/GIT/INITIALSIERT/WERDEN/SOLL
git init
# Alle Files im Ordner
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin https://github.com/<user>/<repo>.git
git push -u origin main
```

Repository klonen und Änderungen holen

```
git clone https://github.com/<user>/<repo>.git
cd <repo>
git pull
```

Änderungen hinzufügen und hochladen

```
git status
git add file.txt README.md
git commit -m "Neue Funktion hinzugefügt"
git push
```

Branch erstellen, pushen und Pull Request auf GitHub anlegen

```
# Neuen Branch lokal erstellen und in diesen wechseln
git switch -c feature/neue-funktion
# oder ältere Variante:
git checkout -b feature/neue-funktion

# Änderungen durchführen und committen
git add .
git commit -m "Implementiert neue Funktion"

# Branch erstmals zum Remote-Repository hochladen -> -u setzt die Upstream
Verknüpfung zwischen lokalem und Remote-Branch, danach kann man immer git push
nehmen
git push -u origin feature/neue-funktion
```

Pull Request in der GitHub-Oberfläche erstellen

1. Öffne dein Repository auf **GitHub**.
2. GitHub erkennt automatisch, dass ein neuer Branch (`feature/neue-funktion`) gepusht wurde, und zeigt oben den Hinweis
   "Compare & pull request" an.
3. Klicke darauf oder unter Pull requests -> New Pull request
4. Überprüfe:
   - **Base branch**: `main`
   - **Compare branch**: `feature/neue-funktion`
5. Gib einen **Titel** und eine **Beschreibung** deines Pull Requests ein (z. B. welche Änderungen du gemacht hast).
6. Klicke auf **"Create pull request"**.

Änderungen aus main in den aktuellen Feature-Branch übernehmen

```
# Stelle sicher, dass du dich im Feature-Branch befindest
git switch feature/name
# oder (ältere Variante)
git checkout feature/name

# Lade die neuesten Änderungen vom Remote-Repository
git fetch origin

# Führe die Änderungen aus main in deinen Feature-Branch zusammen
git merge origin/main

# Alternativ kannst du deine Änderungen auch mithilfe von Rebase auf den neuesten
Stand bringen:
git rebase origin/main
```

Merge vs. Rebase (Ergebnis & Historie):

- **Merge**: Bewahrt die verzweigte Historie und erzeugt einen **Merge-Commit**.
- **Rebase**: Schreibt die Feature-Commits neu (neue Commit-Hashes) und ergibt eine **lineare** Historie ohne Merge-Commit.
  Inhaltlich endest du in beiden Fällen bei denselben Dateien; nur der Verlauf unterscheidet sich.

1.3. Nützliche Plattformen und Tools

Es existieren hilfreiche Plattformen und Tools, um die Arbeit mit git angenehmer zu gestalten und individueller zu gestalten. Dazu zählen u.a. Graphical Tools, PlugIns für IDEs, Unix-Shells für die Kommandozeile

PaaS-Dienste (Platform as a Service)

Diese Plattformen bieten oft kostenlosen Speicherplatz und sind sofort nach der Registrierung nutzbar. Dort werden die Inhalte gehostet, gemeinsam bearbeitet und versioniert. Einige Beispiele:

- **Github** (viele Unternehmen setzen auf GitHub, z.B. PostgreSQ, Android, Mozilla, LinkedIn); Facebook für Programmierer, zusammenarbeiten, verfolgen, bewerten und Beiträge leisten; Selbsthosting-Möglichkeit nur gegen Aufpreis
- **GitLab** (kostenlose Selbsthosting-Möglichkeit, kleinere Community)

- **bitbucket** (bietet Git und Mercurial als Protokoll)

Tools

Es existieren zahlreiche Tools für die Arbeiten mit dem Git-System. Einige Beispiele:

- **Git Bash** (UnixShell zum Arbeiten mit der Kommandozeile)
- **TortoiseGit** (kostenloser Git-Client für Windows, Integration in den Datei-Explorer, Git-Befehle per Rechtsklick auf einen Ordner ausführbar, ohne Terminal bzw. Kommandozeile nutzbar; ideal für Einsteiger)
- **GitDesktop** (kostenloses Programm für Windows und macOS, mit dem Git und GitHub über eine *grafische* Oberfläche genutzt werden kann, ohne Terminal bzw. Kommandozeile nutzbar, ideal für Einsteiger)

Erweiterungen für IDEs

- VS Code Git Graph: Visualisiert Branches und Commits
- IntelliJ Vollständige Git-Integration in JetBrainsIDEs

Nachschlagewerke

Hilfreich bei jeglichen Fragen sind Nachschlagewerke, Dokumentationen zu git an sich oder den jeweiligen PaaS oder Tools. Einige Beispiele:

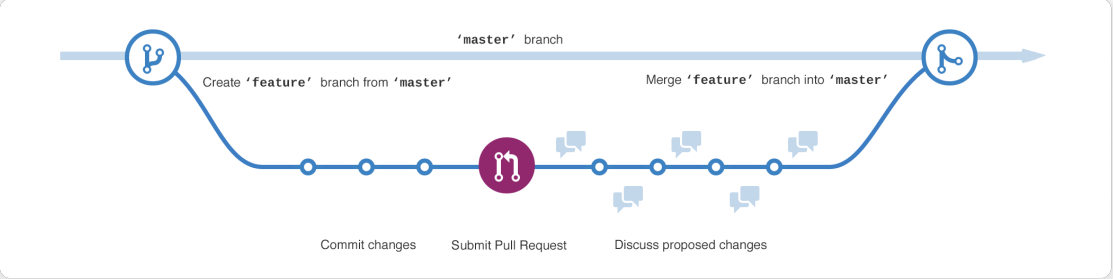
- **git** Kommandozeile: $ git help <'Befehl'>
- **gitHub**: Dokumentation https://docs.github.com/
- **gitLab**: Dokumentation https://docs.gitlab.com/

1.4. Branches und ihre Nutzung, Umgang mit Merge-Konflikten

- Zweck eines Branches:
    - Ein Branch ist ein separater Entwicklungszweig, der paralleles Arbeiten ermöglicht, ohne den main-Branch zu verändern
    - Dadurch können mehrere Personen gleichzeitig an verschiedenen Features oder Bugfixes arbeiten, ohne sich gegenseitig zu stören, bevor diese dann in den main-Branch integriert werden
- Wichtige Befehle für Branches:
    - git branch (Zeigt an, in welchem Branch man sich derzeit befindet)
    - git branch name-des-branches (Erstellt einen neuen Branch)
    - git checkout name-des-branches (Wechseln in einen anderen Branch)
    - git branch -d name-des-branches (Einen Branch löschen)
- Zurückführen des Branches in den main-Branch (PullRequest):
    - Mit einem Pull Request können Änderungen aus einem Branch in den main-Branch übernommen werden
    - Ein Pull Request dient zur Überprüfung und Freigabe des Codes durch andere Teammitglieder
    - Der Pull Request kann in der Weboberfläche (z. B. GitHub) erstellt werden: Dort kann eine Beschreibung hinzugefügt und ein Reviewer ausgewählt werden
    - Nach Freigabe kann der Branch in den main-Branch gemergt werden
- Wichtige Befehle:

    - git push origin name-des-branches (Branch auf Remote-Repository hochladen, um Pull Request zu erstellen)

- Mit Merge-Konflikten umgehen:
    - Ein Merge-Konflikt entsteht, wenn Git Änderungen aus verschiedenen Branches nicht automatisch zusammenführen kann, z.B. wenn dieselbe Datei in den Branches unterschiedlich geändert wurde
    - Git zeigt die betroffenen Dateien und markiert Konfliktstellen, diese können manuell z. B. direkt in der IDE gelöst werden
    - Nach der Anpassung wird die Datei mit git add markiert und der Merge mit git commit abgeschlossen
- Wichtige Befehle:
    - git merge name-des-branches (Führt die Änderungen aus dem angegebenen Branch in den aktuellen Branch ein)
    - git status (Zeigt, welche Dateien Konflikte enthalten oder gelöst wurden)



1.5. Git mit IntelliJ/PyCharm benutzen: Local Repository und Remote Repository

- REMOTE repository liegt in GitHub (cloud computing)
- LOCAL repository liegt lokal auf dem eigenen Computer
- für den Zugriff auf ein REMOTE (GitHub) repository kann man eine lokale client software, z.B. GitHub Desktop verwenden (muss man nicht). Diese verbindet sich dann mit dem REMOTE repository (über das Internet).
- für den Zugriff auf LOCAL repositories kann man eine lokale client software, z.B. GitHub Desktop verwenden (muss man nicht)
- IDE's bzw. Editoren, die mit Git / GitHub kompatibel sind, sind z.B. IntelliJ IDEA Community Edition (von JetBrains s.r.o.), PyCharm (von JetBrains s.r.o.)
- benötigte Plugins sind: Git und GitHub
- JetBrains IntelliJ IDEA Community Edition: https://www.jetbrains.com/help/idea/github.html
- JetBrains PyCharm: https://www.jetbrains.com/help/pycharm/github.html

---

2. Aufgabe 2 - CI/CD-Pipeline

2.1. Definition

- Eine CI/CD-Pipeline ist ein automatisierter Prozess, der die Erstellung, das Testen und die Bereitstellung von Software optimiert. Ziel ist es, Änderungen schnell, zuverlässig und reproduzierbar in produktionsreife Versionen zu überführen

- CI (Continuous Integration) bedeutet, dass Entwickler ihre Codeänderungen regelmäßig in gemeinsame Branches einpflegen. Jede Änderung wird dabei automatisch gebaut und getestet, um Integrationsprobleme früh zu erkennen und sicherzustellen, dass der Code stets in einem stabilen und lauffähigen Zustand bleibt
- CD (Continuous Delivery) baut auf CI auf und stellt sicher, dass die getesteten und integrierten Änderungen automatisch für den Rollout vorbereitet werden. Dadurch ist die Software jederzeit auslieferbar, da alle notwendigen Konfigurationen für eine Bereitstellung in beliebige Umgebungen vorhanden sind
- die 5 zentralen Komponenten in GitHub Actions:

1. **Events** lösen einen Workflow aus.
2. **Jobs** sind eine Gruppe von Arbeitsschritten (Steps) und können parallel oder sequenziell ablaufen.
3. **Steps** werden innerhalb eines Jobs einzeln durchlaufen, laufen im selben Runner und können Daten teilen.
4. **Actions** sind vordefinierte Befehle oder Skripte, welche in den Steps verwendet werden, um einen Job auszuführen.
5. **Runners** sind die Ausführungsumgebungen, auf denen die Jobs laufen. Diese Runner können von GitHub oder selbst gehostet werden.

2.2. Vor- und Nachteile

- Vorteile:
  - Höhere Benutzer-Zufriedenheit durch weniger Bugs
  - Produkte können schneller auf den Markt gebracht werden
  - Entlastung der Entwickler
- Nachteile:
  - Technische Infrastruktur muss erst geschaffen werden

2.3. Protokoll

Die verschiedenen Tools wurden ausgiebig betrachtet. Sie unterscheiden sich in der Art der Bereitstellung (Selfhosting vs. Hosting in der Cloud), dem Integrationsgrad mit bestimmten Plattformen (z.B. GitHub, GitLab, Azure, AWS), der Benutzerfreundlichkeit, der Erweiterbarkeit und anderen Features. So gibt es für unterschiedliche Teamgrößen und Anforderungen passende Lösungen.

Wir haben uns für GitHub Actions entschieden, da wir mit unserem Repository bereits auf Github arbeiten. GitHub Actions ist bereits in GitHub integriert, es ist keine externe Konfiguration notwendig.

Das Setup und die Bedienung sind einfach. GitHub Actions ist schnell und direkt im Repository per YAML-Dateien konfigurierbar. Es ist optimal für Teams, die keine eigene CI-CD-Infrastruktur selbst betreiben wollen oder können.

Wir haben zunächst folgende Pipelines gebaut:

1. **java-ci.yml** Java-ci wird ausgeführt, wenn Änderungen im Source-Ordner vorgenommen werden. Diese Pipeline sorgt für Linting (Superlinter), eine automatisierte Codeprüfung, Tests und die Dokumentation bei jedem Commit und jedem Pull Request.

2. **readme-pdf.yml** Diese Pipeline wird ausgeführt, wenn:

   - Änderungen an der Readme-Datei in den feature-Branches gemacht werden
   - Änderungen an der Readme-Datei in der Main-Branch gemacht werden

- Ein Pull Request gemacht wird. Dies sorgt dafür, dass eine README.md-Datei automatisch in eine PDF-Datei umgewandelt wird, so dass immer eine aktuelle PDF-Version der Readme-Datei im Repository zu finden ist.

2.4. Dokumentation der Tests

Dauer der Workflows teilweise zu lang. Konflikte beim erzwingen von Checks, wenn vom Feature Branch in den Main gemerged werden soll. SuperLinter teilweise zu aggressiv beim linten. Zunächst haben wir eine einzelne Pipeline für den Superlinter eingerichtet. Dann haben wir uns jedoch dafür entschieden diesen mit in die java-ci Pipeline zu integrieren, da diese spezifisch auf Java zugeschnitten ist und der Linter dort benötigt wird. Außerdem wollten wir testen, ob in einer Pipeline mehrere Jobs laufen und dadurch Abhängigkeiten geschaffen werden können. (Ein Job abhängig von einem anderen)