Andreas Nicolai

GIT Workshop

Grundlagen und gemeinschaftliches Arbeiten im Team

Teil 1 – Grundlagen (lokales git)

Repository erstellen

In Windows Kommandozeile oder Linux Bash (oder in SmartGit direkt, siehe Beispiel später)

Ein lokales Repository und Arbeitskopie erstellen:
Enthält ein **.git**-Verzeichnis mit der Änderungshistorie.
Alle anderen Dateien/Verzeichnisse sind Teil
der **lokalen Arbeitskopie**.

- `git init <reponame>`

Bare-Repository erstellen:
Das Verzeichnis enthält dann den Inhalt
des .git-Verzeichnisses. Solche bare-Repos liegen
normalerweise auf dem Server.

- `git init --bare <reponame>`

Dateizugriffsrechte regeln, wer auf das Repository zugreifen kann. Bei mehreren Gruppen mit individuellen Rechten sollte man unter Linux die ACL verwenden.

Lokales Repo in bare-Repo wandeln

Wenn man bspw. lokal angefangen hat, und später das Verzeichnis auf einen Server hochladen will:

1. Das .git-Verzeichnis auf den Server hochladen und umbenennen, bspw. MeinRepo.git.
2. In das Verzeichnis wechseln und folgenden Befehl ausführen:

```
➢ git config core.bare true
```

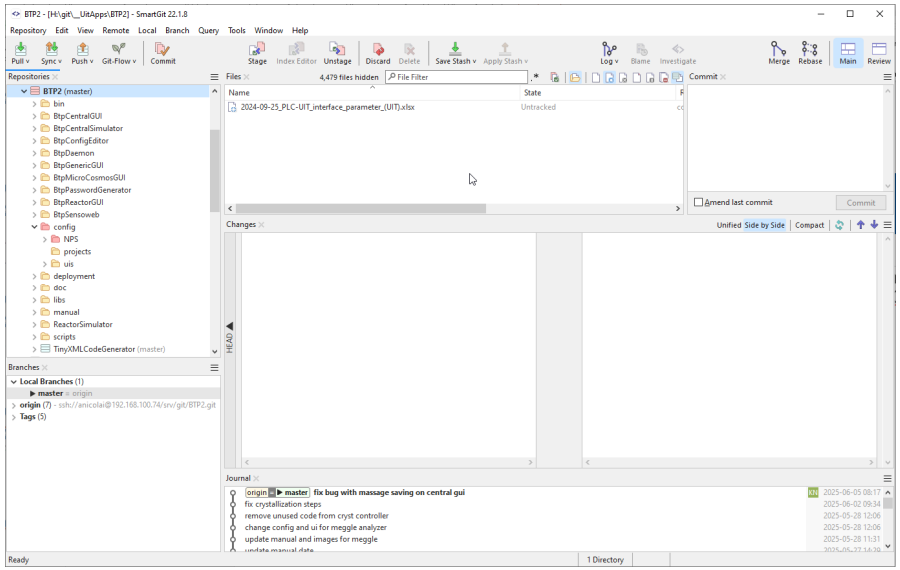
Das ist gleichbedeutend mit dem Bearbeiten der Datei `MainRepo.git/config`:

```
[core]
        …
        bare = true
        …
```

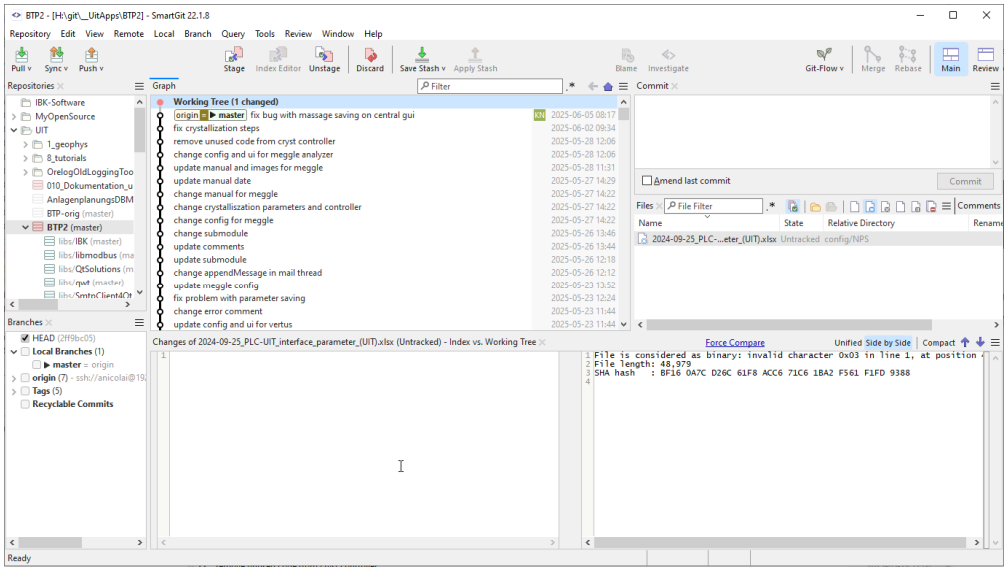
Grundlagen – SmartGit

git Kommandozeilenbedienung ist sehr komplex, fehleranfällig und zeitraubend, d.h. macht keinen Spaß… daher besser alles mit GUI-Frontend machen (wie in diesem Workshop). Viele Verschiedene Frontends sind verfügbar – wir verwenden SmartGit (funktioniert in Windows, Mac, Linux und hat eigentlich alles, was man braucht).

SmartGit Konfiguration



Working Directory View (recommended)



Log View

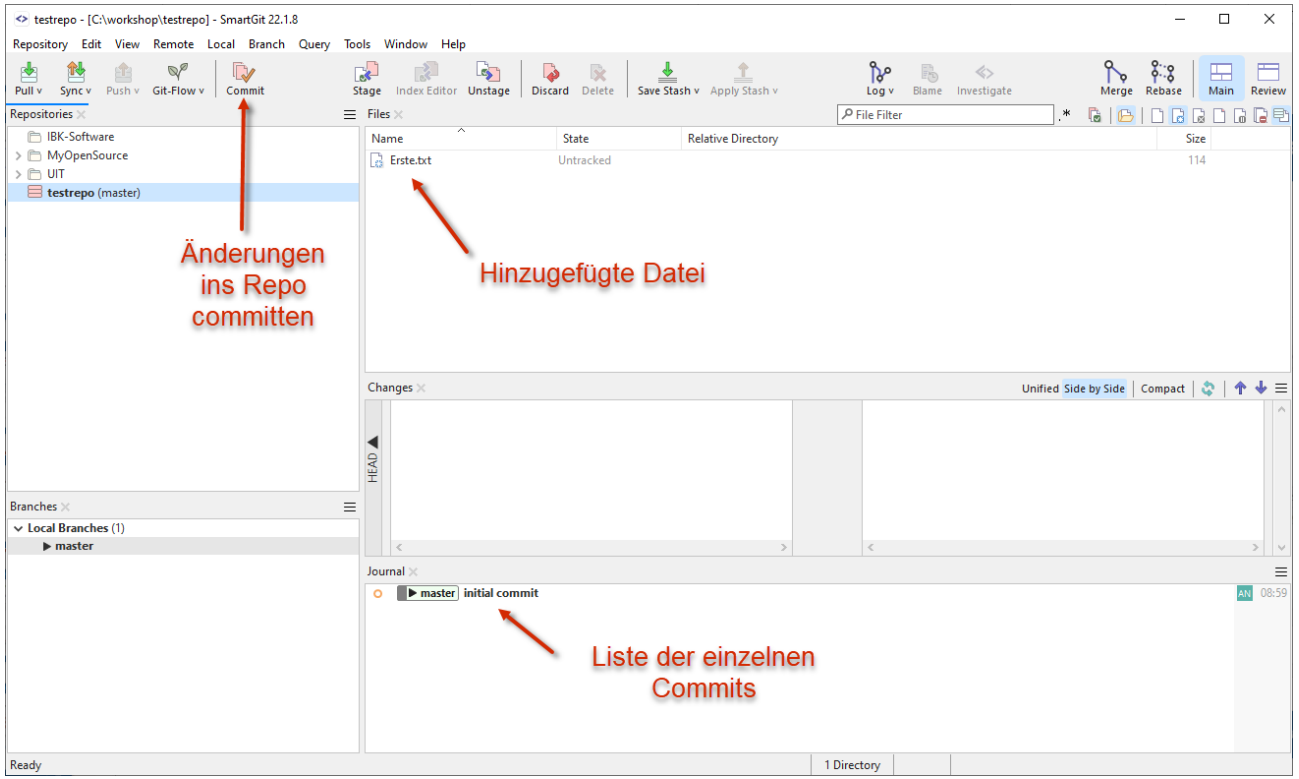
Grundlagen – Commits machen

Änderungen im lokalen Arbeitsverzeichnis:

- Hinzufügen neuer Dateien
- Dateien umbenennen (ist sehr datensparsam, da sich git nur die geänderten Dateinamen merkt)
- Dateien ändern
- Dateien löschen

Analog für Verzeichnisse.

Leere Verzeichnisse werden nicht ins git-repo übertragen, da git selbst keine Verzeichnisse kennt, sondern nur Pfade relativ zum Repo-Wurzelverzeichnis – Verzeichnisse ergeben sich implizit daraus.



Grundlagen – Arbeiten mit Commits und dem Log

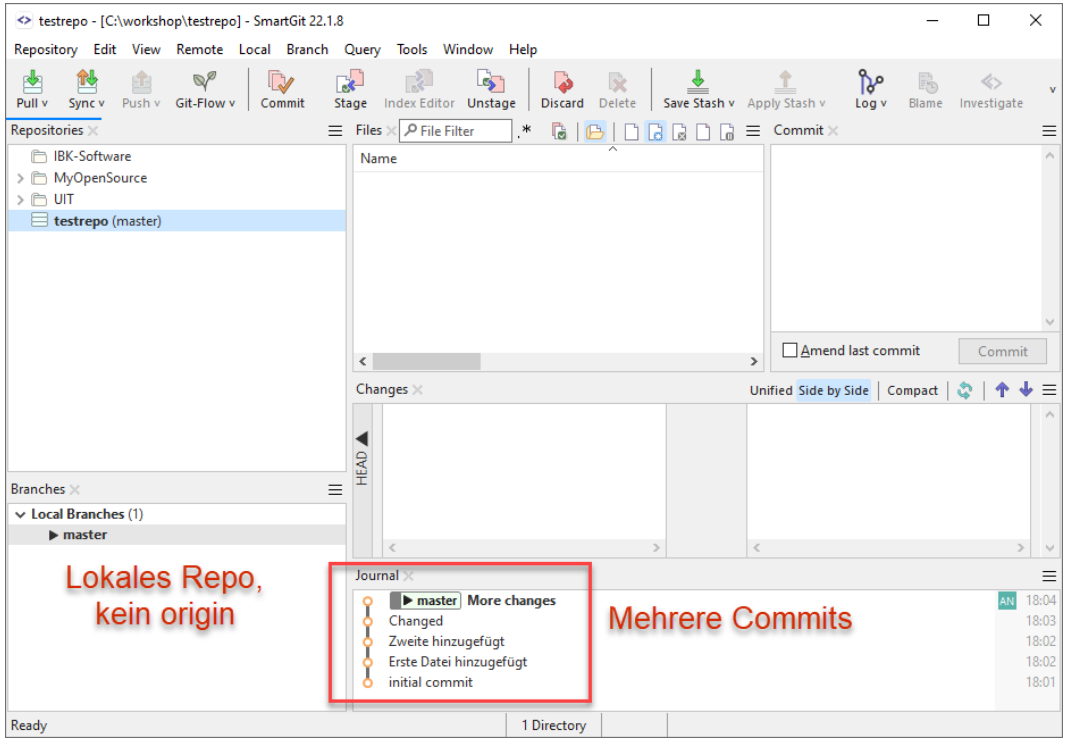
UMWELTLEISTUNGEN

Jetzt ein paar Beispiele mit einer lokal erstellten Arbeitskopie (zum Mitmachen).

In SmartGit: *Repository->Add or Create…*

Dann Pfad für neues Repo eingeben (bspw. c:\workshop\testrepo) (kein Bare-Repository, also keine Endung .git)

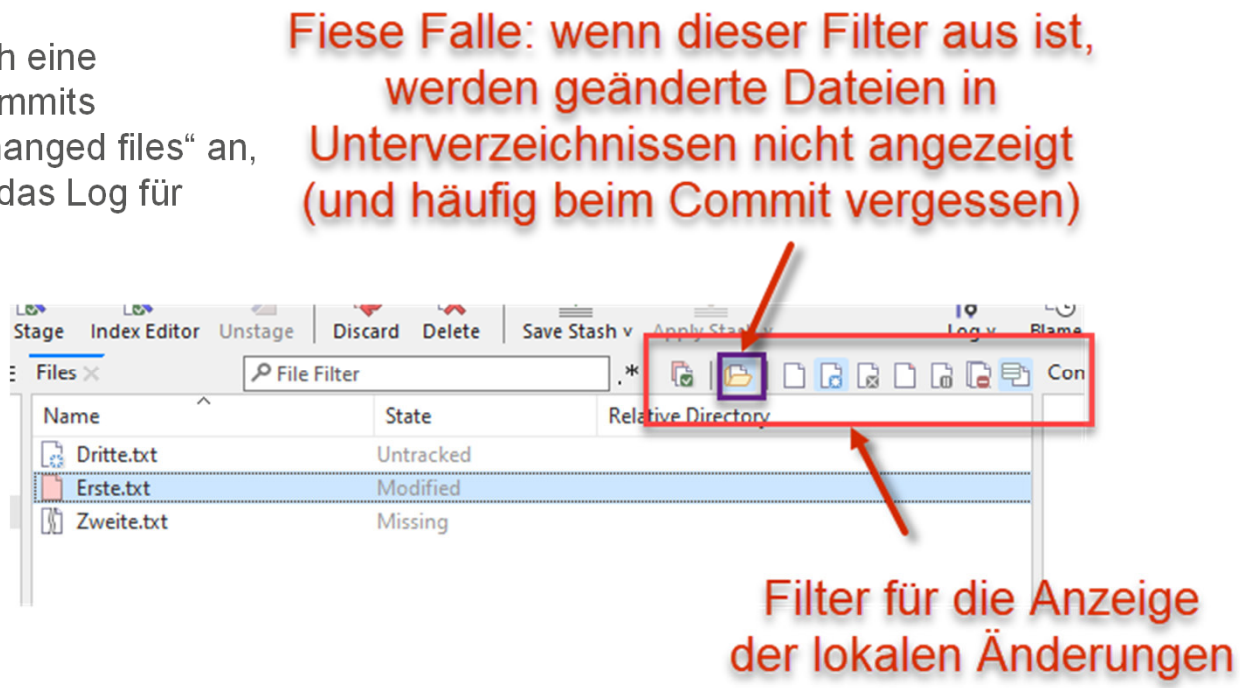
Workshop-Selber-Machen: Mehrfach Dateien einchecken, oder verändern, und schließlich:



Änderungen filtern

Die Liste der lokal geänderten Dateien lässt sich nach Zustand „State“ sortieren oder nach verschiedenen Kriterien filtern.

TIPP: wenn man untersuchen will, wie sich eine bestimmte Datei im Verlauf der letzten Commits verändert hat, macht man den Filter „unchanged files“ an, markiert die gewünschte Datei und öffnet das Log für diese Datei.



UMWELTLEISTUNGEN

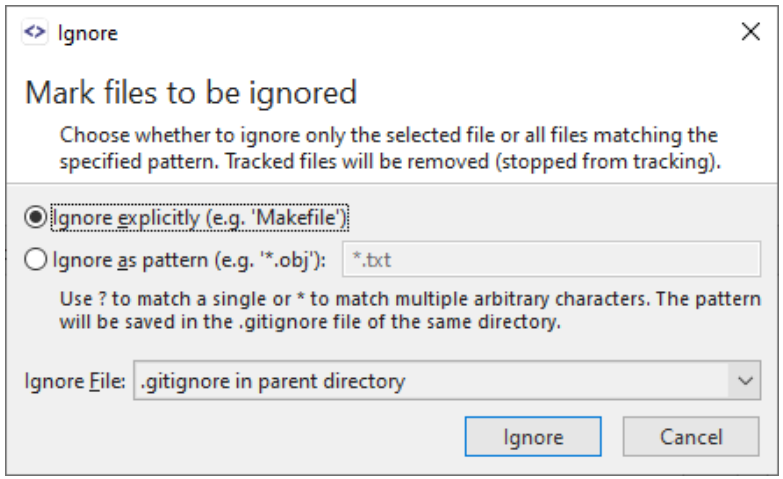
Nicht zu committende Dateien ausblenden/ignorieren

ACHTUNG: Niemals generierte (Binär-)Dateien ins Repo einchecken. Auch große Dateien vermeiden.

Um Dateien auszuschließen (zu ignorieren) die Datei `.gitignore` anlegen und dort explizite Dateipfade (relativ zur `.gitignore`) oder Dateinamen oder Dateinamen mit Wildcards eintragen.

In SmartGit:

- Zu ignorierende Datei auswählen und Strg+I: dann entweder explizit ignorieren oder Suchmaske festlegen
- Zu ignorierendes Verzeichnis (z.B. build-Verzeichnis) im Verzeichnisbaum markieren und Strg+I drücken



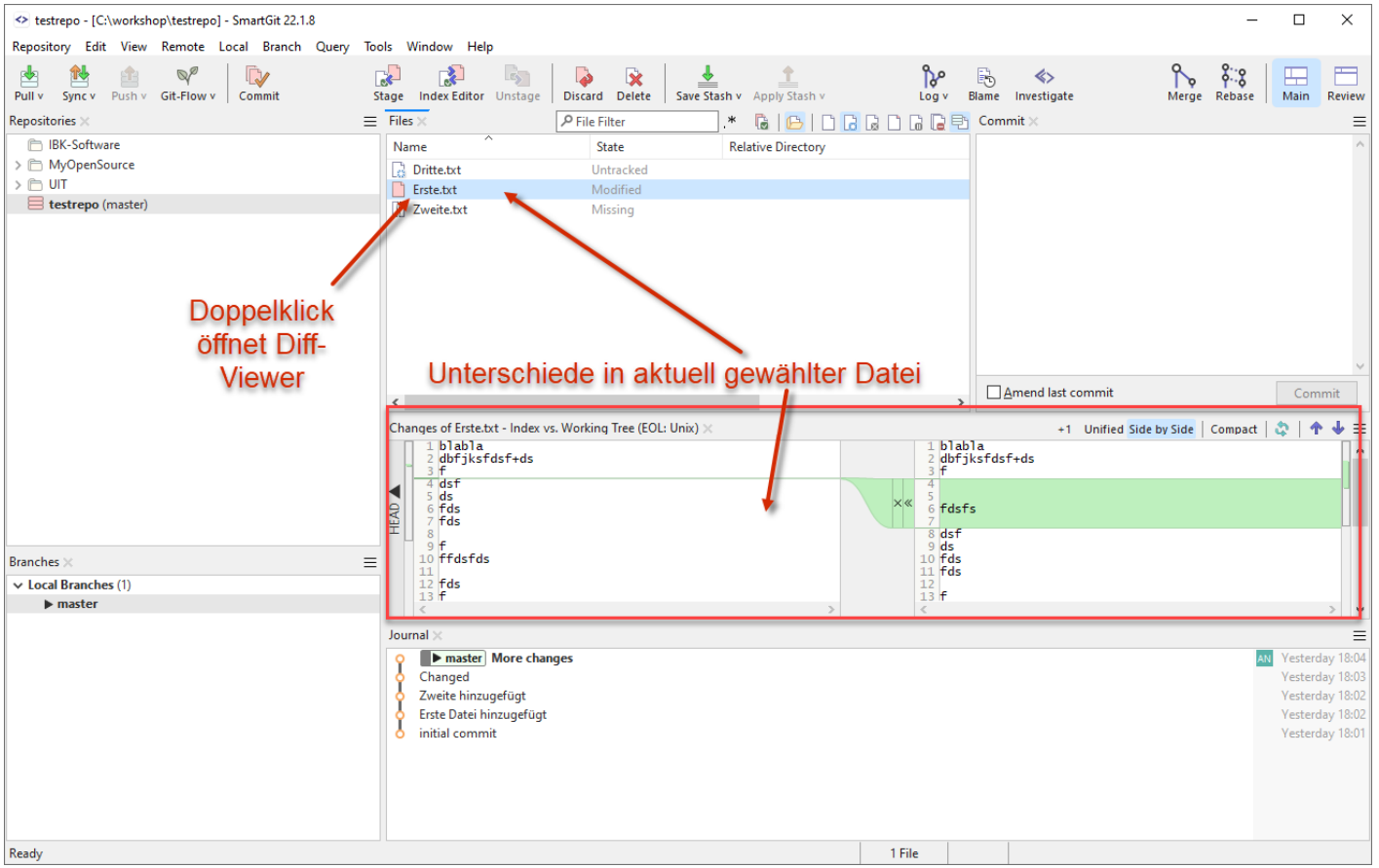
TIPP: `.gitignore`-Dateien gelten immer für das Verzeichnis und alle Kindverzeichnisse (recursiv).

Änderungen in Dateien reviewen

Normales Fenster zeigt Änderungen.

TIPP: in der „Review“ – Perspektive (Window-Menü) bekommt das Diff-Fenster mehr Platz. Am Besten die Buttons *Main* und *Review* in die Toolbar aufnehmen.

Doppelklick auf eine Datei in Dateiliste öffnet Diff-Viewer:

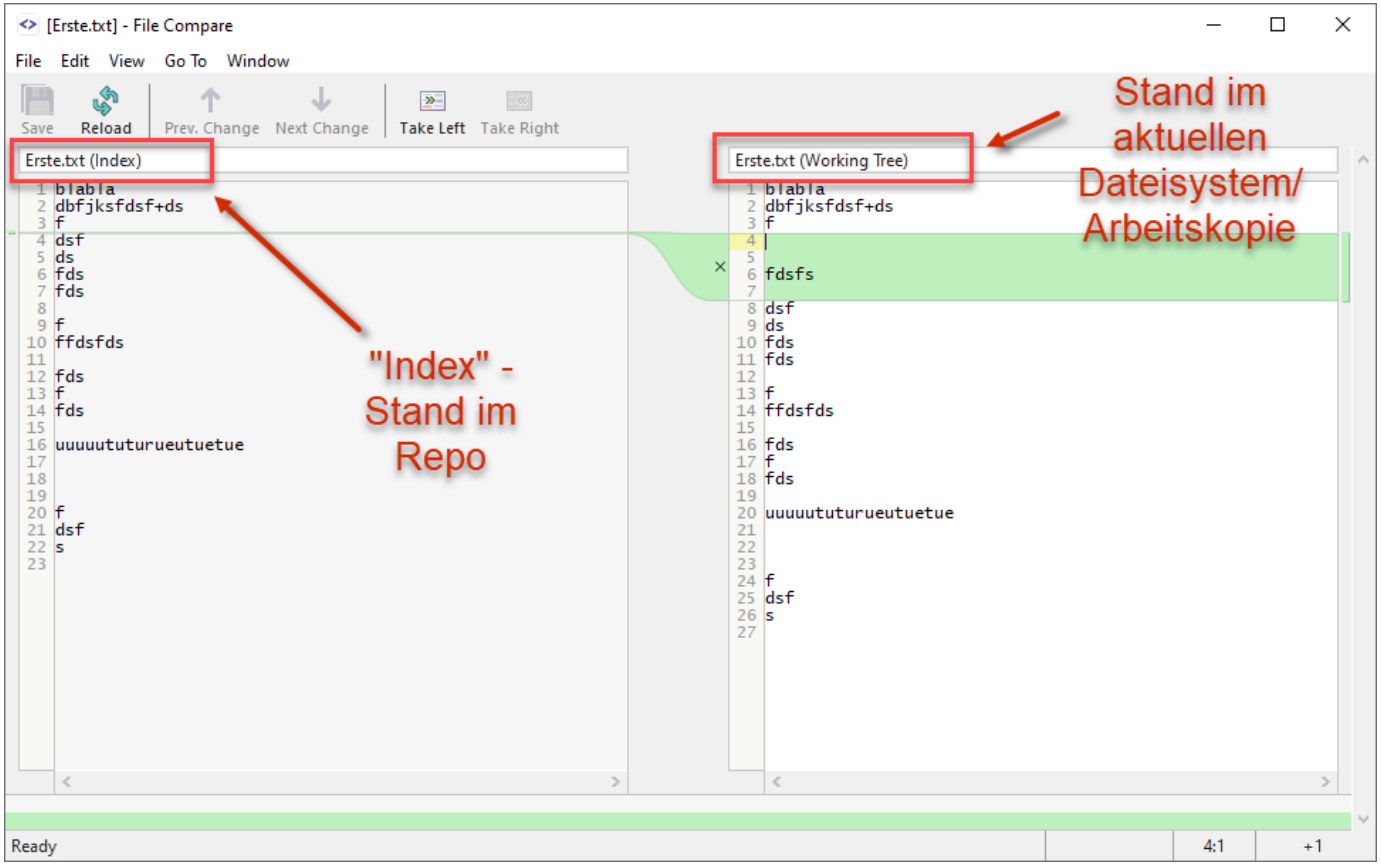


UMWELTLEISTUNGEN

Änderungen in Dateien reviewen

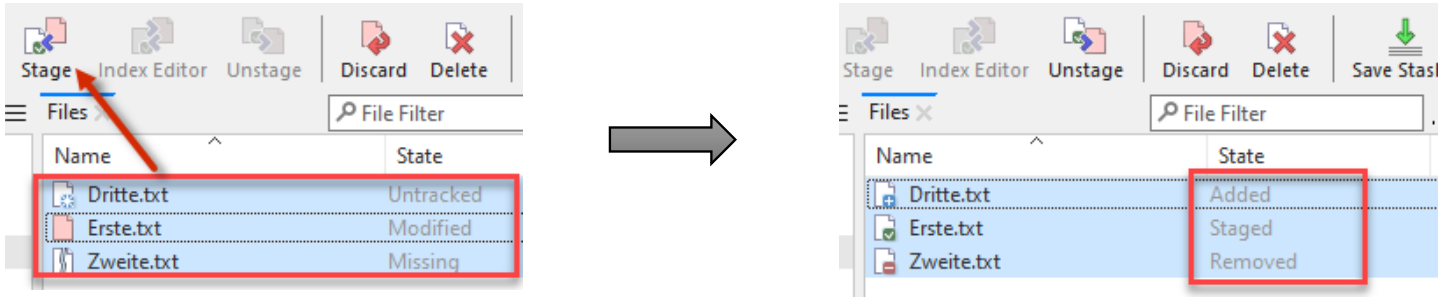
Diff-Viewer erlaubt direkt nochmal Änderungen zu reviewen und zu bearbeiten.

ACHTUNG: Wenn die Datei schon „gestaged“ ist (dazu gleich mehr), kann man die lokale Version NICHT MEHR bearbeiten. Dann muss man die Datei erst „unstagen“.



Dateien für ein Commit vormerken („stagen“)

Der Stage-Bereich von git ist eine Liste von vorgemerkten **Kopien der Dateien**, welche demnächst comitted werden sollen.



- Untracked (neue Dateien) -> Added
- Modified (geänderte Dateien) -> Staged
- Missing (fehlende Dateien) -> Removed

ACHTUNG: Wenn die Datei schon „gestaged“ ist, kann man die lokale Version NICHT MEHR im Diff-Viewer bearbeiten. Dann muss man die Datei erst „unstagen“.

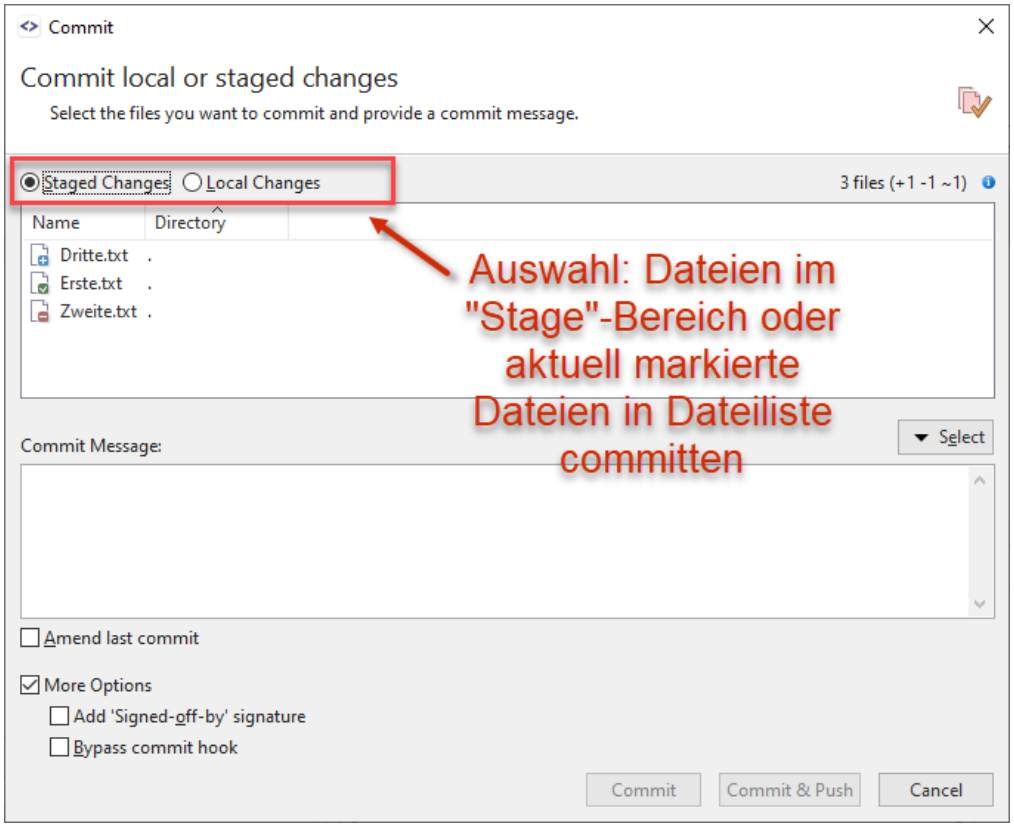
Wird die Datei im Dateisystem geändert, wird der Status als „Staged Modified“.

UMWELTLEISTUNGEN

Dateien im Stage-Bereich oder aktuell markierte Dateien committen

TIPP: Bei größeren Änderungen oder vielen unterschiedlichen Änderungen lohnt es sich, die Änderungen im Stage-Bereich passend zu einem bestimmten Thema/Issue/Ticket zu sammeln und so die Änderungen in mehrere Commits zu unterteilen. Somit lassen sich später Änderungen leichter nachvollziehen.

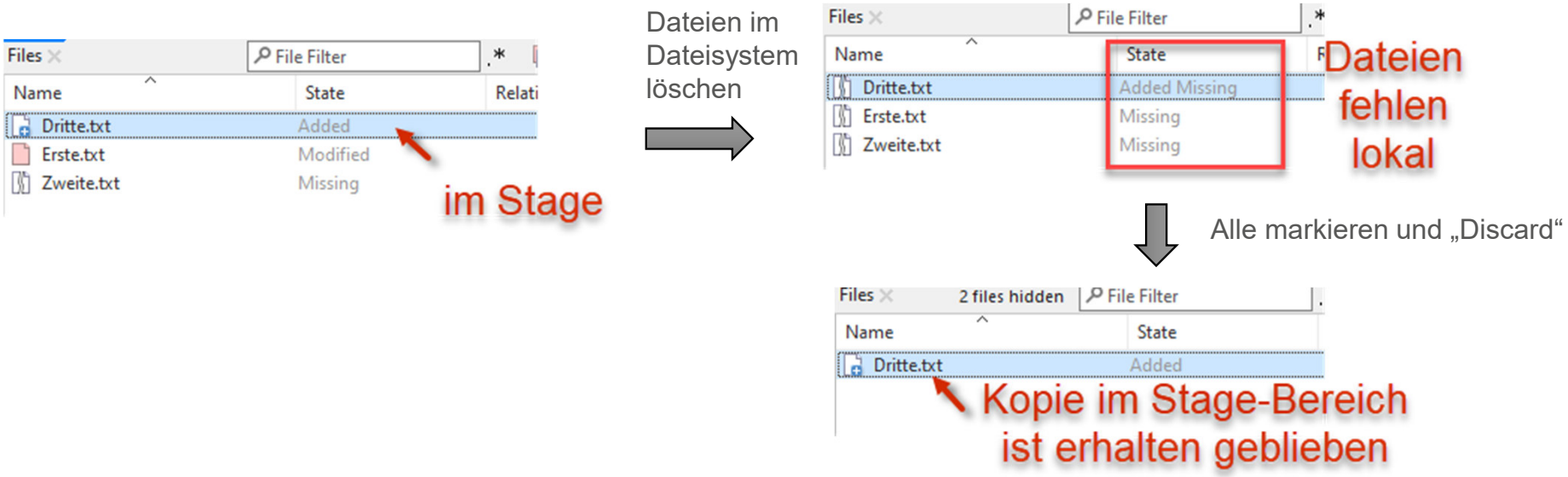
ACHTUNG: Wird eine Datei „Stage“-Bereich später nochmal im Dateisystem geändert, wird der Status als „Staged Modified“ angezeigt. Es gibt dann *zwei Kopien* der gleichen Datei. Wenn man im Commit-Dialog dann „Staged Changes“ auswählt, wird nur die vorher in den Stage geschobene Datei committed – die danach gemachten Änderungen werden nicht committed. Daher besser vorher nochmal die geänderte Datei „stagen“.



Lokale Änderungen rückgängig machen

Lokale Änderungen können, wenn noch nicht gestaged, durch „Discard“ rückgängig gemacht werden. Falls Dateien bereits gestaged sind, erst auf „Unstage“ und dann auf „Discard“.

TIPP: Ein komplettes Repo wieder auf den frisch geklonten Zustand zu versetzen kann man einfach machen, wenn man alle Dateien bis auf das .git-Verzeichnis löscht, dann alle Änderungen markiert und mit „Discard“ zurücksetzt.



Commit rückgängig machen

Wenn man in einem Commit was vergessen hat, kann man diesen einfach rückgängig machen:
Local->Undo Last Commit…

Die im Commit geänderten Dateien werden zurück in den Staging-Bereich überführt.

ACHTUNG:

Für jede im letzten Commit geänderte Datei gilt dann:

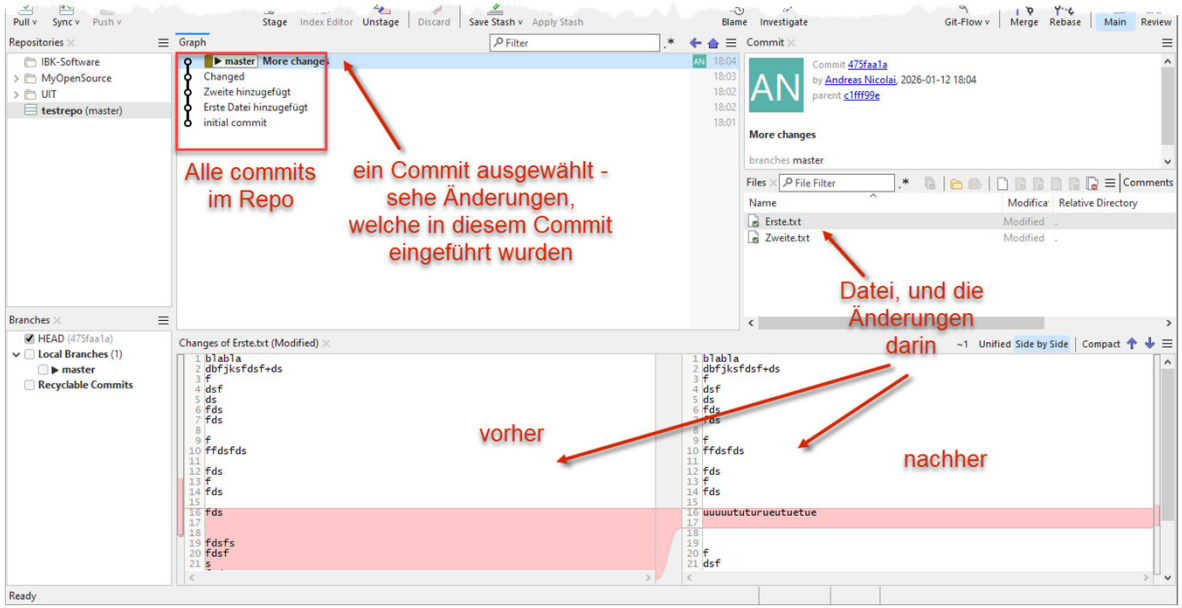
- Gibt es die Datei schon im Stage-Bereich (d.h. wurde zwischenzeitlich erneut geändert und in den Stage-Bereich geschoben), so bleibt die aktuelle Version der Datei im Stage unverändert erhalten.
- Gibt es die Datei noch nicht im Stage, wird die Dateiversion aus dem Commit in den Stage überführt.

TIPP: Empfohlener Workflow bei „vergessenen Dateien“ in einem Commit:

1. Fehlende Dateien oder Dateien mit fehlenden Änderungen in den Stage schieben
2. Letzten Commit rückgängig machen
3. Erneut Commit der „Staged Changes“ machen

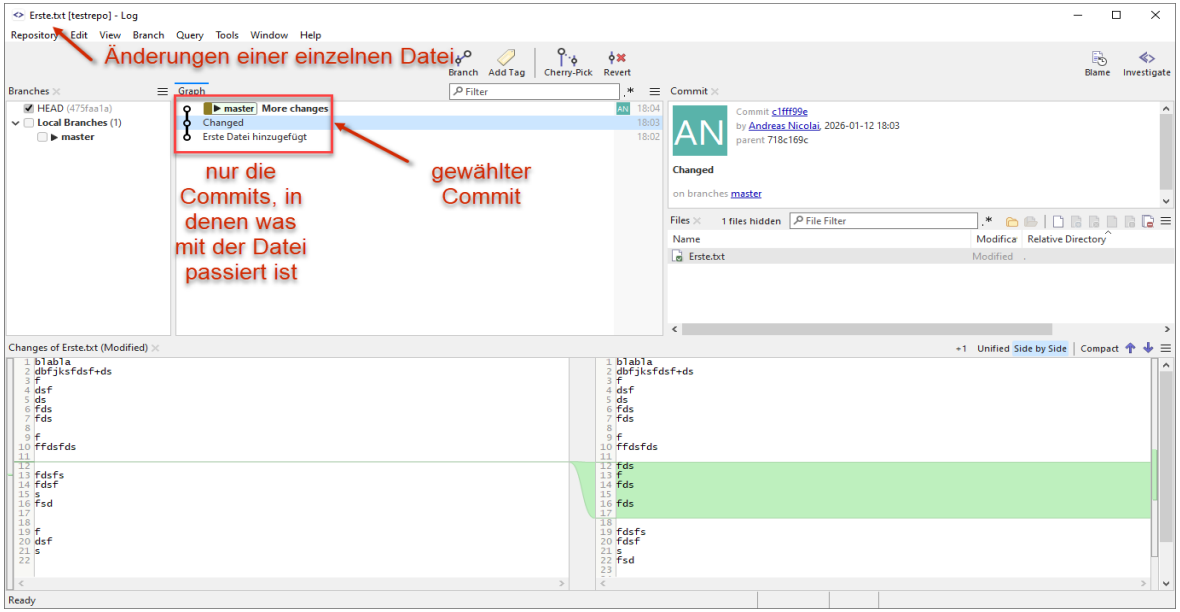
Änderungshistorie ansehen - Repository Log öffnen

- Window->Show Log Window -> zeigt **komplette Historie**
- Ein Verzeichnis markieren und Strg+L (oder im Kontextmenü): zeigt Änderungen der Dateien unterhalb dieses Verzeichnisses
- Eine Datei markieren und Strg+L (oder im Kontextmenü): zeigt ausschließlich Änderungen an dieser Datei



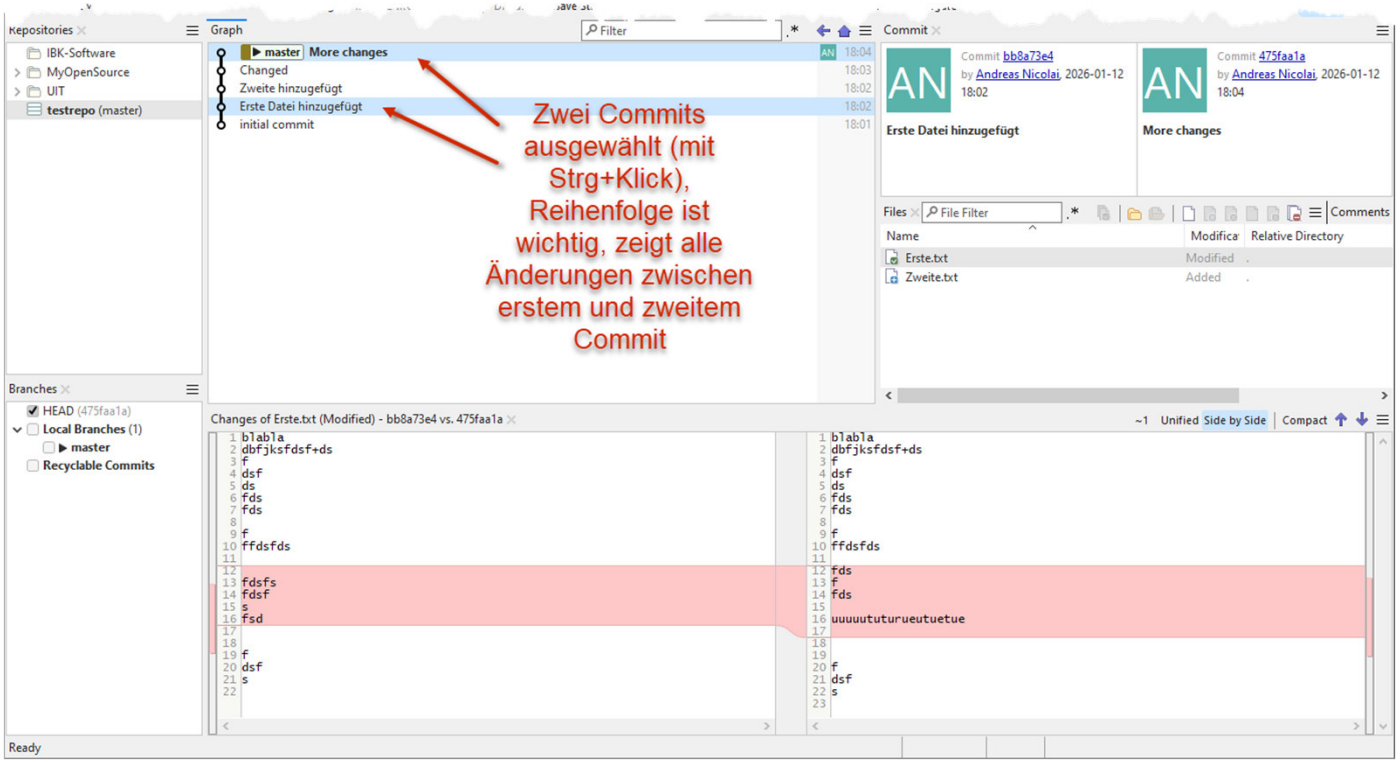
Änderungshistorie ansehen - Repository Log öffnen

- Window->Show Log Window -> zeigt komplette Historie
- Ein Verzeichnis markieren und Strg+L (oder im Kontextmenü):
  zeigt Änderungen der Dateien unterhalb dieses Verzeichnisses
- Eine Datei markieren und Strg+L (oder im Kontextmenü):
  zeigt **ausschließlich Änderungen an dieser Datei**



Repository - Gesamtlog über mehrere Commits anzeigen

Wenn man zwei Commits auswählt, werden alle Änderungen zwischen dem zuerst angeklickten und dem als zweites angeklickten Commit vereint gezeigt, also wäre man direkt mit einem Commit vom ersten zum zweiten gelangt.



Teil 2 – Grundlagen (git-server)

Grundlagen – Repo clonen

Ausgangspunkt: repo wurde auf einem Server (selbst gehosteter ssh-Server, github, gitlab, …) erstellt.

Bei eigenem SSH-Server mit git-server, muss jeder Nutzer zunächst die vertrauenswürdigen git-Verzeichnisse hinzufügen.
Das muss auf dem SSH-git-Server gemacht werden, nicht auf dem Desktop-System, wo der git-client läuft.

Eingeloggt auf dem SSH-git-Server:

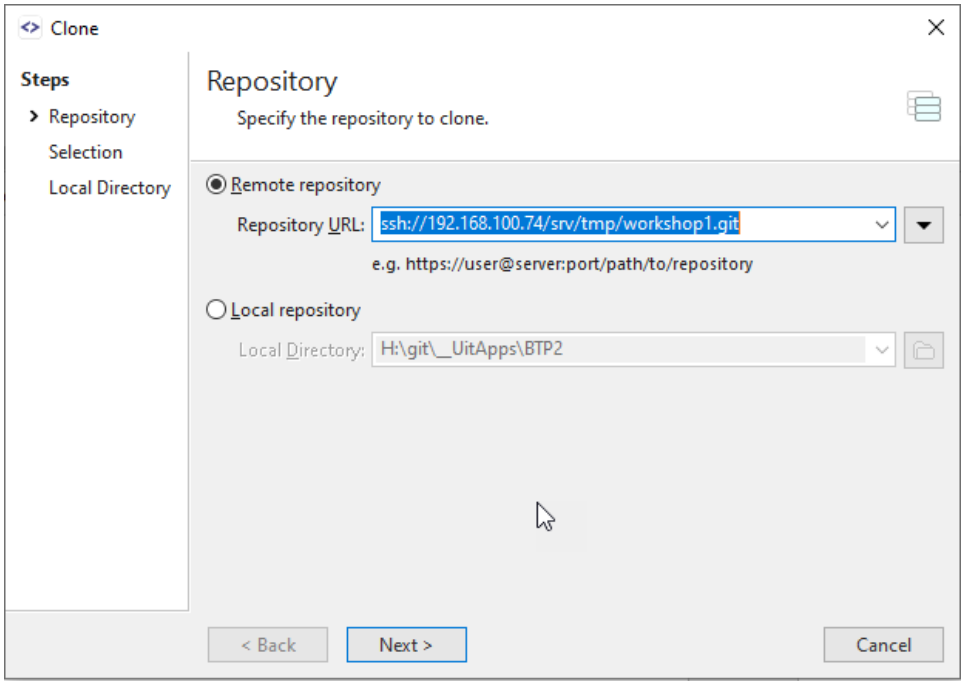
- `git config --global --add safe.directory *`

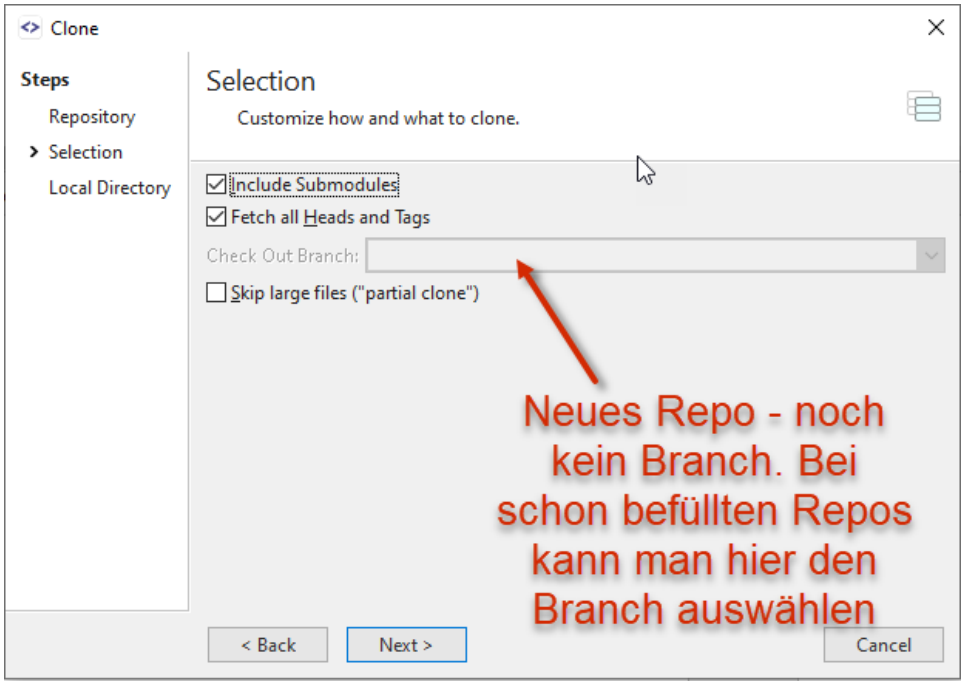
ACHTUNG: kein `'*'` sondern `*` verwenden!

Grundlagen – Repo clonen

Ausgangspunkt: repo wurde auf einem Server (selbst gehosteter ssh-Server, github, gitlab, …) erstellt.

SmartGit: *Repository->Clone…*



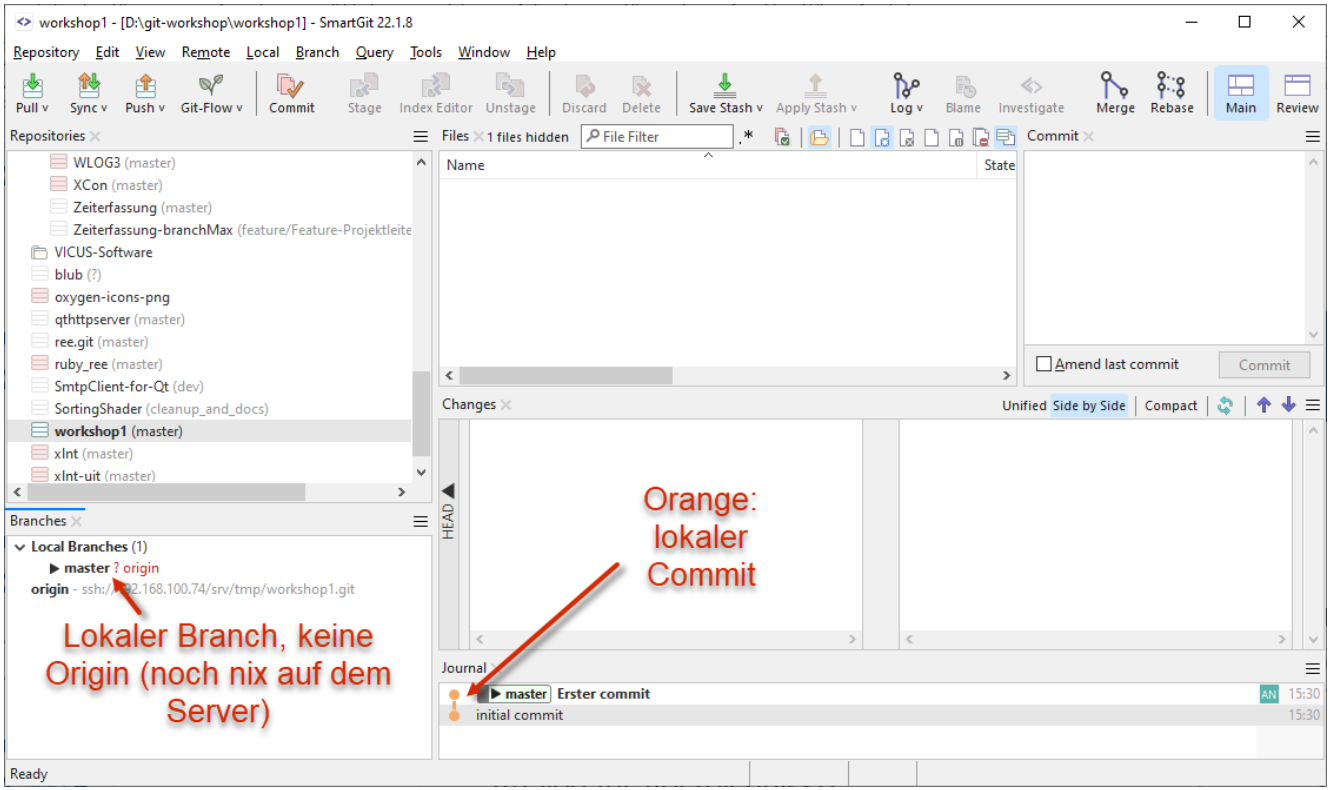


Dann noch das Zielverzeichnis für die lokale Arbeitskopie angeben und fertig.

Grundlagen – Commits machen

UMWELTLEISTUNGEN

Lokale Commits und aktiver Branch:



Commit:
Set von geänderten Dateien
(die „Perlen“)

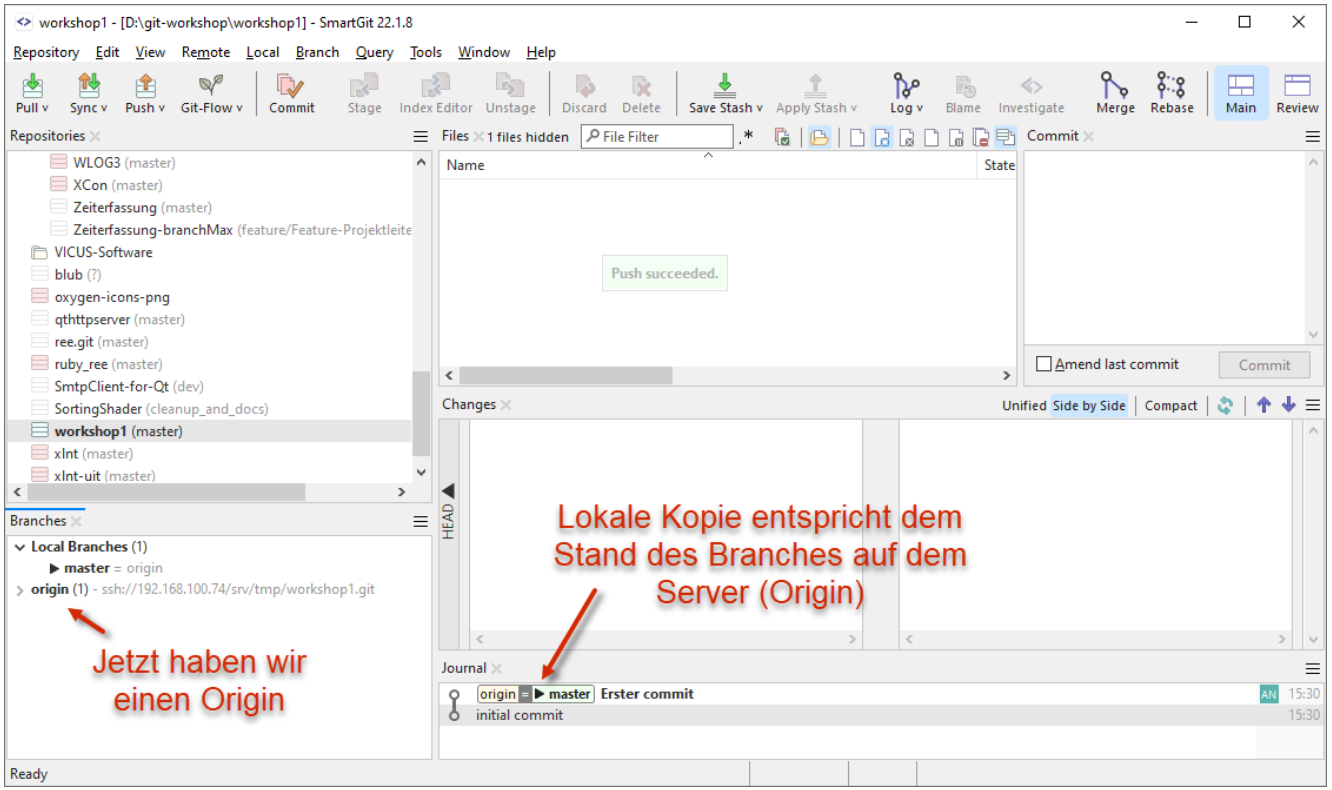
Branch:
Aneinanderreihung
aufeinanderfolgender Commits
(die „Perlenschnur“)

HEAD: Der aktive Branch wird auch als HEAD bezeichnet.

TIPP: es kann mehrere „Perlenschnüre“ geben, die die gleichen „Perlen“ auffädeln!

Grundlagen – Commits pushen

Commits via **push** auf den Server übertragen:

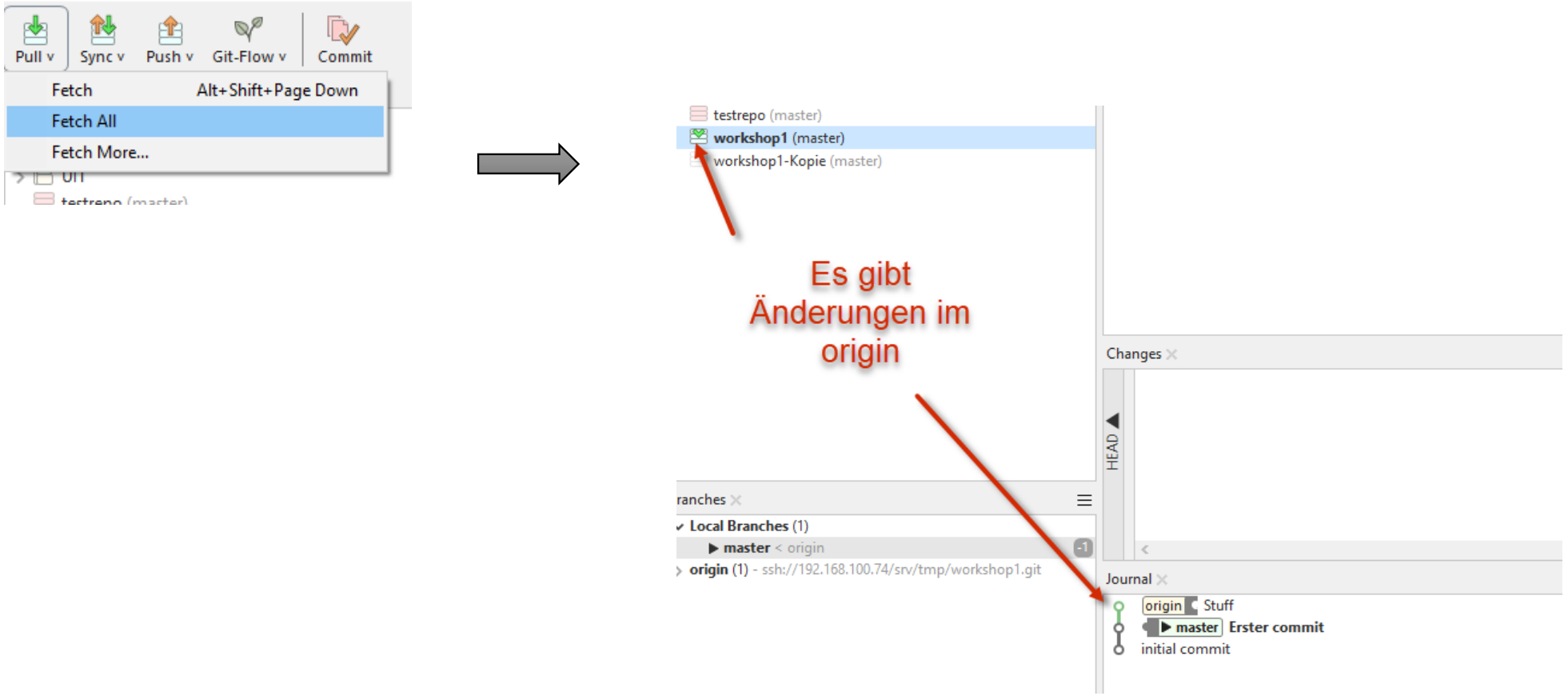


origin:
Ein oder mehrere referenzierte Repositories, mit denen ich das lokale Repository via pull/push abgleichen.

master:
übliche Bezeichnung des Hauptentwicklungszweiges (manchmal auch „main“).

Grundlagen – Entfernte Änderungen holen (fetch/pull)

Fetch – Informationen über entfernte Änderungen holen



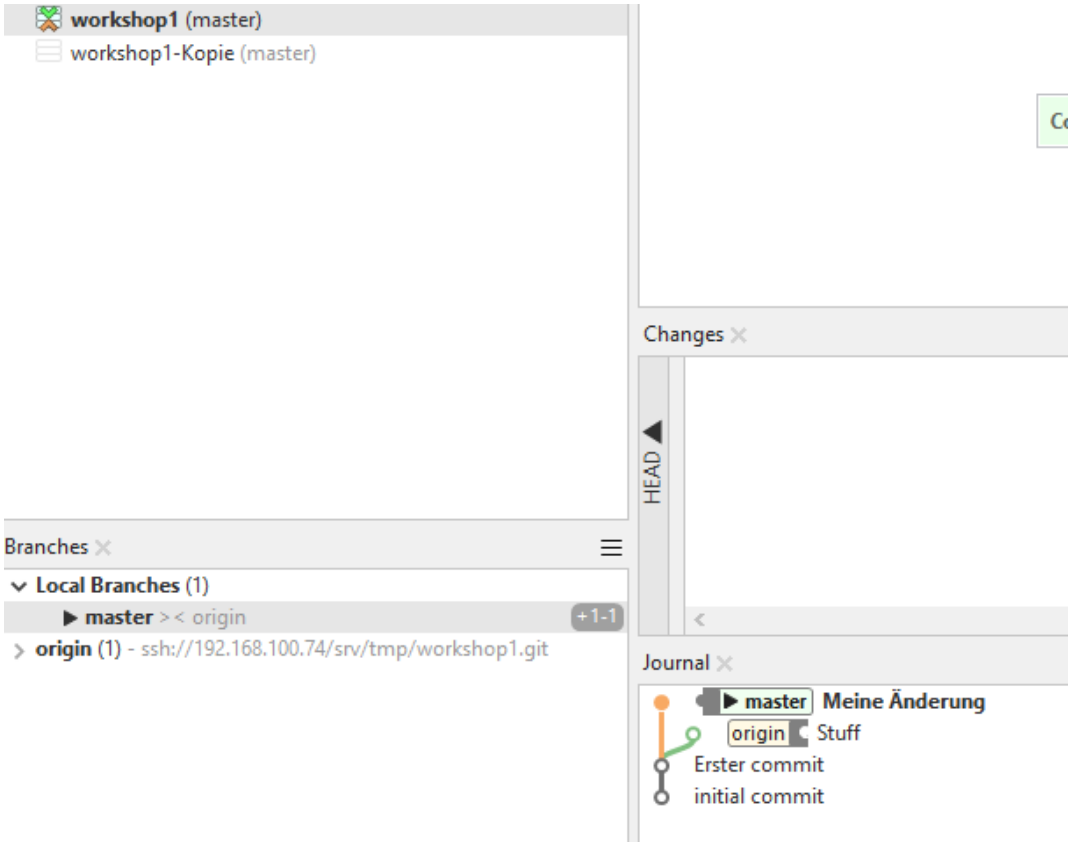
Grundlagen – Entfernte Änderungen holen (fetch/pull)

Fetch – Informationen über entfernte Änderungen holen

Farbkodierung in der Commit-Liste:

Änderungen im entfernten Repo (origin)

Lokale Änderungen



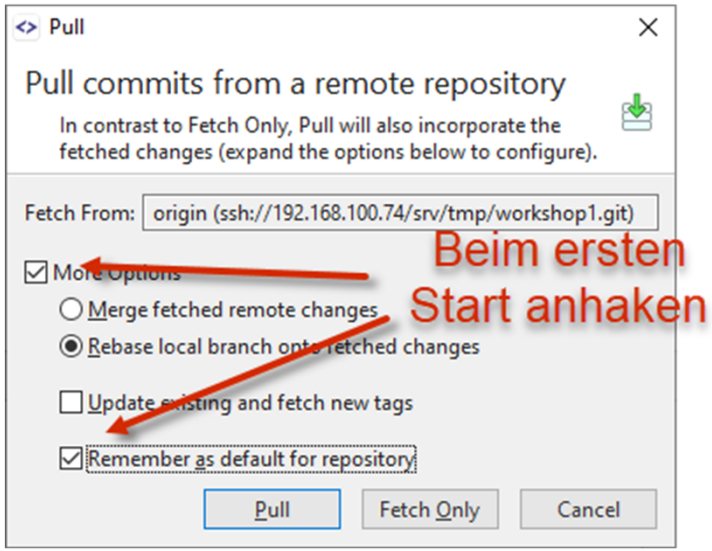
Grundlagen – Entfernte Änderungen holen (fetch/pull)

UMWELTLEISTUNGEN

Pull – Entfernte Änderungen holen und lokal anwenden

Optionen:

- **Rebase** – die eigenen Commits werden mit den entfernten Änderungen verknüpft und es sieht so aus, als wären alle Änderungen hintereinander linear eingecheckt worden
  (empfohlen, da man so die Änderungshistorie besser nachvollziehen kann)
- **Merge-Commit** – die lokalen Änderungen landen in einem individuellen Commit



ACHTUNG: Lokal geänderte Dateien, welche noch nicht committed wurden, können hierbei überschrieben werden. Daher besser zuerst lokal committen (und notfalls später den Commit wieder rückgängig machen).

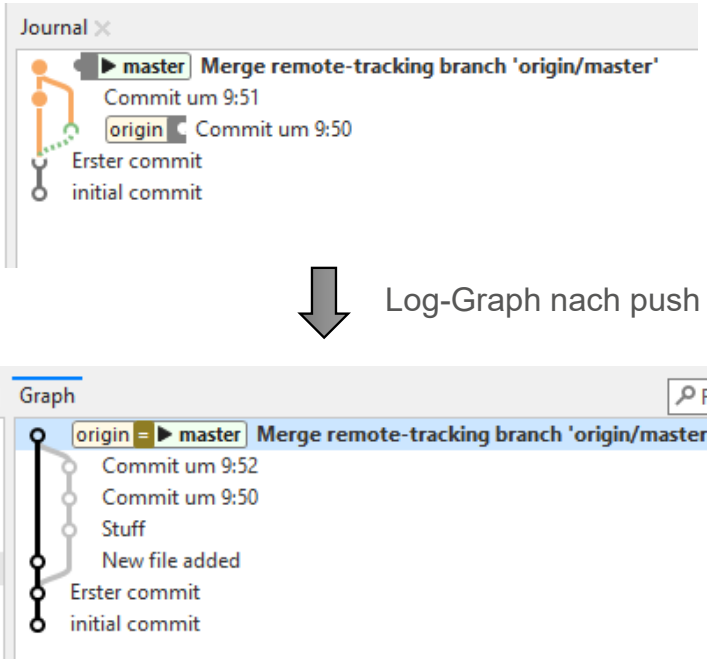
Alternativ die geänderten Dateien in den „Stash“ schieben (wird von SmartGit bei Pull automatisch vorgeschlagen).

Grundlagen – Entfernte Änderungen holen (fetch/pull)

UMWELTLEISTUNGEN

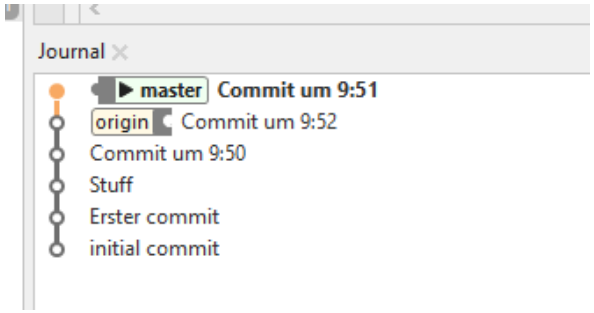
Pull – Entfernte Änderungen holen und lokal anwenden

Merge-Commit nach Pull



Rebase nach Pull

Die Reihenfolge ist nicht zwingend chronologisch, sondern nach dem Motto „first-come-first-served“.



Grundlagen – Entfernte Änderungen holen (fetch/pull)

UMWELTLEISTUNGEN

Pull – Entfernte Änderungen holen und lokal anwenden

Möglichkeiten je Datei, welche im Upstream geändert wurde:

- Datei existiert lokal, wurde aber seit dem letzten Abgleich lokal nicht verändert -> Änderungen vom externen Repo werden angewendet (Datei bekommt den Zustand vom origin)
- Datei existiert lokal nicht, wird hinzugefügt
- Datei wurde lokal gelöscht – Nutzer muss entscheiden, ob die Datei gelöscht bleiben soll, oder die Version aus dem Repo genommen werden soll
- Datei wurde upstream gelöscht, lokal nicht verändert -> Datei wird lokal gelöscht
- Datei wurde upstream gelöscht, lokal aber verändert -> Nutzer muss entscheiden, ob die Datei gelöscht werden soll, oder die lokale Version genommen werden soll
- Datei wurde lokal geändert, aber Änderungen widersprechen sich nicht -> Änderungen werden vereint
- Datei wurde lokal geändert und es gibt einen **Merge-Conflict** (Uh oh…)

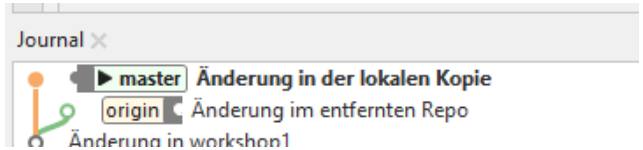
Grundlagen – Konflike Lösen

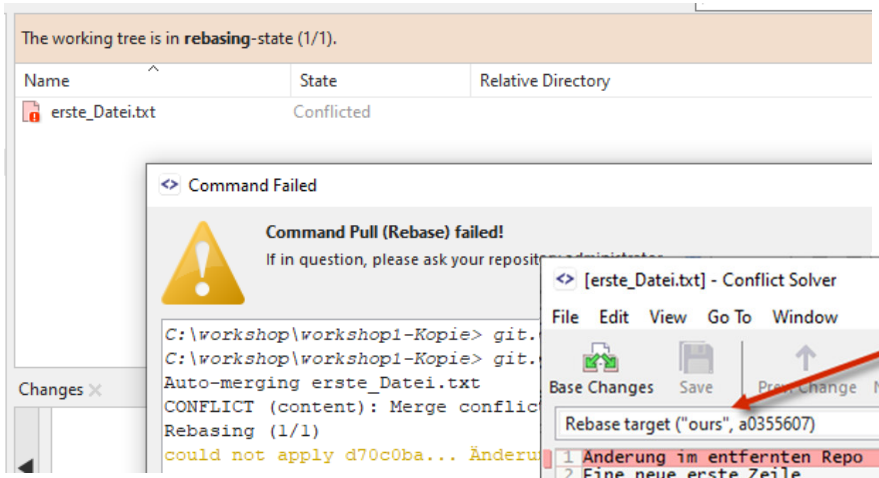
UMWELTLEISTUNGEN

Lokale Datei geändert und committed, entfernte Datei geändert

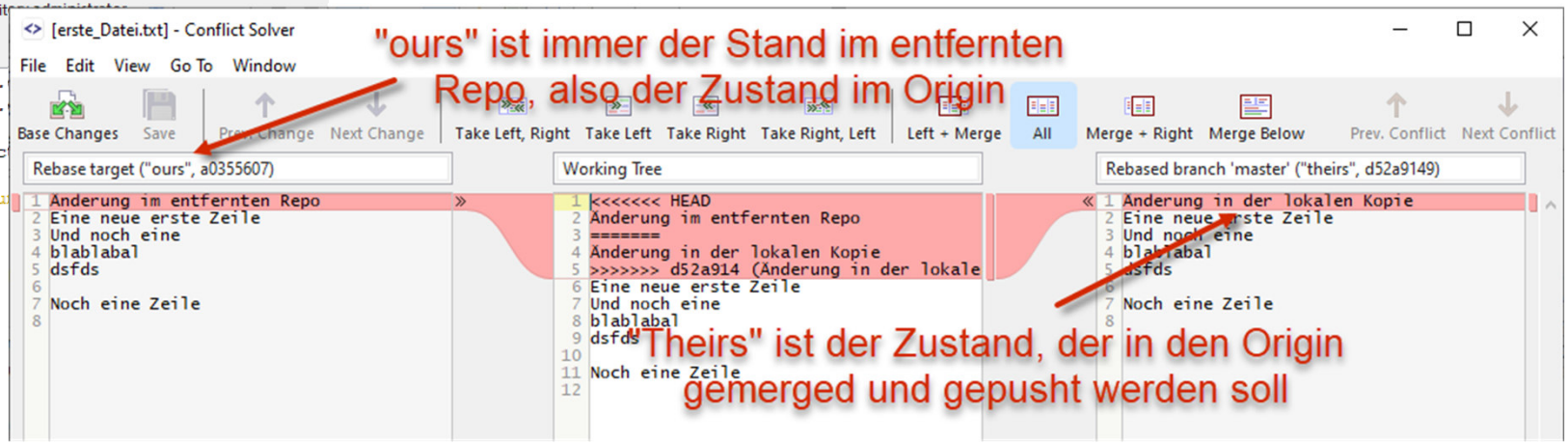
Eine Datei wurde gleichzeitig geändert, es gibt eine Conflict-Warnung und die lokale Arbeitskopie wird in den Zustand „Conflicted“ versetzt.

Jetzt MUSS MAN die Konflikte lösen.





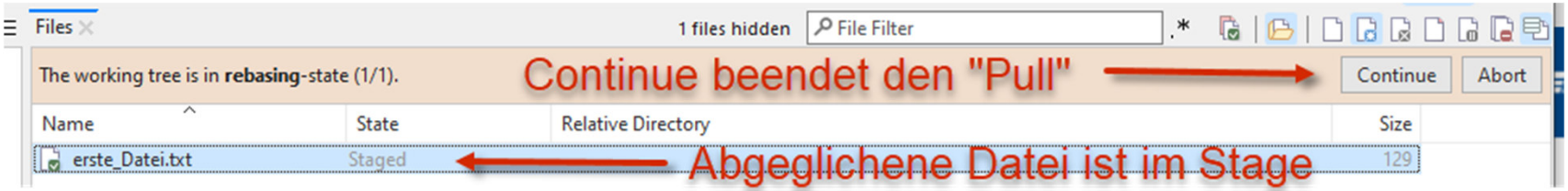
Doppelklick auf Datei mit Konflikt (oder „Conflict Solver“ aus dem Kontextmenu) öffnet den 3-Wege-Merge:

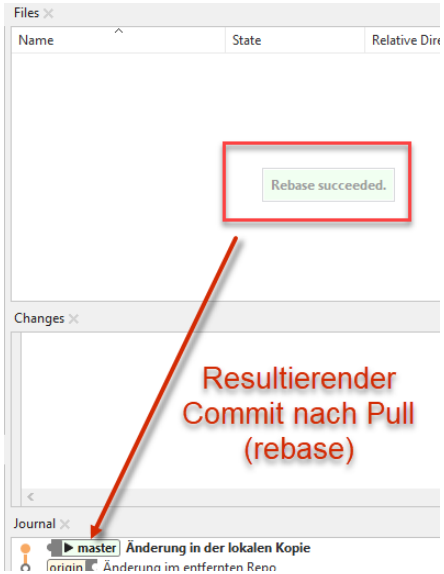


Grundlagen – Konflike Lösen

UMWELTLEISTUNGEN

Wenn alle Konflikte gelöst wurden, Repo-Zustand wieder normalisieren





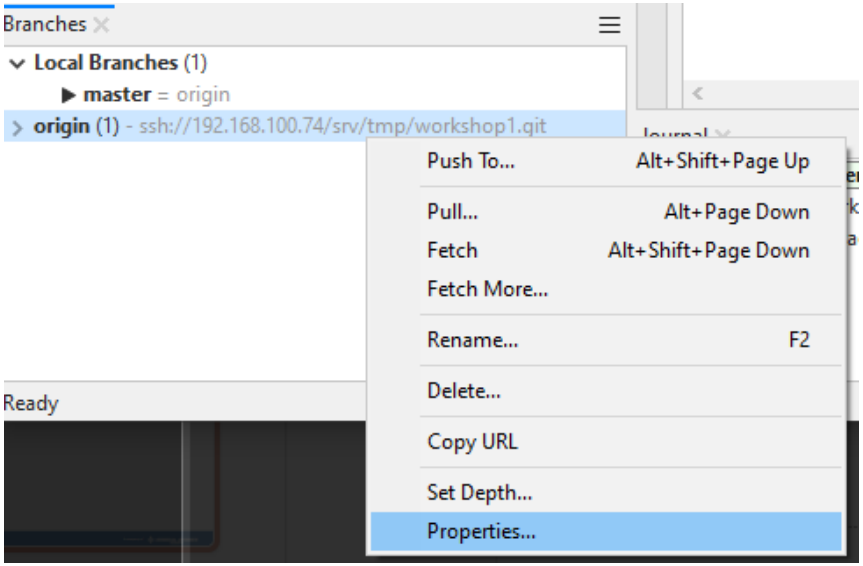
Commit enthält nun die verbliebenen Änderungen gegenüber dem Origin (entfernten Repo).

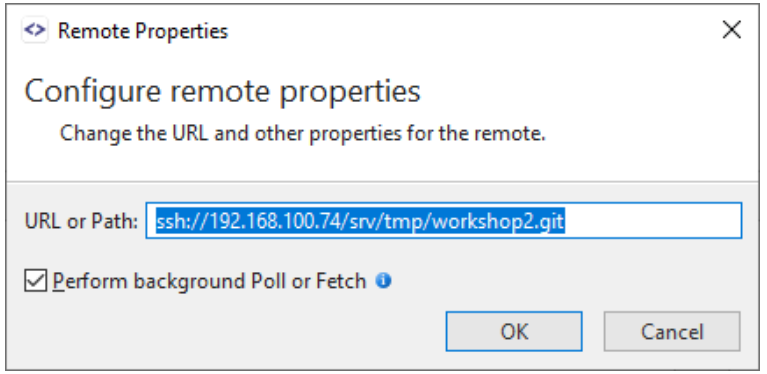
Workshop-Selber-Machen: Änderungen in der gleichen Datei einchecken und gegenseitig Konflikte erzeugen und lösen.

TIPP: Workflow ist immer „Pull“ vor „Push“.

Grundlagen – Origin ändern

Wenn das Server-Repo „umzieht“ (Verzeichnis umbenannte, Pfad verschoben, neuer Server) kann man in der lokalen Arbeitskopie einfach einen neuen Origin angeben.





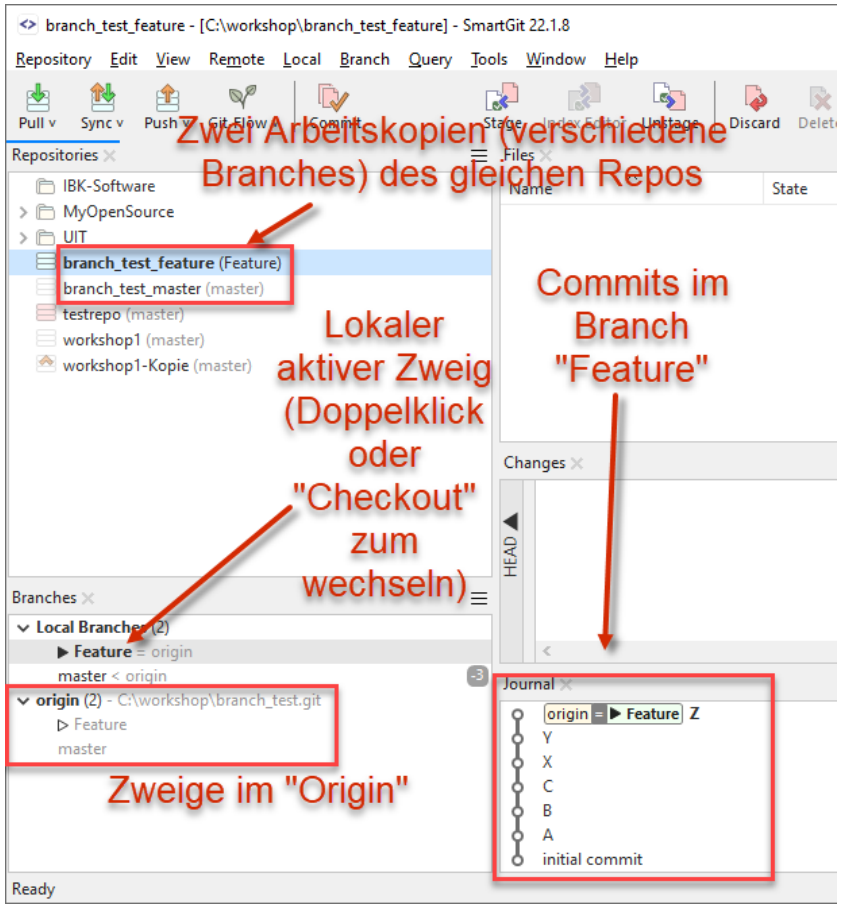
Teil 3 – Zweige (Branches/Tags)

Zweige/Branches

UMWELTLEISTUNGEN

Zweige/Branches erlauben parallele Änderungen in verschiedenen Entwicklungslinien

- Branches wie Perlenketten, die unterschiedliche Perlen (die Commits) auffädeln
- Branches können sich Commits teilen
- Man kann nur committed, wenn man einen Branch aktiv ausgewählt hat (sonst kann man den neuen Commit nicht „auffädeln“)
- Branch neu Erstellen im Menü: *Branch->Add Branch…*
- „Checkout“ wechselt den aktiven Branch
- Anwendungsfälle: Feature-Branches, Review-Branches, Release-Branches, Tags



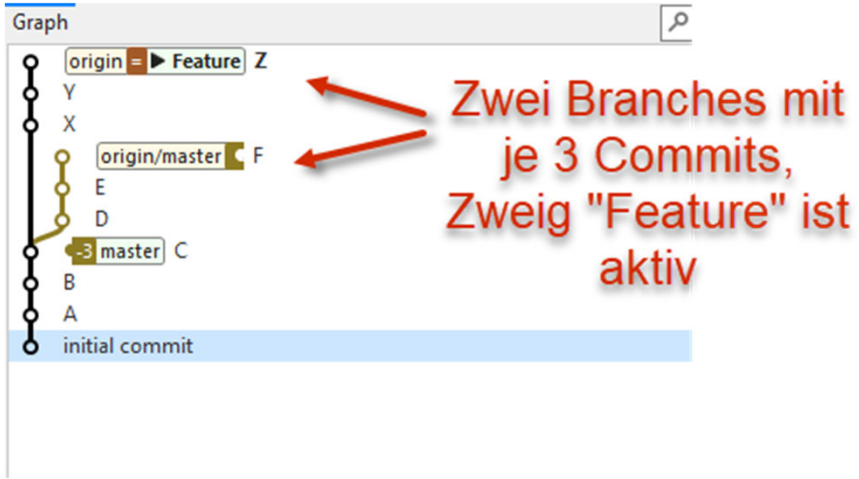
Zweige/Branches

UMWELTLEISTUNGEN

Arbeiten zusammenführen (Merge)

Grundsätzlich: man merged immer die Änderungen fremder Branches in den eigenen, aktiven Branch und erstellt damit neue Commits in der eigenen Perlenkette.

Beispiel-Ausgangssituation:
Ein Featurebranch „Feature“,
abstammend von „master“; in beiden Zweigen
jeweils 3 Commits



Merge-Optionen:

- Von „master“ zu „Feature“ (bzw. „integrate development from master“)
- „Feature“ zu „master“ („finish feature“ oder „integrate development from feature“)
- Mit Merge-Commit, Merge ins Arbeitsverzeichnis oder Squash-Merge

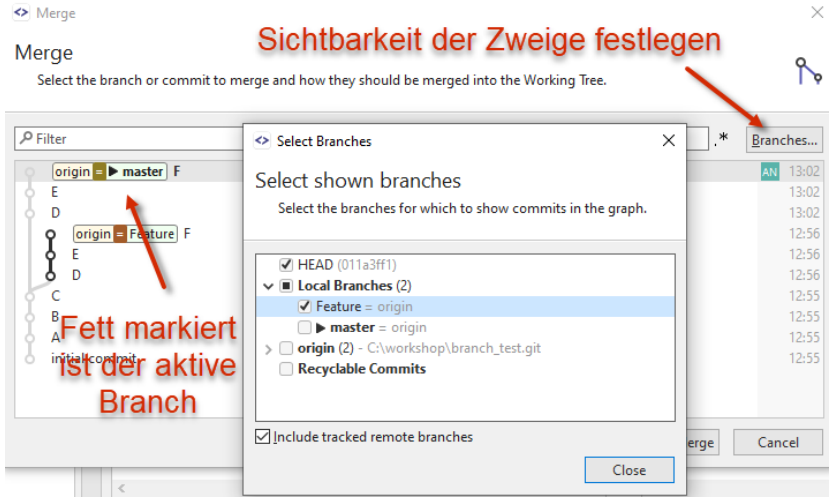
UMWELTLEISTUNGEN

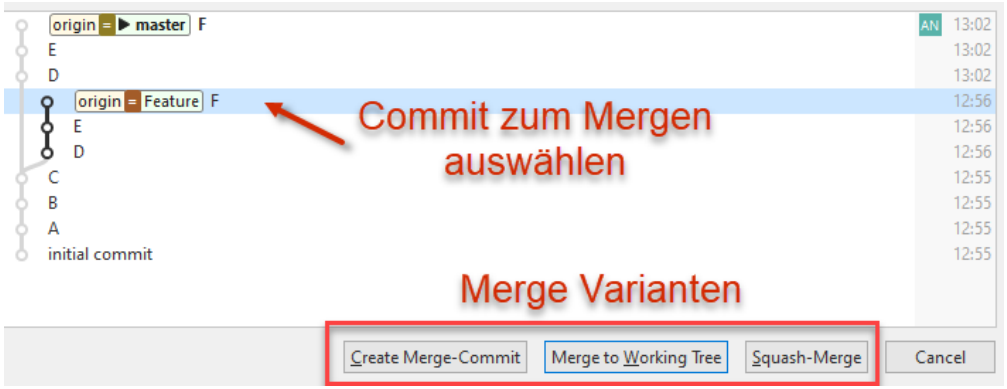
Arbeiten zusammenführen (Merge)

Ausgangsposition:

```
                ,D-E-F          < master
...-A-B-C
                `X-Y-Z          < feature
```

Merge „Feature“ in „master“: master als aktiven Branch einstellen, dann auf im Menü: *Branch->Merge…*





UMWELTLEISTUNGEN

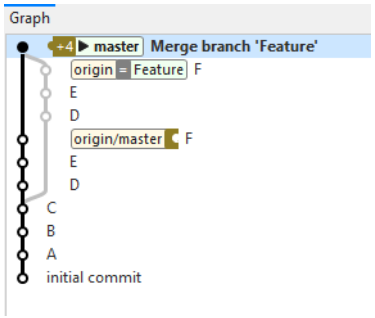
Arbeiten zusammenführen (Merge)

Vorher:

```
                ,D-E-F          < master
...-A-B-C
                `X-Y-Z          < feature
```

Merge-Commit:
Alle sich aus dem Merge ergebenden Änderungen werden automatisch in einem Commit dem aktiven Branch hinzugefügt.

```
              ,D-E-F-M        < master
...-A-B-C       /
              `X-Y-Z          < feature
```



Der so entstandene Merge-Commit ändert nicht wirklich was, er vermerkt nur die Zusammenführung der Zweige.

Wenn man diesen Merge-Commit rückgängig macht, ändert sich auch nichts im Dateisystem.

Zweige/Branches

UMWELTLEISTUNGEN

Arbeiten zusammenführen (Merge)

Vorher:

```
                ,D-E-F          < master
...-A-B-C
                `X-Y-Z          < feature
```

Merge to Working Tree:
Im Prinzip with „Merge-Commit“, nur dass die Arbeitskopie in den Zustand „Merge“ versetzt wird und erst mit dem abschließenden Commit wieder normal wird. Vor dem Commit kann man die geänderten Dateien noch im Dateisystem prüfen oder die Compilierung testen (obwohl man das besser schon vor dem Merge machen sollte).

Nach dem abschließenden Commit ergibt sich das gleiche Bild wie bei Option „Merge-Commit“.

```
              ,D-E-F-M        < master
...-A-B-C      /
              `X-Y-Z          < feature
```

Beim Erstellen des Commits hat man die Wahl zwischen „multiple Parents“ und „Squash“-Merge. Letzteres funktioniert anders, siehe nächste Folie.

UMWELTLEISTUNGEN

Arbeiten zusammenführen (Merge)

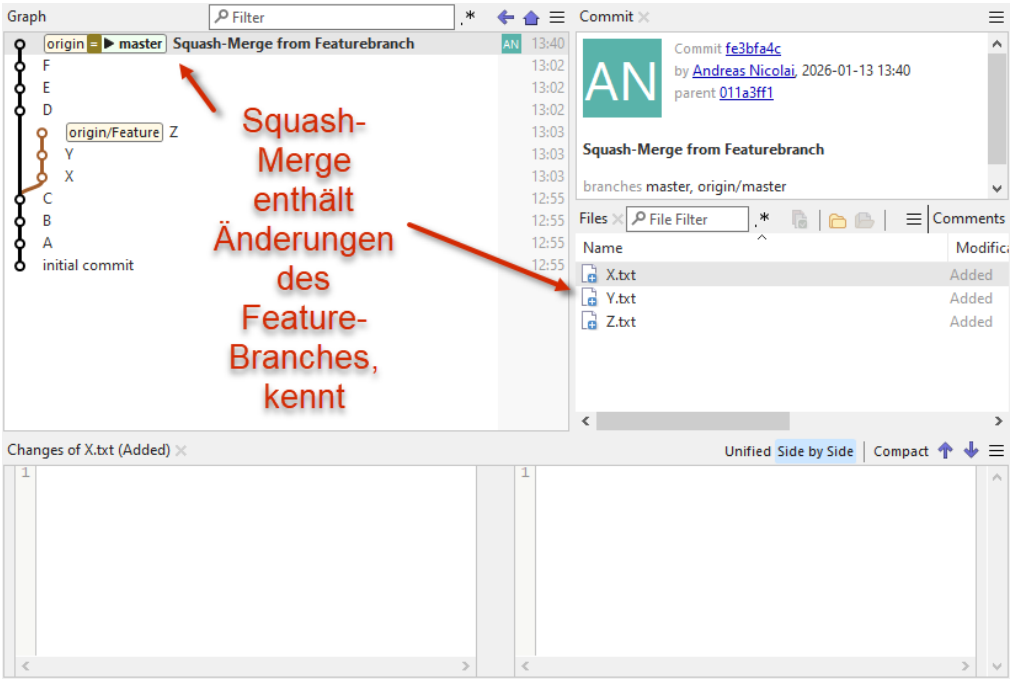
Vorher:

```
                ,D-E-F          < master
...-A-B-C
                `X-Y-Z          < feature
```

Squash-Merge:
Bei dieser Variante werden alle individuellen Änderungen im Featurebranch gegenüber dem aktiven Branch in lokal geänderte Dateien zusammengefasst. Wenn man diese Committed, ist das für das Repository so, als hätte man händisch die Änderungen selbst im aktiven Branch verfasst und eingecheckt. Es entsteht ein neuer Commit, der zwar inhaltlich den Änderungen im Featurebranch entspricht, aber keinen Bezug mehr dazu hat.

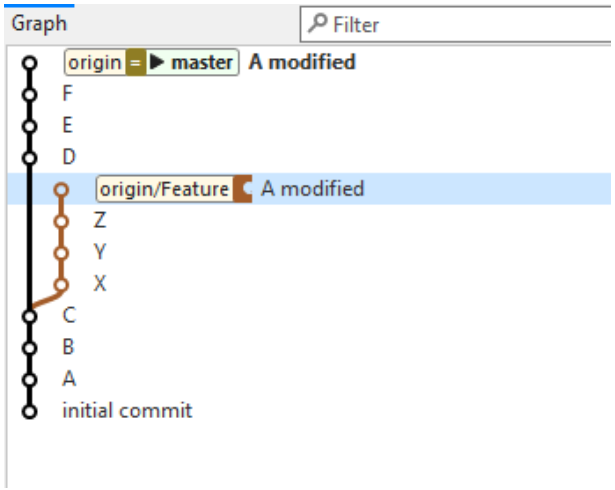
```
                ,D-E-F-S        < master
...-A-B-C
                `X-Y-Z          < feature
```

Wird der Featurebranch gelöscht, sind alle individuellen Commits dort weg.

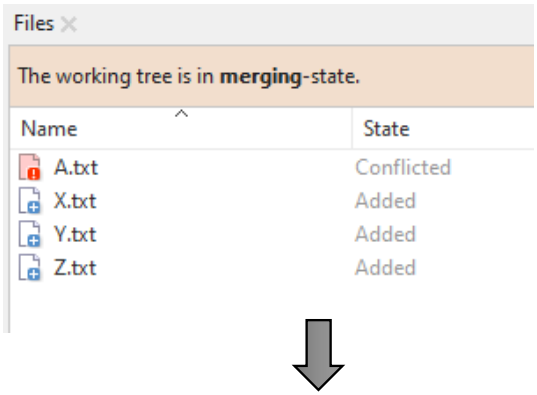


Merge-Konflikte Auflösen

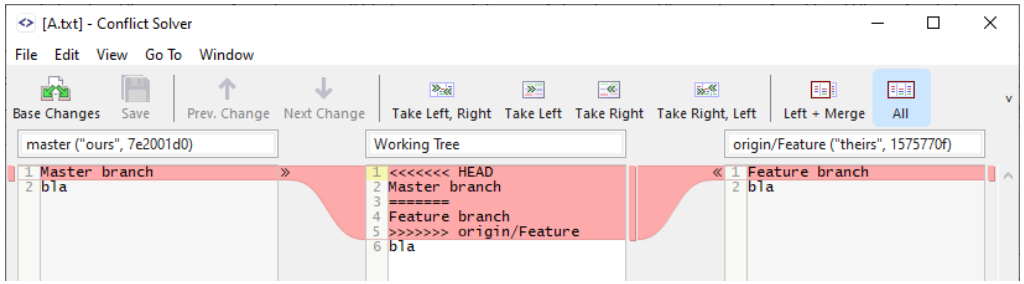
Vorher:



Merge Feature in master:



Auch hier „**ours**“ ist der aktive Branch in den gemergt werden soll, „**theirs**“ ist der Quellbranch, wo die Änderungen her kommen.

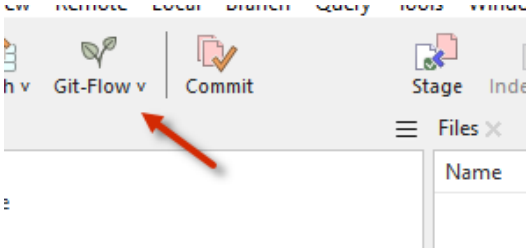


SmartGit – Git-Flow

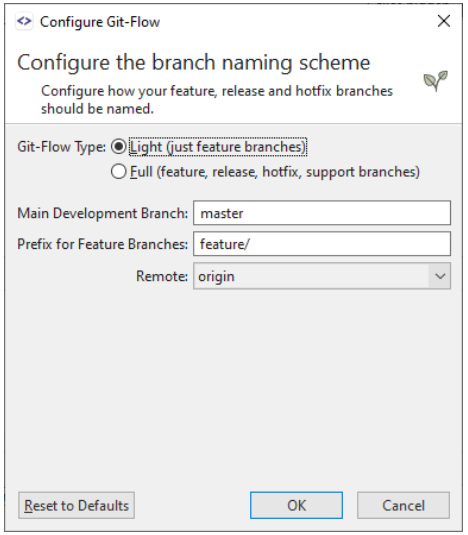
Typische Workflows/Aufgaben:

1. Feature Branch erstellen (möglichst mit einheitlichem Namensschema)
2. Änderung vom master regelmäßig in den Featurebranch übernehmen („am Ball bleiben“)
3. Feature beenden und in den master mergen (vor diesem Schritt sollte man zwingend den aktuellen master-Arbeitsstand übernehmen, siehe Punkt 2)

All das wird durch die Git-Flow Funktion von SmartGit vereinfacht:



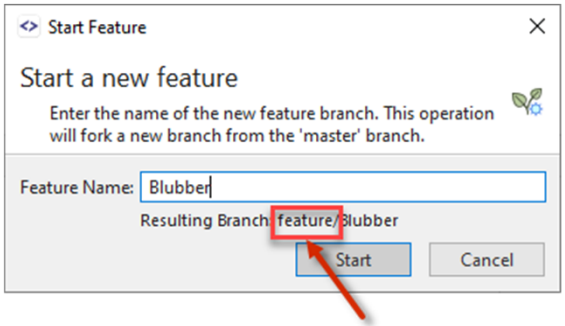
Standardkonfiguration „Light“ reicht aus



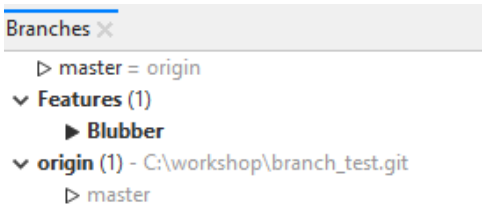
SmartGit – Git-Flow

UMWELTLEISTUNGEN

1. Feature Branch erstellen (möglichst mit einheitlichem Namensschema)
   Einfacher Klick auf „Git-Flow“ oder Auswahl „Start feature…“

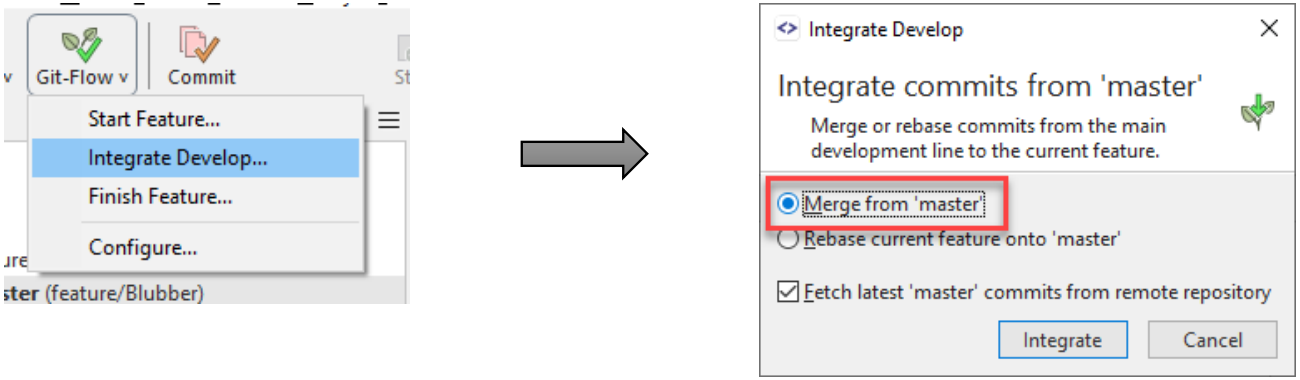


Featurebranches erhalten
Präfix feature/
(SmartGit erkennt daran
Git-Flow Features)



SmartGit – Git-Flow

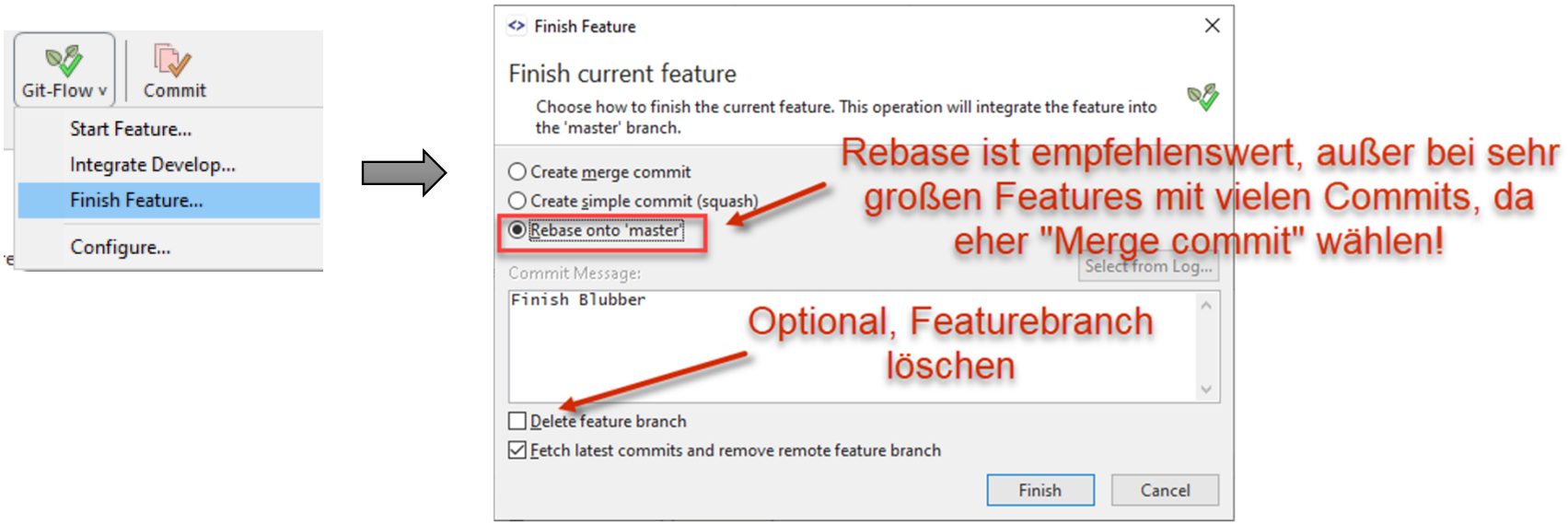
2. Änderung vom master regelmäßig in den Featurebranch übernehmen („am Ball bleiben“)



WICHTIG: während der Entwicklung im Featurebranch regelmäßig „Merge from master“ machen, damit beide Zweige nicht zu weit auseinanderlaufen. Spätestens kurz vor dem Merge zurück in den Master (also bei „Finish Feature“), muss man sowieso erst den master in den Featurebranch integrieren.

SmartGit – Git-Flow

3. Feature beenden und zurück in den master mergen (Klick auf Git-Flow oder Auswahl aus dem Menü):



WICHTIG: Wenn man vor dem „Finish Feature“ den master-Branch integriert hat und dabei alle Merge-Konflikte aufgelöst hat, sollte ein sauberer Finish-Feature-Merge-Commit entstehen.

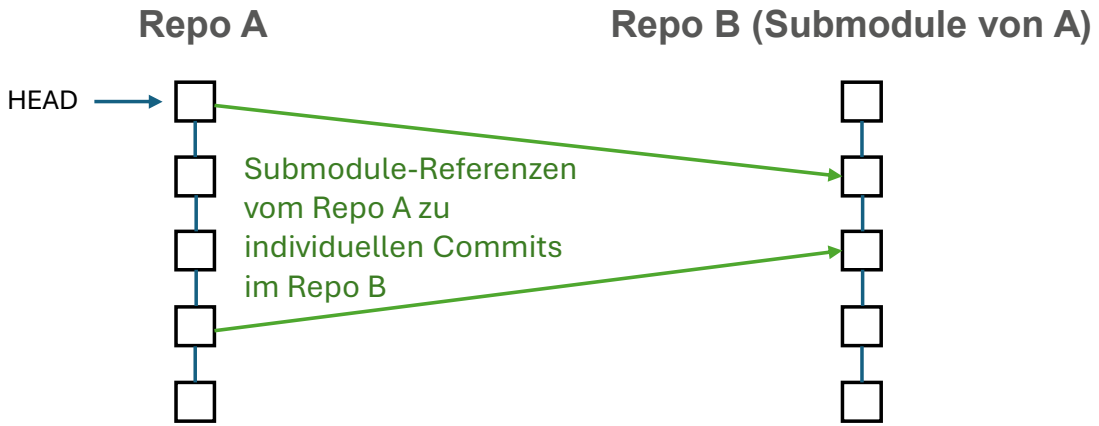
Teil 4 – submodules

Submodules

Ein (Toplevel-) Repository kann andere Repositories, sogenannte ***Submodules*** einbinden.

Grundidee:

- Ein Commit im Toplevel-Repository referenziert eindeutig einen globalen Arbeitsstand, einschließlich der Arbeitsstände in referenzierten Submodules
- Submodule-Repos können weiterentwickelt werden, ohne die Arbeit im Toplevel-Repository zu stören

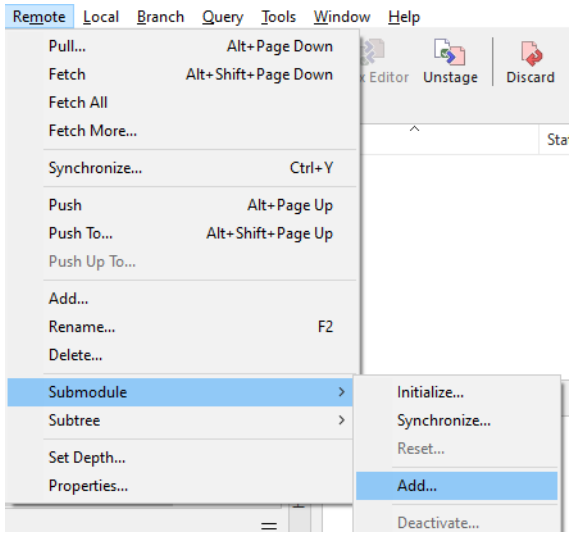


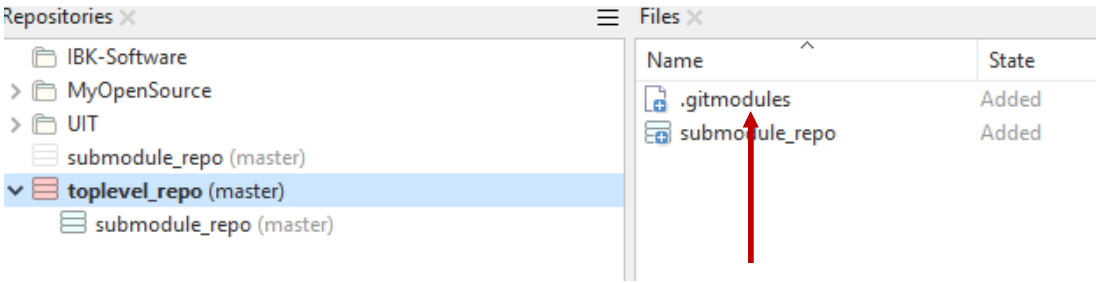
Submodule-Referenzen sind quasi Dateien mit Verweisen auf einzelne Commits in Submodules, und werden genau so als Änderungen in Commits verändert. Ohne Änderungen bleiben die Verweise auf die Commits im Submodule unverändert, egal was da im Submodule-Repo passiert.

Submodules

Submodule anlegen

In SmartGit:





Datei enthält Info über referenzierte Submodules:

```
[submodule "submodule_repo"]
          path = submodule_repo
          url = ../submodule_repo.git
```

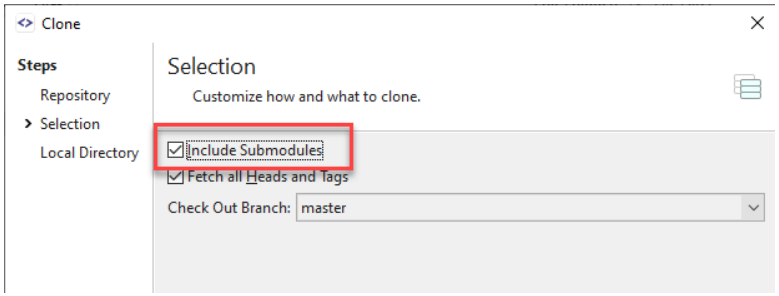
ACHTUNG: Bei der Angabe des Pfades die Angabe von Nutzernamen vermeiden. Relative URLs verwenden, wenn Submodule-Repos auf dem gleichen Server liegen.

Submodules

UMWELTLEISTUNGEN

Repo mit Submodules clonen

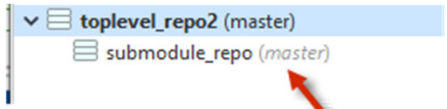
In SmartGit beim clonen:



In der Kommandozeile:

➢ `git clone <repo name> --recurse-submodules`

Anzeige in SmartGit nach dem clonen:



Kursiver Text - kein aktiver Branch (Commits führen zu Fehlermeldung "Detached Head")

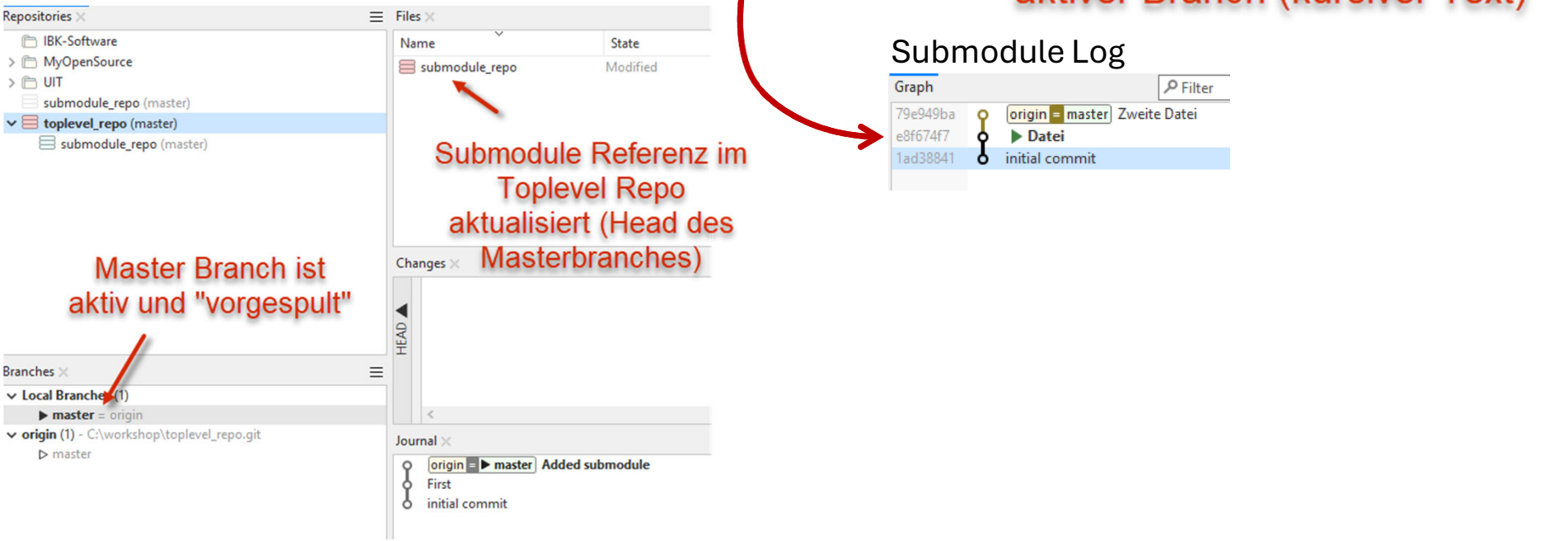
ACHTUNG: Bevor man Änderungen in ein referenziertes Submodule einchecken kann, muss man auch dort ein Branch auswählen (Doppelklick/Checkout). Ohne einen solchen ausgewählten Branch bekommt man die Fehlermeldung, dass ein Commit an einen „Detached Head“ nicht möglich ist.

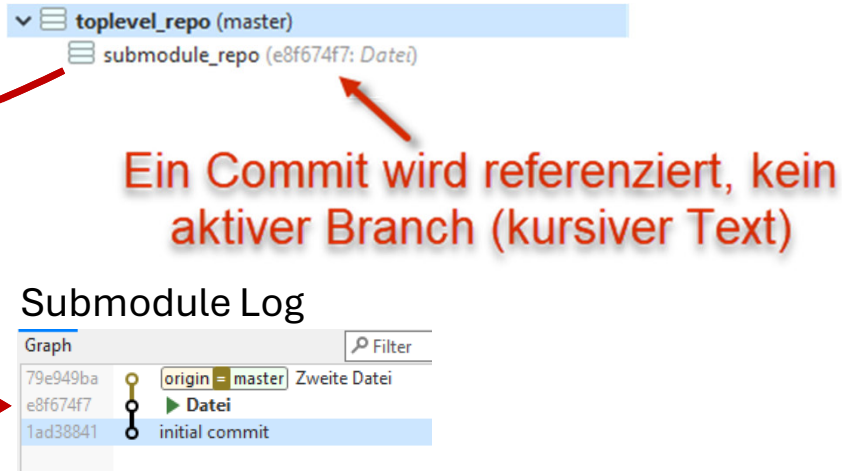
Submodules

Submodules aktualisieren/ändern:

Ausgangsposition: frisch geklontes Repo mit Submodule, wobei ein älterer Commit im Submodule referenziert wird

Checkout des „master“-Branch im Submodul:

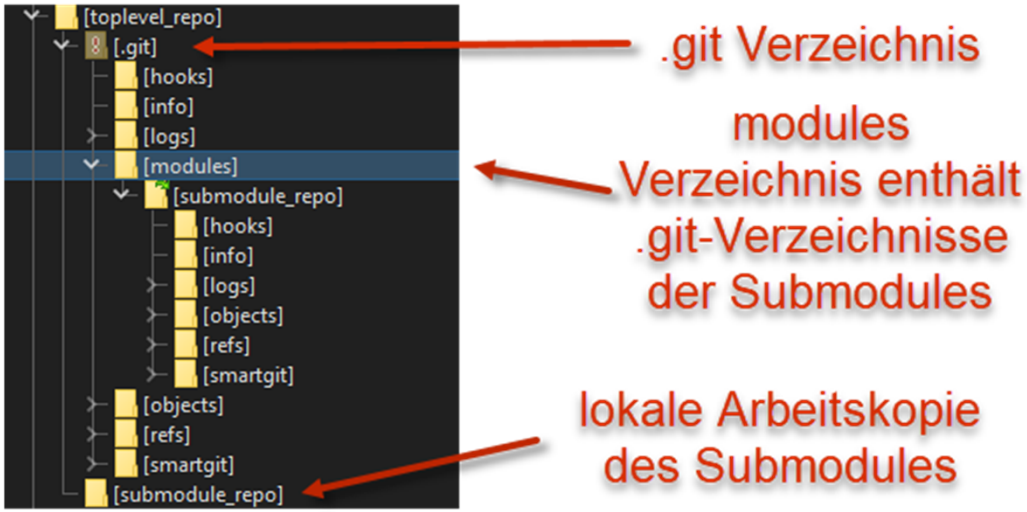




Submodules

Submodules reparieren:

Verzeichnisstruktur einer Arbeitskopie mit Submodule:



Bei Problemen das .git-Verzeichnis (benannt wie das Submodule selbst) aus dem .git/modules-Verzeichnis löschen und die lokale Arbeitskopie löschen.

Dann in SmartGit: „missing submodule“ mit „Discard“ zurücksetzen oder via Menü: *Remote->Submodule->Initialize…*

Teil 5 – Teamarbeit

Best-Practice Tipps für die Teamarbeit

UMWELTLEISTUNGEN

Allgemeines:

- Commits zu einzelnen Themen erstellen (Dateien entsprechend stagen und committed), keine Monstercommits mit zig Bugfixes und Features gleichzeitig
- Aussagekräftige Commit-Kommentare benutzen
- Niemals generierte (Binär-)Dateien ins Repo committen, .gitignore-Dateien verwenden!
- Jeder Commit im „master“ sollte lauffähigen Code erzeugen (um die Arbeit anderer nicht zu behindern). Im eigenen Featurebranch darf man unfertige Zwischenstände Committen

Featurebranches:

- regelmäßig den master-Branch in den Featurebranch integrieren, um ein Divergieren der Zweige zu vermeiden
- kleine Featurebranches (ein Feature = ein Branch)
- Können nur lokal gehalten werden, sofern andere nicht gleichzeitig daran arbeiten. Gelegentlich ins Repo pushen dient aber auch als Backup.