GIT in der Schule

Inhalt

1. Administration der Online-Umgebung . . . 4
   1.1. Nutzerinnen und Nutzer zentral anlegen . . . 4
      1.1.1. CSV-Upload . . . 4
      1.1.2. Nutzer . . . 4
         Nutzer anlegen . . . 4
         Nutzerzuordnung . . . 5
   1.2. Passwörter . . . 5
      1.2.1. Initialpasswort . . . 5
      1.2.2. Passwortreset . . . 5
   1.3. Klassen . . . 5
   1.4. Repository . . . 6
   1.5. Löschroutine . . . 6
   1.6. Quota . . . 6
2. Umgang mit Git (Online) . . . 8
   2.1. Anlegen von Repositories . . . 8
      2.1.1. Navigation zur Anlegemaske . . . 8
      2.1.2. Ausfüllen der Anlegemaske . . . 8
      2.1.3. Befüllen mit Daten . . . 9
   2.2. Einstellungen bei Repositories . . . 9
      2.2.1. Zugriffsrechte / Repositories freigeben . . . 10
      2.2.2. Issues zulassen . . . 11
   2.3. Forken eines Repositories . . . 12
   2.4. Herunterladen von Repositories / einzelnen Branches . . . 13
   2.5. Löschen von Repositories . . . 14
3. Umgang mit Git am lokalen PC . . . 15
   3.1. Clienten . . . 15
      3.1.1. GitButler . . . 15
         Repository eröffnen/ klonen . . . 15
         Hauptansicht . . . 16
         Mergekonflikte . . . 18
      3.1.2. SmartGit . . . 19
         Installation . . . 19
         Repository eröffnen . . . 20
         Repository klonen . . . 22

Übersichten . . . 23
Mergen und Mergekonflikte . . . 24
3.2. Git-Konsole . . . 28
3.2.1. Anlegen eines Repositories . . . 28
Konsole starten . . . 28
Navigation . . . 28
Initialisierung . . . 28
Verzeichnis prüfen . . . 28
3.2.2. Clonen eines Repositories . . . 29
Konsole starten . . . 29
Navigation . . . 29
Klonvorgang anstoßen . . . 29
Erfolg prüfen . . . 29
3.2.3. Stage / Commit . . . 29
Überprüfen der Änderungen . . . 29
Stagen der Änderungen . . . 29
Überprüfen der gestagten Änderungen . . . 30
Commiten der Änderungen . . . 30
Überprüfen des Commit-Erfolgs . . . 30
3.2.4. Check out . . . 30
Überprüfen des Repository-Status . . . 30
Aus-checken des Gewünschten Branchs oder Commits . . . 30
Überprüfen des Wechsel-Erfolgs . . . 31
3.2.5. Push . . . 31
Überprüfen des Repository-Status . . . 31
Pushen der Änderungen . . . 31
Authentifizierung (falls erforderlich) . . . 31
Überprüfen des Push-Erfolgs . . . 31
3.2.6. Pull . . . 32
Pullen der neuesten Änderungen . . . 32
Authentifizierung (falls erforderlich) . . . 32
Überprüfen des Pull-Erfolgs . . . 32
3.2.7. Merge / Rebase . . . 32
Navigieren zum Ziel-Branch . . . 32
Mergen des Quell-Branchs . . . 33
Bearbeiten von Merge-Konflikten (falls erforderlich) . . . 33
3.2.8. Branches . . . 33
anlegen . . . 33
mergen . . . 34

4. GIT auf der Konsole . . . 36
4.1. Konfiguration . . . 36
4.2. Erste Schritte mit Git . . . 36
4.2.1. Initialisieren . . . 37
4.2.2. Repository Status anzeigen lassen . . . 37
4.2.3. Ein erster Tagebucheintrag . . . 37
4.2.4. Ein erster Commit . . . 38
4.2.5. Commit: Schritt für Schritt . . . 39
4.2.6. Frühstück . . . 41
4.2.7. Mittagessen . . . 43
4.3. Versionshistorie untersuchen . . . 44
4.3.1. Blick in die Vergangenheit . . . 44
4.3.2. Änderungen zwischen Commits ansehen . . . 46
4.3.3. Reise in die Vergangenheit . . . 47
4.3.4. Manipulation der Vergangenheit . . . 51
4.4. Speichern in der Cloud . . . 52
4.4.1. Klonen eines Repositorys . . . 53
4.4.2. Änderung auf den Server zurückkopieren . . . 54
5. Anwendungsszenarien Materialtausch . . . 57
5.1. Dateitypen . . . 57
5.1.1. Software . . . 57
5.1.2. Skripte & Arbeitsblätter . . . 57
5.1.3. Präsentationen . . . 57
5.2. Szenarien für den Materialtausch . . . 57
5.2.1. Fortbildner untereinander . . . 58
5.2.2. Fortbildner - Lehrer . . . 58
5.2.3. Lehrer - Lehrer . . . 59
6. Anwendungsszenarien Unterricht . . . 60
6.1. SuS versionieren lokal . . . 60
6.2. Lehrperson stellt Material per GIT bereit, SuS versionieren lokal . . . 60
6.3. SuS versionieren online (SuS arbeiten zu Hause und in der Schule) . . . 61
6.4. Zusammenarbeit in Gruppenprojekten . . . 63
7. Übungen für die Fortbildung . . . 65
7.1. Lokal Versionieren . . . 65
7.2. Mit Vorlagen in einem Online-Repository arbeiten . . . 66
7.2.1. Vorlage übernehmen und lokal versionieren . . . 67
7.2.2. Änderungen an der Vorlage (Rebase) . . . 68
7.2.3. Musterlösung bereitstellen . . . 68
7.3. Eigenes Repository online ablegen . . . 69

7.3.1. Anlegen eines Repositories auf dem GIT-Camp Server . . . 70
7.3.2. Arbeiten an verschiedenen Orten . . . 72
7.4. Zusammenarbeit Lehrer-Schüler . . . 72
7.5. Kooperatives Arbeiten . . . 75
7.6. Mit ZPG-Materialien arbeiten (Verändern / Fehler melden usw.) . . . 78
8. Glossar (Kopie aus Urs Dokument) . . . 80

1. Administration der Online-Umgebung

1.1. Nutzerinnen und Nutzer zentral anlegen

Beim Upload einer CSV werden diverse benutzerbezogene Kriterien vom Controller überprüft und bei Bedarf angepasst. Der CSV-Upload findet getrennt nach Lehrkräften sowie Schülerinnen und Schülern statt (in Umsetzung). Es muss beim Upload der Radiobutton gesetzt werden (Lehrkräfte oder Schülerinnen und Schüler). Dadurch wird Nutzerinnen und Nutzern auch die jeweilige Rolle zugewiesen.

1.1.1. CSV-Upload

Zunächst findet ein Abgleich der hochgeladenen CSV-Datei statt. Dabei wird wie folgt verfahren:

- Benutzer in DB und CSV vorhanden, dann Aktualisierungen übernehmen
- Benutzer in DB vorhanden, aber nicht in CSV, dann Benutzer/in deaktivieren
- Benutzer in DB nicht vorhanden, aber in CSV, dann Nutzer/-in neu anlegen

1.1.2. Nutzer

Folgende Daten werden aus der CSV-Datei importiert, verarbeitet, generiert und gespeichert:

- Vorname (importiert)
- Nachname (importiert)
- Primärschlüssel (importiert)
- Klassenzuordnung (importiert)
- E-Mail Adresse (importiert, falls vorhanden, ansonsten Vergabe einer Platzhalter E-Mail Adresse)
- Initialpasswort (generiert)
- Rolle (ergibt sich aus dem Upload)

Nutzer anlegen

Dies aus der CSV übernommenen Vor- und Nachnamen werden:

- transponiert
- Umlaute werden entfernt
- doppelte Vornamen werden entfernt. Z.B.: aus "Ben Marlon MüllerHofholz" wird "Ben.MuellerHofholz"
- Der Benutzer jedes Nutzers Users wird nach folgendem Schema erzeugt: Vorname.Nachname
- Bei identischen Vornamen, Nachnamen und/oder Geburtsnamen jedoch unterschiedlicher ID wird an den generierten Nutzernamen eine fortlaufende Nummer angehängt.

Nutzerzuordnung

Beim Import von Nutzern aus der CSV-Datei werden diese bereits in die richtigen Klassen (Organisationen) eingepflegt. Nutzer, die sowohl in der CSV, als auch in der DB vorhanden sind, werden bei Bedarf neuen Klassen (Neues Schuljahr) oder zusätzlichen Organisationen (zB neue Lerngruppe) zugeordnet. Vorhandene Nutzer (DB + CSV) werden aus allen, nicht in der CSV zu diesem jeweiligen Nutzer hinterlegten Klassen gelöscht.

1.2. Passwörter

1.2.1. Initialpasswort

Beim Import wird automatisch ein Initialpasswort generiert, welches bei der ersten Anmeldung geändert werden muss. Die Lehrkraft erhält per E-Mail eine Liste mit den Nutzerdaten der Klasse(n) in denen sie unterrichtet.

1.2.2. Passwortreset

Der Controller spielt je nach Lehrkraft die von ihr unterrichteten Klassen (Organisationen) in der GUI aus. Nach Auswahl der Klasse (Organisation) zeigt der Controller die Schülerinnen und Schüler der Klasse (Organisation) an. Nun kann die Lehrkraft in der jeweiligen Spalte der Schülerin oder des Schülers den Button „Passwort zurücksetzen“ anklicken. Anschließend wird der Lehrkraft ein temporäres Passwort angezeigt, welches bei der nächsten Anmeldung geändert werden muss.

1.3. Klassen

Klassen (Organisationen) werden nach folgendem Schema erstellt:

- Klassen (Organisationen) die beim CSV-Import erzeugt werden, erhalten automatisch ein Suffix mit dem Erstellungsjahr. Muster: 10b-2023
- Jede automatisch angelegte Klasse wird in einer Organisation abgebildet und die entsprechenden Nutzer werden dieser Organisation zugeordnet.
- Eine Organisation "Lehrkräfte" wird einmalig erstellt. Die importierten Lehrkräften werden

der Organisation automatisch zugeordnet.

Schülerinnen und Schüler können keine Klassen anlegen.

1.4. Repository

Jede Organisation (Klasse) erhält automatisch ein Repository. Der Besitzer des Repository ist die unterrichtende Lehrkraft. Repositories werden nach folgendem Schema erzeugt:

- Der Zeitpunkt der Repoanlage wird automatisch im Repotitel als Suffix angehängt, sofern die Anlage durch einen Schüler erfolgt. Z.B. Robotik-2023
- Repos die in Organisationen liegen können ausschließlich von Lehrkräften erzeugt werden.
- Maximal 50 Repos je Nutzer.

1.5. Löschroutine

Die automatische Löschung von Klassen (Organisationen), Repos und Nutzern wird von einer Routine übernommen. Dabei wird nach folgendem Schema vorgegangen:

- **Organisationen**:
  - Organisationen mit - im Namen oder automatisch über CSV erzeugte Organisationen werden automatisch am 30. September des nachfolgenden Jahres archiviert.
  - Wenn eine langfristige Organisation benötigt wird (z. B. "Informatik-AG"), muss diese beim Schul-Verwalter beantragt werden.
  - Nach 365 Tagen erfolgt die automatische Löschung von archivierten Inhalten, dies kann nur durch den Schul-Verwalter verhindert werden.
  - Die Organisation "Lehrer" beinhaltet keine automatische Archivierung von Repos.
- **Nutzer**:
  - 365 Tage nach Deaktivierung erfolgt die automatische Löschung von deaktivierten Benutzern und deren Inhalten, dies kann nur durch den Schul-Verwalter verhindert werden
- **Repos**:
  - Repos, die In Organisationen liegen, werden 1 Jahr nach der Anlage automatisch archiviert, wenn sie ein - im Namen tragen.
  - 365 Tage nach Archivierung erfolgt die automatische Löschung von archivierten Inhalten, dies kann nur durch den Schul-Verwalter verhindert werden. Logs
  - Zugriffs-Logs der Schulinstanz werden automatisch nach 21-90 Tagen gelöscht

1.6. Quota

Die Löschroutine trägt einen wesentlichen Teil dazu bei, dass keine unnötigen Daten dauerhaft

gespeichert werden. Zusätzlich wird GitCamp eine Quota beinhalten (Aktuell noch in der Testphase). Die Quota kann über eine Weboberfläche innerhalb GitCamps geändert werden. Für jede Instanz ist die Quota bei der Einrichtung identisch. Sollte die Quota von einem Nutzer überschritten werden, so werden der Schul-Admin und der Nutzer per E-Mail benachrichtigt.

2. Umgang mit Git (Online)

2.1. Anlegen von Repositories

Das Anlegen eines neuen Repositories gestaltet sich auf der Weboberfläche von Gitcamp denkbar einfach. Dazu gilt es folgende Schritte zu befolgen:

2.1.1. Navigation zur Anlegemaske

Um ein Repository anzulegen drücken Sie zunächst auf das Plus am oberen, rechten Bildschirmrand.

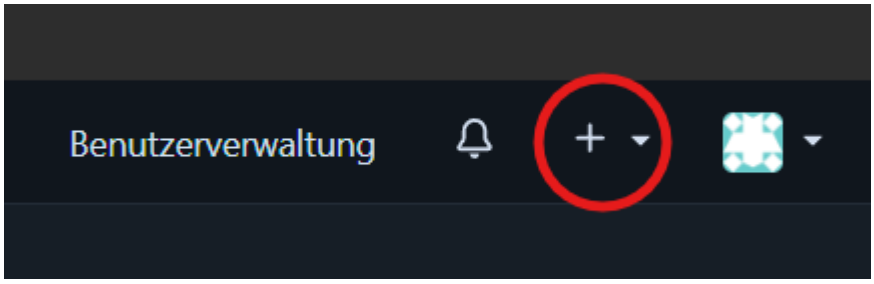


Abb. 1. Position des Plus-Buttons

Im Anschluss öffnet sich das Kontextmenü und man erhält die Wahl zum Anlegen eines neuen Repositories.

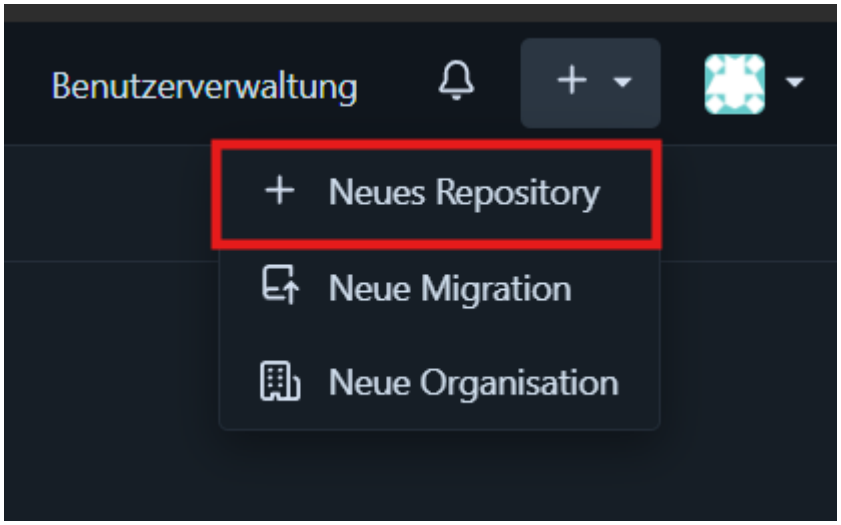


Abb. 2. Auswahl der "Neues Repository"-Option

2.1.2. Ausfüllen der Anlegemaske

Innerhalb der nun geöffneten Maske lassen sich verschiedene Optionen für das neue Repository auswählen.

- verpflichtend ist hier der Name
- auch die beiden beiden Einstellungen für die Sichtbarkeit können nicht übergangen werden.

Es empfiehlt sich für Testprojekte immer private Repositories zu verwenden, während die Vorlagen für andere öffentlich sein sollten.

- Eine Beschreibung (gerne in Ascii-Doc) hilft bei der Übersichtlichkeit
- als Lizenz empfiehlt sich die MIT-Lizenz
- Bei Bedarf bieten auch die anderen Einstellungsmöglichkeiten hervorragende Anpassungen

Durch Klicken auf den Button "Repository erstellen" wird das entsprechende Repository beim Nutzer angelegt.

2.1.3. Befüllen mit Daten

Im Nachgang zum Anlegen des Repsotories wird einem eine Kurzanleitung direkt eingeblendet. Diese gibt einem sogar die direkten Befehle vor um Lokal ein entsprechendes Repository anzulegen.

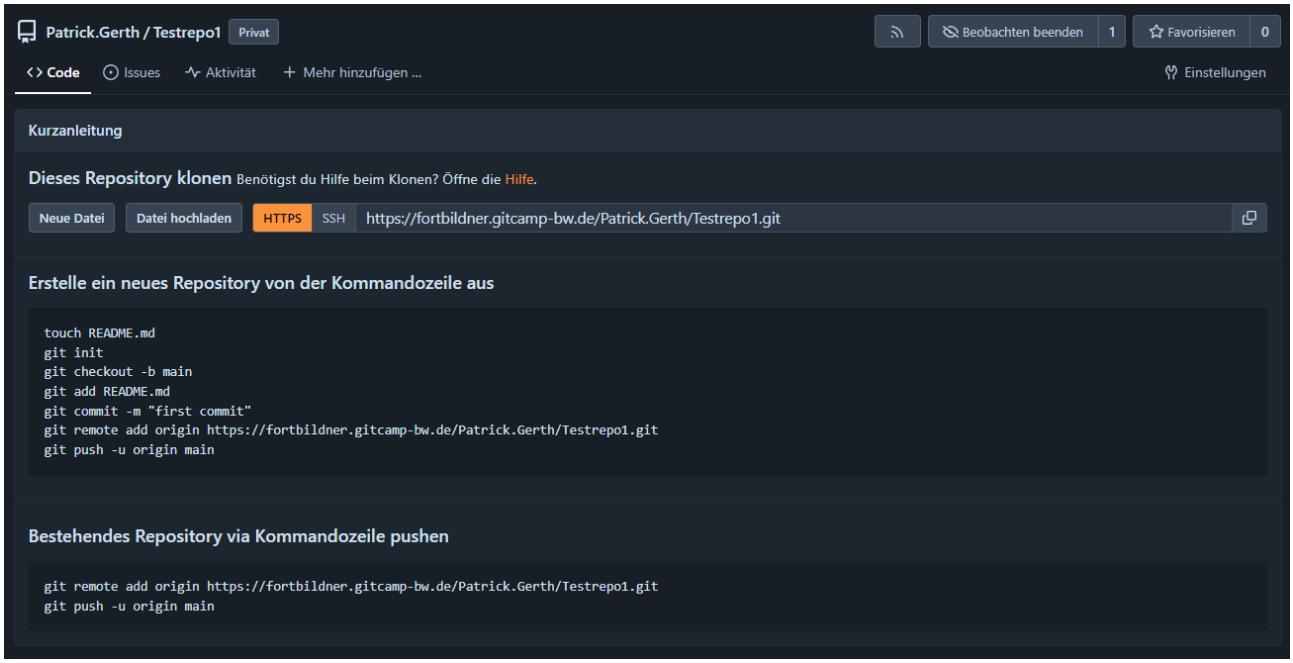


Abb. 3. Maske nach dem Erstellen

Selbstverständlich sind die entsprechenden Werte auf Ihr persönliches Repository anzupassen.

Sind Ihre initialen Daten erstmal hinterlegt können Sie umgehend mit der Arbeit beginnen. Auf ihrer Startseite finden Sie ab jetzt am rechten Rand eine Auflistung Ihrer Repositories.

2.2. Einstellungen bei Repositories

Befinden Sie sich auf der Seite Ihres Repositories finden sie oben rechts den Einstellungsbutton:

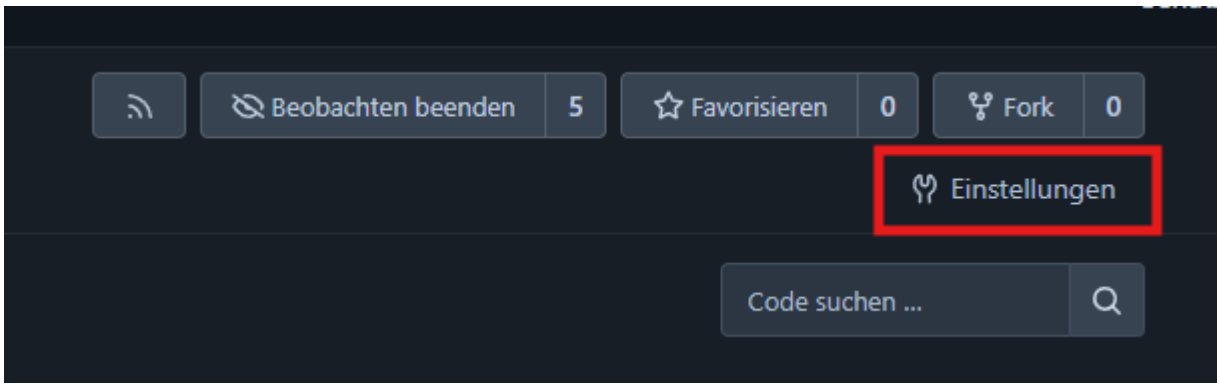


Abb. 4. Auswahl der Einstellungs-Option

Hier gelangen Sie zunächst auf eine Seite, auf welcher Sie sämtliche Einstellungen anpassen können, welche Sie bei der Erstellung getroffen haben. Außerdem finden sich hier auch komplexere Einstellungsmöglichkeiten, was das Signaturvertrauensmodell angeht (sofern Sie ein solches in Ihren Projekten verwenden möchten) und die besonderen Verwaltungseinstellungen des Administrators eines Projekts.

Abb. 5. Admineinstellungen

Die beistehenden Texte sind hier selbsterklärend.

2.2.1. Zugriffsrechte / Repositories freigeben

Wie im vorherigen Abschnitt zu sehen ist lässt sich ein Repository direkt an andere Benutzer übertragen. Allerdings ist dies natürlich nicht immer sinnvoll. Statt dessen benötigt ein entsprechendes Projekt oft Mitarbeiter oder, sollte es sich um ein privates Repository handeln, Betrachter.

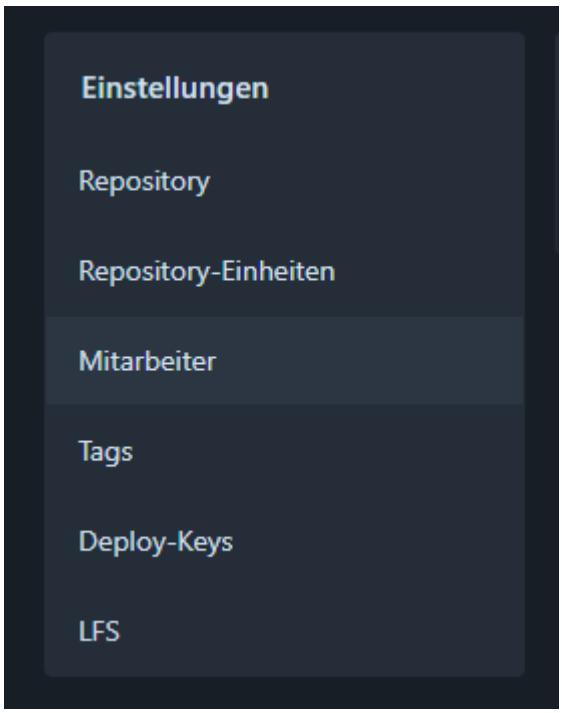


Abb. 6. Mitarbeiterbereich in den Einstellungen

Hier lassen sich Mitarbeiter hinzufügen, entfernen und individuell einstellen, ob sie

- Nur leserechte haben
  - also nur die vorhandenen Dateien einsehen können und ggf. selbst von diesen einen Fork erstellen können
- Schreibrechte haben
  - um damit auch Veränderungen an den Dateien vornehmen können
- oder Administratorrechte haben
  - und damit das komplette Projekt samt Mitarbeiterliste etc. verändern können. Hier wird entschieden davon abgeraten leichtfertig dieses Recht zu vergeben.

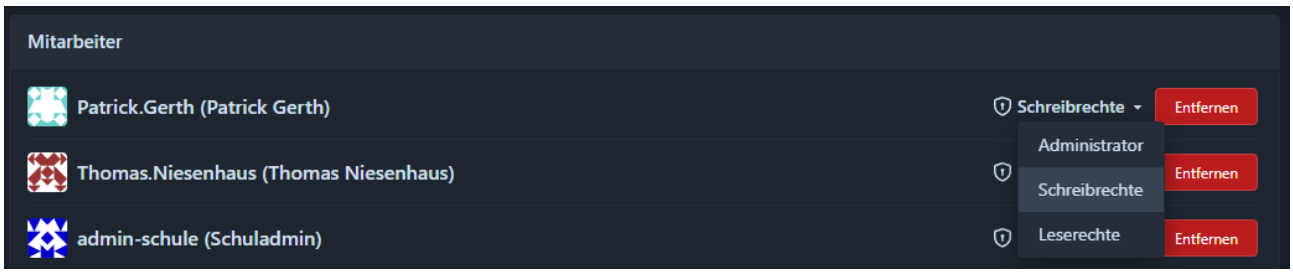


Abb. 7. Mitarbeitereinstellungen

2.2.2. Issues zulassen

Um Rückmeldungen, Fehlermeldungen, Funktionswünsche etc. an Projekte zu richten oder um entsprechende Dinge selbst am Projekt zu verfolgen eignet sich die Erstellung sog. Issues. Diese werden im entsprechenden Projekt unter "Issues" verwaltet.

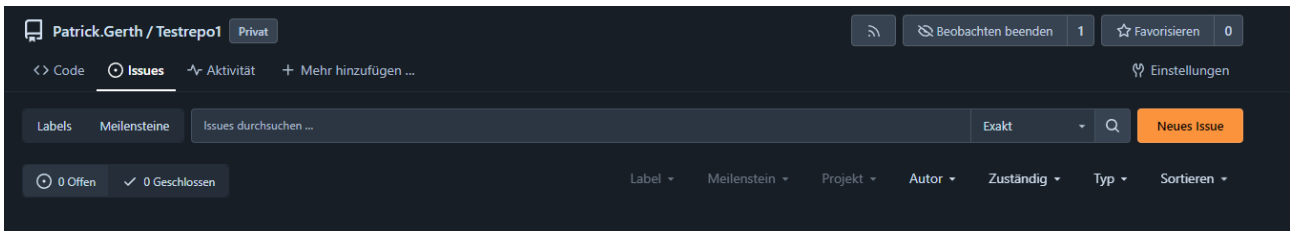


Abb. 8. Issues-Tab

Hier können Kommentare angehängt werden, welche den aktuellen Status beschreiben oder diskutieren. Durch Schließen des Issues seitens der Administration gilt dieses als abgearbeitet oder abgelehnt. Stellt sich im Nachhinein heraus, dass dem nicht so ist, so kann das Issue wieder eröffnet werden. Bei Öffnen des Issues lassen sich außerdem Zuständige, Meilensteine, Fälligkeitsdaten etc. anhängen. Außerdem kann man die Kommentarsektion sperren, das Issue oben in der Issueliste anheften oder das ganze Issue löschen.

2.3. Forken eines Repositories

Betrachtet man ein fremdes Repository, oder eines, in welchem zumindest Inhalte vorhanden sind, so lässt sich davon ein sog. Fork erstellen. Dabei handelt es sich um Kopien des entsprechenden Repositories zum Zeitpunkt des Forkens, welche beim forkenden Benutzer verortet sind. Zu finden ist der Knopf zum Forken auf der Seite des entsprechenden Repositories:

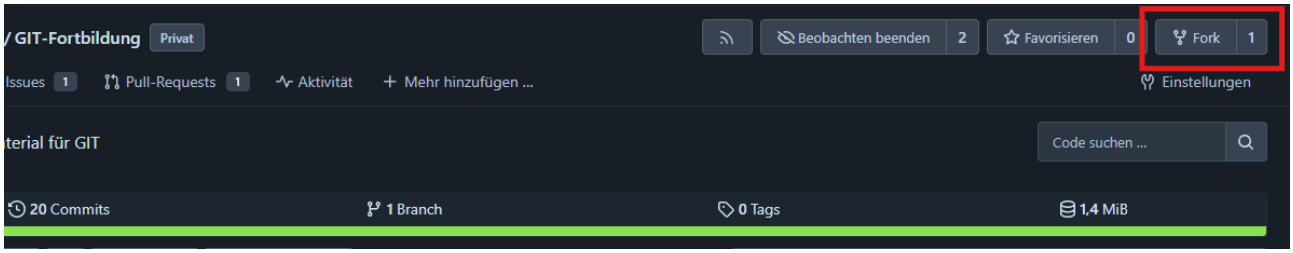


Abb. 9. Fork-Button

Durch Drücken dieses Knopfes öffnet sich ein weiteres Dialogfenster, welches verschiedene Optionen anbietet.

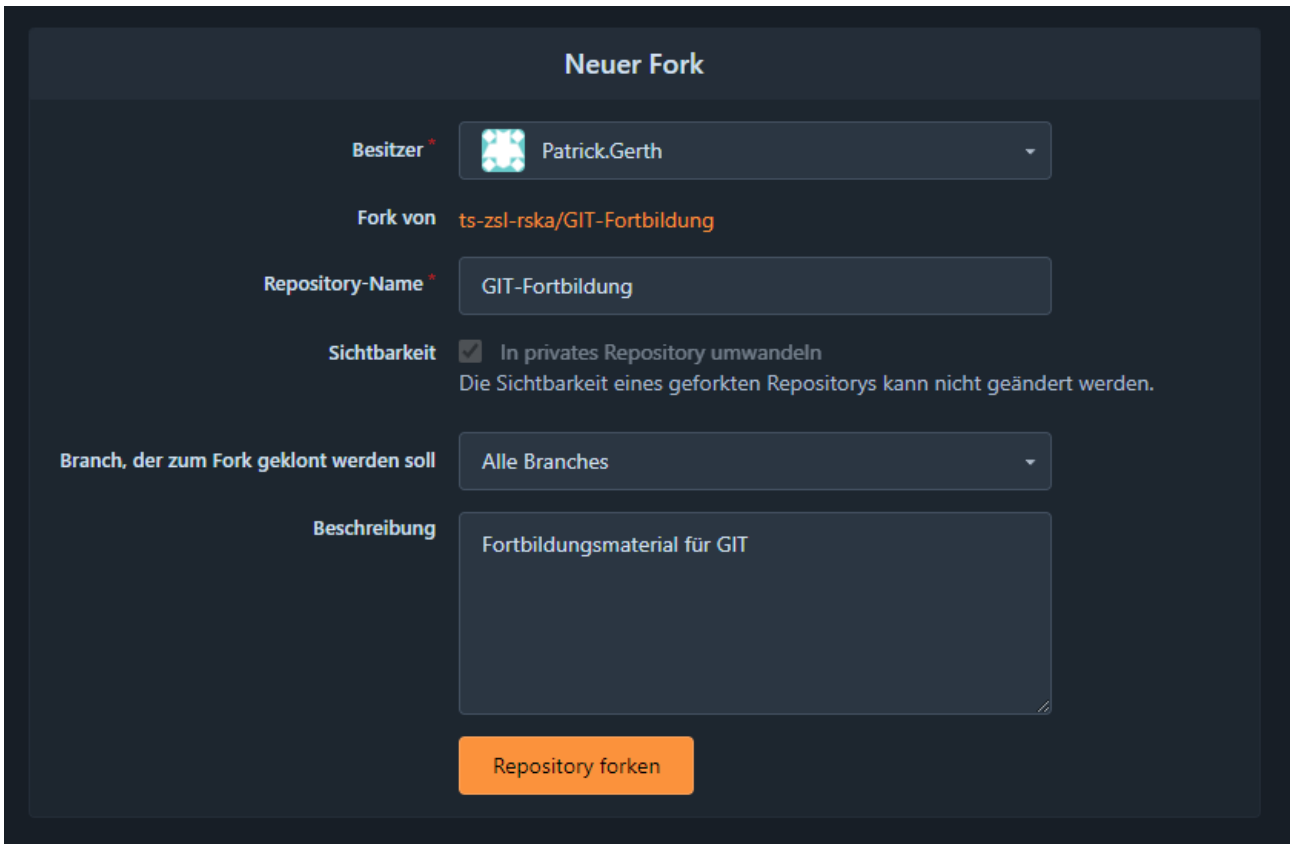


Abb. 10. Fork-Dialog

Durch Anlegen des Forks öffnet sich die entsprechende Seite des neuen Repositories. Damit sind Sie fortan Besitzer einer entsprechenden Kopie des Originalrepositories. Updates dessen werden nicht zu Ihnen weitergespielt und Ihre Änderungen werden nicht an das Originalrepository weitergegeben.

2.4. Herunterladen von Repositories / einzelnen Branches

Möchten Sie Ihr Repository von Ihrer lokalen Maschine aus herunterladen, so hängen die nächsten Schritte massiv von Ihrer benutzten Entwicklungs- / Git-Umgebung ab. Zentral festzuhalten gilt, dass Sie den Zugriffsschlüssel auf der Projektseite direkt über der Dateiliste am rechten Rand abfragen können:

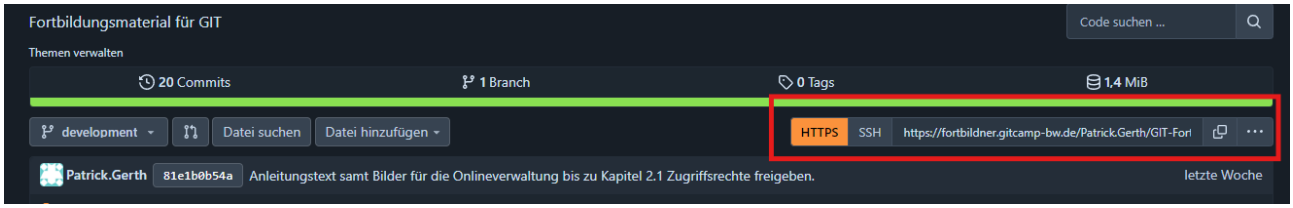


Abb. 11. Access-Link

Die Wahl zwischen HTTPS oder SSH sollte für die meisten Benutzer auf HTTPS fallen, wenn sie sich nicht über einen entsprechenden SSH-Schlüssel authentifizieren wollen. Das Erstellen und Verwenden von SSH-Schlüsseln würde den Rahmen dieser Dokumentation allerdings sprengen.

2.5. Löschen von Repositories

Möchten Sie ein Repository löschen geschieht dies auf der "Einstellungen"-Seite des Repositories ganz unten. Dies ist endgültig! Folgen Sie hier den Anweisungen des Dialogfensters.

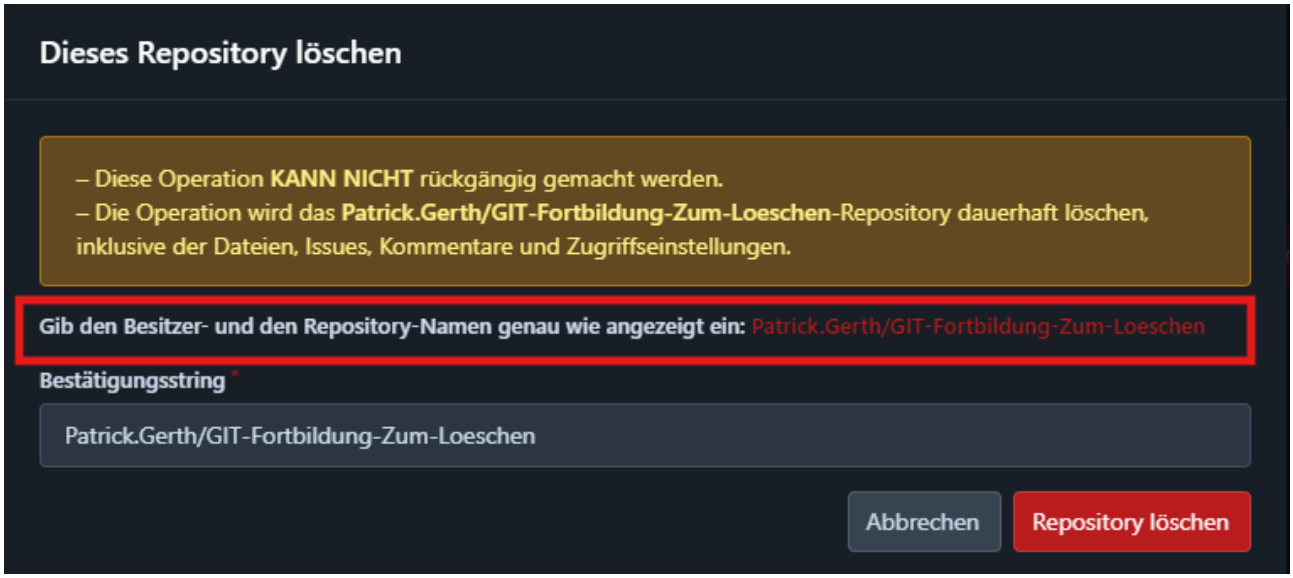


Abb. 12. Löschen-Dialog

Im Anschluss finden Sie oben in der Mitte des neuen Fensters die Meldung "Das Repository wurde gelöscht." wenn der Vorgang erfolgreich war.

3. Umgang mit Git am lokalen PC

3.1. Clienten

Zum Zeitpunkt des Erstellens dieser Materialien bieten sich verschiedene Lösungen für Desktopanwendungen an. Beispiele wären hier:

- GitButler, welches unter https://gitbutler.com/ erreichbar ist und open source eingesehen werden kann.
- GitHub Desktop ist unter https://desktop.github.com/download/ verfügbar und ebenfalls open source.
- SmartGit ist ein kommerzielles Produkt aus dem Hause Syntevo. Die Projektseite findet sich unter https://www.syntevo.com/smartgit/

3.1.1. GitButler

Nach der Installation findet man sich im Startfenster von GitButler.

Abb. 13. Dialogfeld Neuanlegung

Repository eröffnen/ klonen

GitButler kann zum Zeitpunkt der Erstellung dieses Dokuments Repositories nur verwalten, nicht frisch anlegen. Daher wäre der Ablauf der, dass man zunächst ein Repository in der Webmaske erstellt und dieses dann via "Clone repository" auf den lokalen Rechner klont.

GitButler tut sich erfahrungsgemäß schwer damit leere Repositories zu klonen. Um dies zu umgehen kann man eine leere Textdatei in der Online-Ansicht des Repositories anlegen.

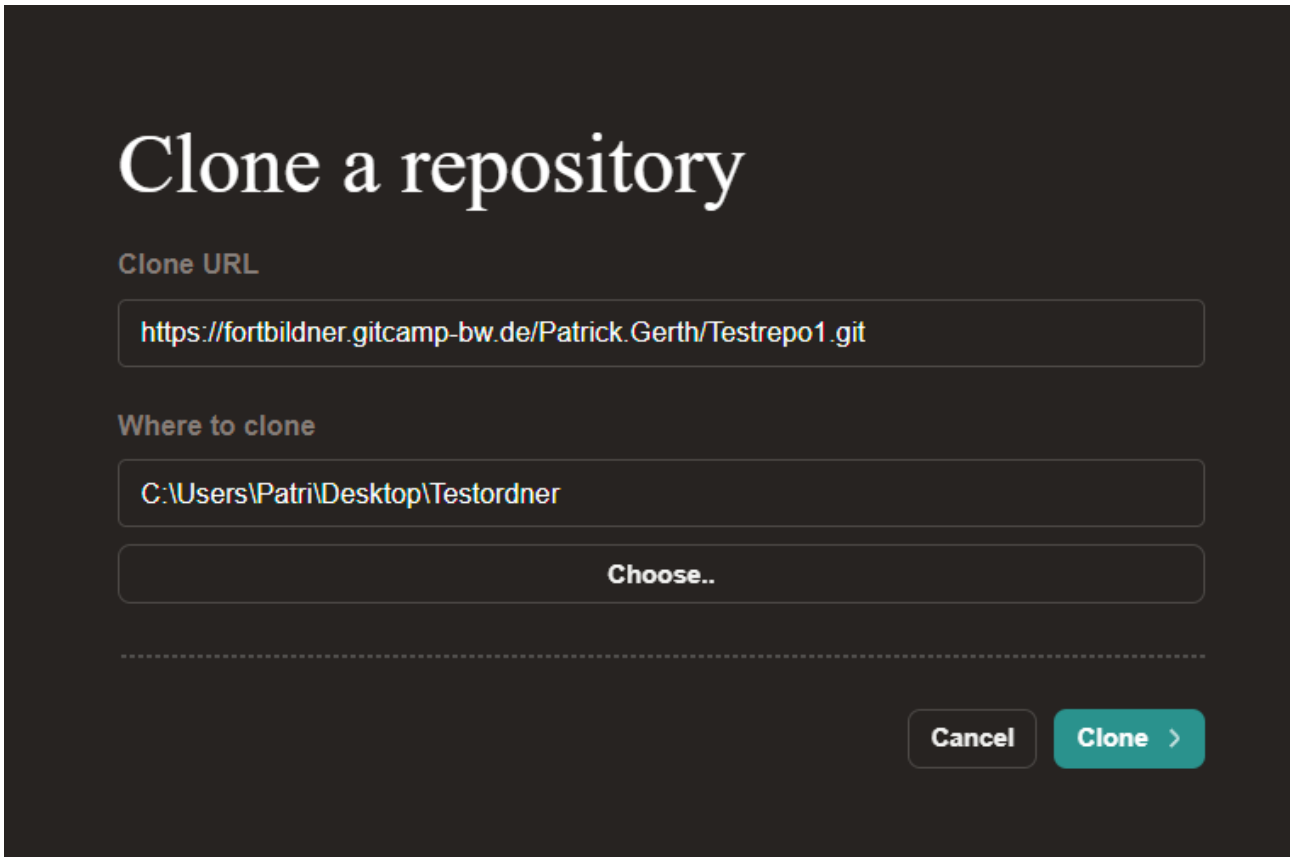


Abb. 14. Dialogfeld Klonen

Im Anschluss öffnet sich die Hauptansicht.

Hauptansicht

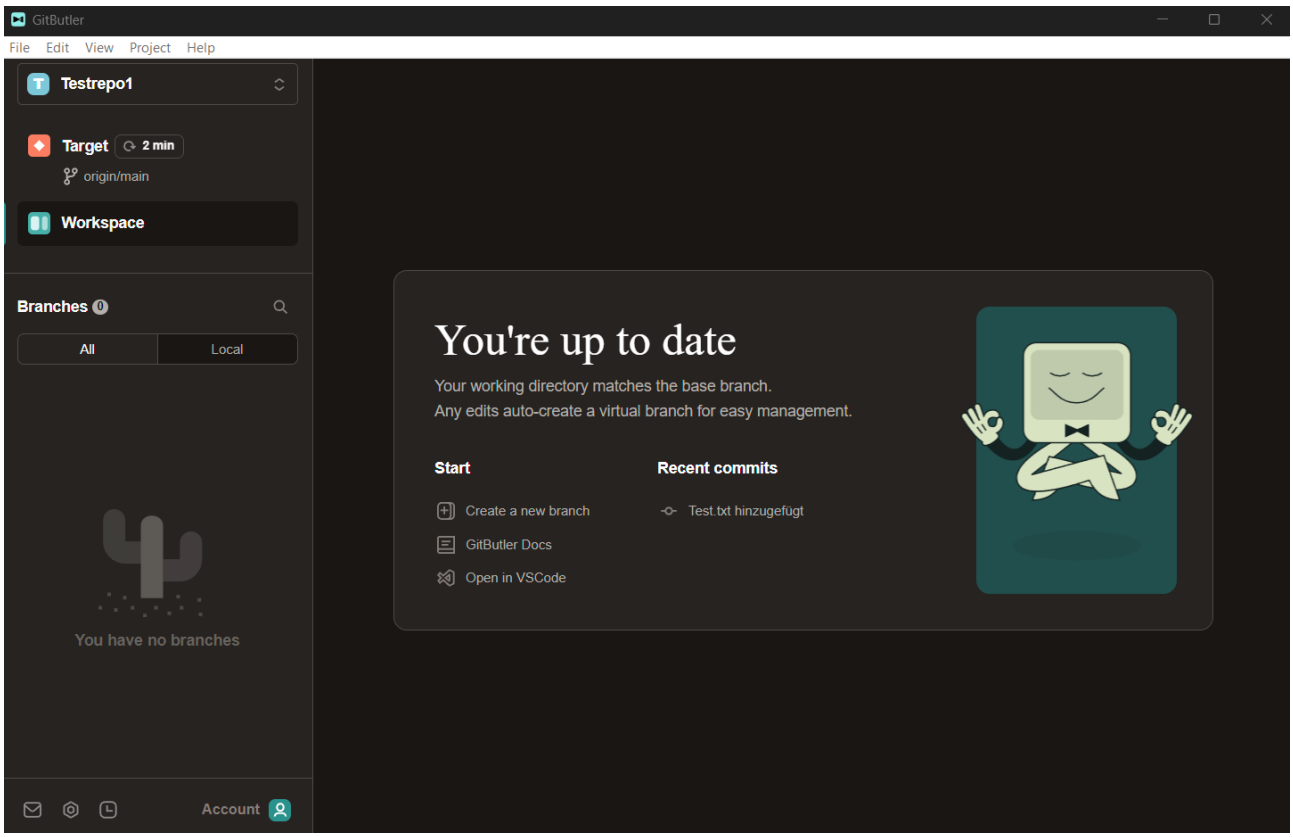


Abb. 15. Hauptansicht

Hier lassen sich verschiedene Branches anlegen, in welche die entsprechenden Dateien per drag-and-drop hineingezogen werden können:

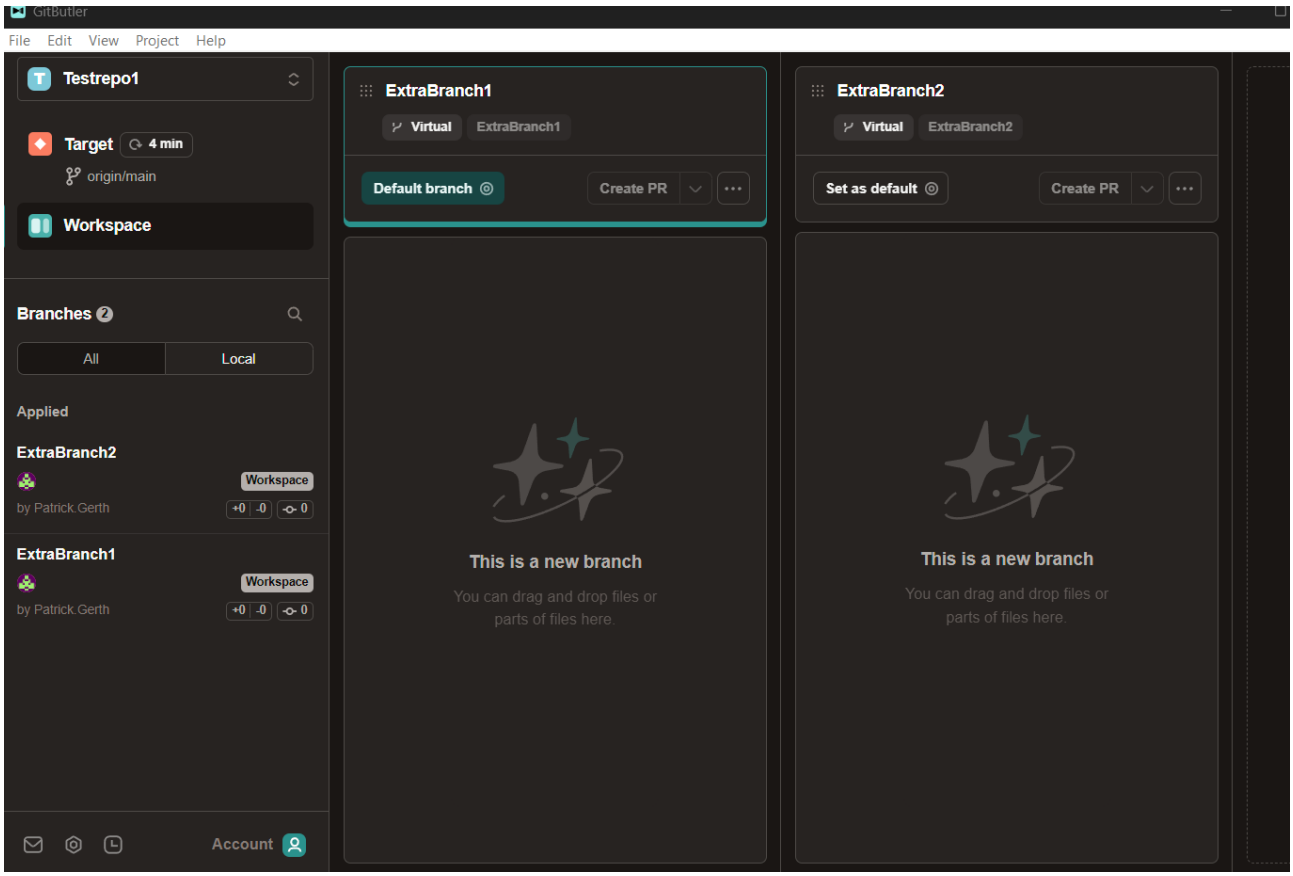


Abb. 16. Branch-Ansicht

Für eine detaillierte Führung durch die Benutzung empfiehlt sich ein Blick in die Dokumentation unter https://docs.gitbutler.com/

Mergekonflikte

In seiner aktuellen Fassung (Stand 10.09.2024) verfügt Gitbutler nur über sehr eingeschränkte Möglichkeiten Mergekonflikte zu lösen. Statt dessen wird jedes mal lokal ein neuer Entwicklungsbranch (Virtualbranch) angelegt, welcher dann per Konsole oder Webanwendung wieder gemerged werden muss.

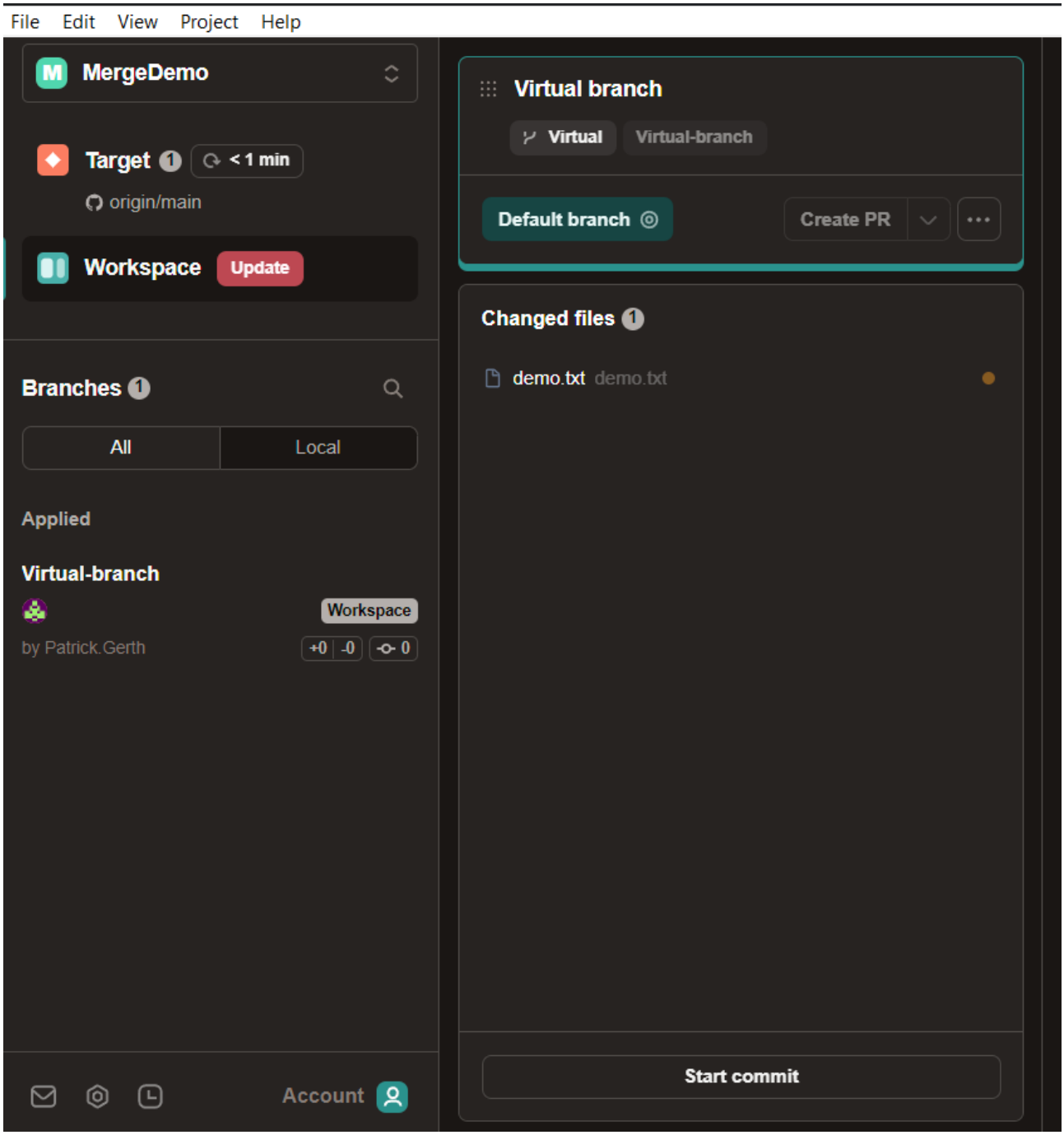


Abb. 17. Branch-Ansicht mit Virtual branch

3.1.2. SmartGit

Installation

Smartgit gibt es in unterschiedlichen Varianten (https://www.syntevo.com/register-non-commercial/#academic): eine kostenpflichtige Version, eine kostenlose Education-Version und eine kostenlose Privatuser (Hobby-Edition). Für einfache Anwendungen (ohne kollaboratives Arbeiten) reicht die Hobby-Edition. Diese können Schülerinnen und Schüler leicht zu Hause installieren. In der Schule empfiehlt sich die Education-Version. Dafür muss die Software mit einer E-Mail-Adresse der Schule registriert werden. Das Lizenzdatei (nach Absprache mit der Firma) darf dann von allen Lehrern und Schülern der Schule benutzt werden.

Installation unter Windows: Die Software kann als portable Version auf dem Server der Schule abgelegt werden, wenn einige zusätzliche Einstellungen vorgenommen werden:

1. Die Datei `smartgit\bin\smartgit.vmoptions` muss so angepasst werden, dass die Settings-Einstellungen im Homeverzeichnis der Schüler gespeichert werden:

   -Dsmartboot.sourceDirectory=H:\smartgit-settings\updates -Dsmartgit.settings=H:\smartgit-settings -XX:ErrorFile=%EXE4J_EXEDIR%..\.settings\hs_err_pid%p.log ---

2. Dieses Verzeichnis muss beim ersten Starten von Smartgit mit den Standardwerten gefüllt werden (insbesondere der Lizenzdatei und den Proxy-Einstellungen). Daher führt man Smartgit als Admin einmalig aus und konfiguriert die Lizenzdatei und die Starteinstellungen. Die kompletten Einstellungen werden in dem Verzeichnis `.settings` bzw. `H:\smartgit-settings` (wenn man Punkt 1 schon durchgeführt hat) gespeichert.

   Dieses Verzeichnis kopiert man auf dem Server in ein eigenes Verzeichnis z. B. `smartgit-settings-schulexy`.

3. Proxy-Einstellungen: Damit der Proxy der Schule benutzt wird, muss man in der Datei `preferences.yml` folgende Zeilen anpassen:

   ---

   proxy: {user: %user%, authenticate: true, host: 10.10.0.1, enabled: true, port: 8080, autoDetect: false} ---

   Die Variable %user% wird später durch ein Skript durch den Usernamen ersetzt.

4. Statt direkt smartgit.exe zu starten, muss die Verknüpfung so angelegt werden, dass eine Batch-Datei mit folgendem Inhalt gestartet wird:

   ---

   if exist H:\smartgit-settings\ (echo Settingsverzeichnis schon vorhanden) else (mkdir H:\smartgit-settings xcopy P:\informatik\smartgit\smartgit-settings-schulexy H:\smartgit-settings /s /e cscript P:\informatik\smartgit\replaceusername.vbs)

echo starte smartgit xcopy H:\smartgit-settings\.gitconfig %USERPROFILE%\ /I /Y p:\informatik\smartgit\bin\smartgit.exe xcopy %USERPROFILE%\.gitconfig H:\smartgit-settings /Y /I ---

+ Der erste Teil kopiert das Settingsverzeichnis, wenn es noch nicht existiert. Der Username wird durch ein vbs-Skript ausgelesen und angepasst.

+

Const ForReading = 1 Const ForWriting = 2

strFileName = "H:\smartgit-settings\preferences.yml" strOldText = "%user%" strNewText = CreateObject("WScript.Network").UserName

Set objFSO = CreateObject("Scripting.FileSystemObject") Set objFile = objFSO.OpenTextFile(strFileName, ForReading) strText = objFile.ReadAll objFile.Close

strNewText = Replace(strText, strOldText, strNewText) Set objFile = objFSO.OpenTextFile(strFileName, ForWriting) objFile.Write strNewText objFile.Close ---

+ Der zweite Teil ist notwendig, wenn das %USERPROFILE% der Schüler nicht gespeichert wird. In der .gitconfig-Datei werden globale Git-Einstellungen gespeichert. Durch die Befehle wird die in smartgit-settings gesicherte .gitconfig an die richtige Stelle kopiert, dann Smartgit gestartet und eventuelle Änderungen nach Beendigung wieder gesichert. Das ist notwendig, um Benutzer und Email-Konfiguration in Git dauerhaft zu speichern.

+

Repository eröffnen

Nach der Installation und dem Einstellen der Initialparameter findet sich die Einstellung zum Einbinden eines neuen Repositories oben links.

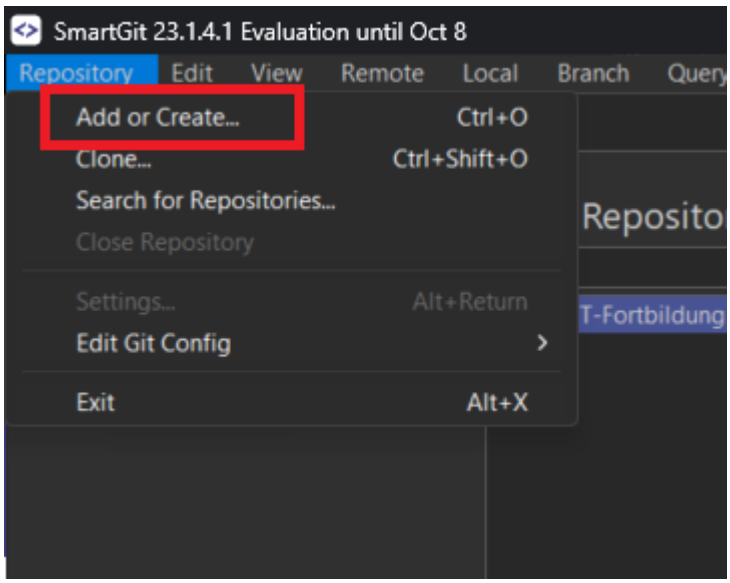


Abb. 18. Neues Repository anlegen oder abrufen

Durch Clicken der Option öffnet sich ein neues Dialogfeld:

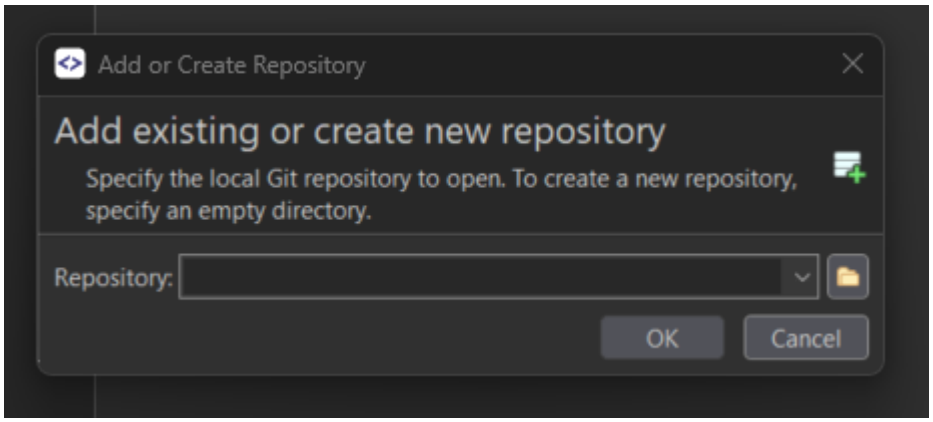


Abb. 19. Dialogfeld Neuanlegung

Wählen Sie hier einen Speicherort für das entsprechende Repository aus. Der Ordner sollte im Regelfall leer sein.

Durch das Auswählen von "Remote" in der Leiste am oberen Rand lässt sich eine Verknüpfung zu einem leeren Repository auf dem Git-Server anlegen

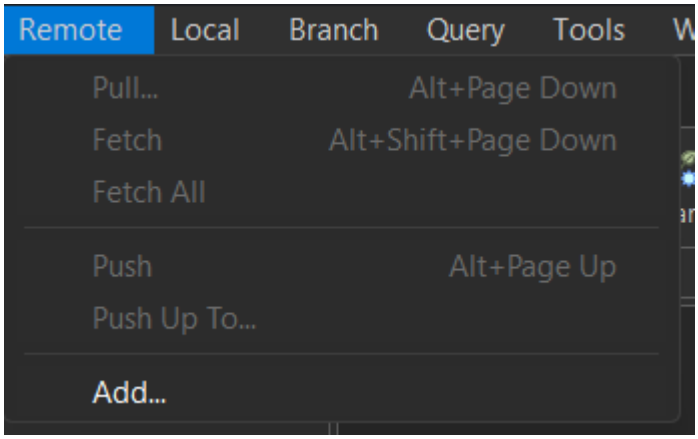


Abb. 20. Repository auf Server auswählen

Nach der Authentifizierung sind nun die beiden Repositories verknüpft und lassen sich fortan vollumfänglich benutzen.

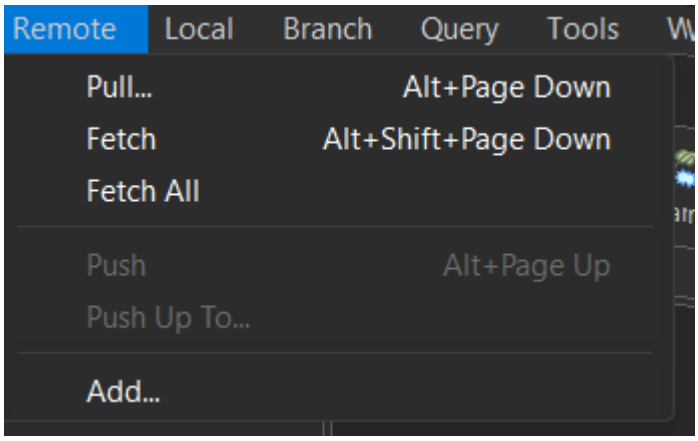


Abb. 21. Pull-Optionen bei vollständiger Verknüpfung

Repository klonen

Möchten Sie ein bestehendes Repository auf Ihren lokalen Rechner übertragen sprechen wir ja bekanntlich vom Klonen. Über Repositoy → Clone öffnet sich die Dialogmaske zum Eintragen des entsprechenden Links.

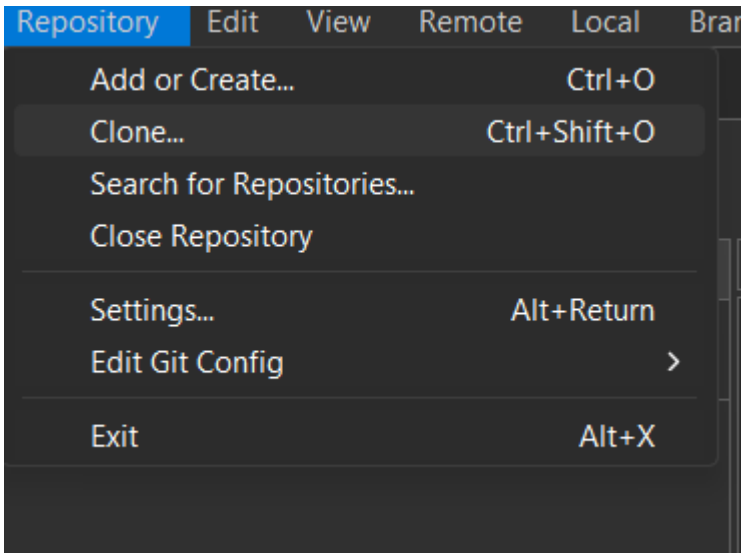


Abb. 22. Auswahl der Clone-Option

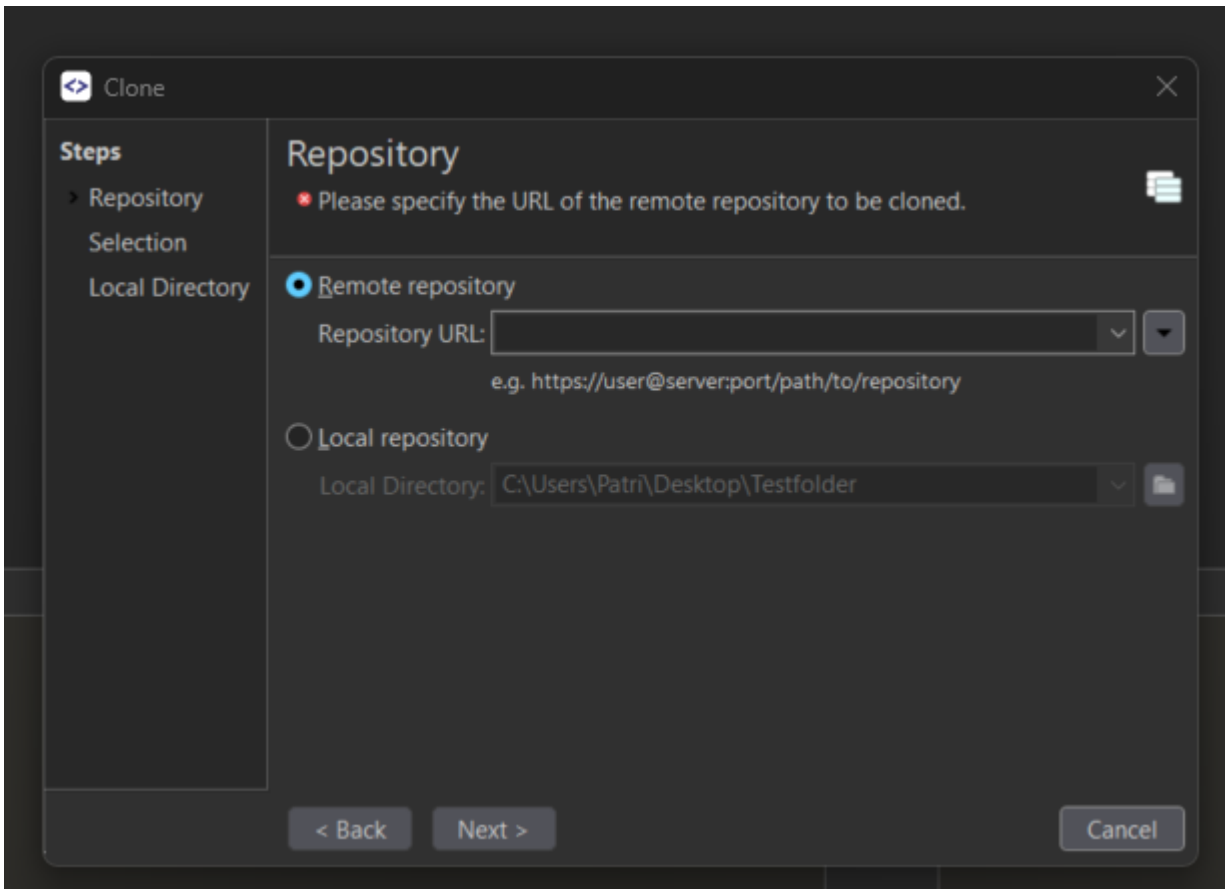


Abb. 23. Clone-Dialog

Da die beiden Optionen selbsterklärend sind wird nicht weiter darauf eingegangen. Die folgenden Dialoge fragen danach nach dem Speicherort und den Klonoptionen. Sobald dies ausgewählt wurde öffnet sich das Projekt im Betrachter.

Übersichten

Nachdem das entsprechende Repository erfolgreich in SmartGit eingebunden wurde bietet es sich an zwischen den entsprechenden Hauptansichten zu wählen.

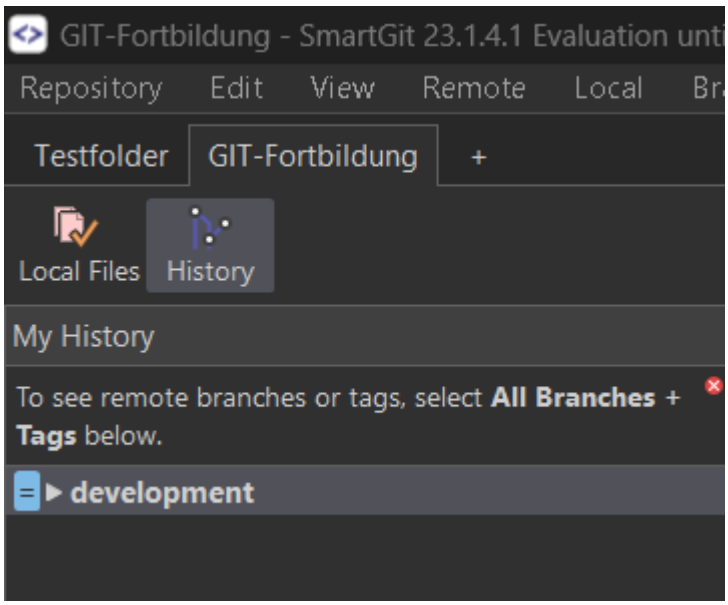


Abb. 24. Hauptansichten

- Local Files dient dabei dazu, die Dateien direkt zu verwalten. Hier sieht man eine Übersicht der Dateien seit dem letzten Pull und kann entscheiden welche Dateien damit im nächsten Commit landen.
- History zeigt den Branchverlauf der letzten Commits und deren entsprechenden Branches. Durch Anklicken der entsprechenden Commits wird ein sog. Dif erstellt, welches die Veränderungen durch den entsprechenden Commit darstellt. Gelöschte Zeilen sind dabei rot, grüne wurden neu hinzugefügt.

In beiden Ansichten finden sich die Branching-Optionen oben in der Mitte:

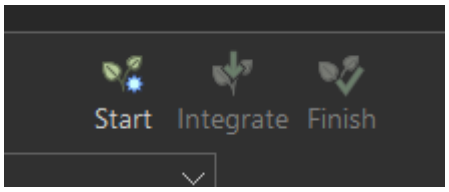


Abb. 25. BranchingOptions

Branches werden hier Feature genannt.

Mergen und Mergekonflikte

Um den Mergevorgang nachzustellen wurde folgendes Szenario erschaffen: Eine Datei wurde von zwei Rechnern gleichzeitig bearbeitet und hinterher zunächst von einem erfolgreich gepusht. Der zweite Rechner versucht nun seine Version ebenfalls zu pushen.

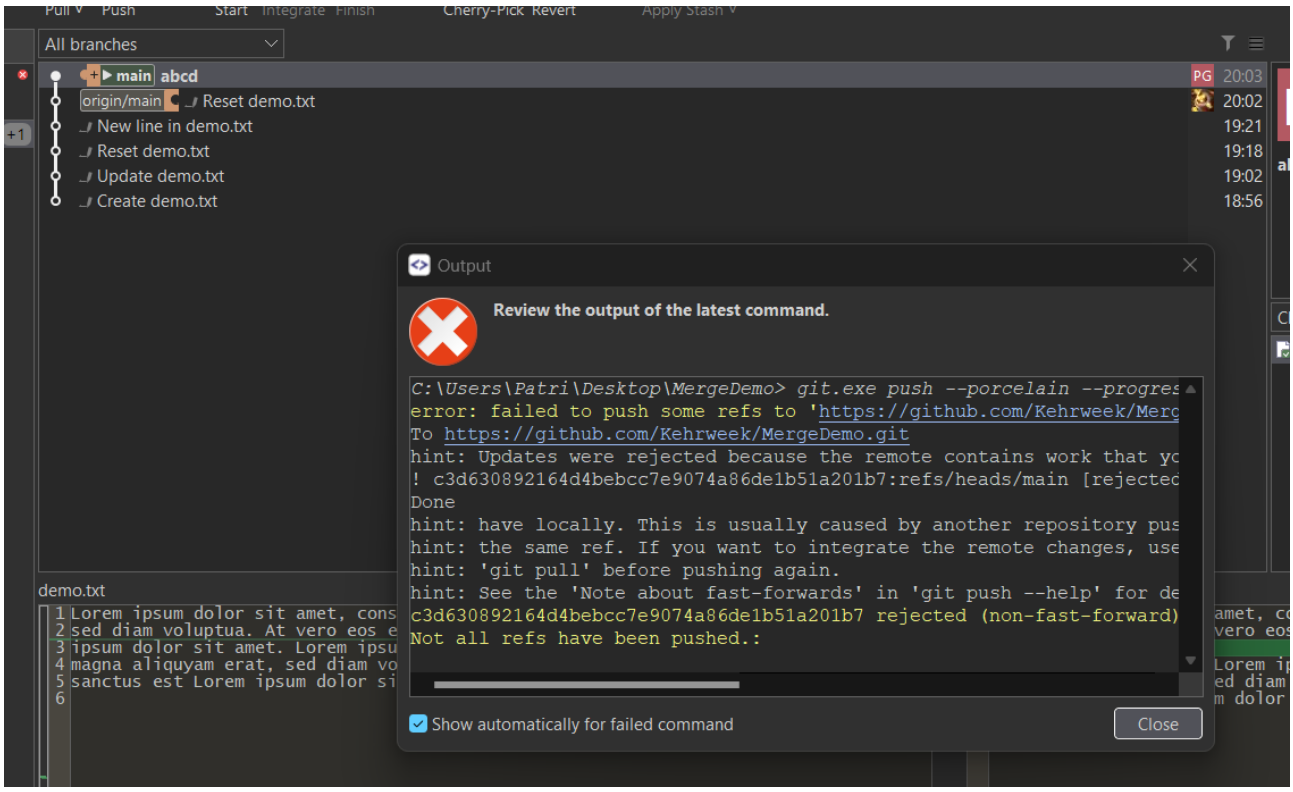


Abb. 26. Smartgit Mergekonflikt

Daraufhin erstellt Smartgit eine Auflistung der Mergekonflikte.

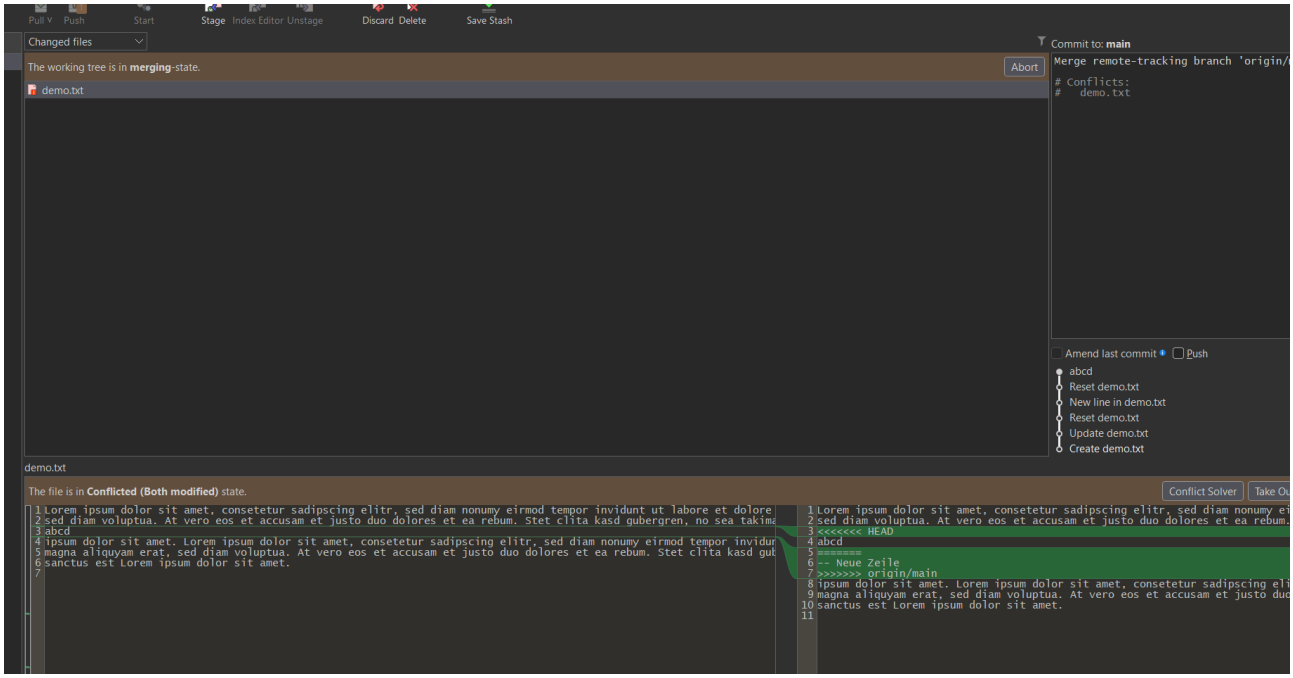


Abb. 27. Smartgit Merging Teil 1

Außerdem werden verschiedene Buttons angeboten um den Konflikt zügig zu lösen.

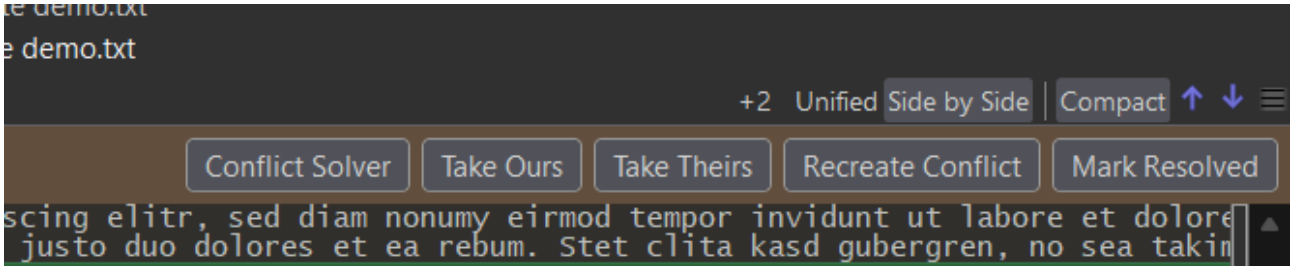


Abb. 28. Smartgit Merging Teil 2

Durch Klicken auf die Datei öffnet sich ein detailliertes Fenster um in dieser jeweils zeilenweise zu entscheiden welcher Teil übernommen werden soll.

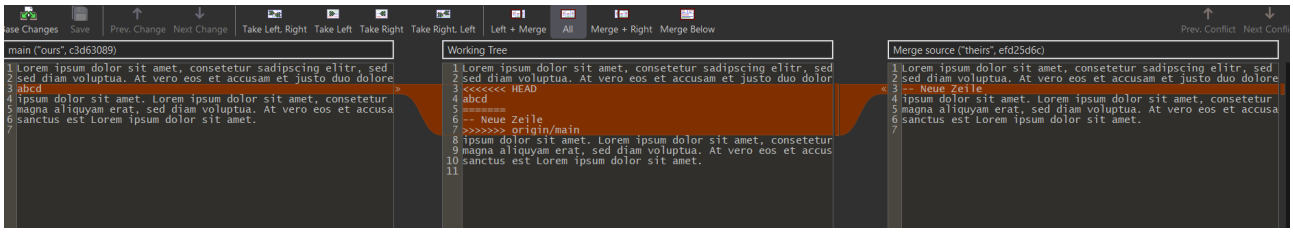


Abb. 29. Smartgit Merging Teil 3

Die Entscheidung wird durch das Klicken auf die kleinen Pfeile getätigt.,

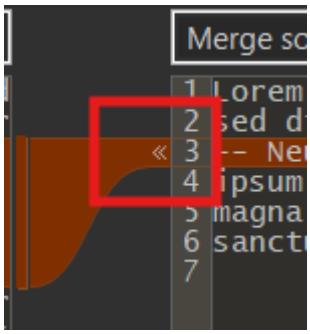


Abb. 30. Smartgit Merging Teil 4

Ist der Konflikt entschieden wird die ignorierte / herausgelöschte Seite rot dargestellt.

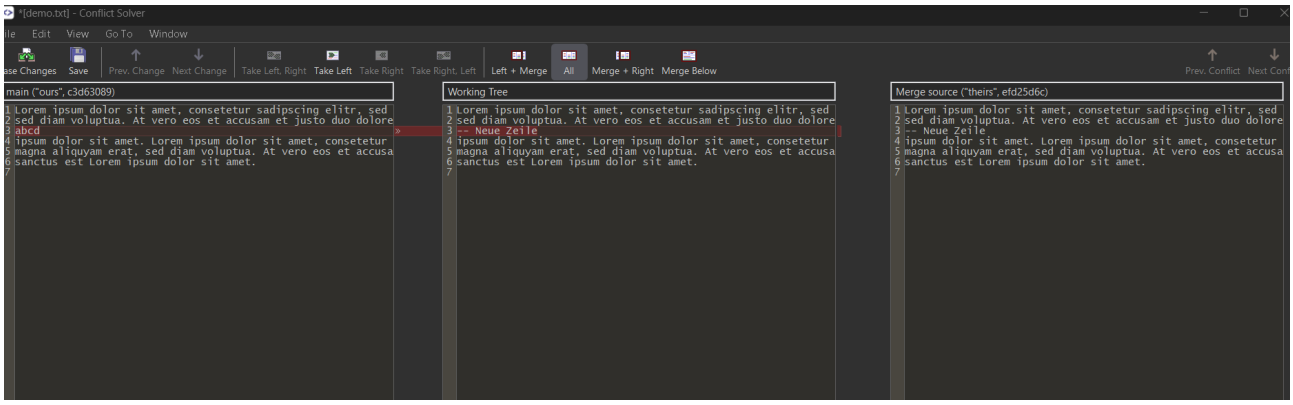


Abb. 31. Smartgit Merging Teil 5

Sind mehrere Konflikte vorhanden kann über diese beiden Pfeile zwischen diesen gewechselt werden.

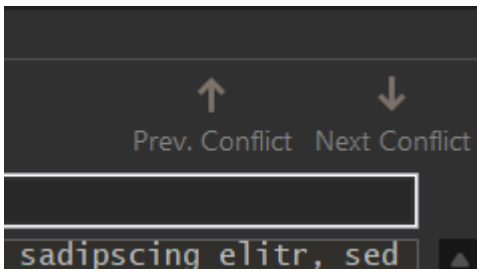


Abb. 32. Smartgit Merging Teil 6

Ist die Konfliktentscheidung abgeschlossen muss die entsprechende Datei als resolved markiert werden.

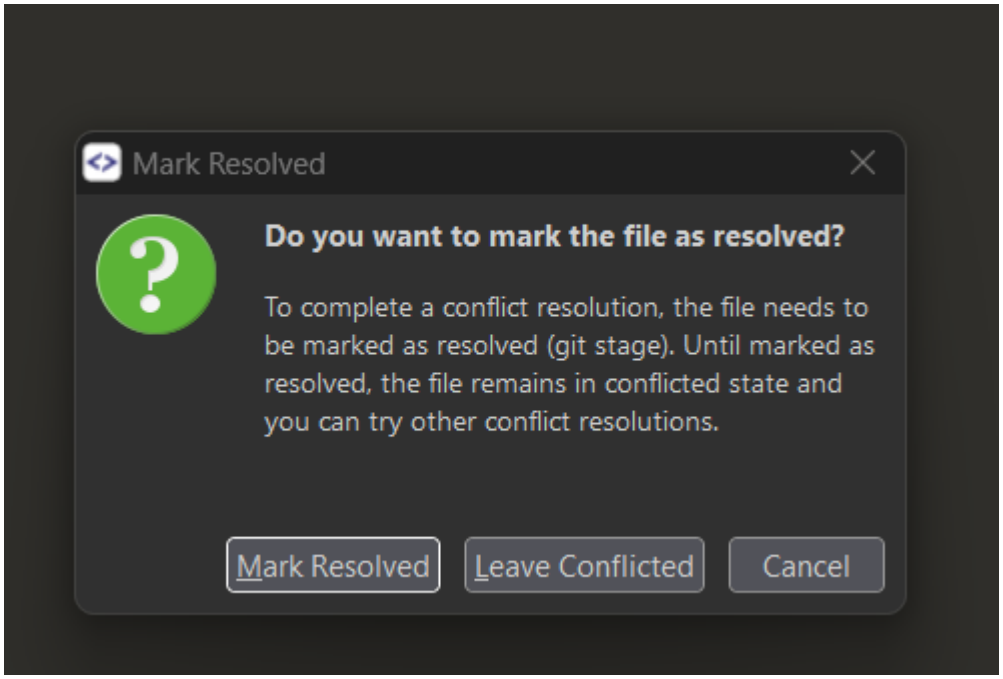


Abb. 33. Smartgit Merging Teil 7

Der gelöste Mergekonflikt wird hinterher in der History durch die Vereinigung der jeweiligen Branches visualisiert.

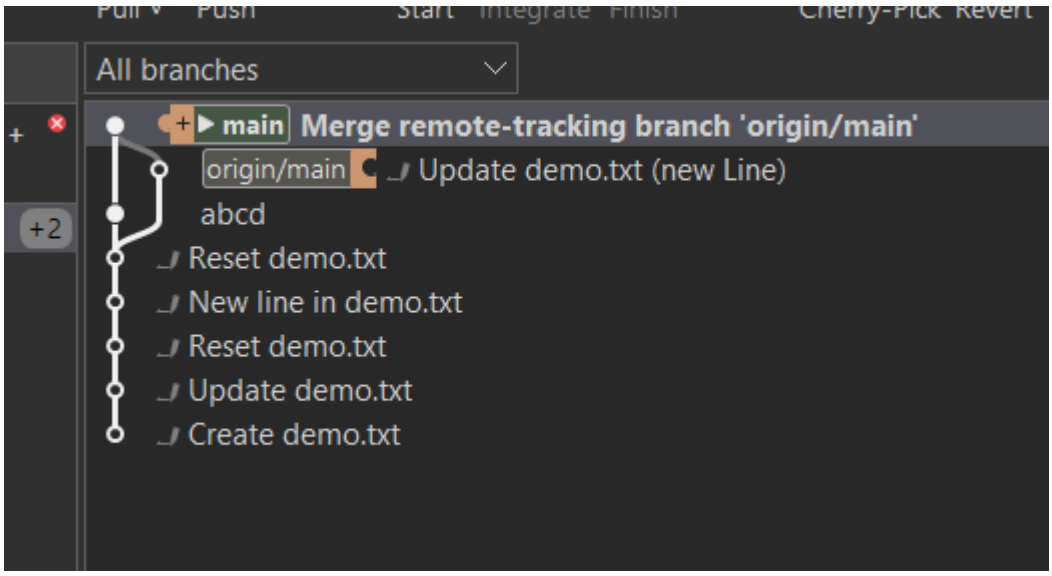


Abb. 34. Smartgit Merging Teil 8

Entwickelt man ohne entsprechende Mergekonflikte werden Branches über Pull-Requests auf der Website vereinigt.

3.2. Git-Konsole

3.2.1. Anlegen eines Repositories

Um ein neues Git-Repository mit der Git-Konsole anzulegen, befolgen Sie bitte diese Schritte:

Konsole starten

Starten Sie Ihre bevorzugte Git-Konsole. Dies könnte die Befehlszeile (Command Line) oder eine integrierte Konsole in Ihrer Entwicklungsumgebung sein, wie zum Beispiel Git Bash.

Navigation

Verwende den Befehl `cd` (Change Directory), um zum gewünschten Speicherort zu navigieren, an dem Sie das neue Repository erstellen möchten.

Initialisierung

Verwenden Sie den Befehl `git init`, um ein neues Git-Repository im aktuellen Verzeichnis zu initialisieren. Dieser Befehl legt ein neues leeres Repository an.

Verzeichnis prüfen

Bestätigen Sie, dass das neue Git-Repository erfolgreich initialisiert wurde, indem Sie den Befehl `ls -a` (List All) verwenden, um den Inhalt des aktuellen Verzeichnisses anzuzeigen. Sie sollten ein verstecktes `.git`-Verzeichnis sehen, das das Repository enthält.

3.2.2. Clonen eines Repositories

Wenn Sie ein vorhandenes Git-Repository auf Ihrem lokalen System klonen möchten, können Sie dies mit der Git-Konsole tun. Hier sind die Schritte dazu:

Konsole starten

Öffnen Sie die Git-Konsole Ihrer Wahl. Sie können die Befehlszeile (Command Line) verwenden oder eine integrierte Konsole in Ihrer Entwicklungsumgebung wie Git Bash oder das integrierte Terminal.

Navigation

Verwenden Sie den Befehl `cd` (Change Directory), um zum Verzeichnis zu navigieren, in dem Sie das geklonte Repository speichern möchten.

Klonvorgang anstoßen

Verwenden Sie den Befehl `git clone` gefolgt von der URL des Repositorys, das Sie klonen möchten. Zum Beispiel:

```
git clone https://fortbildner.gitcamp-bw.de/ts-zsl-rska/GIT-Fortbildung.git
```

Hier werden Sie nach Ihren Login-Daten gefragt, welche Sie eingeben müssen, um Zugang zu den Dateien zu erlangen.

Erfolg prüfen

Sobald der Klonvorgang abgeschlossen ist, können Sie mit `ls` den Inhalt des aktuellen Verzeichnisses überprüfen. Sie sollten das geklonte Repository als neues Verzeichnis sehen.

3.2.3. Stage / Commit

Wenn Sie Änderungen an Ihrem Git-Projekt vorgenommen haben und diese für einen Commit vorbereiten möchten, können Sie dies einfach über die Git-Konsole tun. Hier sind die Schritte dazu:

Überprüfen der Änderungen

Verwenden Sie den Befehl `git status`, um den aktuellen Status Ihres Repositorys zu überprüfen. Dadurch erhalten Sie eine Liste der geänderten, ungestageten Dateien.

Stagen der Änderungen

Verwenden Sie den Befehl `git add`, um die gewünschten Änderungen zur Staging-Area hinzuzufügen. Sie können einzelne Dateien oder Verzeichnisse hinzufügen, indem Sie ihren Pfad angeben, oder alle Änderungen auf einmal mit einem Punkt (`.`) für alle Dateien im

aktuellen Verzeichnis.

Beispiel für das Stagen einer einzelnen Datei:

```
git add Dateiname
```

Beispiel für das Stagen aller Änderungen im aktuellen Verzeichnis:

```
git add .
```

Überprüfen der gestagten Änderungen

Verwenden Sie erneut den Befehl `git status`, um sicherzustellen, dass die gewünschten Änderungen erfolgreich zur Staging-Area hinzugefügt wurden. Sie sollten eine Liste der gestagten Änderungen sehen.

Commiten der Änderungen

Verwenden Sie den Befehl `git commit`, um die gestagten Änderungen zu committen. Geben Sie eine aussagekräftige Commit-Nachricht ein, um die durchgeführten Änderungen zu beschreiben.

```
git commit -m "Hier ist Ihre Commit-Nachricht"
```

Überprüfen des Commit-Erfolgs

Nachdem Sie die Änderungen committet haben, können Sie mit `git log` den Commit-Verlauf anzeigen und sicherstellen, dass Ihr Commit erfolgreich war.

3.2.4. Check out

Wenn Sie an einem bestimmten Branch oder einem früheren Commit in Ihrem Git-Repository arbeiten möchten, können Sie dies über die Git-Konsole tun. Hier sind die Schritte dazu:

Überprüfen des Repository-Status

Verwenden Sie den Befehl `git status`, um den aktuellen Status Ihres Repositorys zu überprüfen. Dadurch erhalten Sie Informationen darüber, ob Sie Änderungen haben, die committet oder gestaged werden müssen.

Aus-checken des Gewünschten Branchs oder Commits

Verwenden Sie den Befehl `git checkout`, um zum gewünschten Branch oder Commit zu wechseln. Geben Sie den Namen des Branchs oder die Commit-ID an, zu der Sie wechseln möchten.

Beispiel für das Auschecken eines Branchs:

```
git checkout Branch-Name
```

Beispiel für das Auschecken eines früheren Commits:

```
git checkout Commit-ID
```

Überprüfen des Wechsel-Erfolgs

Nachdem Sie zum gewünschten Branch oder Commit gewechselt haben, verwenden Sie `git status`, um sicherzustellen, dass der Wechsel erfolgreich war und Ihr Arbeitsverzeichnis auf dem neuen Stand ist.

3.2.5. Push

enn Sie Ihre lokalen Änderungen an ein entferntes Git-Repository hochladen möchten, können Sie dies über die Git-Konsole tun. Hier sind die Schritte dazu:

Überprüfen des Repository-Status

Verwenden Sie den Befehl `git status`, um den aktuellen Status Ihres Repositorys zu überprüfen. Dadurch erhalten Sie Informationen darüber, ob Sie Änderungen haben, die committet oder gestaged werden müssen.

Pushen der Änderungen

Verwenden Sie den Befehl `git push`, um Ihre lokalen Änderungen auf das entfernte Repository hochzuladen. Geben Sie den Namen des entfernten Repositories und den Namen des Branchs an, auf den Sie pushen möchten.

Beispiel für das Pushen auf den Hauptbranch (üblicherweise "master" oder "main"):

```
git push origin master
```

Authentifizierung (falls erforderlich)

Je nach Konfiguration des entfernten Repositories kann es sein, dass Sie sich authentifizieren müssen, um den Push-Vorgang abzuschließen. Geben Sie Ihre Anmeldeinformationen ein, wenn Sie dazu aufgefordert werden.

Überprüfen des Push-Erfolgs

Nachdem der Push-Vorgang abgeschlossen ist, können Sie die Repository-Website besuchen oder den Befehl `git log` verwenden, um sicherzustellen, dass Ihre Änderungen erfolgreich auf

das entfernte Repository hochgeladen wurden.

3.2.6. Pull

Wenn Sie die neuesten Änderungen aus einem entfernten Git-Repository auf Ihr lokales Repository herunterladen möchten, können Sie dies über die Git-Konsole tun. Hier sind die Schritte dazu:

Pullen der neuesten Änderungen

Verwenden Sie den Befehl `git pull`, um die neuesten Änderungen aus dem entfernten Repository herunterzuladen und mit Ihrem lokalen Repository zu fusionieren. Geben Sie den Namen des entfernten Repositories und den Namen des Branchs an, den Sie pullen möchten.

Beispiel für das Pullen aus dem Hauptbranch (üblicherweise "master" oder "main"):

```
git pull origin master
```

Authentifizierung (falls erforderlich)

Je nach Konfiguration des entfernten Repositories kann es sein, dass Sie sich authentifizieren müssen, um den Pull-Vorgang abzuschließen. Geben Sie Ihre Anmeldeinformationen ein, wenn Sie dazu aufgefordert werden.

Überprüfen des Pull-Erfolgs

Nachdem der Pull-Vorgang abgeschlossen ist, verwenden Sie den Befehl `git log` oder andere Git-Befehle, um sicherzustellen, dass die neuesten Änderungen erfolgreich in Ihr lokales Repository gezogen wurden.

3.2.7. Merge / Rebase

Wenn Sie Änderungen aus einem anderen Branch in Ihren aktuellen Branch integrieren möchten, können Sie dies über die Git-Konsole tun. Hier sind die Schritte dazu:

Navigieren zum Ziel-Branch

Verwenden Sie den Befehl `git checkout`, um zum Branch zu wechseln, in den Sie die Änderungen integrieren möchten. Stellen Sie sicher, dass Sie in dem Branch sind, in den Sie die Änderungen mergen möchten.

Beispiel für das Wechseln zum Ziel-Branch:

```
git checkout Ziel-Branch-Name
```

Mergen des Quell-Branchs

Verwenden Sie den Befehl `git merge`, um den Quell-Branch in den Ziel-Branch zu mergen. Geben Sie den Namen des Quell-Branchs an, den Sie mergen möchten.

Beispiel für das Mergen des Quell-Branchs:

```
git merge Quell-Branch-Name
```

Bearbeiten von Merge-Konflikten (falls erforderlich)

Wenn es Merge-Konflikte gibt, die nicht automatisch gelöst werden können, müssen Sie diese manuell bearbeiten. Öffnen Sie die betroffenen Dateien in einem Texteditor, beheben Sie die Konflikte und führen Sie dann den Merge-Vorgang erneut aus. Dieser Schritt wird durch moderne Entwicklungsumgebungen stark vereinfacht.

3.2.8. Branches

Branches dienen dazu einzelne Features bzw. Teilprojekte separat anzulegen ohne dabei die Integrität des Hauptprojektes zu gefährden. In einer kollaborativen Entwicklungssituation sind sie unabdingbar.

anlegen

Sollten Sie einen bisherigen branch um einen neuen erweitern wollen, so sind folgende Schritte zu beachten:

Überprüfen des Repository-Status

Verwenden Sie den Befehl `git status`, um den aktuellen Status Ihres Repositorys zu überprüfen. Dadurch erhalten Sie Informationen darüber, ob Sie ungespeicherte Änderungen haben, die vor dem Branchen committet oder gestaged werden müssen.

Anlegen des Neuen Branchs

Verwenden Sie den Befehl `git branch`, um einen neuen Branch anzulegen. Geben Sie den Namen des neuen Branchs an, den Sie erstellen möchten.

Beispiel für das Anlegen eines neuen Branchs:

```
git branch Neuer-Branch-Name
```

Wechseln zum Neuen Branch

Verwenden Sie den Befehl `git checkout`, um zum neu erstellten Branch zu wechseln und dort zu arbeiten.

Beispiel für das Wechseln zum neuen Branch:

```
git checkout Neuer-Branch-Name
```

Überprüfen des Branch-Erfolgs

Nachdem Sie den neuen Branch erstellt haben, verwenden Sie den Befehl `git branch` erneut oder andere Git-Befehle, um sicherzustellen, dass der neue Branch erfolgreich erstellt wurde und Sie sich in ihm befinden.

mergen

Wenn Sie die Änderungen aus einem anderen Branch in Ihren aktuellen Branch integrieren möchten, können Sie dies über die Git-Konsole tun. Hier sind die Schritte dazu:

Überprüfen des Repository-Status

Verwenden Sie den Befehl `git status`, um den aktuellen Status Ihres Repositorys zu überprüfen. Dadurch erhalten Sie Informationen darüber, ob Sie ungespeicherte Änderungen haben, die vor dem Mergen committet oder gestaged werden müssen.

Wechseln zum Ziel-Branch

Verwenden Sie den Befehl `git checkout`, um zum Branch zu wechseln, in den Sie die Änderungen mergen möchten. Stellen Sie sicher, dass Sie sich im Ziel-Branch befinden, in den Sie die Änderungen integrieren möchten.

Beispiel für das Wechseln zum Ziel-Branch:

```
git checkout Ziel-Branch-Name
```

Mergen des Quell-Branchs

Verwenden Sie den Befehl `git merge`, um den Quell-Branch in den Ziel-Branch zu mergen. Geben Sie den Namen des Quell-Branchs an, den Sie mergen möchten.

Beispiel für das Mergen des Quell-Branchs:

```
git merge Quell-Branch-Name
```

Bearbeiten von Merge-Konflikten (falls erforderlich)

Wenn es Merge-Konflikte gibt, die nicht automatisch gelöst werden können, müssen Sie diese manuell bearbeiten. Öffnen Sie die betroffenen Dateien in einem Texteditor oder Entwicklungsumgebung, beheben Sie die Konflikte und führen Sie dann den Merge-Vorgang erneut aus.

Überprüfen des Merge-Erfolgs

Nachdem der Merge-Vorgang abgeschlossen ist, verwenden Sie den Befehl `git status` oder andere Git-Befehle, um sicherzustellen, dass der Merge erfolgreich war und keine Konflikte mehr vorhanden sind.

4. GIT auf der Konsole

Quelle: Diese Dokumentation und die darin enthaltenen Bilder beruhen alle auf dem Material von Frank Schiebel (https://info-bw.de/faecher:informatik:oberstufe:git:start). Dort finden sich auch weitergehende Anleitungen für die Arbeit mit GIT auf der Konsole.

Nach der Installation von GIT stehen eine GIT-Konsole und eine einfache Git-GUI zur Verwaltung der Repositorys zur Verfügung. Für die produktiven Arbeit werden in der Regel in die Entwicklungsumgebungen (z. B. bei IntelliJ oder Visual Studio Code) integrierte GIT-Clients verwendet. Es gibt auch spezielle Git-Verwaltungsprogramme, die eine umfangreichere Verwaltung der GIT-Projekte zulassen (z. B. Git-Cola, Smart-Git).

Im Unterricht kann es sinnvoll sein, zunächst mit der GIT-Konsole zu starten. In der GIT-Konsole müssen alle Befehle von Hand eingetippt werden. Daher wird jeder Befehl mit mehr Bedacht ausgeführt. In einer GUI ist schnell mal ein falscher Klick gemacht, der nur schwer rückgängig gemacht werden kann. Hat man allerdings die Arbeitsweise von GIT verstanden, ist es wesentlich komfortabler, mit einer GUI zu arbeiten.

4.1. Konfiguration

Bevor mit git gearbeitet werden kann, müssen Name und Mailadresse festgelegt werden. Dazu sind in einer Shell die folgenden Kommandos auszuführen. Unter Windows ist das Programm "Git Bash" zu starten, in Linux/MacOS reicht ein gewöhnliches Terminal aus. Name und Mailadresse sind durch passende Werte zu ersetzen.

```
$ git config --global user.name "Max Mustermann"
$ git config --global user.email max@example.org
```

Sollte keine Berechtigung bestehen, diese Einstellungen systemweit ("global") vorzunehmen, z.B. an den PCs in der Schule, ist es erforderlich, sie für jedes Repository einzeln festzulegen, **nachdem** dieses initialisiert wurde:

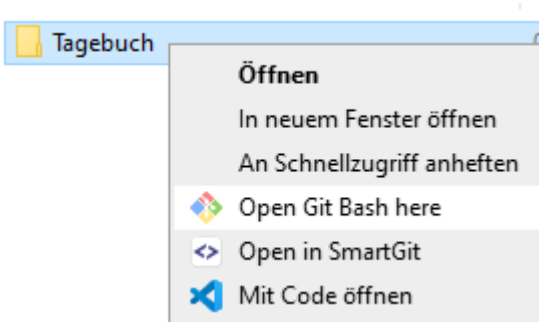
```
$ git config user.email "meine@mail.adresse.hier"
$ git config user.name "John Doe"
```

Diese Befehle speichern die Einstellungen nur für das Repository, in dem gerade gearbeitet wird. Für ein weiteres Repository muss die Email-Adresse erneut konfiguriert werden.

4.2. Erste Schritte mit Git

4.2.1. Initialisieren

Um die Abläufe und die Funktionsweise zu erproben, soll zunächst ein Verzeichnis unter Versionskontrolle gestellt werden, in dem ein Tagebuch angelegt wird. Es wird also ein Verzeichnis `tagebuch` erstellt und dort ein Git-Repository initialisiert:




```
$ git init
Initialized empty Git repository in F:/Tagebuch/.git/
```

Nun steht das Verzeichnis `tagebuch` unter Versionskontrolle. Das lokale Git-Repository befindet sich im Unterverzeichnis `.git`:

```
$ ls -la
total 52
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:19 ./
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:18 ../
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:19 .git/
```

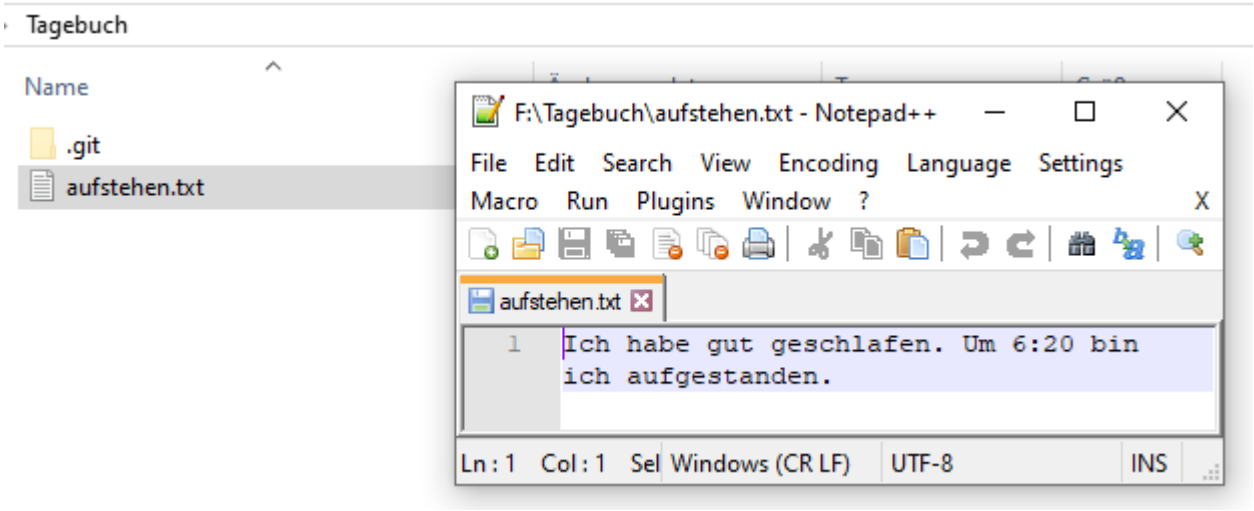
4.2.2. Repository Status anzeigen lassen

Das Verzeichnis `tagebuch` ist jetzt ein "git-Repository" - es wird von git "beobachtet", so dass Änderungen in diesem Verzeichnis und seinen Unterverzeichnissen nachverfolgt werden können. Mit dem Befehl `git status` kann der aktuelle Status des "Repos" angezeigt werden[1] :

```
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

4.2.3. Ein erster Tagebucheintrag

Mit einem Texteditor (z. B. notepad++ bei Windows oder kate bei Linux, **nicht** mit Word oder Writer!) wird eine Datei `aufstehen.txt` angelegt. In diese Datei kann beispielsweise hineingeschrieben werden, wie geschlafen wurde und wann aufgestanden wurde. Wichtig ist, dass die Datei im Verzeichnis `tagebuch` abgespeichert wird.

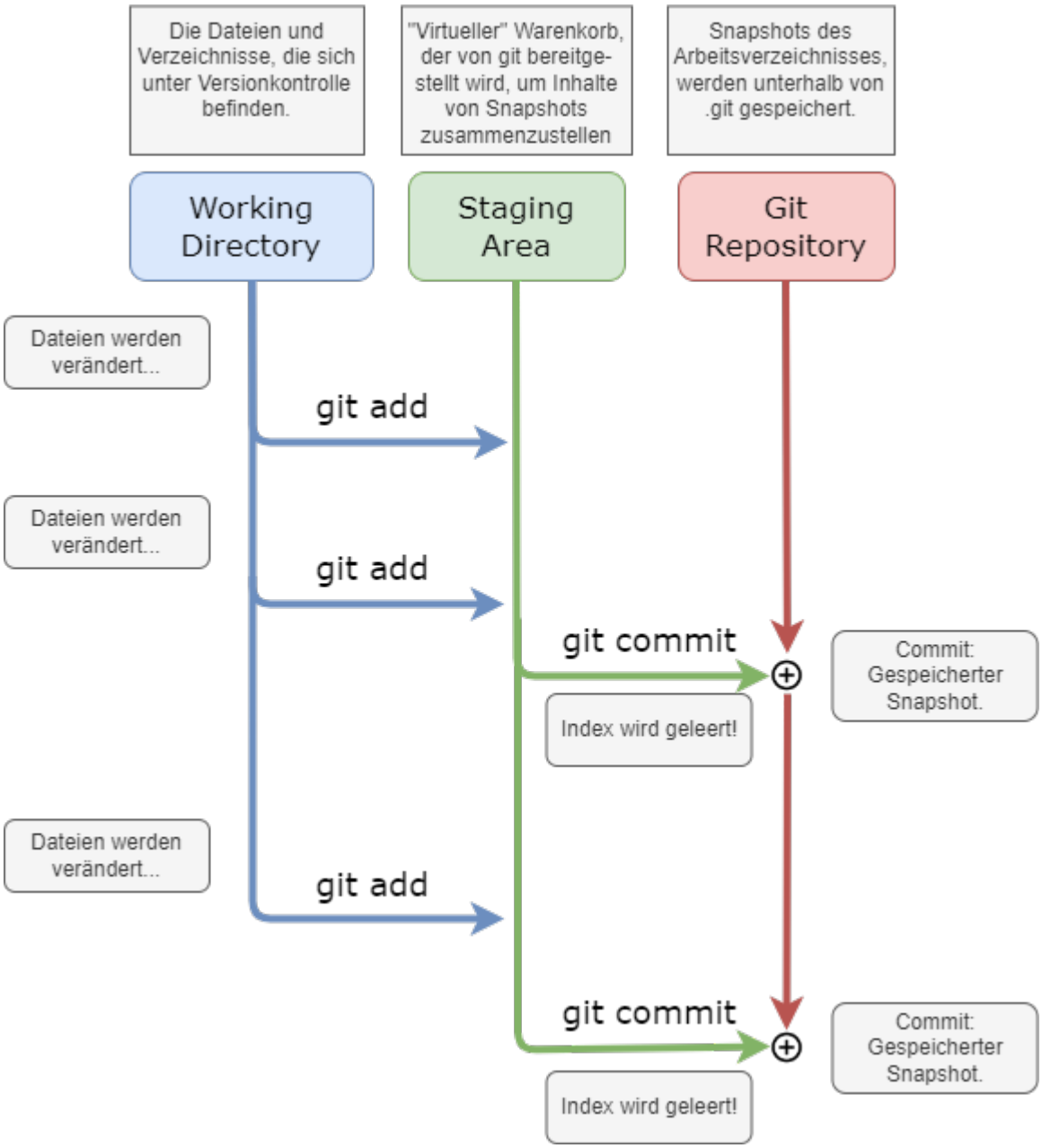


Das Tagebuch enthält nun einen Eintrag in `aufstehen.txt`. Es soll jetzt der Zustand des Tagebuchs an dieser Stelle so in der Versionsverwaltung festgehalten werden, dass er später wieder verwendet werden kann.

4.2.4. Ein erster Commit

Um den git-Workflow zu verstehen, müssen drei Begriffe unterschieden werden: Das Arbeitsverzeichnis ("Working Directory"), die Staging Area ("Index") und das eigentliche Repository.

1. **Arbeitsverzeichnis ("Working Directory"):** Das ist das Verzeichnis, welches zuvor mit `git init` unter Versionskontrolle gestellt wurde, mit allen seinen Dateien und Unterverzeichnissen, so wie es auf der Festplatte vorgefunden wird. Das "spezielle" Verzeichnis `.git` wird dabei ignoriert, es dient der internen Verwaltung der Abläufe durch git.
2. **Staging Area ("Index")**: In der Staging Area werden zunächst alle Dateien zusammengetragen, die in einem nächsten Schritt zu einem Snapshot zusammengefasst und im Repository gespeichert werden sollen. Es gibt zahlreiche Anwendungsfälle, bei denen nicht alle Änderungen des Arbeitsverzeichnisses in einem Snapshot festgehalten werden möchten: Es kann z.B. sinnvoll sein, die Änderungen auf mehrere Snapshots aufzuteilen oder Dateien auszuschließen, die gar nicht unter Versionskontrolle gestellt werden sollen, beispielsweise Compilate von Java Programmen (class-Dateien). Daher ist es sinnvoll, zunächst über die Staging Area auszuwählen, welche Dateien gesichert werden sollen.
3. **Repository:** Wenn in der Staging Area alle Dateien für den nächsten Snapshot zusammengestellt wurden, kann ein neuer Snapshot erstellt werden. Ein solcher Snapshot heißt **Commit** und wird durch einen Hashwert identifiziert, außerdem werden Metainformationen wie Zeit und Name des Commiters festgehalten. Ein Commit wird mit dem Befehl `git commit` durchgeführt. Bei einem Commit wird außerdem die Staging Area wieder geleert, da alle Änderungen, die dort vorgemerkt waren, in den Snapshot überführt wurden. Für den nächsten Commit müssen die Dateien dort wieder hinzugefügt werden.



4.2.5. Commit: Schritt für Schritt

Neue Dateien befinden sich zunächst "nur" im Arbeitsverzeichnis und werden von git ignoriert. Mit `git status` kann dies überprüft werden, solche Dateien tauchen dort in der Liste der "Unversionierten Dateien" auf, für das Tagebuch sieht das so aus:

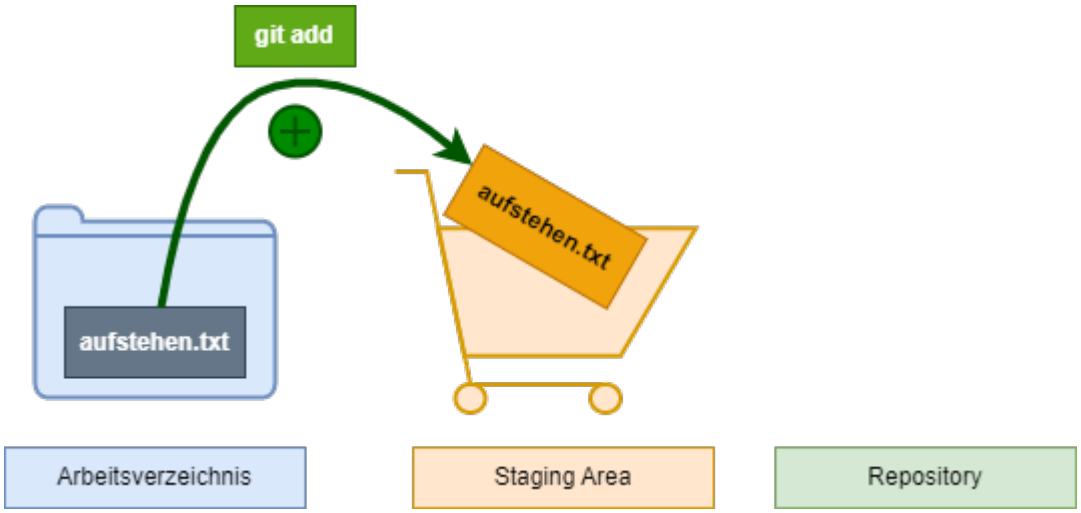
```
$ git status
On branch main

No commits yet

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

```
nothing added to commit but untracked files present (use "git add" to track)
```

Mit dem Befehl `git add` wird eine Datei im Index vorgemerkt - dies kann man sich wie ein Einkaufswagen vorstellen, in dem neue Dateien und Änderungen gesammelt werden, bis ein Punkt erreicht ist, den man sich "merken" möchte.

Im Folgenden wird die einzige Datei `aufstehen.txt` zum Index hinzugefügt:

```
$ git add aufstehen.txt
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   aufstehen.txt
```

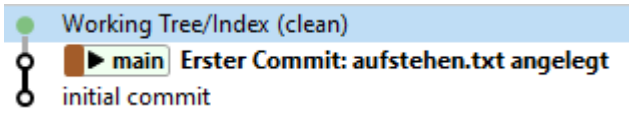


Wenn man mit den im Index vorgemerkten Änderungen zufrieden ist, kann ein "Commit" durchgeführt werden, um den Zustand aller im Index befindlichen Dateien zu speichern. Mit dem Befehl `git commit -m "Erster Commit: aufstehen.txt angelegt"` wird ein Commit mit einer Commit-Message angelegt (Parameter `-m`).

```
$ git commit -m "Erster Commit: aufstehen.txt angelegt"
[main (root-commit) 9eefa56] Erster Commit: aufstehen.txt angelegt
 1 file changed, 1 insertion(+)
 create mode 100644 aufstehen.txt
```

Wenn der Status des Arbeitsverzeichnisses jetzt erneut abgefragt wird, erhält man folgende Ausgabe:

```
$ git status
On branch main
nothing to commit, working tree clean
```



Man erkennt, dass der Index wieder leer ist ("nichts zum Commit vorgemerkt"). Nun können weitere Änderungen im Tagebuch vorgenommen werden und zu allen wichtigen Zeitpunkten der Zustand der Dateien in einem Commit festgehalten werden.

4.2.6. Frühstück

Aufgabe:

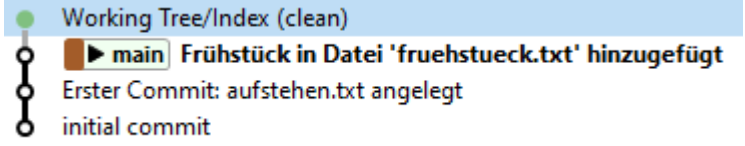
1. Halten Sie in der Datei `fruehstueck.txt` fest, was es zum Frühstück gab.
2. Kontrollieren Sie mit `git status`, dass die Datei jetzt existiert, sie jedoch noch nicht unter Versionskontrolle steht.
3. Fügen Sie die Datei `fruehstueck.txt` mit dem Befehl `git add fruehstueck.txt` zum Index hinzu.
4. Erstellen Sie einen Commit für das Frühstück. Vergessen Sie die Commit-Message nach der Option `-m` nicht.
5. Überprüfen Sie den Zustande des Repositorys erneut.

Es wurde nun ein zweiter Commit erstellt:

Es wurde zwar nur die Datei `fruehstueck.txt` zum Commit vorgemerkt und anschließend mit `git commit` "commited", **ein Commit beinhaltet jedoch stets den Zustand aller unter Versionskontrolle stehender Dateien im Arbeitsverzeichnis**, in diesem Fall ist in dem zweiten Commit also auch die (unveränderte) Datei `aufstehen.txt` enthalten!

Ein Commit kann also wie im Bild dargestellt als Archivbox betrachtet werden, in dem jeweils der Zustand aller versionierten Dateien festgehalten ist. Ein Commit wird durch einen hexadezimalen "Hashwert" identifiziert, das ist gewissermaßen die eindeutige Nummer eines Commits, z.B. `2f40bf7`. Mit dem Befehl `git log` können die Commits aufgelistet werden:

```
$ git log
commit 2f40bf7011e453259db1621014979353003262df (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Man erkennt hier auch, dass die eigentlichen Commit-Hashes sehr viel länger sind, als das Beispiel oben vermuten lässt; für die Identifizierung eines Commits reichen die ersten 7 Stellen des Hashes aus.

4.2.7. Mittagessen

Aufgabe:

1. Fügen Sie dem Tagebuch den Eintrag `mittagessen.txt` als Datei hinzu, zunächst ohne diese zu versionieren.
2. Ändern Sie die Datei `fruehstueck.txt` und schreiben Sie zusätzlich `Schokolade` in die Datei.
3. Überprüfen Sie mit `git status` den Zustand des Repositorys. Das Repo sollte ungefähr so aussehen:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   fruehstueck.txt

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

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

Es wurden jetzt **zwei** Dinge geändert: In der Datei `fruehstueck.txt` wurde eine Änderung vorgenommen. Die Datei `mittagessen.txt` wurde neu hinzugefügt.

Wenn nun der nächste Commit vorbereitet wird, kann mit dem Befehl `git add` wieder ausgewählt werden, welche Änderungen in den Commit übernommen werden. Um dies zu demonstrieren, werden die beiden vorgenommenen Änderungen im Folgenden auf zwei Commits aufgeteilt.

```
$ git add fruehstueck.txt

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   fruehstueck.txt

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

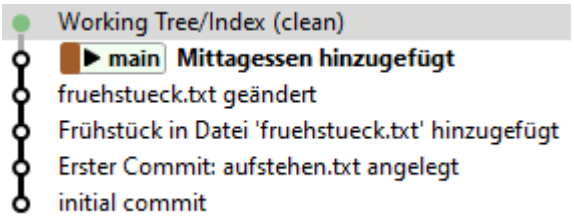
Jetzt wurden die Änderungen von `fruehstueck.txt` für den nächsten Commit vorgemerkt, die neue Datei `mittagessen.txt` wird allerdings nicht in diesen übernommen. Mit `git commit -m`

`"fruehstueck.txt geändert"` wird der Commit ausgeführt:

```
$ git commit -m "fruehstueck.txt geändert"
[main 1c669a2] fruehstueck.txt geändert
 1 file changed, 2 insertions(+), 1 deletion(-)

$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        mittagessen.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Für den nächsten Commit wird jetzt die Datei `mittagessen.txt` übernommen:

```
$ git add mittagessen.txt

$ git commit -m "Mittagessen hinzugefügt"
[main 311cb29] Mittagessen hinzugefügt
 1 file changed, 1 insertion(+)
 create mode 100644 mittagessen.txt

$ git status
On branch main
nothing to commit, working tree clean
```



Sie können nun Dateien selektiv in einem Verzeichnis unter Versionskontrolle stellen. Mit jedem Commit wird ein **Snapshot** des Zustands erzeugt, den die versionierten Dateien zum Zeitpunkt des Commits haben. Dateien, die nicht mit `git add` unter Versionskontrolle gestellt wurden, werden von git nicht beachtet. Im folgenden Kapitel wird die Versionsgeschichte genauer untersucht und Zeitsprünge durchgeführt, um ältere Versionen zu betrachten.

4.3. Versionshistorie untersuchen

4.3.1. Blick in die Vergangenheit

Nachdem einige Commits gemacht wurden, kann nun das Repository untersucht werden:

```
$ git log
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db1621014979353003262df
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200
```

```
    Erster Commit: aufstehen.txt angelegt
```

Es werden einige Informationen für jedem Zeitpunkt angezeigt, an dem ein Snapshot der unter Versionskontrolle stehenden Dateien im Arbeitsverzeichnis erstellt wurde:

- Den Hashwert des Commits
- Den Autor
- Datum und Zeit, zu der der Snapshot erstellt wurde
- Commit-Message

Außerdem wird durch den Zeiger `HEAD` angezeigt, in welchem Zustand sich das Arbeitsverzeichnis befindet.

4.3.2. Änderungen zwischen Commits ansehen

Um den Unterschied des aktuellen Arbeitsstandes (HEAD) zu vorhergehenden zu untersuchen, kann folgender Befehl verwendet werden: `git diff HEAD~1`. Dies bedeutet: "Zeige alle Unterschiede im Verzeichnis zwischen dem Commit, auf den HEAD gerade zeigt, und dem vorigen Commit" - die Ausgabe ist zunächst etwas gewöhnungsbedürftig:

```
$ git diff head~1
diff --git a/mittagessen.txt b/mittagessen.txt
new file mode 100644
index 0000000..54777bb
--- /dev/null
+++ b/mittagessen.txt
@@ -0,0 +1 @@
+Blumenkohl
\ No newline at end of file
```

Die Ausgabe sagt, dass in der Datei mittagessen.txt neu angelegt wurde und in diese eine Zeile eingefügt wurde: Blumenkohl.

Der Vergleich kann auch mit weiter zurückliegenden Commits durchgeführt werden: z. B. zwei Commits in die Vergangenheit: `git diff HEAD~2`:

```
$ git diff head~2
diff --git a/fruehstueck.txt b/fruehstueck.txt
index 307fc9b..b7a56aa 100644
--- a/fruehstueck.txt
+++ b/fruehstueck.txt
@@ -1,3 +1,4 @@
 Müsli
 Kaffee
-Brot
```

```
\ No newline at end of file
+Brot^M
+Schokolade
\ No newline at end of file
diff --git a/mittagessen.txt b/mittagessen.txt
new file mode 100644
index 0000000..54777bb
--- /dev/null
+++ b/mittagessen.txt
@@ -0,0 +1 @@
+Blumenkohl
\ No newline at end of file
```

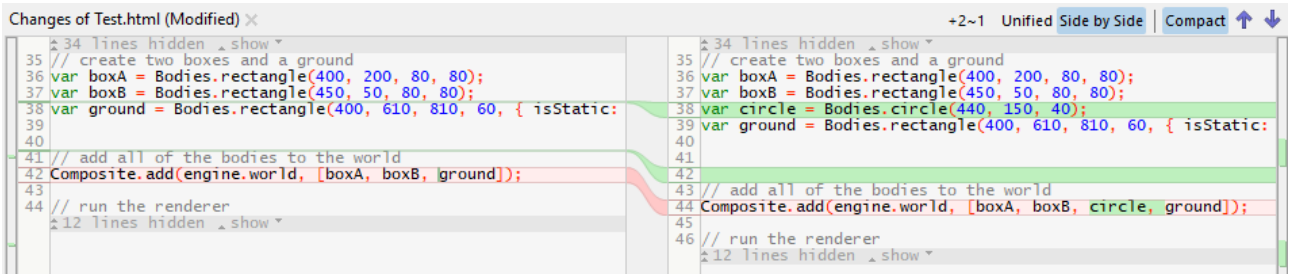
Die Ausgabe sagt, dass in den letzten zwei Commits bis zum aktuellen HEAD die folgenden Änderungen im Repo stattgefunden haben:

Es wurde eine neue Datei angelegt - mittagessen.txt und in diese eine Zeile eingefügt: Blumenkohl. In der zuvor bereits vorhandenen Datei fruehstueck.txt wurde nach den drei schon vorhandenen Zeilen eine weitere Zeile Schokolade eingefügt. Am Ende der Zeile Brot wurde außerdem ein Zeilenumbruch hinzugefügt.

Aufgabe:

1. Untersuchen Sie die Unterschiede in Ihrem Repo zwischen dem HEAD auf main und einigen vorigen Commits.
2. Erstellen Sie auf dem main Branch einen weiteren Commit, bei dem in einer Ihrer Dateien eine Zeile entfernt wird. Untersuchen Sie, wie die Ausgabe von git diff jetzt aussieht - woran erkennt man, dass die Zeile entfernt wurde?
3. Erstellen Sie einen Commit, bei dem eine Datei entfernt wird (git rm <Dateiname>, dann einen Commit erstellen). Untersuchen Sie, wie die Ausgabe von git diff jetzt aussieht.

Für eine bessere Darstellung der Unterschiede sollte eine GUI für GIT genutzt werden.

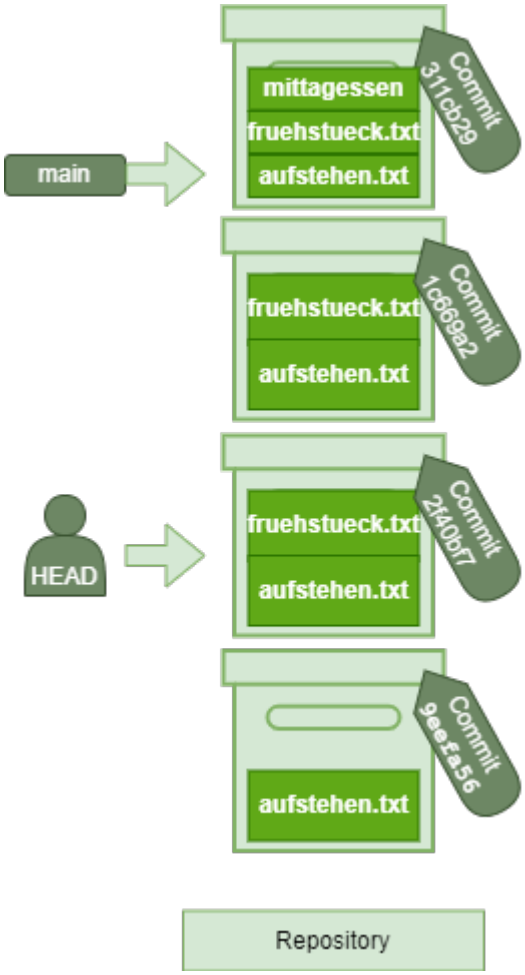


4.3.3. Reise in die Vergangenheit

Man sollte nur dann zu einem älteren Stand der Dateien zurückkehren, wenn das Arbeitsverzeichnis keine nicht committeten Änderungen beinhaltet. ("Sauberes Arbeitsverzeichnis")

```
$ git status
On branch main
nothing to commit, working tree clean
```

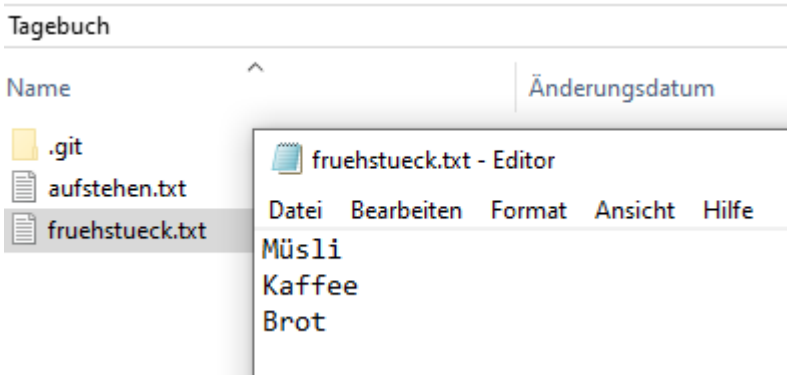
Der Zustand des Arbeitsverzeichnisses kann aus den Commits wiederhergestellt werden. Es ist also möglich, das Arbeitsverzeichnis in genau den Zustand zurückzuversetzen, in dem es beim Commit `2f40bf7011e453259db162101497935300 3262df` - "Frühstück in Datei 'fruehstueck.txt' hinzugefügt" war – oder eben zu jedem anderen Zeitpunkt, an dem der Zustand des Arbeitsverzeichnisses als Snapshot commitet wurde. Der Befehl dazu ist **`git checkout <Commit-Hash>`**. Es reicht dabei, die ersten 7 Zeichen des Hashwertes, z.B. `2f40bf7`, anzugeben:




```
$ git log
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (HEAD -> main)
...
commit 2f40bf7011e453259db162101497935300 3262df
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt
...
```

```
$ git checkout 2f40bf7
Note: switching to '2f40bf7'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
  git switch -c <new-branch-name>

Or undo this operation with:
  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 2f40bf7 Frühstück in Datei 'fruehstueck.txt' hinzugefügt

$ git status
HEAD detached at 2f40bf7
nothing to commit, working tree clean
```

Was ist denn jetzt passiert?



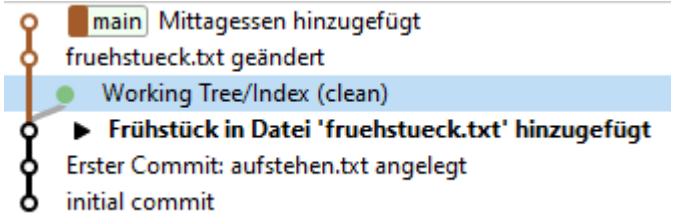
1. Zunächst einmal befinden sich die Dateien im Verzeichnis jetzt in dem Zustand, in dem sie waren, als der Commit `2f40bf7` erstellt wurde. Überprüfen Sie das.
2. Der HEAD ist zum Commit `2f40bf7` gewandert. HEAD ist ein Zeiger. Er springt in gewisser Weise in der Versionsgeschichte herum und zeigt immer auf denjenigen Commit, der im Arbeitsverzeichnis gerade "ausgecheckt" ist.
3. Die Versionsgeschichte ist kürzer geworden:

   ```
   $ git log
   commit 2f40bf7011e453259db16210149793530032 62df (HEAD)
   Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
   Date:   Wed Oct 9 14:35:50 2024 +0200
   ```

```
    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```

Git zeigt mit `git log` standardmäßig die Versionsgeschichte an, die zu dem Commit geführt hat, den man im Arbeitsverzeichnis gerade angezeigt bekommt. Die anderen Commits sind jedoch nicht verloren. Die Option `--all` zeigt dies. Es kann also auch in die Gegenwart zurückgekehrt werden.

```
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db16210149793530032 62df (HEAD)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Neben dem HEAD gibt es einen weiteren Zeiger `main`, der auf den aktuellen Stand der Arbeit[2] zeigt - hier auf den Commit `311cb29`.

Aufgabe:

1. Wechseln Sie durch einen Checkout zu den verschiedenen Zeitpunkten in der Versionsgeschichte und untersuchen Sie, welche Dateien vorhanden sind und welchen Text sie enthalten.

4.3.4. Manipulation der Vergangenheit

Was passiert, wenn nun die Vergangenheit geändert wird? Welche Auswirkungen hat dies auf die aktuelle Gegenwart? Die Antwort auf die zweite Frage ist kurz: keine! Wenn in der Versionsgeschichte zurückgegangen und Änderungen durchgeführt werden, wird immer eine parallele Zeitschiene begonnen. Diese parallele Zeitschiene hat eine neue Zukunft. Jede Zeitschiene ist ein sogenannter **Branch**. `Main` ist der Name des Standard-Branches. In der Schule sollte man nicht mit mehreren Branches arbeiten, da dies leicht zur Verwirrung führen kann.

Wenn mit checkout in die Vergangenheit zurückgekehrt wird, wird lediglich der HEAD verschoben. Es wird kein neuer Branch erstellt. Der HEAD ist also losgelöst (**detached HEAD**) von einem Branch. Man kann sich umsehen und auch Dateien verändern; wenn anschließend jedoch wieder in der Versiongeschichte "gesprungen" wird, gehen diese Änderungen verloren.

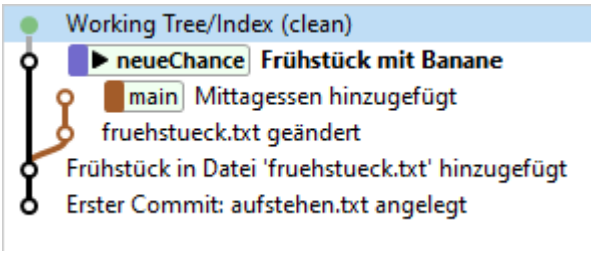
Wenn eine neue Zeitschiene begonnen wird, kann man sich entscheiden, was mit der aktuellen Zeitschiene passieren soll:

1. Sie ist weiterhin verfügbar, dann muss aber ein neuer Branch abgezweigt werden. (git switch)
2. Sie wird gelöscht und steht nicht mehr zur Verfügung (git reset).

Variante 1 (nur für Experten zu empfehlen): neuer Branch

```
$ git switch -c "neueChance"
Switched to a new branch 'neueChance'
$ git log --all
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db162101497935300 3262df (HEAD -> neueChance)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt
```

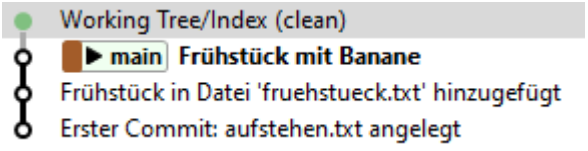
```
commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Neben main gibt es jetzt auch noch den Branch "neueChance". Der HEAD zeigt auf diesen Branch. Damit können Änderungen nun in diesem Branch commited werden. Eine parallele Zeitschiene beginnt. Mit `git switch <branchname>` kann zwischen den verschiedenen Zeitschienen hin- und hergewechselt werden.

Variante 2: die aktuelle Zeitschiene löschen Soll unwiderruflich zu einem alten Arbeitsstand zurückgekehrt werden, können die letzten Änderungen verworfen werden:

```
$ git reset --hard 2f40bf7
HEAD is now at 2f40bf7 Frühstück in Datei 'fruehstueck.txt' hinzugefügt

$ git log --all
commit 2f40bf7011e453259db1621014979353003262df (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



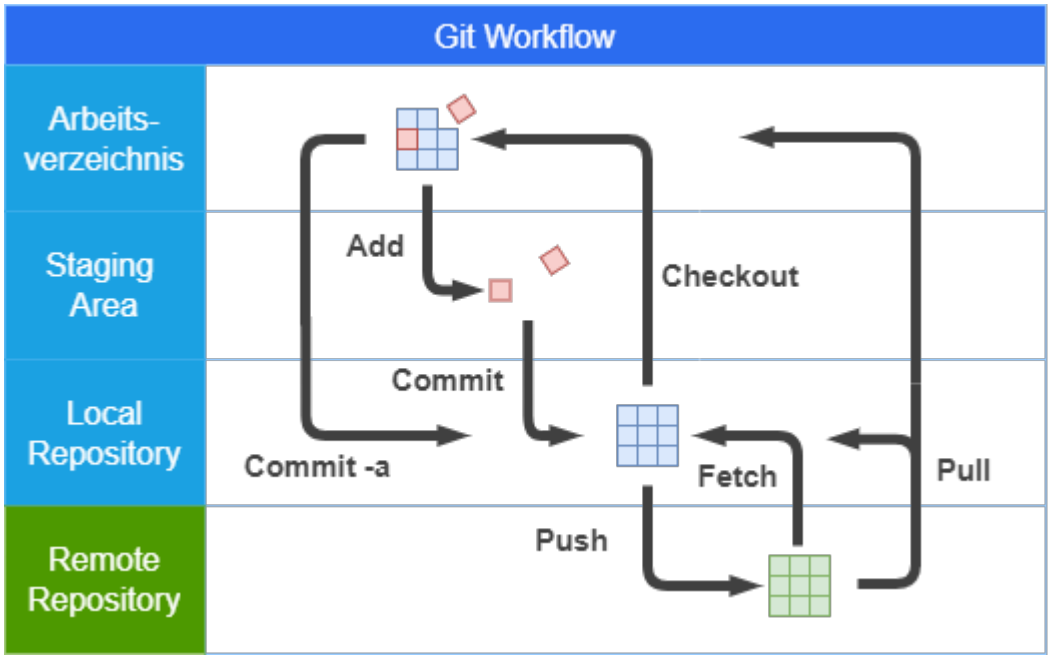
Der HEAD ist nicht losgelöst, da er weiterhin auf main zeigt. Die nachfolgenden Commits sind nicht mehr vorhanden und daher auch bei der Option `--all` nicht mehr zu sehen.

4.4. Speichern in der Cloud

Zunächst ist ein Git-Repo eine vollkommen lokale Angelegenheit - alle wichtigen Informationen

und die Snapshots werden im .git-Verzeichnis gespeichert.

Um besser zusammenarbeiten zu können, ist es möglich, ein Repo über einen Cloud-Speicher anderen zur Verfügung zu stellen. Das Land Baden-württemberg stellt für diesen Zweck eine datenschutzkonforme Plattform "Gitcamp" bereit.



4.4.1. Klonen eines Repositorys

Ein so veröffentlichtes Repo kann man "klonen". Dabei wird ein Repository aus der Cloud auf den eigenen Computer kopiert. Es wird in ein Unterverzeichnis kopiert, das genauso heißt, wie das Repository. Git merkt sich, von welcher Quelle das Repository stammt (origin) und auf welchem Arbeitsstand die Quelle ist (origin/main, origin/HEAD).

```
$ git clone https://fortierung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
Cloning into 'UebungVorlage'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 7.17 KiB | 152.00 KiB/s, done.

$ git log
commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977 (HEAD -> main, origin/main,
origin/HEAD)
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Der Begriff des Klonens ist hier wörtlich zu nehmen - jetzt existiert eine vollständige Kopie des

Repos auf dem lokalen Rechner, die alle Commits des ursprünglichen Repos nachverfolgbar enthält.

4.4.2. Änderung auf den Server zurückkopieren

Nun kann man mit dem Repo lokal ganz normal arbeiten, der wesentliche Unterschied zum ausschließlich "lokalen" Repository ist, dass dieses Repository "weiss", woher es kommt – das ermöglicht es, Änderungen auch wieder auf den entfernten Server zurück zu übertragen. Der dazu verwendete Befehl lautet `git push`.

Zunächst bearbeitet man lokal Dateien im Repo und erzeugt einen (oder mehrere Commits) :

```
$ git log
commit d81cb957d62bccb2f68378aea7f6232fc2f81026 (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt

commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977 (origin/main, origin/HEAD)
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Das Cloud-Verzeichnis (origin) bleibt dabei auf dem vorherigen Stand. Nur das lokale Repository (HEAD→main) kennt die Änderungen. Durch einen push werden diese Änderungen hochgeladen (origin ist nun auch auf dem neusten Stand):

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 324 bytes | 162.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://fortbildung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
   33f53e8..d81cb95  main -> main

$ git log
commit d81cb957d62bccb2f68378aea7f6232fc2f81026 (HEAD -> main, origin/main,
origin/HEAD)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt
```

```
commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Damit landen die Änderungen auf dem Server, von dem das Repo zuvor geklont wurde. Es darf aber natürlich nicht jeder auf jedes im Internet zugänglich Repository Änderung zurückspielen, sondern nur diejenigen, die dazu berechtigt sind.

Andere Personen oder man selbst an einem anderen Rechner kann diese Änderungen nun herunterladen und in das eigene, vorher geklonte Repository integrieren. Dazu gibt es den Befehl `git pull`.

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 260 bytes | 9.00 KiB/s, done.
From https://fortbildung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
   d81cb95..2ce4edf  main       -> origin/main
Updating d81cb95..2ce4edf
Fast-forward
 neu.txt | 1 +
 1 file changed, 1 insertion(+)

$ git log
commit 2ce4edf2df9254891d16dce195dc0a51fc0db357 (HEAD -> main, origin/main,
origin/HEAD)
Author: Thomas.Schaller <thomas.schaller@noreply.Domain>
Date:   Mon Oct 14 16:48:51 2024 +0200

    neu.txt aktualisiert

commit d81cb957d62bccb2f68378aea7f6232fc2f81026
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt

commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Hier wurden von einer anderen Person an der Datei `neu.txt` Änderungen vorgenommen und gepusht. Diese Änderungen stehen nun im lokalen Repository auch zur Verfügung und werden sofort in das Arbeitsverzeichnis kopiert. (HEAD → main, origin/main, origin/HEAD zeigen alle auf den gleichen Commit). Der Befehl `git fetch` lädt das Repository auch herunter, überträgt die Änderungen aber nicht sofort ins Arbeitsverzeichnis.

Nehmen mehrere Personen gleichzeitig Änderungen an einem Repository vor, kommt es in der Regel zu sogenannten Konflikten: mehrere Personen haben die gleiche Datei bearbeitet. Die verschiedenen Versionen müssen nun zusammengeführt werden (merge). Als Git-Neuling sollte man diesem Problem aus dem Weg gehen, indem man zunächst alleine an einem Repository arbeitet und bei der Arbeit an mehreren Rechnern, vor jedem Arbeitsschritt ein Pull durchführt und am Ende einer Arbeitsphase sofort einen Push macht.

5. Anwendungsszenarien Materialtausch

Für den Informatikunterricht gibt es eine Vielzahl von Fortbildungsmaterialien, die zum Teil von den Fortbildnern entwickelte Software oder Programmieranwendungen für den Unterricht enthalten. Diese werden fortlaufend weiterentwickelt und Fehler bereinigt.

GIT bietet die Möglichkeit, eine professionelle Versionsverwaltung dafür zu realisieren. Alle Lehrenden haben nun jederzeit die Möglichkeit, sich über neue Versionen zu informieren und die neuste Version abzurufen.

Lehrende soll auf strukturierte Weise Projekte aus der Lehrerfortbildung weiterentwickeln können und so zu der Weiterentwicklung der Unterrichtsmaterialien beitragen können.

Auch eigene, erprobte Projekte sollen von jedem Lehrenden anderen Lehrenden angeboten und getauscht werden können.

All dies kann mit GIT realisiert werden.

5.1. Dateitypen

GIT arbeitet textbasiert. Daher bieten sich textbasierte Dateiformate an, um eine effektive Versionskontrolle gewährleisten zu können.

5.1.1. Software

Bei Softwareprojekten ist dies kein Problem, da diese ohnehin alle textbasiert sind.

5.1.2. Skripte & Arbeitsblätter

Als Format für Skripte sollte in Zukunft ASCII-Doc verwendet werden. ASCII-Doc bietet die notwendigen Formatierungsbefehle für die üblichen Textgestaltung und entwickelt sich zu einen Standard weiter. Markdown wäre eine Alternative, die aber aufgrund vieler verschiedener Markdown-Dialekte problematischer ist. Außerdem werden Dokumente im ASCII-Doc Format auf Git-Camp BW automatisch gerendert und formatiert angezeigt.

5.1.3. Präsentationen

Präsentationen können durch reveal.js einfach im HTML-Format erstellt werden. Dadurch ist eine Versionskontrolle mit GIT möglich. Die Darstellung kann auf jedem Endgerät erfolgen, ist responsive-fähig. Die Präsentation kann duch direkt auf der Webseite von Git-Camp BW angezeigt werden. Ohne Download steht immer die neuste Version bereit.

5.2. Szenarien für den Materialtausch

5.2.1. Fortbildner untereinander

Für jedes Projekt gibt es einen oder mehrere **hauptverantwortliche Fachberatende**. In einem Development-Branch entwickeln diese die Projekte und arbeiten Änderungswünsche ein.

Sie besitzen Lese- und Schreibrechte auf den Repositorys, die auf dem **Git Camp Fortbildner** angelegt werden. Alle anderen FBUs können Repositorys forken und Verbesserungen einbauen. Die Hauptverantwortlichen entscheiden, ob diese übernommen werden.

Die Hauptverantwortlichen arbeiten größere Änderungen in separaten Branches ein und bitten andere FBU um die Begutachtung der Änderungen. Ggf. wird der Branch mit dem Hauptbranch gemerged. Kleinere Bugfixes erfolgen auf dem Hauptbranch.

Davon abgeleitet wird ein Branch zum Veröffentlichen, der dann mit Versionsnummern versehen wird. Jede zu veröffentliche Version wird in ein zweites Repository kopiert, das einen Deployment-Branch enthält und auf dem **GitCamp Lehrerfortbildung** verfügbar gemacht wird, so dass alle Lehrpersonen darauf zugreifen können. Dazu müssen die Repositorys so eingestellt werden, dass die GRUPPE XXX lesenden Zugriff auf das Repository hat.

Alle Arbeitsversionen während des Development-Prozesse bleiben daher den Lehrenden verborgen. Daher sind Urheberprobleme von Zwischenversionen ein geringeres Problem.

5.2.2. Fortbildner - Lehrer

Die Materialien aus Lehrerfortbildungen (insbesondere Softwareprojekte) werden von den Fachberatern im **GitCamp Lehrerfortbildung** bereit gestellt. Das Repository XXX enthält eine Übersicht über alle bereitgestellten Projekte mit der Zuordnung zu den möglichen Einsatzbereichen. Links verweisen auf die Repositorys mit den dazugehörigen Materialien.

Jede Lehrperson sollte diese Repositorys

1. clonen, wenn sie die Projekte nutzen möchte und ihren SuS zur Verfügung stellen möchte. Dazu fügt sie dem Projekt ein weiteres Remote-Verzeichnis hinzu, das im **schuleigenen GitCamp** liegt und pushed dort das Projekt.
2. forken, wenn sie vor hat, das Projekt zu verändern (Verbesserungen einzuprogrammieren) und diese Veränderungen der Allgemeinheit zur Verfügung zu stellen. Nachdem die Veränderungen eingefügt wurden, stellt die Lehrperson einen Pull-Request für das ursprüngliche Repository mit einer aussagekräftigen Beschreibung, welche Änderungen eingefügt wurden. Der betreuende Fachberater entscheidet, ob die Änderungen in das Originalprojekt übernommen werden soll.

Fehler melden: Finden Lehrpersonen Fehler (Schreibfehler, inhaltliche Fehler, Bugs) in den Materialien können sie Issues im normalen Projekt anmelden, so dass der betreuende Fachberater diese korrigieren kann. Auch Änderungswünsche können auf diesem Wege angezeigt werden.

5.2.3. Lehrer - Lehrer

Jede Lehrperson hat jederzeit die Möglichkeit eigene Repositorys im **GitCamp Lehrerfortbildung** anzulegen. Sie kann einzelnen anderen Lehrpersonen Leserechte oder allen Lehrpersonen (GRUPPE XXX) auf diese Repositorys einräumen, so dass diese die Repositorys für den eigenen Unterricht nutzen sollen.

Soll alle Lehrpersonen Zugriff erhalten, müssen die Fachberater informiert werden und gebeten werden, einen Link auf der Übersichtsseite mit allen Projekten zu erstellen.

6. Anwendungsszenarien Unterricht

6.1. SuS versionieren lokal

Einsatzbereich: IMP ab Klasse 8

Voraussetzungen

- kein Account bei einem GIT Hoster notwendig
- Stage, Commit, Anlegen eines Repositories, Check out

Oft haben SuS Probleme, ihre Dateien ordentlich zu verwalten. Oft existieren mehrere Versionen in verschiedenen Ordnern, die Arbeit eines Tages wird gar nicht gespeichert oder durch ungwollte Änderungen wird der Code nicht mehr ausführbar. Den SuS fällt es schwer, diesen Fehler zu korrigieren, da oft unbeabsichtig notwendiger Code gelöscht wurde, den die SuS nicht wiederherstellen können.

Durch die Verwendung von GIT von Beginn des Programmierunterrichts an, werden diese Probleme minimiert. Es muss zur Routine werden, am Ende jeder Arbeitsphase / Unterrichtsstunde einen Commit zu machen. Dabei wird die Arbeit des Tages rekapituliert und der aktuelle Arbeitsstand im Commit festgehalten.

Bei ungewollte Änderungen kann jederzeit zum Arbeitsstand des Vortages zurückgekehrt werden oder die Unterschiede zum Vortag angezeigt werden.

Die Sus initialisieren dazu einen Ordner als GIT-Repository. In diesem bearbeiten sie ihr Projekt und speichern die aktuelle Version am Ende jeder Stunde. Dazu werden alle Änderungen der Stunde mit einem entsprechenden Kommentar commited.

Solange die SuS nur in der Schule am Projekt arbeiten, ist es nicht erforderlich das Repository auf Git Camp zu pushen. Dadurch ist es unmöglich Versionskonflikte zu bekommen. Die SuS werden auf sehr einfache Art und Weise an GIT herangeführt.

6.2. Lehrperson stellt Material per GIT bereit, SuS versionieren lokal

Einsatzbereich: IMP ab Klasse 8

Voraussetzungen

- öffentlich verfügbares Repository oder schuleigene Git Camp Instanz mit Zugängen für die Schüler
- Clonen, Pull, Stage, Commit, Check out, (Rebase, Fetch)

Die Lehrperson stellt ein Repository online für Schüler zur Verfügung. Die Schüler clonen dieses Repository lokal in der Schule und arbeiten lokal. SuS machen am Ende der Stunde einen

Sicherungsschritt (commit).

Dabei können die Unterschiede zur Vorversion sichtbar gemacht werden. Außerdem ist es möglich, zu einer Vorversion zurückzukehren. Dies entspricht den Möglichkeiten des ersten Anwendungsszenarios. Hier kommt im ersten Schritt ein Clonen eines vorhanden Repositorys hinzu.

Variante Update bei Fehler:

Gegebenenfalls kann die Lehrperson ein Update des Repository zur Verfügung stellen. Die Sus fetch/pullen die Neuerungen und führen diese mit ihrer Version so zusammen, dass das Update die Originalvorlage ersetzt und nachträglich die Änderungen der SuS angewendet werden. Dies bezeichnet man als **Rebase**. Dadurch vermeidet man den Versionskonflikt, der entstehen kann, wenn beide Änderungen (Update der Lehrperson und eigene Arbeit der SuS) gleichberechtigt zusammengeführt werden würden (merge).

In der Regel betreffen die Änderung an der Vorlage durch die Lehrkraft aber ohnehin Bereiche, in denen die SuS nicht selbst programmiert haben. Daher sollten selbst bei Merge nur selten Versionskonflikte auftreten.

Variante Lösungen zur Verfügung stellen:

Die Lehrperson kann die Lösungen der Aufgaben im Repository als Commit bereitstellen. Die SuS pullen die neue Variante und mergen mit ihrer Lösung. Die SuS können dabei ihre Version mit der Lehrerversion vergleichen und ggf. durch Musterlösung ersetzen. Dafür ist aber eine Konfliktbehandlung beim Mergen erforderlich.

6.3. SuS versionieren online (SuS arbeiten zu Hause und in der Schule)

Einsatzbereich: IMP ab Klasse 10

Voraussetzungen

- schuleigene GIT-Camp Instanz mit Zugängen für SuS
- Fork, Clone, Stage, Commit, Pull, Push

Die Lehrperson legt Accounts für die SuS auf der schuleigenen GIT Camp Instanz an und stellt dort Material in einem Repository. Die SuS brauchen einen lesenden Zugriff auf das Repository.

Die SuS forken dieses Repository direkt in der Web-Oberfläche von GIT Camp. Dadurch haben sie einen eigenen Ableger der Projekts, bei dem sie volle Zugriffsrechte haben. Sie sollten auch der Lehrperson volle Zugriffsrechte (zumindest Leserechte) einräumen.

Es ist alternativ auch möglich, das Repository der Lehrkraft zu clonen. Grundsätzlich verwendet man einen Fork, wenn man beabsichtigt ein Projekt eigenständig weiterzuentwickeln, welches unabhängig vom Original existiert. Clone wird verwendet, wenn man in einem Team gemeinsam an einem Repository arbeitet und einzelne Teile des Projekts bearbeitet. Daher ist in

diesem Szenario ein Fork sinnvoller, auch wenn ein Clone genauso benutzt werden könnte.

Das eigene Repository clonen die SuS sowohl in der Schule als auch zu Hause und führen am Ende jeder Stunde ein Commit mit anschließendem Push durch, so dass sie zu Hause mit einem Pull auf die aktuelle Version zugreifen können.

Machen Sie den SuS deutlich, dass es sehr wichtig ist, immer einen Pull durchzuführen, bevor man anfängt zu arbeiten. Vergessen die SuS dies, kann es Änderungen in der Schule und zu Hause geben, die im Konflikt zueinander stehen. Die SuS merken dies daran, dass sie die zweite Änderung nicht mehr pushen können. GIT meldet die Konflikt und verlangt zunächst ein Pull/Merge bevor die nächste Push Aktion möglich ist. Es muss also aufwendig ermittelt werden, welche Version die richtige ist (Conflict Solver), was während einer Unterrichtsstunde nicht immer so einfach ist.

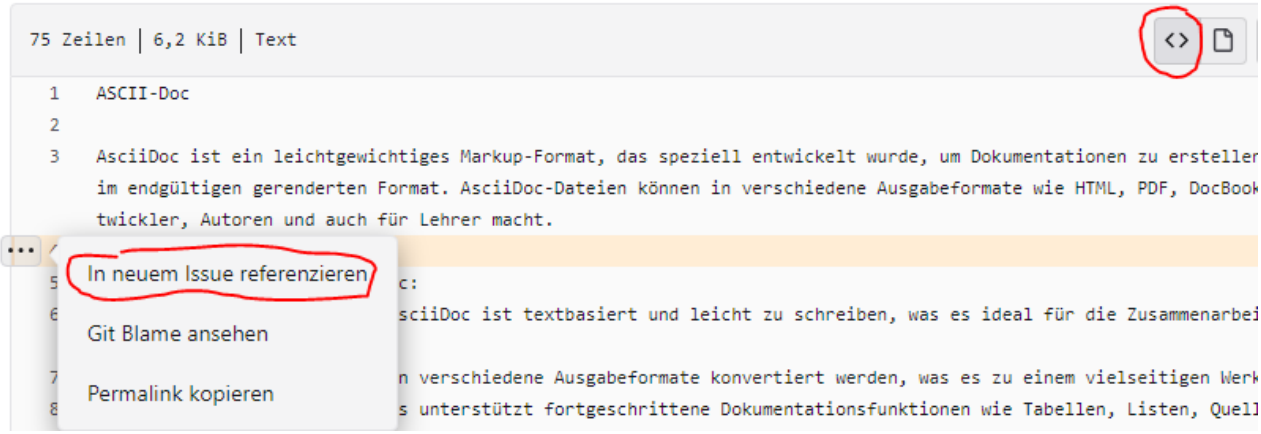
Die Lehrperson hat über die GitCamp-Webseite jederzeit Einblick in den Stand der Schülerrepositorys. Sie sieht alle Forks im eigenen Projekt.

Die Lehrperson kann im eigenen Projekt lokal weitere Branches für jeden SuS anlegen und den tracking branch auf die Repositorys der SuS legen. Dann kann die Lehrperson den aktuellen Stand der Schülerprojekte jederzeit lokal anschauen und überprüfen.

Bei Fehlern hat die Lehrperson die Möglichkeit die Änderungen direkt am Code durchzuführen und Kommentare dazu beim Commit unterzubringen. Anschließend push die Lehrperson die Änderungen. Die Sus sehen die Änderungen, die von der Lehrperson vorgenommen wurden und lesen die Kommentare. Auch hier ist es dringend geboten, dass die SuS zunächst ein Pull durchführen, bevor sie selbst weiterarbeiten. Ansonsten können auch hier Konflikte auftreten. Da die Lehrkraft die eigenen Änderungen aber kennt, sind sie meist leichter zu lösen.

Variante: Kommentieren von Schülerprojekten: In GitCamp hat eine Lehrerin automatisch Leserechte auf den Repos der Schülerinenn. Daher kann eine Lehrerin jederzeit den Arbeitsstand der Schülerinnen ansehen. Dies kann sie entweder online oder indem sie einen lokalen Clone anlegt. Dazu müssen die Repos der Schülerinnen nicht als Fork des Lehrerprojekts entstanden sein.

Statt die Schülerinnenprojekte nun zu ändern und zu pushen, kann die Lehrerin den Schülerinnen auch einen Issue im online-Repository eintragen. Dieser Issue kann an eine bestimmte Zeile im Code gebunden werden. Dazu lässt sich die Lehrerin im Gitcamp den Quellcode einer Datei anzeigen, so dass links Zeilennummern zu sehen sind (Quellcode muss explizit rechts oben gewählt werden!). Dann kann in der fehlerhaften Zeile ganz links geklickt und ein Issue zu dieser Zeile erstellt werden.



6.4. Zusammenarbeit in Gruppenprojekten

Einsatzbereich: BF in der Kursstufe

Voraussetzungen

- schuleigene GIT-Camp Instanz mit Zugängen für SuS
- Clone, Stage, Commit, Push, Pull, Merge

Eine einfache Version der Zusammenarbeit von zwei (mehreren) Schülerinnen kann durch die Verwendung eines gemeinsamen online Repositorys erreicht werden. Dabei legt eine Schülerin das Online-Repo in Gitcamp an und räumt den anderen Teammitgliederinnen Lese- und Schreibrechte darauf ein.

Das ganze Team startet mit einer gemeinsamen Vorlage (ggf. ein Fork oder Clone eines Lehrerinnen-Repositorys). Diese Vorlage wird allen Teammitgliederinnen gecloned, damit es lokal auf ihrem Rechner vorliegt. Gleichzeitig arbeiten sie an verschiedenen Aspekten des Projekts. Die geringsten Konflikte erzielt man, wenn jedes Teammitglied an einer eigenen Datei (Klasse) arbeitet.

Nach Abschluss eines Arbeitsschrittes wird die eigene Arbeit durch einen Commit lokal gesichert. Bevor der Arbeitsstand gepusht werden kann, muss vor ein Pull durchgeführt werden, da die anderen Teammitgliederinnen auch Änderungen vorgenommen haben könnten. Aufgrund des gleichzeitigen Arbeitens wird ein Merge der beiden Änderungen notwendig sein. Es sollte stets die Option "Merge to working tree" gewählt werden. Nachdem alle Merge-Konflikte beseitigt sind, das Projekt erneut getestet wurde, wird das zusammengeführte Projekt durch einen weitere Commit gesichert und dann gepusht. Der Push ist nun möglich, da das zusammengeführte Projekt ein Nachfolger der zuerst gepushten Änderungen ist.

Einsatzbereich: LF in der Kursstufe

Voraussetzungen

- schuleigene GIT-Camp Instanz mit Zugängen für SuS
- Fork, Clone, Stage, Commit, Push, Pull, Merge

Die Rohversion eines Projektes wird ggf. immer noch von der Lehrperson per Repository bereit

gestellt. Ein Mitglied wird zum "Sprecher" der Gruppe. Dieser forked das Lehrerprojekt oder legt ein eigenes Repository an, wenn es keine Vorlage gibt. Er nimmt alle anderen Mitglieder der Gruppe als "Developer" in das Repository auf. Alle Gruppenmitglieder clonen diesen Fork und haben damit ihr lokalen Versionen.

Die Gruppenmitglieder arbeiten lokal, committen ihr Änderungen und pushen sie danach. Ab dem Push des zweiten Gruppenmitglieds ist ein Zusammenführen der verschiedenen Änderungen erforderlich. Arbeiten die SuS an verschiedenen Dateien ist dies problemlos möglich. Bei Änderungen an gleichen Dateien müssen die Änderungen gemerged werden und ggf. Konfiktbehandlung durchgeführt werden.

Die SuS könnten auch in verschiedenen Branches arbeiten und erst am Ende ihre Branches zu einem gemeinsamen Projekt zusammenführen. Das Arbeiten mit mehreren Branches setzt aber ein sehr diszipliniertes Arbeiten voraus. Sobald einmal ein falscher Branch aktiv ist, wird es schwierig die Fehler wieder zu korrigieren. Daher ist nur bei sehr fitten SuS zu empfehlen, dieses Konzept einzuführen.

7. Übungen für die Fortbildung

Für die Fortbildung sind Übungen mit GIT vorgesehen, die den Anwendungsszenarien für den Unterricht entsprechen. Dabei schlüpfen die Teilnehmenden sowohl in die Rolle der Lehrer als auch in die Rolle der Schüler. Die Fortbildenden sollten daher genau darauf achten, die jeweilige Rolle deutlich herauszuarbeiten.

> **Fortbildner:** Das Arbeiten mit GIT ist sehr anspruchsvoll und für GIT-Anfänger oftmals sehr verwirrend. Beschränken Sie sich daher in der ersten Fortbildung auf die Übungen mit einem lokalen Repository (Kapitel Section 7.1, “Lokal Versionieren”), das Anlegen eines Online-Repositorys und des Sicherns des lokalen Repos auf dem Online-Speicher (Kapitel Section 7.2.1, “Vorlage übernehmen und lokal versionieren”), damit an verschiedenen Rechnern gearbeitet werden kann. Alle anderen Übungen sind optional und können ggf. in einer Anschlussveranstaltung durchgeführt werden.

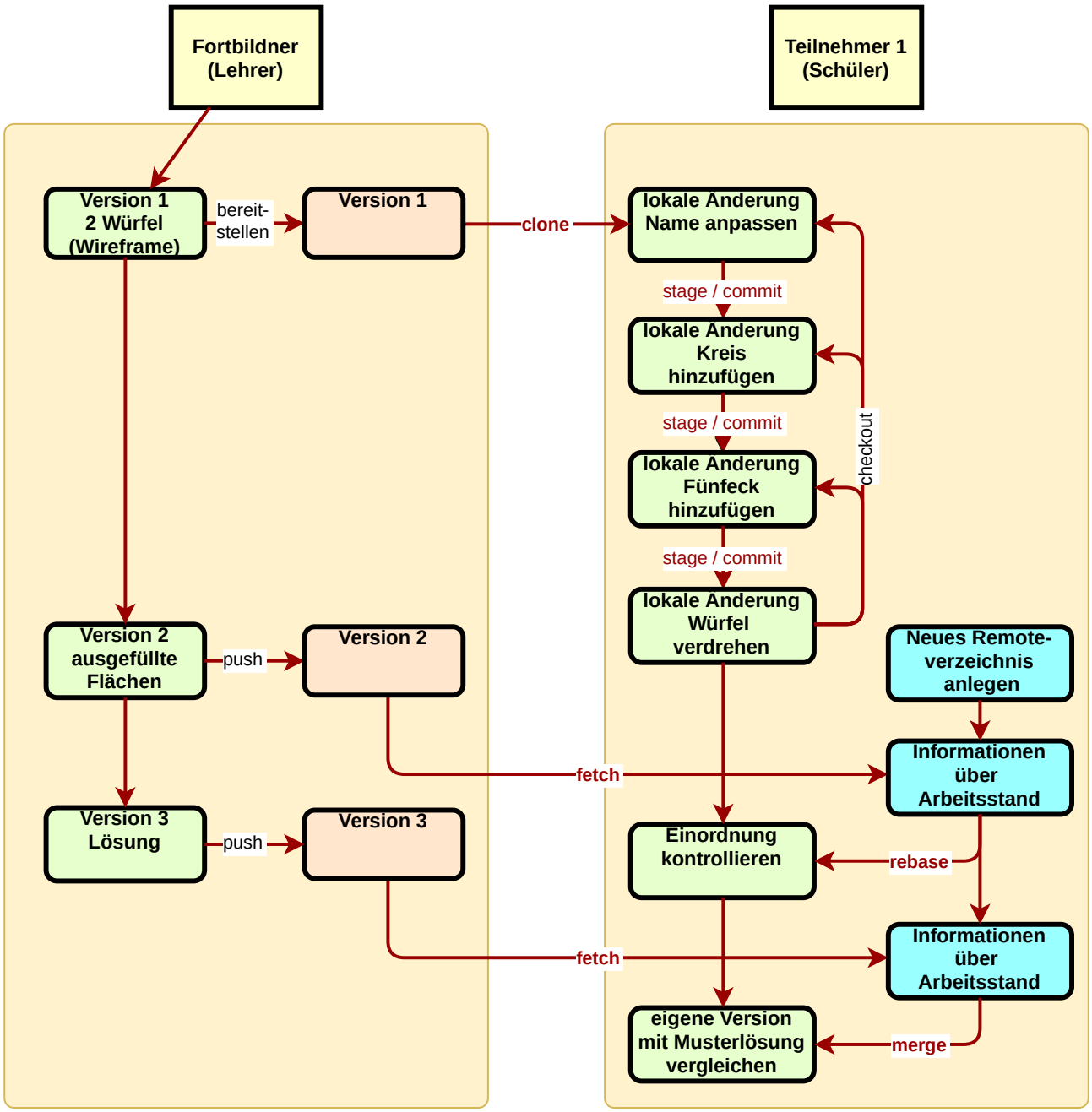
7.1. Lokal Versionieren

> **Fortbildner:** Das lokale Versionieren wird in den folgenden Übungen noch intensiver geübt. Daher geht es hier nur darum eine erste Erfahrung zu sammeln und zu erkennen, wie eine Versionsverwaltung arbeitet.
>
> Entscheiden Sie, ob der Arbeit mit einer GUI eine kurze Arbeitsphase mit der GIT-Konsole vorangestellt werden sollte.

Aufgaben:

1. Legen Sie mit Smartgit ein neues Repository an (Menü Repository→Add or create). Wählen Sie dort einen Ordner aus. Da er noch kein Repository enthält, müssen Sie das Repository zunächst initialisieren.
2. Erstellen Sie in diesem Ordner eine Text-Datei und schreiben Sie "Version 1" als Text hinein.
3. Sie sehen in Smartgit, dass eine Datei hinzugekommen ist. Erstellen Sie eine "Commit"-Nachricht, fügen die Datei dem Stage-Bereich hinzu und commiten diesen Zustand.
4. Verändern Sie die Textdatei (z. B. "Version 2" hineinschreiben) und legen Sie eine weitere Datei an. Commiten Sie diese Veränderungen auch.
5. Sie können nun zwischen den beiden Version hin- und herwechseln und sehen in Smartgit, welche Veränderungen gemacht wurden. Es biete sich an, die Schülerinnen und Schüler (mindestens) jeweils am Ende der Unterrichtsstunde einen Commit machen zu lassen, dann ist der jeweilige Arbeitsstand des Tages gesichert.

7.2. Mit Vorlagen in einem Online-Repository arbeiten



Ziel der zweiten Übung ist es, selbst ein Repository durch Clonen eines online bereitgestellten Repositorys anlegen zu können, mit diesem zu arbeiten und die Arbeitsschritte lokal zu versionieren.

In der Schule wird in der Regel, das vom Lehrer bereitgestellte Repository von den Schülern geclont. Sie befinden sich also im Moment in der Rolle der Schüler.

Vorbereitung Fortbildner: Erstellen Sie ein neues Repository und kopieren Sie die Version 1 des Matter-Projekts dort hinein. Stellen Sie das Repository online bereit und

teilen Sie den Teilnehmern den Link zum Repo mit.

7.2.1. Vorlage übernehmen und lokal versionieren

Aufgaben:

1. Erzeugen Sie ein neues lokales Repository durch Clonen des vom Fortbildner angegebenen Online-Repository.
2. Öffnen Sie die Datei Test.html mit einem Browser.
3. Öffnen Sie die Datei Test.html mit einem beliebigen Texteditor (z. B. notepad++). Fügen Sie Ihren Namen bei "Autor" hinzu. Speichern Sie die Datei, überzeugen Sie sich, dass die Änderungen im Browser zu sehen sind.
4. Sie sehen auch in Smartgit, dass die Datei verändert wurde und bekommen die Änderungen im Detail angezeigt. Tragen Sie eine Commit-Nachricht ein, stagen Sie die Datei und commiten Sie die Änderungen.
5. Fügen Sie einen Kreis (Bodies.circle verlangt die X- und die Y-Koordinate des Mittelpunkts und den Radius als Parameter) in der Welt hinzu.

   ```
   var circle = Bodies.circle(440, 150, 40);
   ```

   Beachten Sie, dass er auch beim Composite.add Befehl ergänzt werden muss. Stage/Commiten Sie die Änderungen.

6. Fügen Sie ein Fünfeck (Bodies.polygon verlangt die X- und die Y-Koordinate des Mittelpunkts, die Anzahl der Ecken und den Radius als Parameter).

   ```
   var fuenfeck = Bodies.polygon(380, 250, 5, 30);
   ```

   Stage/Commiten Sie die Änderungen.

7. Verdrehen Sie die Würfel um einen beliebigen Winkel:

   ```
    var boxA = Bodies.rectangle(400, 200, 80, 80, {angle: 0.5} );
   ```

   Stage/Commiten Sie die Änderungen.

8. Wechseln Sie in Smartgit zu den verschiedenen Entwicklungsstufen (checkout). Es reicht, wenn Sie ohne lokalen Branch (read only) arbeiten. Kontrollieren Sie im Browser, dass sich die Datei Test.html im jeweiligen Entwicklungszustand befindet.

7.2.2. Änderungen an der Vorlage (Rebase)

Es kommt immer wieder vor, dass ein Fehler in den bereitgestellten Dateien erst im Verlauf des Unterrichts bemerkt wird. Nun wäre es gut, die Originaldateien ändern zu können und trotzdem alle von den Schülern gemachten Änderungen beibehalten zu können.

> **Vorbereitung Fortbildner:** Kopieren Sie Version 2 der Matter-Projekt-Vorlage in ihr Repo und pushen Sie die neue Version.

9. Sollte noch kein Remote-Verzeichnis in Smartgit existieren, legen Sie in Smartgit ein neues für das Online-Repo des Fortbildners an, das Sie vorhin geclont haben. Rufen Sie mit Fetch die Information über dieses Repo ab.
10. Führen Sie ein Rebase durch. Wählen Sie ihren lokalen Branch aus und rufen Sie das Kontextmenü auf. Achten Sie darauf, dass ihr lokaler Branch fett markiert ist. Wählen Sie dann "Rebase" aus und führen Sie das Rebase automatisch durch ("Rebase HEAD to").
    Kontrollieren Sie im Browser, dass die Objekte nur gefüllt sind. Untersuchen Sie, an welcher Stelle der Versionshistorie die Umstellung auf "wireframes: false" stattgefunden hat.

7.2.3. Musterlösung bereitstellen

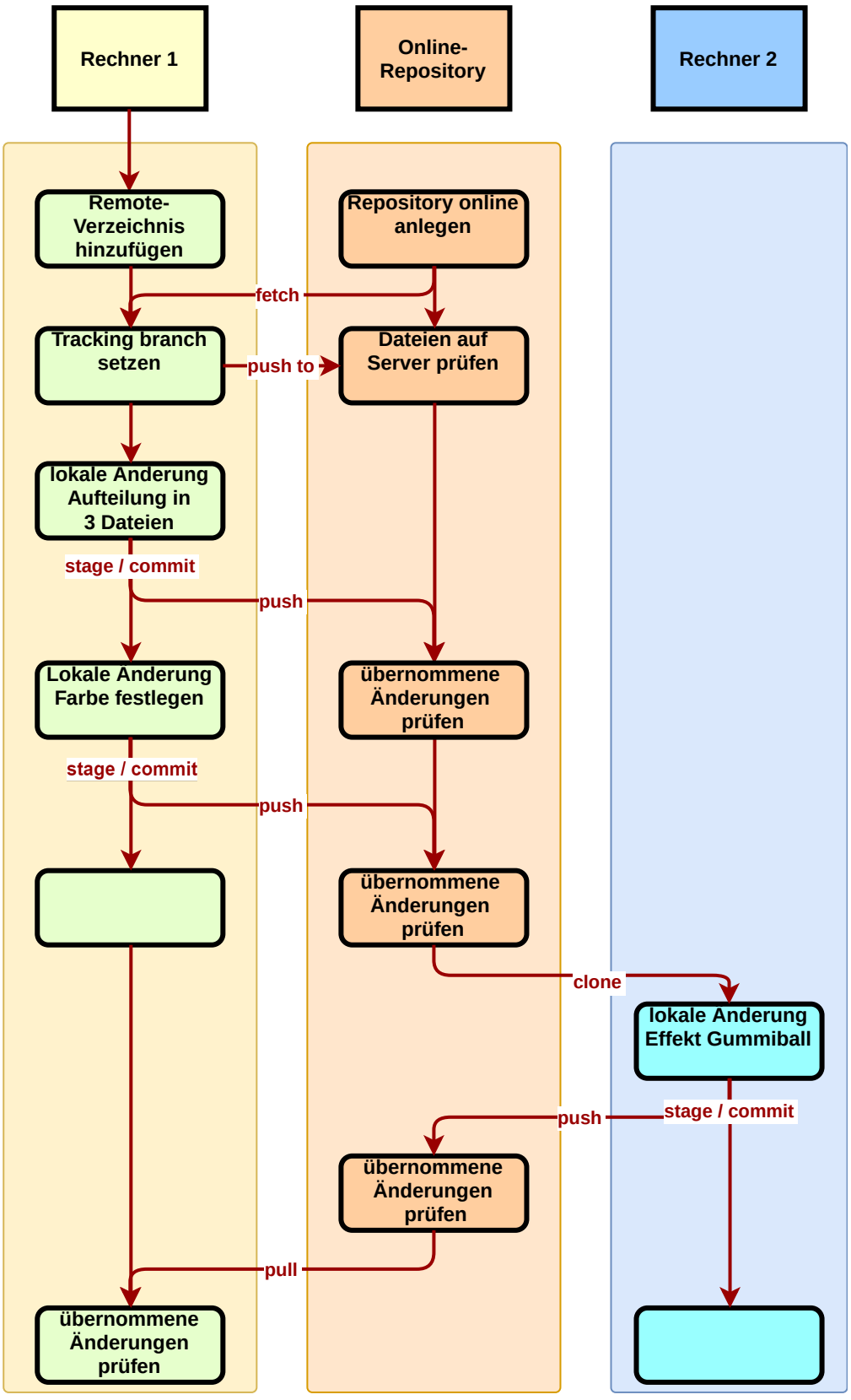
Gerne möchte man als Lehrender am Ende oder als Zwischenstand eine Musterlösung bereitstellen. Die Schülerinnen und Schüler sollen ihre Lösungen mit der Musterlösung vergleichen und nur dort, wo sie nicht fertig wurden oder Fehler erkannt wurden, die Musterlösung übernehmen.

> **Vorbereitung Fortbildner:** Kopieren Sie Version "Würfel verdreht" der Matter-Projekt-Vorlage in ihr Repo und pushen Sie dieses.

11. Führen Sie erneut einen Fetch für das Online-Repo des Fortbildners durch.
12. Führen Sie nun über das Kontextmenü des lokalen Branches ein Merge durch. Wählen Sie dazu "Merge to Working Tree". Dies hat den Vorteil, dass die aktuellen Änderungen jederzeit verworfen werden können, wenn der Merge Probleme bereitet.
    Vergleichen Sie dabei die Musterlösung der Fortbildungsmaterialien mit ihrer Lösung. Sie sehen, dass die Datei Test.html sowohl von Ihnen wie auch von der Musterlösung verändert wurde (The file is in Conflicted (both modified) state). Rufen Sie den Conflict Solver auf. Links sehen Sie ihre Lösung (ours), rechts die heruntergeladene Musterlösung (theirs). Sie können nun über die ">>" bzw. "<<" Buttons entscheiden, welche Variante sie übernehmen wollen. Übernehmen Sie nur dann die Musterlösung, wenn ihre Lösung Fehler enthält oder unvollständig ist. Speichern Sie die aktualisierte Version, schließen Sie den Conflict Solver und markieren sie den Konflikt als gelöst.
    Zum Abschluss müssen Sie noch einen Commit durchführen, um den Merge-Prozess abzuschließen.

7.3. Eigenes Repository online ablegen

Ein lokales Image ermöglicht die Versionsverwaltung auf einem Rechner. Sowohl Schüler als auch Lehrkräfte arbeiten aber sowohl in der Schule als auch zu Hause. Daher ist ein Datenaustausch zwischen verschiedenen Rechnern notwendig. Dieser erfolgt über die Speicherung auf einem GIT-Server. Das Land Baden-Württemberg stellt dafür den GIT-Camp Server zur Verfügung. In dieser Übung lernen Sie den Datenaustausch mit dem GIT-Camp Server kennen.



7.3.1. Anlegen eines Repositories auf dem GIT-Camp Server

1. Loggen Sie sich mit Ihren GIT-Camp Login-Daten auf dem Server ein.
2. Legen Sie dort ein neues Repository an:
    - Wählen Sie einen Namen.

- Stellen Sie die Sichtbarkeit auf "privat".
- Geben Sie eine kurze Beschreibung ein : z. B. "Matter-Projekt der GIT-Fortbildung".
- Die restlichen Felder können Sie leer lassen. Der Haken bei "Repository initialisieren" darf **nicht** gesetzt sein.

3. Kopieren Sie die URL des Repositorys und legen Sie in Smartgit in Ihrem Repository ein neues Remote-Verzeichnis (=Online-Repository) an (Menü Remote→Add). Fügen Sie die kopiere URL ein und wählen als Namen "GIT-Camp".
4. Führen sie zunächst ein Fetch auf diesem Remote-Verzeichnis durch, damit Smartgit die Informationen über das Repositiory abfragt. Führen Sie danach ein "Push to..." mit ihrem lokalen Repository durch und wählen das Remote-Verzeichnis als Ziel. Prüfen Sie auf "Git-Camp", dass die Dateien übertragen wurden.
5. Damit ihr lokales und das Remote-Verzeichnis verknüpft sind, können Sie außerdem "Set tracked branch" auf ihr Remote-Verzeichnis setzen. Sie müssen dann in Zukunft nicht immer angeben, mit welchem Remote-Verzeichnis ein pull oder ein push durchgeführt werden soll. Dazu muss allerdings vorher einmal ein Push auf dieses Remote-Verzeichnis durchgeführt werden.

Sie führen nun noch eine weitere Änderungen an den Dateien durch und legen diese auf dem Server ab.

6. Teilen Sie die Datei Test.html in drei Teile auf:
   - Schneiden Sie aus dem JavaScript-Teil alles aus, was mit den Objekten in der Welt zu tun hat (Erstellen der Objekte und Einfügen in die Welt) aus und fügen Sie es in eine neue Datei "objekte.js" ein. Ein umgebender <script>-Tag ist nicht notwendig.
   - Schneiden Sie aus dem JavaScript-Teil alles aus, was mit der Welt zu hat (also den gesamten Rest des Scripts) und fügen Sie es in eine neue Datei "welt.js" ein.
   - Ändern Sie "Test.html" so ab, dass die beiden Scripte geladen werden:

     ```
     <script src="welt.js"></script>
     <script src="objekte.js"></script>
     ```

     Der umgebende Rest an HTML-Code bleibt erhalten.
7. Stage/Commiten Sie die Änderungen lokal. Pushen Sie die Änderungen auf den Git-Camp-Server.
8. Wählen Sie die Farben der Objekte selbst aus, z. B.:

   ```
   boxA.render.fillStyle = "#FF66AC"; // z.B. für Pink
   ```

   Stage/Commiten Sie die Änderungen lokal. Pushen Sie die Änderungen auf den Git-Camp-Server.

9. Überzeugen Sie sich, dass die verschiedenen Versionen nun auf dem Git-Camp-Server verfügbar sind und die gesamte Änderungshistorie nachvollziehbar ist (auch die Änderungen, die erfolgt sind, bevor das Repository auf den Git-Camp-Server hochgeladen wurde).

7.3.2. Arbeiten an verschiedenen Orten

In dieser Übung sollen Sie lernen, wie Sie an verschiedenen Orten mit ihrer jeweils aktuellen Version arbeiten können. Wechseln Sie daher mit ihrem Nachbarn den Rechner.

1. Clonen Sie ihr eigenes Repository auf dem Rechner des Nachbarn. Wählen Sie dazu in Smartgit Menü Repository→Clone und geben die URL ihres Repositories auf dem Git-Camp-Server an. Wählen Sie als lokales Verzeichnis ein neues, leeres Verzeichnis.
2. Überzeugen Sie sich, dass ihre aktuelle Version nun lokal auf dem neuen Rechner verfügbar ist. Untersuchen Sie, ob auch alle vorherigen Versionen vorhanden sind.
3. Die Objekte soll nun wie ein Gummiball herumspringen. Dazu muss der Boden gummiartig werden:

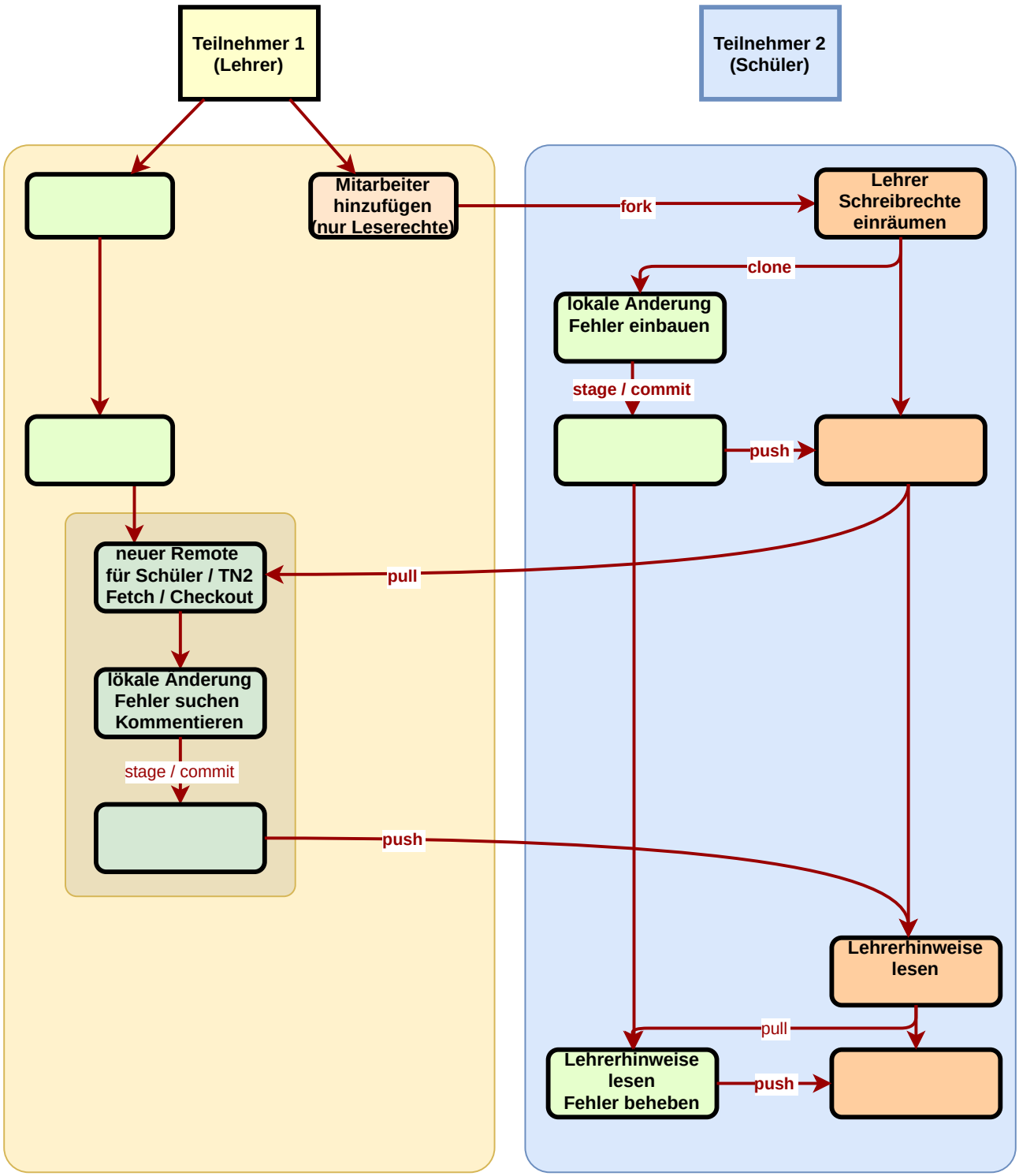
   ```
   ground.restitution = 1.3;
   ```

   Stagen/Commiten Sie die Änderung lokal und pushen Sie die Änderung auf den Git-Camp-Server. Überzeugen Sie sich davon, dass die Änderungen auf dem Git-Server übernommen wurden.

4. Wechseln Sie zu ihrem eigenen Rechner zurück und pullen Sie aktuelle Version vom Git-Camp-Server. Überzeugen Sie sich, dass die Objekte jetzt auch auf diesem Rechner herumspringen.

7.4. Zusammenarbeit Lehrer-Schüler

In dieser Übung wird die Situation simuliert, dass die Schüler mit einem online-Repository arbeiten, das auf einer Vorlage des Lehrers beruht, und der Lehrer den aktuellen Arbeitsstand der Schüler sichtet und Fehler behebt oder zumindest kommentiert.



Dazu ist es notwendig, dass der Schüler lesenden Zugriff auf das Repository des Lehrers und der Lehrer lesenden und schreibenden Zugriff auf das Repository des Schülers besitzt. Nur so kann er die Korrekturen an den Schüler zurückgeben.

In dieser Übung agieren Sie sowohl als Lehrer als auch als Schüler: Als Lehrer stellen Sie ein Repository bereit und gewähren den Schülern lesenden Zugriff. "Schüler" ist in diesem Fall Ihr linker Nachbar. Dieser legt einen Fork[3] ihres Repositorys an und baut einen Fehler in das Programm ein. Sie übernehmen diesen Part für Ihren rechten Nachbarn. Danach holen Sie sich das Repositorys des "Schülers" (ihres linken Nachbarn) und suchen dessen Fehler. Die korrigierte und kommentierte Version geben Sie an den "Schüler" zurück.

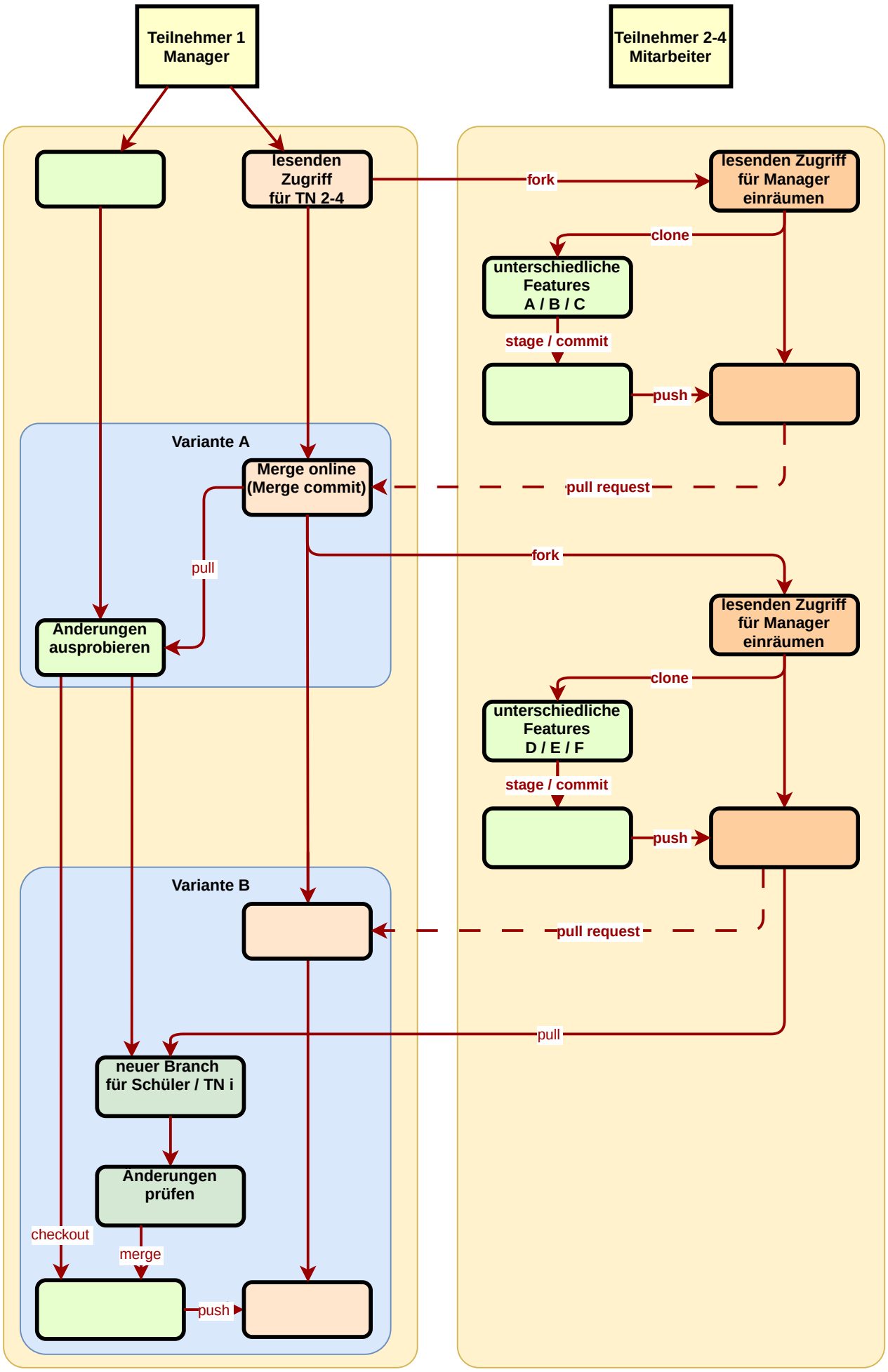
1. Lehrerrolle: Räumen Sie ihrem linken Nachbarn (dem "Schüler") im Git-Camp Leserechte auf Ihr Online-Repository ein. Gehen Sie dazu in der Onlineumgebung von Git-Camp beim Repo auf Einstellungen→Mitarbeiter und fügen den Namen ihres linken Nachbarn hinzu. Gewähren Sie nur Leserechte! In der Schule müssten Sie das für die ganze Klasse machen. (TODO: Geht das auch für Gruppen?)
2. Schülerrolle: Da Ihr "Lehrer" (rechter Nachbar) sein Repository für Sie freigegeben hat, sehen Sie es in Git-Camp in der Liste der verfügbaren Repositories. Wählen Sie es aus und legen Sie einen Fork an. Benennen Sie den Fork so, dass Sie Ihren Namen an den Repository-Namen anhängen.
3. Schülerrolle: Räumen Sie Ihrem "Lehrer" (rechter Nachbar) Lese- und Schreibrechte auf Ihren Fork ein. Im Schulumfeld ist dieser Schritt nicht notwendig, da Sie als Besitzer einer Gruppe in der Organisation automatisch Lese- und Schreibrechte auf die Repos der Gruppenmitglieder besitzen.
4. Schülerrolle: Legen Sie eine lokale Kopie Ihres Forks an, indem Sie das Repository mit SmartGit clonen.
5. Schülerrolle: Bauen Sie an beliebiger Stelle einen Fehler ein. Stage/commiten Sie die geänderte Version und verwenden Sie als Commit-Message "Ich weiß nicht weiter… Es geht nichts mehr." (Sie dürfen das "nichts" auch näher beschreiben, dass sollte man von Schülern verlangen).
6. Lehrerrolle: Legen Sie (für jeden Schüler) im Smartgit ein neues Remote-Verzeichnis an (Menü Remote → Add). Geben Sie dort die URL des Forks Ihres "Schülers" (rechter Nachbar) an. Sie können in Git-Camp alle Forks ihrer Schüler anzeigen lassen, wenn Sie auf die Zahl neben "Fork" klicken. Wählen Sie einen Fork aus, sehen Sie die URL des Forks im Browser.
   Als Name des Remote-Verzeichnis wählen Sie nicht "origin", sondern den Namen Ihres Schülers.
7. Lehrerrolle: Holen Sie sich die Informationen über das neue Remote-Verzeichnis, indem Sie ein "Fetch" auf dem Remote-Verzeichnis ausführen (Rechtsklick). Sie sollten dann die Branches sehen, die das Projekt enthält (in der Regel nur einen master-Branch). Wählen Sie diesen aus, indem Sie ein "Check out" durchführen. Sie bestätigen dann die Frage, ob ein lokaler Branch angelegt werden soll. Auf diese Weise bekommen Sie für jeden Schüler einen Branch, der dem Namen des Schülers entspricht. Sie können dann sehr schnell zwischen den verschiedenen Schülerversionen hin- und herwechseln.
8. Lehrerrolle: Suchen (und korrigieren) Sie den Fehler, den Ihr Schüler eingebaut hat. Schreiben Sie eine Rückmeldung / Erklärung in die Commit-Message, welche Arbeiten der Schüler noch selbst durchführen muss, um seinen Fehler zu beheben. Stage/Commiten Sie die Änderungen lokal und pushen Sie die Änderung Sie Änderungen in den Fork des Schülers. Normalerweise müssen Sie dazu einfach "Push" wählen, da bei dieser Variante der Branch so angelegt wurde, dass der "Tracking branch" automatisch das Remote-Verzeichnis des entsprechenden Schülers ist. Ansonsten müssen Sie den Tracking branch setzen oder "Push to" benutzen.
9. Schülerrolle: Schauen Sie auf Git-Camp nach, ob Ihr "Lehrer" (rechter Nachbar) ihren Fehler gefunden hat und Ihnen Tipps gegeben hat, wie Sie ihn korrigieren können. Pullen Sie die Änderungen des "Lehrers", so dass ihr lokales Repository wieder auf dem aktuellen Stand

ist. Beheben Sie mit den Tipps des Lehrers ggf. noch vorhandene Fehler.
Achtung: Es ist essentiell, dass die SuS zunächst den Pull durchführen und den Fehler sich nicht nur online anschauen. Wird kein Pull durchgeführt und der Fehler lokal behoben, entsteht beim nächsten Push ein Konflikt, da eine neuere Version auf dem Server vorhanden ist. Diese muss erst gepullt und mit der lokalen Version gemerged werden.

7.5. Kooperatives Arbeiten

Die komplexeste Form der Nutzung von GIT ist das kooperative Arbeiten an einem Projekt. Dafür ist GIT konzipiert worden. Hier arbeiten mehrere Personen gleichzeitig an einem Projekt. Eine genaue Absprache, wer welche Teile des Projektes bearbeitet, ist unerlässlich. Trotzdem lässt es sich nicht vermeiden, dass einige Dateien von mehreren Personen bearbeitet werden. Es werden also beim Zusammenführen unweigerlich Konflikte auftreten.

Um das Chaos in einem überschaubaren Rahmen zu halten, sollten nicht alle Gruppenmitglieder volle Schreibrechte auf das Repository haben. Es ist einfacher, wenn zunächst jeder im eigenen Fork arbeitet und ein Gruppenmanager die Ergebnisse kontrolliert zusammenführt.


Teilnehmer 1
Manager
Teilnehmer 2-4
Mitarbeiter
lesenden
Zugriff
für TN 2-4
fork
lesenden Zugriff
für Manager
einräumen
clone
unterschiedliche
Features
A / B / C
stage / commit
push
Variante A
Merge online
(Merge commit)
pull request
pull
fork
Anderungen
ausprobieren
lesenden Zugriff
für Manager
einräumen
clone
unterschiedliche
Features
D / E / F
stage / commit
push
Variante B
pull request
pull
neuer Branch
für Schüler / TN i
Anderungen
prüfen
checkout
merge
push

1. Finden Sie sich in Gruppen von 3-4 Personen zusammen. Jeder von Ihnen ist im schulischen Kontext ein Schüler oder eine Schülerin. Bestimmen Sie einen Gruppen-Manager. Dieser ist verantwortlich dafür, die Arbeiten der einzelnen Gruppenmitglieder zusammenzuführen und online zu stellen.
2. Gruppenmanager: Als Gruppenmanager räumen Sie den Gruppenmitgliedern lesenden Zugriff auf ihr Repository ein.
3. Gruppenmitglieder: Forken Sie das bereitgestellte Repository und räumen Sie ihrem Gruppenmanager lesenden Zugriff auf ihren Fork ein.
4. Gruppenmitglieder (+ Manager): Bearbeiten Sie arbeitsteilig eine der folgenden Aufgaben. Sie sind aufsteigend nach Schwierigkeitsgrad geordnet. Nutzen Sie das Internet oder eine KI, um den Code zu erstellen.
   Tipp: Drücken Sie in Chrome Strg+Shift+I um die Entwickler-Tools angezeigt zu bekommen. Nur dort stehen Fehlermeldungen der Webseite, wenn Programmierfehler auftreten:
   - Lassen Sie die Geschwindigkeiten (velocity) der Objekte in der Welt durch kleine Pfeile anzeigen. Dies ist eine Funktionalität, die im Renderer aktiviert werden kann.
   - Fügen Sie rechts und links Wände hinzu, die sich natürlich nicht bewegen. Die Wände müssen nicht bis ganz oben gehen, so dass die Objekte ab einen gewissen Füllgrad über die Wände hinausfallen.
   - Bauen Sie Balken in der Welt ein, die sich nicht bewegen und auf denen die Objekte herunterrutschen. Die Reibung sollte also nicht zu groß sein.
   - Fügen Sie 1000 kleine Würfelchen in die Welt ein. Diese sollen an zufälligen Positionen in der oberen Hälfte des Bildschirm erzeugt werden. Schöner sieht es aus, wenn auch die Verdrehung zufällig ist.
     Wenn man die Reibung (friction) der Würfel verringert, dann rutschen sie allmählich auseinandern.
   - Fügen Sie der Webseite Schieberegler hinzu, die die Schwerkraft der Simulation in X- und in Y-Richtung verändern. Ordentlich beschriftet sollen sie natürlich auch sein.
   - Fügen Sie der HTML-Seite einen "Wasserhahn"-Button hinzu, der fortlaufend kleine blaue Kreise (Wassertropfen) in der Welt erzeugt, solange die Maus auf dem Button gedrückt gehalten wird.
   - Fügen Sie der HTML-Seite einen "Kanone"-Button hinzu, der größere Kreise von einer bestimmten Position aus verschießt. Die Kreise sollten zunächst schräg nach oben fliegen. Gegebenenfalls können Sie die Stärke des Schußes zufällig wählen oder irgendwie steuren. Wenn das Gewicht der Kreise groß ist, können sie andere Objekte wegschießen.
5. Gruppenmitglieder: Nachdem Sie eine Aufgabe erfüllt haben, stagen/commiten Sie die Änderungen. Pushen Sie die Änderungen auch in ihr Online-Repository. Gehen Sie dann zur Online-Umgebung von Git-Camp wählen Ihr Repository aus. Wählen Sie dann den Reiter "Pull-Request" und erzeugen Sie einen neuen Pull-Request. Das Ziel des Requests muss das Repository des Gruppenmanagers sein. Von Ihrem eigenen Repository soll gepullt werden. Sie sehen, welche Änderungen im Vergleich zum Manager Sie vorgenommen haben.
   Der Manager erhält dadurch eine Nachricht (ggf. sogar eine E-Mail), dass Sie die

Einarbeitungen der Änderungen wünschen. Sie sollten daher dem Pull-Request einen sinnvollen Titel und eine Beschreibung Ihrer Arbeit hinzufügen. Andernfalls wird der Manager den Request vermutlich gar nicht erst in Betracht ziehen.

6. Gruppenmanager: Sie sehen in Git-Camp, dass ein Pull-Request eingetroffen ist.
   - Variante A: Sie können dem Pull-Request direkt online stattgeben und die Repos zusammenführen. Dies ist für sehr einfache Änderungen (z. B. Tippfehler) sinnvoll. Holen Sie sich im Anschluß mit Pull die neue Version in ihr lokales Repo.
   - Variante B: Größere Änderungen sollten vorher begutachtet werden. Dann sollten Sie zunächst das Repo des Gruppenmitgliedes herunterladen und lokal zusammenführen.
     Fügen Sie wie in der Übung "Zusammenarbeit Schüler-Lerher" ein neues Remote-Verzeichnis für das Repo des Gruppenmitgliedes hinzu. Führen Sie einen Fetch durch und checken Sie dann den master-Branch des neues Remote-Verzeichnisses out. Stimmen Sie dem Anlegen eines lokalen Branches zu. Wählen Sie diesen lokalen Branch aus (check out) und testen Sie die Neuerungen.
     Wenn Sie mit diesen Neuerungen einverstanden sind, dann wählen Sie ihren eigenen lokalen Master-Branch aus (check out). Dieser muss unbedingt fett geschrieben und mit einem Pfeil markiert sein! Wählen Sie dann im Kontext-Menü des Branches mit den Neuerungen den Menüpunkt "Merge" (Merge to Working Tree) aus. Dort müssen Sie nun gegebenenfalls Konflikte mit dem Conflict Solver lösen.
     Schließen Sie den Merge-Prozess ab, indem Sie eine sinnvolle Commit-Message schreiben und die Änderungen sichern. Pushen Sie die Änderungen in Ihr Online-Repo.
7. Gruppenmitglieder: Wenn Sie eine Aufgabe erfüllt und den Pull-Request erstellt haben, dann können Sie eine weiter Aufgabe in Angriff nehmen. Erstellen Sie dazu einen neuen Fork, da Sie ja jetzt eine neue Aufgabe bearbeiten. Hat der Gruppenmanager schon eine neue Version hochgeladen, haben Sie auch alle Neuerungen im neuen Fork dabei (auch Ihre eigenen). Wählen Sie also eine weitere Aufgabe aus der obigen Liste aus oder denken Sie sich etwas eigenes aus.

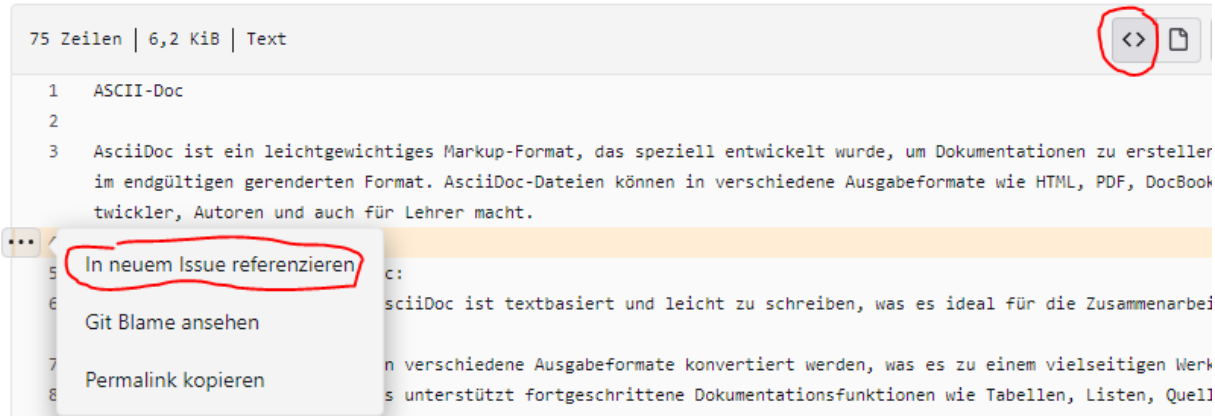
7.6. Mit ZPG-Materialien arbeiten (Verändern / Fehler melden usw.)

Auch die Zusammenarbeit mit uns Fortbildnern soll in Zukunft über GIT vereinfacht werden: Sie können entweder Fehler und Anregungen an uns melden oder selbst Fehler beheben, Verbesserungen implementieren oder Texte überarbeiten.

1. **Fehler melden:** Wählen Sie in Git-Camp das Repo mit unseren Fortbildungsmaterialien aus. Vielleicht haben Sie ja schon einen Tippfehler gefunden oder eine andere Ungenauigkeit festgestellt. Dann melden Sie dies nun. Wählen Sie den Reiter "Issues". Legen Sie einen neuen Issue an. Wenn Sie keinen Fehler gefunden haben, hinterlassen Sie einfach einen (netten) Kommentar für uns.

   Dieser Issue kann an eine bestimmte Zeile im Code gebunden werden. Das erleichtert den Fortbildnerinnen die Arbeit. Dazu lassen Sie sich im Gitcamp den Quellcode einer Datei anzeigen, so dass links Zeilennummern zu sehen sind (Quellcode muss explizit rechts oben

gewählt werden!). Dann kann in der fehlerhaften Zeile ganz links geklickt und ein Issue zu dieser Zeile erstellt werden.



2. **Fehler selbst beheben:** Sie sollen das bereitgestellte Dokument Asciidoc.adoc überarbeiten. Da ist mit der Formatierung noch einiges schief gelaufen. Arbeiten Sie bitte genauso wie in Übung 5 und erstellen Sie einen Fork der Materialien zu dieser Fortbildung, arbeiten Sie die Änderungen an Asciidoc.adoc ein und stellen Sie danach einen Pull-Request. Der oder diejenigen Veranwortlichen für die Materialien werden sich den Request anschauen und eine Rückmeldung geben. Wird er angenommen, haben Sie vielleicht für viele Lehrer zu einer Verbesserung des Unterrichts beigetragen. Sollte er abgelehnt werden, können Sie trotzdem ihre Änderungen selbst verwenden.

   Sollte später einmal ein Request abgelehnt werden, müssen Sie trotzdem Änderungen der Originalmaterialien mit Ihren (abgelehnten) Änderungen zusammenführen. Sonst profitieren Sie entweder nicht mehr von den Verbesserungen am Originalmaterial oder verlieren Ihre eigenen Änderungen.

   Führen Sie dazu folgende Schritte durch:

   - Legen Sie lokal ein neues Remote-Verzeichnis mit dem Repo der Fortbildungsmaterialien an. Führen Sie dort einen Fetch durch und checken Sie dann den master-Branch des neues Remote-Verzeichnisses out. (Vgl. Übung 4)
   - Wählen Sie ihren eigenen lokalen Master-Branch aus (check out). Dieser muss unbedingt fett geschrieben und mit einem Pfeil markiert sein! Wählen Sie dann im Kontext-Menü des Branches der Originalmaterialien den Menüpunkt "Merge" (Merge to Working Tree) aus. Dort müssen Sie nun gegebenenfalls Konflikte mit dem Conflict Solver lösen.
     Schließen Sie den Merge-Prozess ab, indem Sie eine sinnvolle Commit-Message schreiben und die Änderungen sichern.

8. Glossar (Kopie aus Urs Dokument)

1. Arbeitsverzeichnis
   Ordner, der mit Hilfe eines Git-Repository verwaltet werden soll.
2. repository
   Git-eigene Datenbank, die entweder direkt im .git-Unterordner des Arbeitsverzeichnisses liegt (lokales Repo), oder auch auf einem anderen Rechner.
3. staging area (manchmal auch "Index")
   Vorbereitungsschritt, der geänderte Dateien für den nächsten Commit sammelt.
4. commit
   Ein Versionsstand des Projekts, der mit Datum, Kommentar und weiteren Angaben festgehalten wurde; als Verb committen: einen Commit vornehmen.
5. clean
   Das Arbeitsverzeichnis ist clean, wenn es mit dem letzten Commit exakt übereinstimmt.
6. checkout
   bringt das Arbeitsverzeichnis auf den Stand eines bestimmten Commit (oder Branch). Es sollte dafür vor dem Checkout clean sein.
7. merge
   auseinander gelaufene Versionen zusammenführen; das können Differenzen zwischen zwei lokalen Branches sein, aber auch Änderungen im gleichen Branch zwischen zwei Teammitgliedern.
8. push
   neue Commits vom lokalen Repo ins Server-Repo schieben.
9. fetch
   neue Commits vom Server ins lokale Repo holen.
10. pull
    fetch und dann merge.
11. Konflikt
    tritt auf, wenn zwei "auseinandergelaufene" Versionen Änderungen in gleichen Zeilen enthalten, und muss von Hand aufgelöst werden.
12. branch
    Absichtlich eingerichtete Verzweigung der Versionsgeschichte.

[1] Bei älteren git-Versionen heißt der Hauptbranch, der hier angezeigt wird, gelegentlich `master`, in diesem Fall muss man sich stets `main` durch `master` ersetzt denken.

[2] Genau genommen zeigt `main` auf den aktuellen Stand des Branches mit dem Namen main; arbeitet man mit mehreren Branches, gibt es weitere Zeiger.

[3] Diese Anwendungsszenario lässt sich auch mit einem Clone umsetzen. Allerdings ist ein Clone normalerweise dafür vorgesehen, wenn man zusammen an einem einzigen Projekt arbeiten möchte. Jeder Schüler/jede Schülerin arbeitet aber an einem eigenen Projekt. Daher sollte hier ein Fork gewählt werden.