



Tag 1: Einführung in Git und GitLab, Git-Workflow im Team

17.06.2024, Daniel Krämer & Malte Fischer

© Copyright 2024 anderScore GmbH

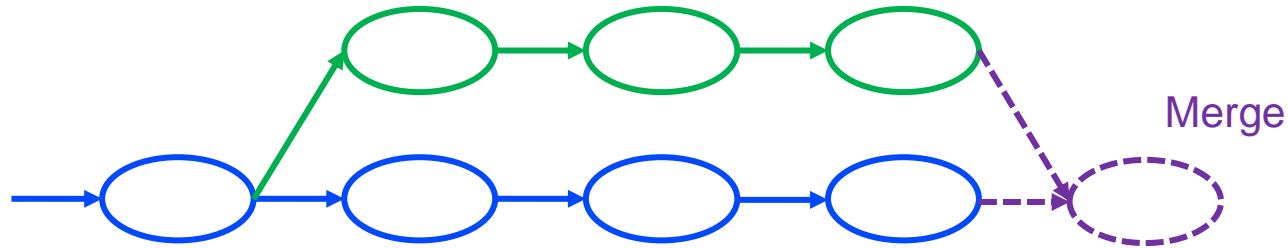
- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

Grundlagen und Konzepte von **Merging in Git**

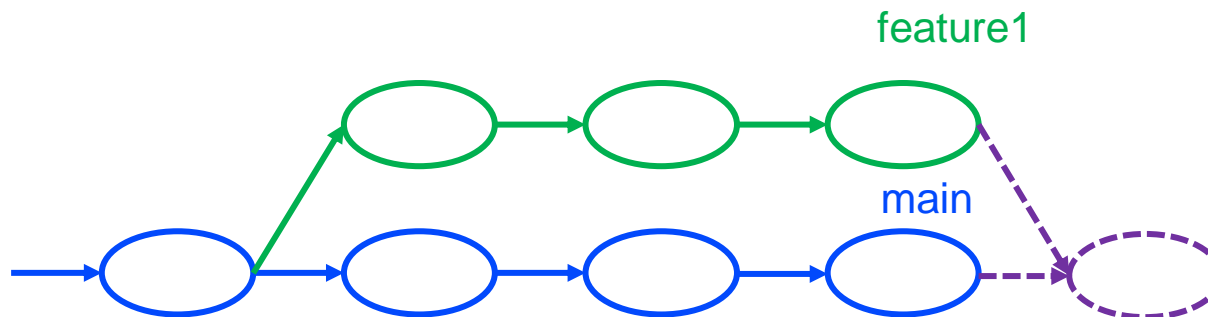
Inhalt

- Merging
 - Konzept
 - Ablauf
 - Merge-Konflikte
 - Befehle
 - Strategien
- Rebase
 - Konzept
 - Befehle und Optionen
 - Konflikte während Rebase
 - Anwendung
- Merge vs Rebase



- Merging bildet das Gegenstück zum Branching
- Ermöglicht Zusammenführung von verschiedenen Branches zu einem einheitlichen Verlauf
- Beispielhafte Use Cases
 - Entwickler hat Hotfix abgeschlossen und möchte Änderungen in Code integrieren
 - Feature ist abgeschlossen und so zum aktuellen Stand hinzugefügt werden
 - Änderungen von Remote Branches einpflegen (mehr dazu später)

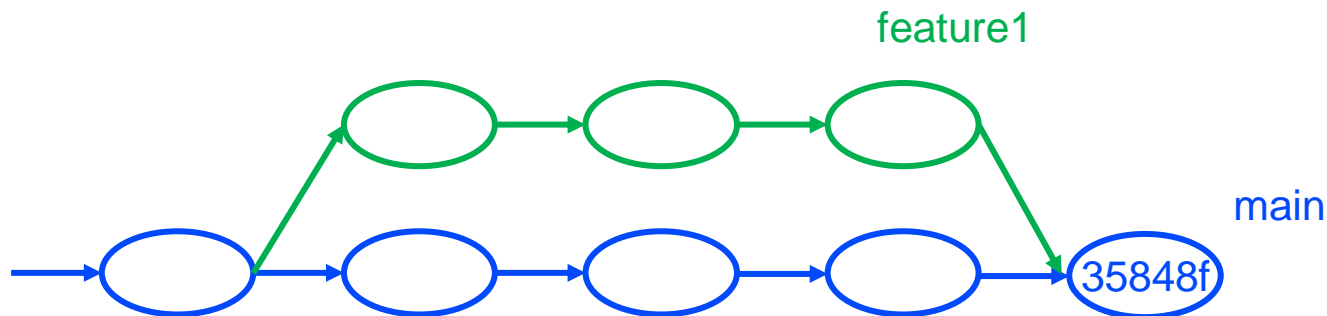
- Einen Branch *branch* in den aktiven Branch mergen mittels
`git merge <branch>`
- Merging verändert immer nur den aktiven Branch
- Erzeugt auf dem aktiven Branch einen Commit, um die Änderungen zu integrieren
- Merge-Commit hat im Gegensatz zu anderen bisher betrachteten Commits zwei Vorfahren
- Gemergeter Branch wird nicht automatisch gelöscht, aber oft als abgeschlossen betrachtet



Beispiel

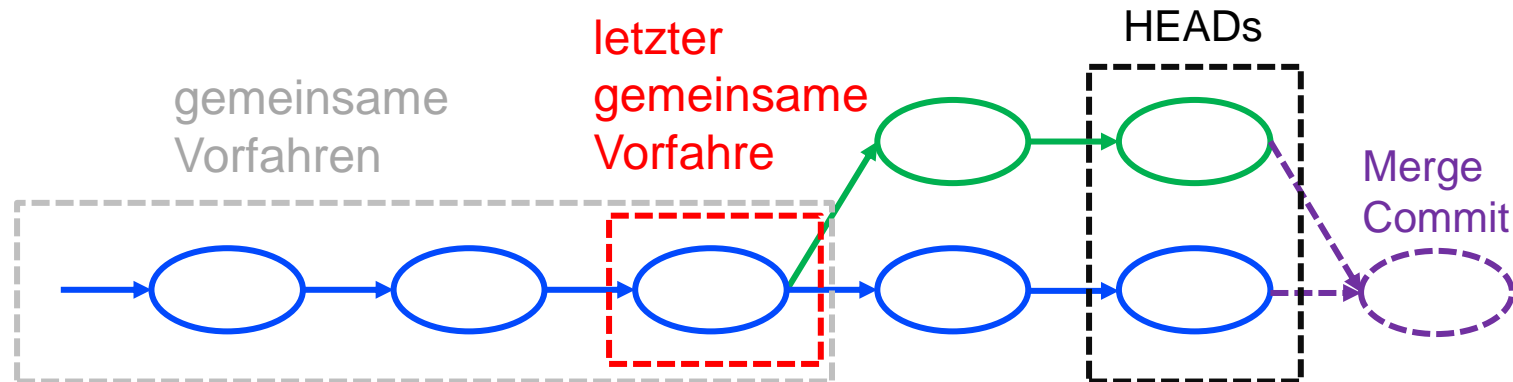
```
$ git merge feature1
Merge made by the 'ort' strategy.
 feature_file1.txt | 0
 feature_file2.txt | 0
 feature_file3.txt | 0
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 feature_file1.txt
create mode 100644 feature_file2.txt
create mode 100644 feature_file3.txt
```

```
$ git log --oneline
358548f (HEAD -> main) Merge branch 'feature1'
...
```



Ablauf eines Merges

- Merging von Dateien mit identischem Inhalt ist am einfachsten
 - Datei auf beiden Branches unverändert
 - Datei auf beiden Branches exakt gleich verändert
- Unterscheiden sich die Inhalte, benötigt man einen anderen Ansatz
- Git nutzt hierzu den Three-Way-Merge Algorithmus
 - Suchen nach dem letzten gemeinsamen Vorfahren
 - Betrachtet für jede modifizierte Datei, ausgehend vom letzten gemeinsamen Vorfahren, die Änderungen
 - Es existiert immer mindestens ein gemeinsamer Vorfahre (Zumindest initialer Commit)



- Mithilfe des letzten gemeinsamen Vorfahren werden dann auf Dateiebene die Änderungen zusammengefügt
- Dateien werden dafür in Sektionen (üblicherweise einzelne Zeilen) aufgeteilt, die dann ausgehend vom l.g.V. verglichen werden
 - Zeile ist auf beiden Branches unverändert → Zeile ins Ergebnis übernehmen
 - Zeile ist auf beiden Branches gleich verändert → Veränderung übernehmen
 - Zeile ist auf einem beiden Branches geändert, auf dem anderen nicht → Veränderung übernehmen
 - Zeile ist in beiden Branches unterschiedlich verändert → Mergekonflikt, kein automatisches Auflösen möglich

- Änderungen inkludiert auch das Hinzufügen einer Zeile bzw. das Löschen einer Zeile
- Git versucht beim zeilenbasierten Mergen zusammenhängende Bereiche zu erkennen
 - Bspw. beim Einfügen einer Zeile wird nur diese als Änderung erkannt, der restliche Dateiinhalt gilt als unverändert, auch nachfolgende Zeilen ggf. verschoben sind

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(firstName, person.firstName)
        && Objects.equals(lastName, person.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

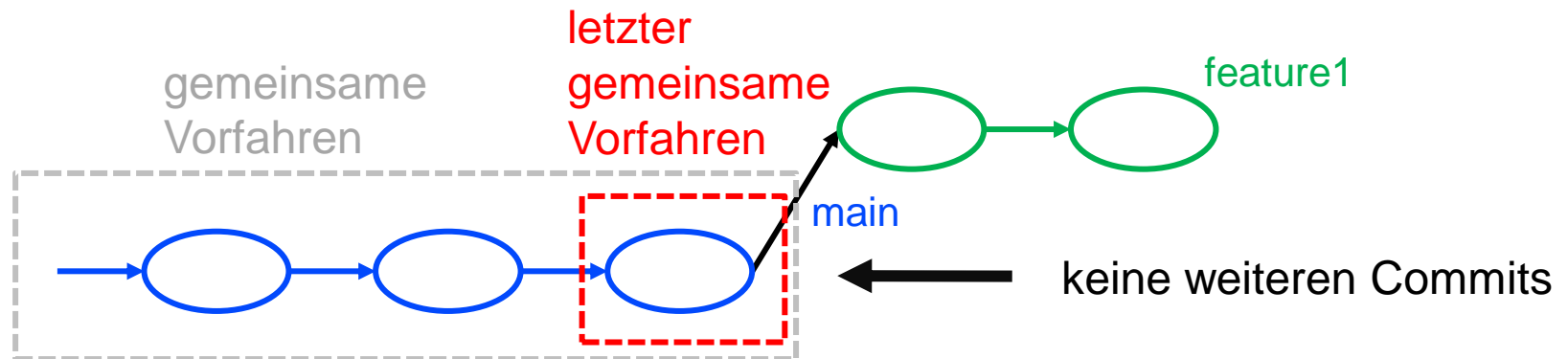
```
35 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
```

```
@Override
public boolean equals(Object o) {
    //first comment
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    //another comment
    return Objects.equals(firstName, person.firstName)
        && Objects.equals(lastName, person.lastName);
}

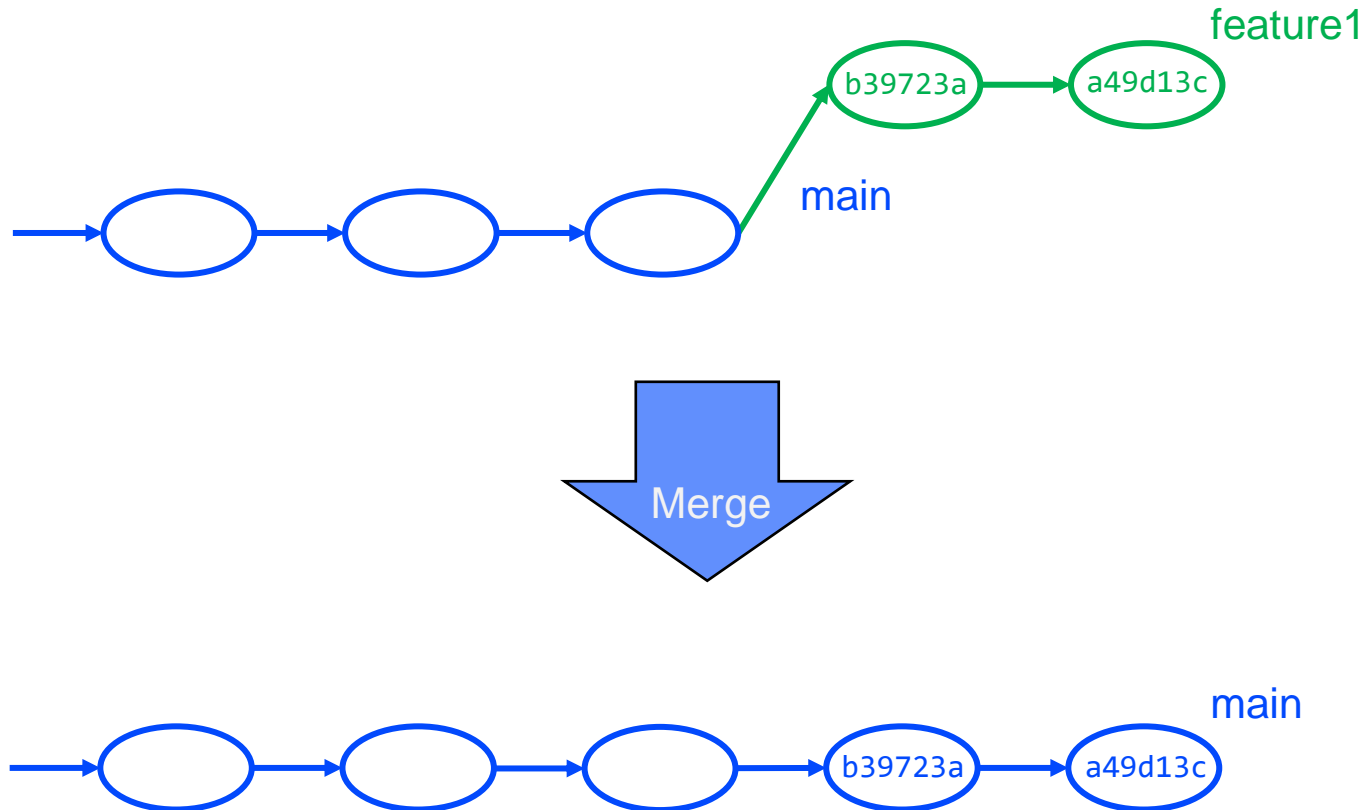
@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

Fast-Forward Merge

- Möglich, wenn zwei Kriterien erfüllt werden
 - Beide Branches haben einen eindeutigen letzten gemeinsamen Vorfahren
 - Bei einem der Branches wurde seit dem letzten gemeinsamen Vorfahren kein Commit hinzugefügt



- Bei einem Fast-Foward Merge entsteht kein Merge-Commit
 - Kann mittels `--no-ff` trotzdem erstellt werden
- Commit des weiter fortgeschrittenen Branches werden als Commits des Zielbranches weiterbehandelt



Aufgabe 8: Fast-Forward Merging

Feature 1 und 2 sind so weit abgeschlossen und die Änderungen sollen in den **main** Branch integriert werden.

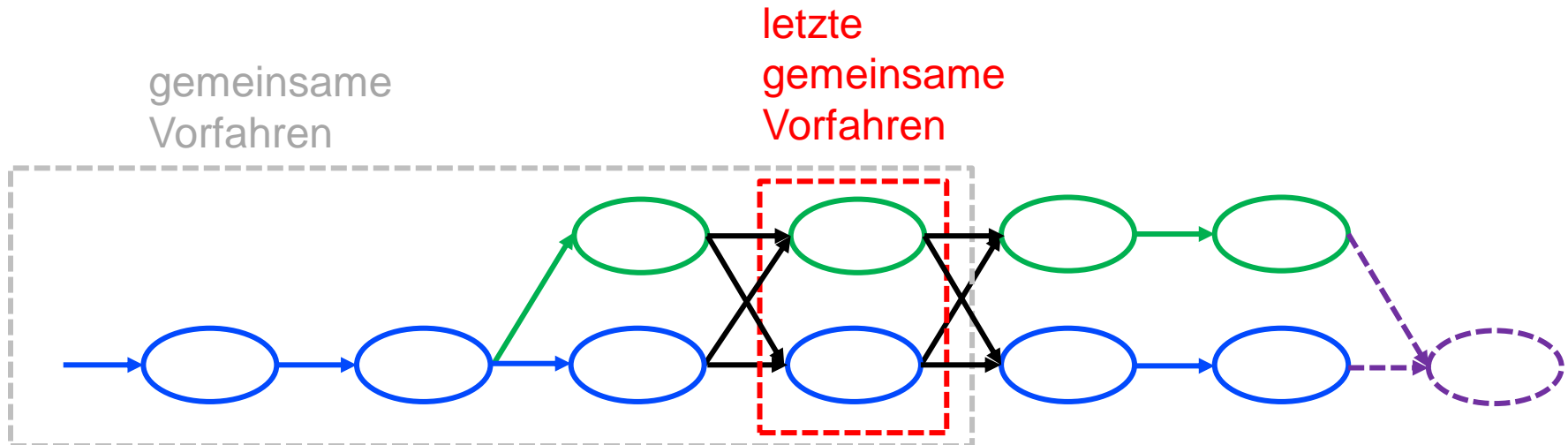
1. Stellen Sie zunächst sicher, dass Sie sich auf dem **main** Branch befinden. Git folgt bis auf wenige Ausnahmen immer dem Prinzip, dass nur der aktive Branch verändert wird.
2. Mergen Sie den **feature1** Branch zurück in den **main** Branch. Da nach der Abzweigung von **feature1** (Merge-Base) auf **main** keine neuen Commits erzeugt wurden, sollte Git den Merge als Fast-Forward Merge umsetzen können. Rückwirkend sieht es von den Logs so aus, als wären die **feature1** Commits direkt auf **main** getätigt worden.

Aufgabe 9: Merging

1. Mergen nun auch **feature2** zurück in den **main** Branch.
Nach der Merge-Base von **feature2** und **main** folgen auf **main** noch weitere Commits, daher kann hier kein Fast-Forward Merge durchgeführt werden.
2. Verifizieren Sie über die Logs, dass Git einen Merge-Commit erstellt hat, um **feature2** in **main** zu mergen.
3. Da **feature1** und **2** erfolgreich gemerged wurden, gelten die Branches als abgeschlossen und werden nicht mehr benötigt.
Löschen Sie **feature1** und **feature2**.

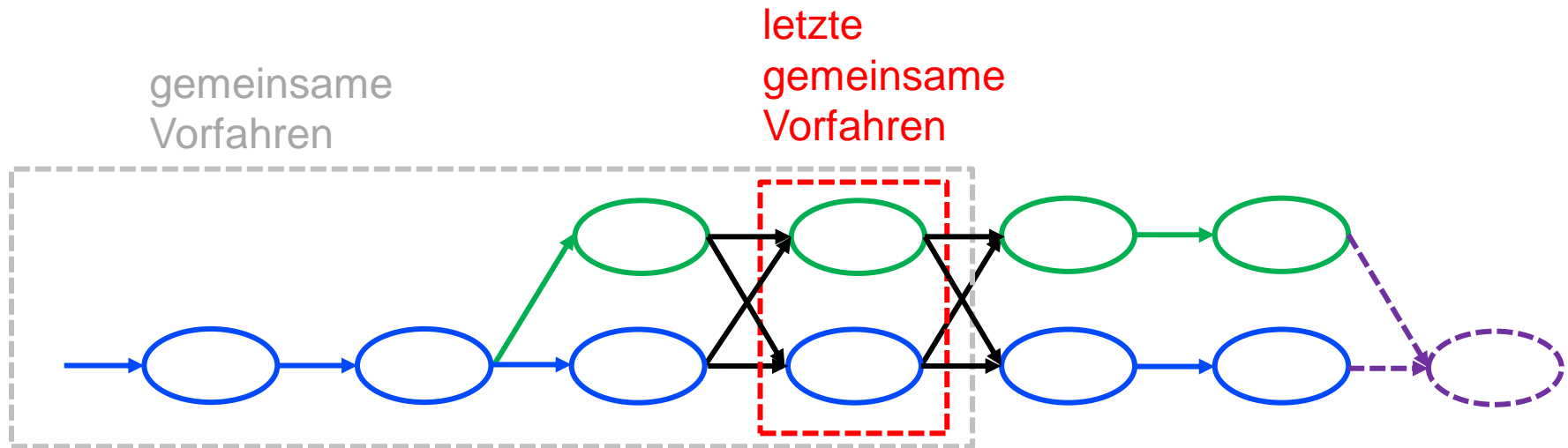
Mehrere gemeinsame Vorfahren

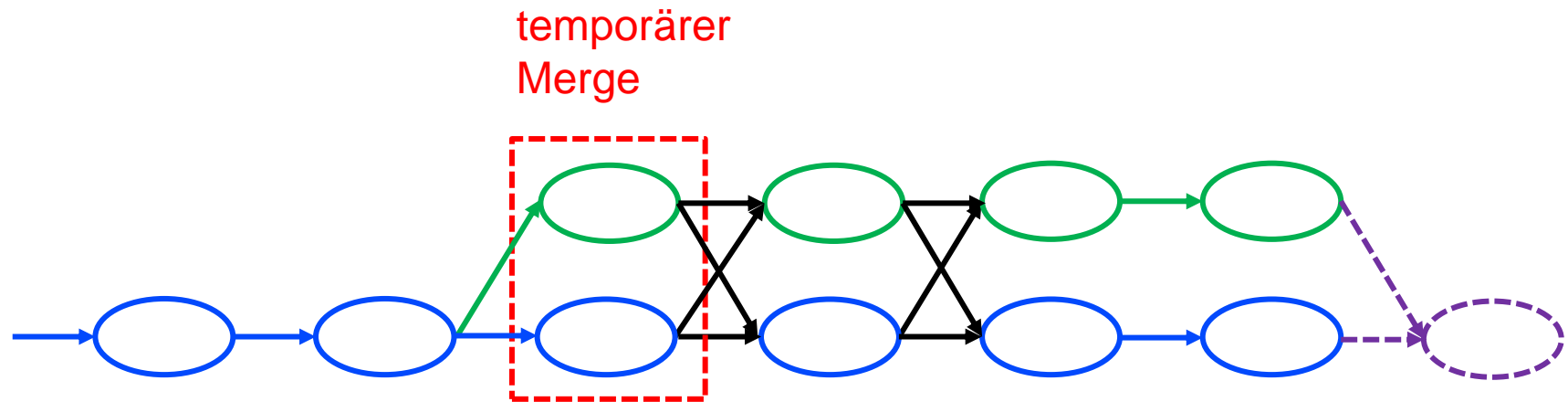
- In Sonderfällen kann es passieren, dass es bei einem Merge mehrere gemeinsame letzte Vorfahren gibt
- Ein Beispiel dafür ist ein sogenannter **Criss-Cross-Merge**

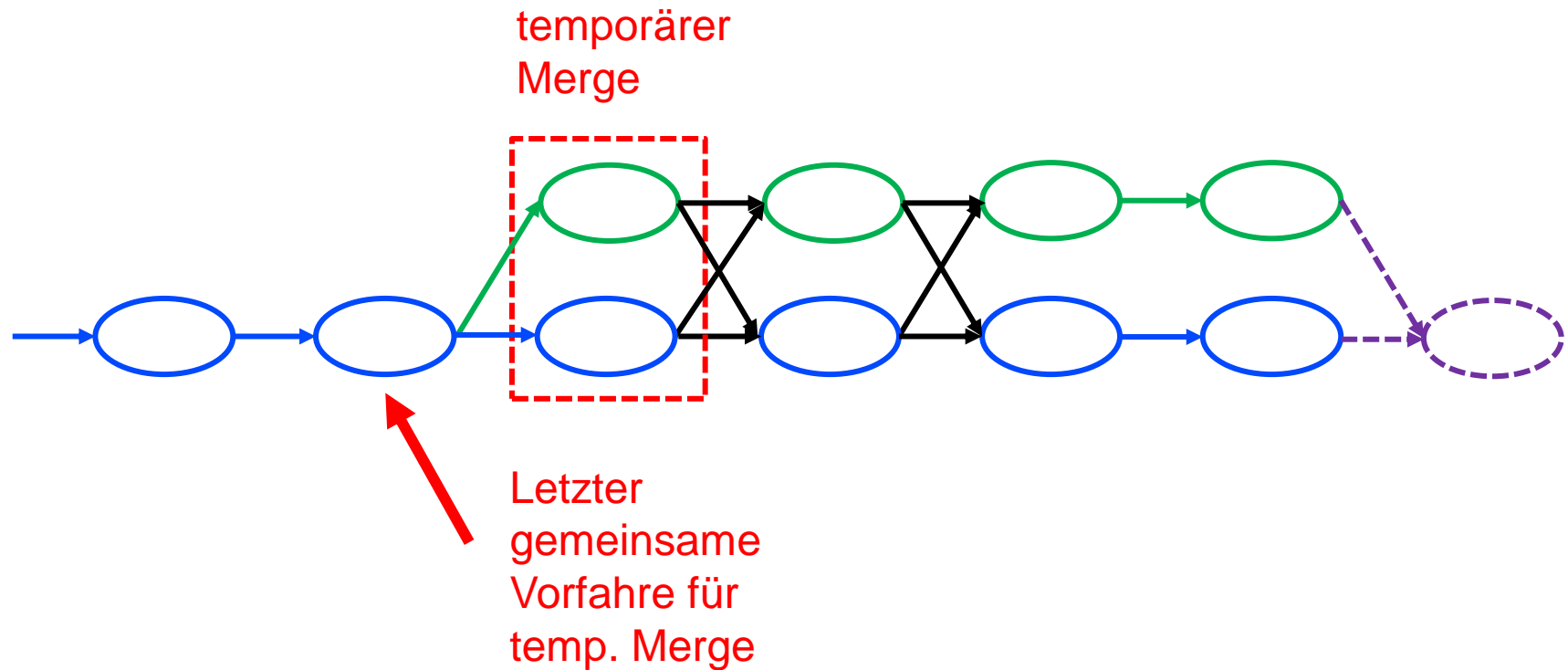


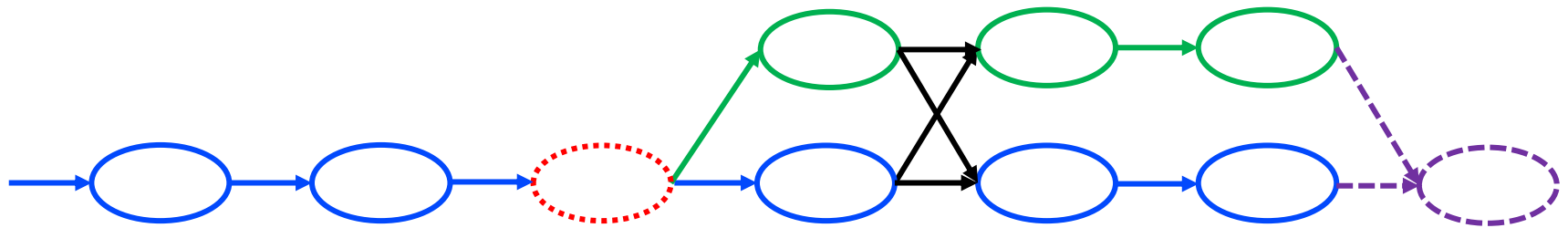
- Git kann hier den rekursiven 3-Way-Merge nutzen
- Über die Vorfahren werden temporäre Merges gebildet, bis ein eindeutiger gemeinsamer letzter Vorfahre existiert

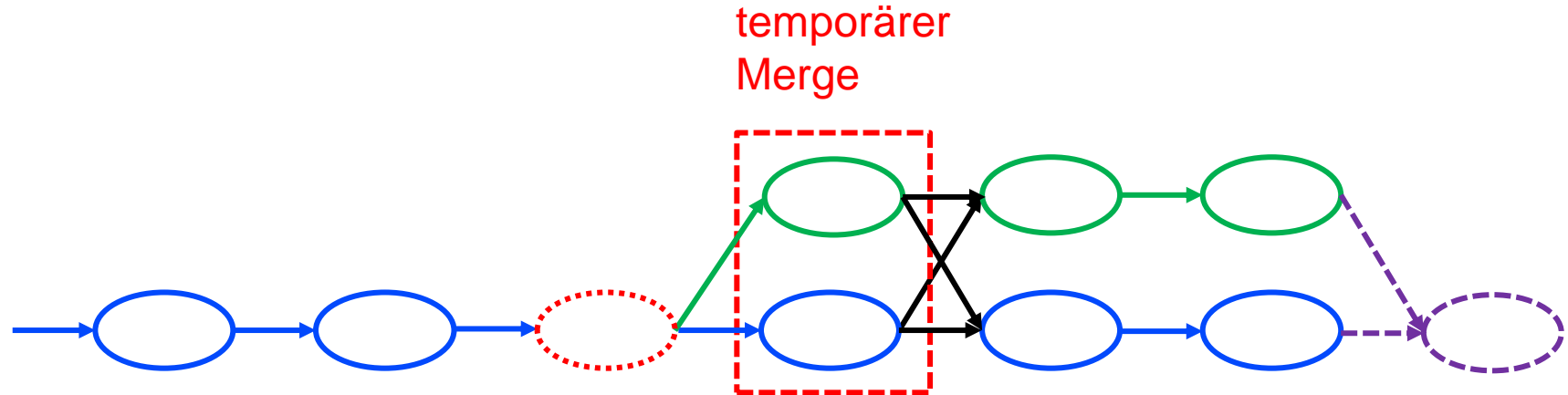
Bispiel

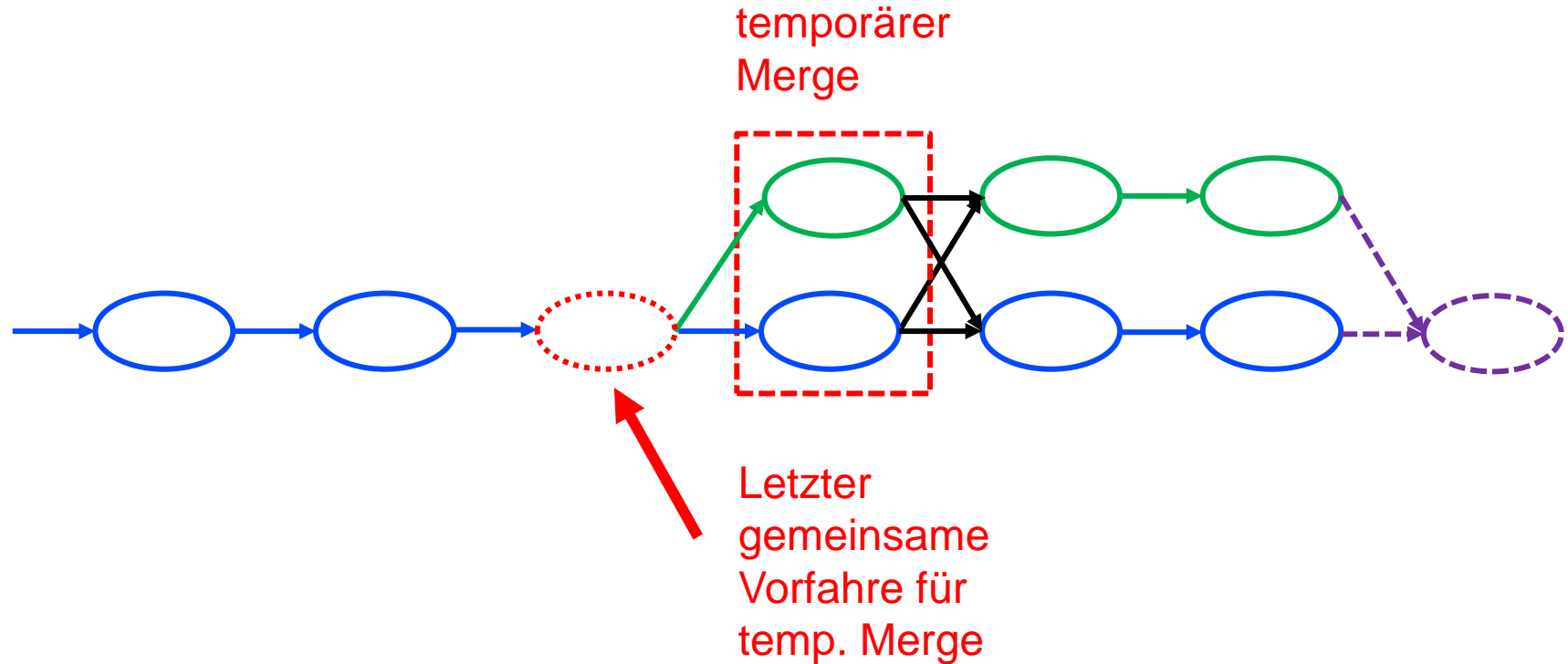


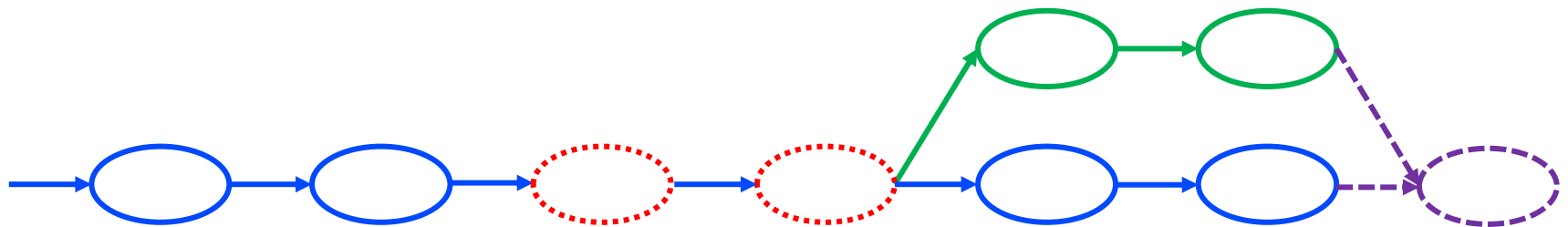


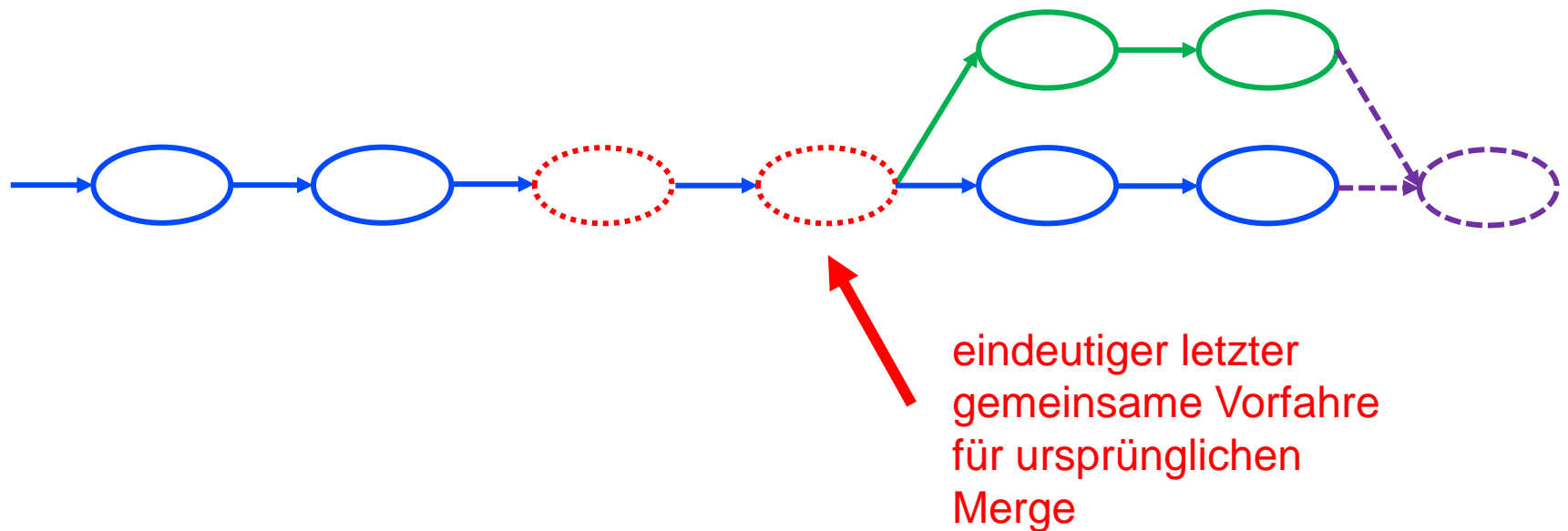


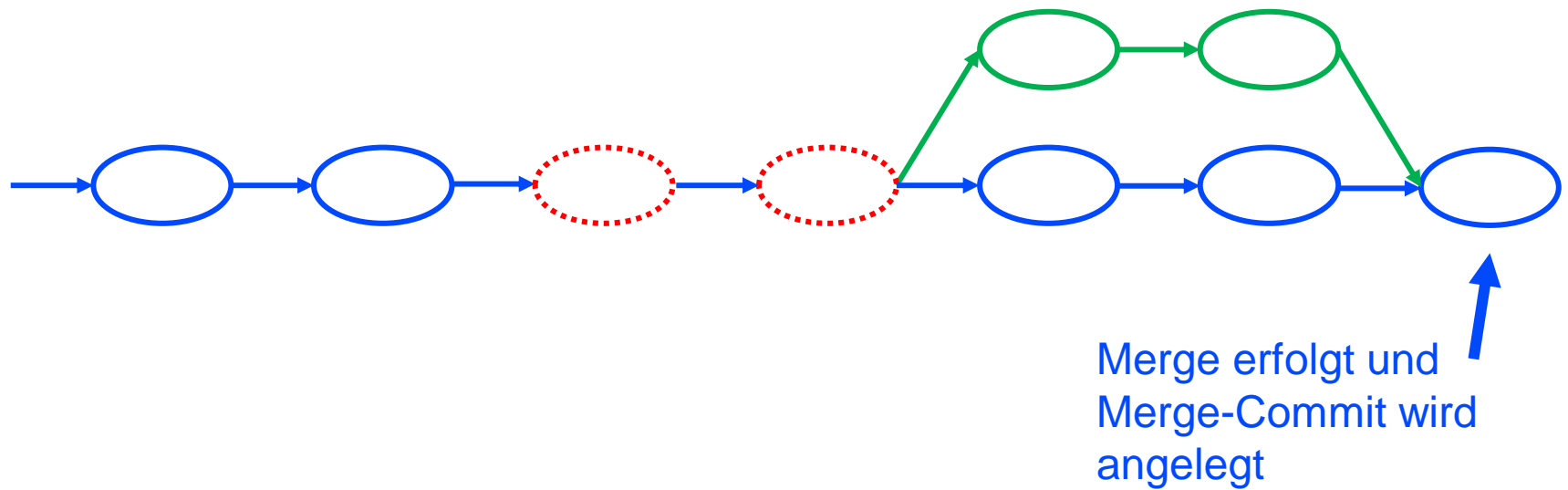


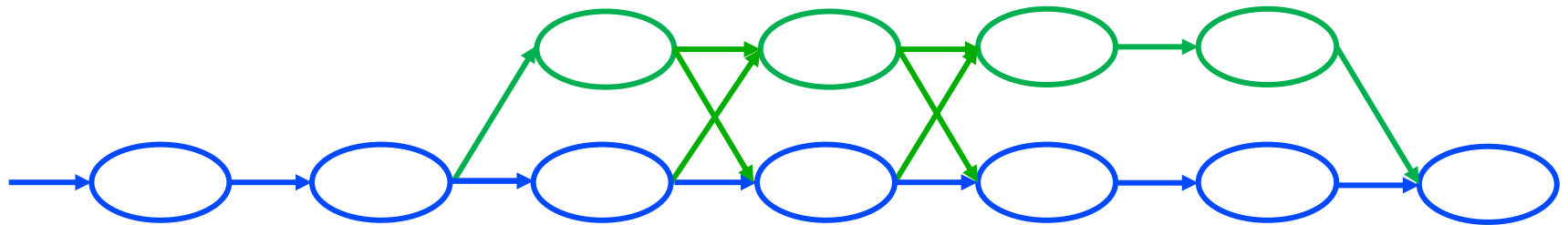












Git

Merge-Konflikte

Merge-Konflikte in Git

- Entstehen, wenn eine Zeile in zwei Branches auf verschiedene Art verändert wurde
- Erfordern manuelles Eingreifen
- Dateien ohne Konflikte werden automatisch gemerged und gestaged
- Entwickler muss nach Auflösen des Konfliktes den Merge-Commit selbst ausführen

Beispiel

```
$ git checkout -b conflict-feature  
Switched to a new branch 'conflict-feature'
```

```
$ echo "hello from feature branch" >  
conflict_file.txt
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Add file on feature"  
[conflict-feature 3984687] Add file on  
feature  
1 file changed, 1 insertion(+)  
create mode 100644 conflict_file.txt
```

```
$ git checkout main  
Switched to branch 'main'
```

```
$ echo "hello from main branch" >  
conflict_file.txt
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Add file on main"  
[main c9bb037] Add file on main  
1 file changed, 1 insertion(+)  
create mode 100644 conflict_file.txt
```

```
$ git merge conflict-feature  
Auto-merging conflict_file.txt  
CONFLICT (add/add): Merge conflict in  
conflict_file.txt  
Automatic merge failed; fix conflicts and  
then commit the result.
```

Auflösen von Konflikten im Editor

- In der betroffenen Datei werden durch Git Indikatoren hinzugefügt

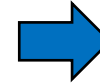
```
$ cat conflict_file.txt
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

- Die Originaldatei wird unter <Dateiname>.orig abgespeichert
- Entwickler muss entscheiden, welche Änderungen übernommen werden sollen und ungewünschte Änderungen löschen

Auflösen des Konfliktes

```
$ nano conflict_file.txt
```

```
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

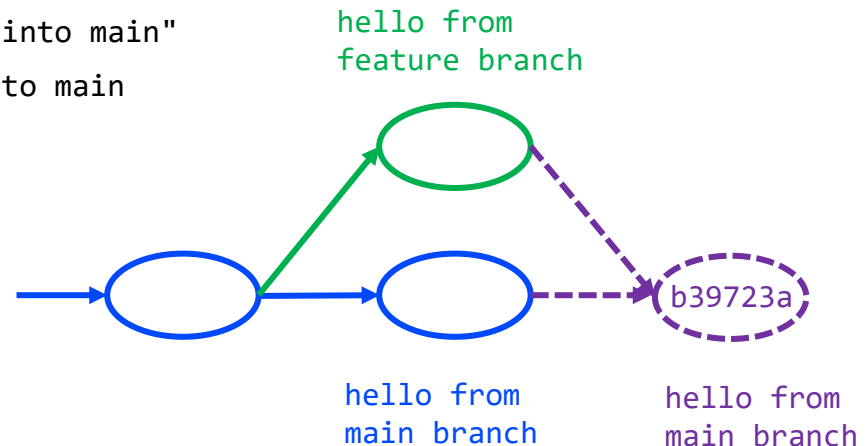


```
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Merge conflict-feature into main"
[main b39723a] Merge conflict-feature into main
```

```
$ cat conflict_file.txt
hello from main branch
```



Aufgabe 10: Merge-Konflikte

Im Nachfolgenden werden Sie beispielhaft einen Merge-Konflikt verursachen und auflösen.

1. Erstellen Sie einen neuen Branch **feature3** und wechseln Sie in diesen.
2. Legen Sie im Hauptordner eine Datei **merge_conflict_file.txt** mit dem Inhalt „Hello from feature3“ an und committen Sie diese.
3. Wechseln Sie zurück in den **main** Branch und legen Sie hier ebenfalls eine Datei **merge_conflict_file.txt** im Hauptordner an. In dieser Datei soll jedoch „Hello from main“ stehen. Committen Sie die Datei.

Aufgabe 10: Merge-Konflikte

Damit besitzen sowohl **main** als auch **feature3** eine Datei **conflict_file.txt**, beide jedoch mit unterschiedlichem Inhalt.

4. Versuchen Sie, **feature3** in **main** zu mergen.
Git sollte Ihnen anzeigen, dass es einen Merge-Konflikt bei der Datei **merge_conflict_file.txt** gab.
5. Öffnen Sie die Datei in einem Editor Ihrer Wahl. Mittels der Markierungen wird Ihnen angezeigt, wie die Branches divergieren.
6. Löschen Sie die den Bereich von **feature3** sowie sämtliche Markierungen, speichern Sie die Datei. Damit haben Sie entschieden, welche Änderungen verworfen bzw. übernommen werden.
7. Stagen Sie die Datei und erzeugen Sie nun selbst den Merge-Commit, um den Merge abzuschließen.

Aufgabe 11: Merge-Konflikte

Das vorherige Beispiel war ein sehr simpler Merge-Konflikt. Grundlegend führt der Fall, dass zwei Branches eine Datei auf verschiedene Art und Weise ändern, auch nicht zwingend zu einem Konflikt.

Ein Merge kann konfliktfrei funktionieren, jedoch semantische Konflikte produzieren, die Git nicht erkennt.

Beides schauen wir uns im Folgenden an.

1. Erstellen Sie auf dem **main** Branch eine Datei **sum.sh** mit dem Inhalt

```
#!/bin/bash
```

```
echo "Result: $((($1 + $2)))"
```

2. Comitten Sie die Datei.

Aufgabe 11: Merge-Konflikte

Als neues Feature soll nun dem Skript noch eine Ausgabe der übergebenen Argumente hinzugefügt werden.

3. Erstellen Sie einen neuen Branch **sum-feature** und wechseln Sie in diesen. Fügen Sie hier mit einem Editor eine Ausgabe **über** der Berechnung ein

```
#!/bin/bash
```

```
echo "Arguments are: $1 and $2"  
echo "Result: $((($1 + $2))"
```

4. Speichern und committen Sie die Änderungen und wechseln Sie zurück auf den **main** Branch.

Aufgabe 11: Merge-Konflikte

Durch ein Versehen werden die geforderten Änderungen ebenfalls direkt auf den **main** Branch committet.

- Öffnen Sie die **sum.sh** Datei im Editor und eine Ausgabe **unter** der Berechnung ein

```
#!/bin/bash
```

```
echo "Result: $(( $1 + $2 ))"  
echo "Arguments were: $1 and $2"
```

- Speichern und committen Sie die Änderungen.
- Mergen Sie nun den **sum-feature** Branch in den **main** Branch. Git sollte hierbei alle Änderungen auflösen können, sodass kein Konflikt entsteht.

Aufgabe 11: Merge-Konflikte

Git merged Dateien immer zeilenbasiert. Dabei werden nicht nur simpel die Zeilennummern verglichen, sondern die intelligenteren Merge-Strategien erkennen neu eingefügte Zeilen.

Git wertet daher die jeweiligen Argumentausgaben als neu hinzugefügte Zeilen.

Diese erzeugen kein Konflikt, denn wenn eine Zeile nur auf einem Branch existiert, auf dem anderen jedoch nicht, wird die Zeile einfach in das Ergebnis der Merges übernommen.

Semantisch sind die Änderungen jedoch falsch, in anderen Szenarien können auch tiefgreifende Fehler entstehen als eine einfache Textausgabe wie in diesem Fall.

8. (Optional) Führen Sie **sum.sh** aus.

Git – Merges

Befehle

- Merging von mehr als zwei Branches ist möglich und in Git implementiert
- Im normalen Workflow aber meist Merge zwischen zwei Branches
- Liste aller in den aktiven Branch gemergeten Branches

```
git branch --merged
```

```
$ git branch --merged  
feature1  
* main
```

- Liste aller vom aktiven Branch abgezweigten Branches, die noch nicht gemerged wurden

```
git branch --no-merged
```

```
$ git branch --no-merged  
feature2
```

- Nachdem ein Branch gemerged wurde, gilt er oft als abgeschlossen (Hotfixes, Features, Bugfixes, ...)
- Gemergeten Branch löschen mittels

```
git branch -d <branch> (oder --delete)
```

- Beispiel

```
$ $ git branch --delete feature1  
Deleted branch feature1 (was edff90f).
```


- Ungemergte Branches erzeugen eine Error Meldung

```
$ git branch --delete feature2
```

```
error: The branch 'feature2' is not fully merged.
```

```
If you are sure you want to delete it, run 'git branch -D feature2'.
```

- Ungemergete Branches löschen mittels einem der folgenden Befehle

```
git branch -D <branch>
```

```
git branch -d -f <branch>
```

```
git branch --delete --force <branch>
```

- Beispiel

```
$ git branch -D feature2
```

```
Deleted branch feature2 (was 609a7d9).
```

Git – Merges

Strategien

- Beim `git merge` Befehl kann mit der Option `-s` eine Mergestrategie ausgewählt werden
- Zusätzliche Optionen können mittels `-X<option>` angegeben werden
- Manual mit weiteren Informationen unter <https://git-scm.com/docs/merge-strategies>

Strategie ort

- Ostensibly Recursive's Twin (kurz **ort**) ist die Standard-Strategie in Git seit v2.33.0
- Merged zwei HEADs mittels 3-Way-Merge Algorithmus
- Bei mehreren gemeinsamen Vorfahren werden diese rekursiv gemerged und als letzter gemeinsame Vorfahre genutzt
→ Verhalten der vorherigen Beispiele

ort – Optionen

- **ours**
 - Übernimmt „ours“ bei Konflikten, also die Dateiversion des aktiven Branches, in den gemerged wird
 - Änderungen des anderen Branches werden übernommen, wenn diese keine Konflikte verursachen
 - Binär-Dateien werden automatisch komplett vom aktiven Branch übernommen
 - Nicht mit der Strategie „ours“ verwechseln, hier nur Option unter **ort**
- **theirs**
 - Gegenteil zu **ours**

- ignore-space-change
ignore-all-space
ignore-space-at-eol
ignore-cr-at-eol
 - Zeilen mit zutreffenden Kriterien gelten als unverändert
 - Enthält eine Datei auf einem Branch nur Whitespace-Änderungen und auf dem anderen Branch inhaltliche Änderungen, werden die inhaltlichen Änderungen übernommen
- renormalize
 - Normalisiert durch virtuellen Check-Out und Check-In in den verschiedenen Phasen des 3-Way-Merge Dateien
 - Nützlich beim Mergen von Branches mit verschiedenen Zeilenenden-Normalisierungen oder anderen Filtern

- `no-renormalize`
 - Deaktiviert Normalisierung und überschreibt **merge.renormalize** Konfigurations-Variable
- `find-renames[=<n>]`
 - Aktiviert Umbenennungserkennung (Standardmäßig an)
 - Ermöglicht setzen eines Thresholds zu Erkennung

Strategie Recursive

- War die Standard-Strategie in Git von v0.99.9k bis v2.33.0
- Nutzt ebenfalls wie **ort** einen 3-Way Merge Algorithmus zu Mergen von zwei Branches
- Bei mehreren gemeinsamen Vorfahren werden diese rekursiv gemerged und als letzter gemeinsame Vorfahre genutzt
 - ➔ Verhalten der vorherigen Beispiele
- Nutzt gleiche Optionen wie **ort**, besitzt jedoch zwei zusätzliche Optionen

Recursive – Optionen

- `diff-algorithm=[patience|minimal|histogram|myers]`
 - Ermöglicht Angabe eines bestimmten Merge Algorithmus
 - **ort** verwendet explizit histogram, während recursive auf die Option `diff.algorithm` Konfiguration zurückgreift
- `no-renames`
 - Deaktiviert Umbenennungserkennung und setzt **merge.rename** Variable

Strategie – Resolve

- Nutzt einfacheren Ansatz als ort/recursive um mittels 3-Way Merge Algorithmus zwei Branches zu mergen
- Simpler aber weniger effektiv in automatischer Konflikt bzw. Problemlösung
- Versucht Criss-Cross Merges zu erkennen
- Umbenennungen werden nicht behandelt

Strategie Octopus

- Kann mehr als zwei Heads mergen
- Standardoption, wenn mehrere Branches beim Merge angegeben werden
- Kann keine komplexen Merges umsetzen, die ein manuelles eingreifen benötigen würden
- Hauptsächlich gedacht, um mehrere Topic Branches zusammenzufügen

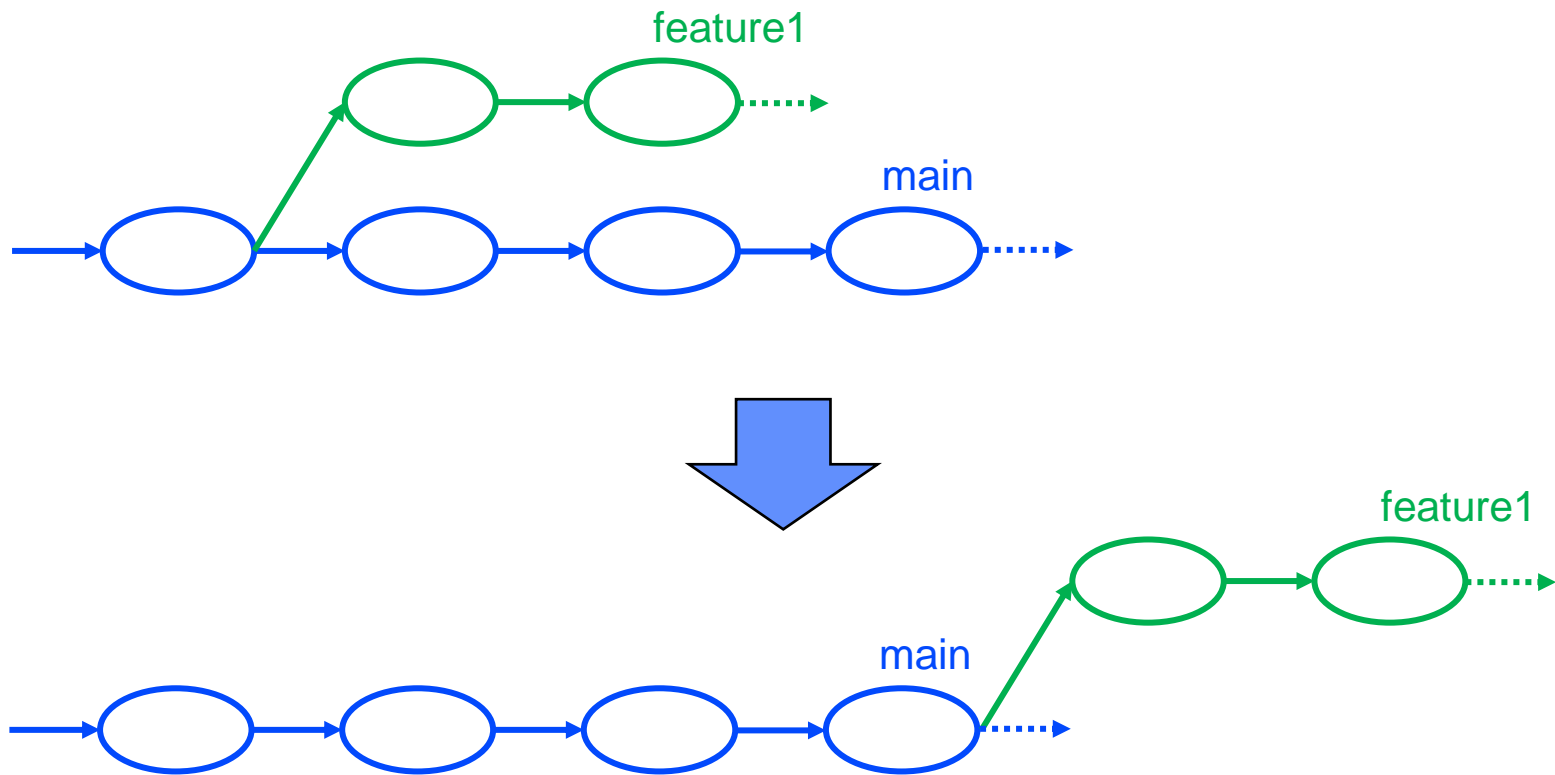
Strategie Ours

- Merged beliebig viele Branches
- Resultat des Merges spiegelt immer den Inhalt des aktiven Branches wider
- Änderungen der anderen Branches werden ignoriert und nicht in das Ergebnis übernommen
- Nützlich um Entwicklungslinie bzw. Historie zu erhalten, anstatt einfach anderen Branch zu löschen

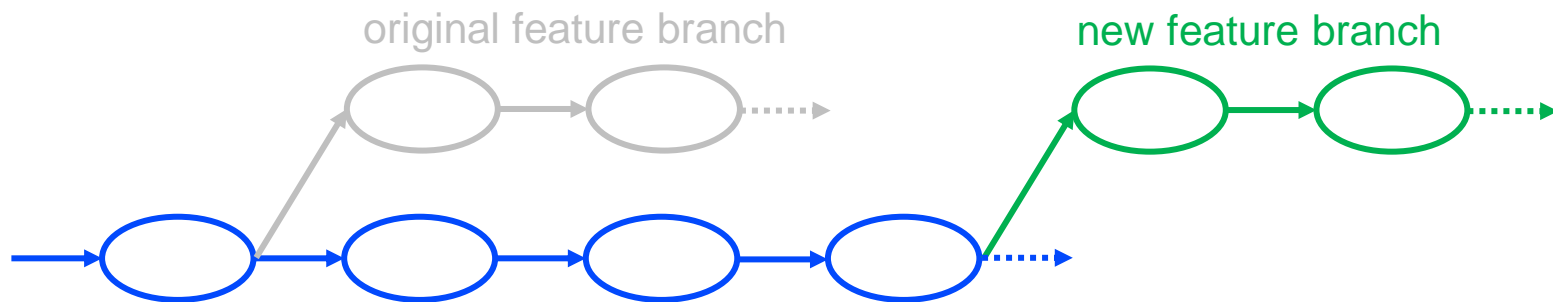
Git

Rebase

- Ein Rebase verändert den Startpunkt eines Branches



- Branch wird von anderem Startpunkt ausgehend neu erstellt
- Commits des zu rebasenden Branches werden nacheinander neu angewendet
 - Änderungen werden vom neuen Startpunkt ausgehend angewendet
 - Kann ähnlich wie Merging zu Konflikten führen
 - Konflikte müssen manuell aufgelöst werden, um Rebase fortzusetzen
 - Durch die sequenzielle Anwendung der einzelnen Commits, kann es bei jedem weiteren Commit erneut zu Konflikten kommen



-
- original feature branch
- new feature branch

- Rebase des aktiven Branches auf *upstream* mittels

```
git rebase <upstream>
```

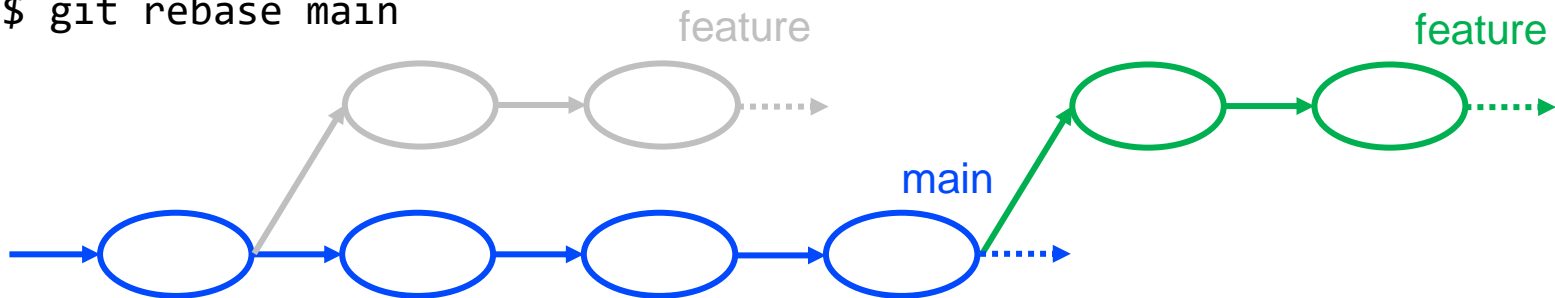
- Rebase eines Branches unabhängig vom aktiven Branch

```
git rebase <upstream> <branch>  
(verwendet intern git switch rebase vor rebase)
```

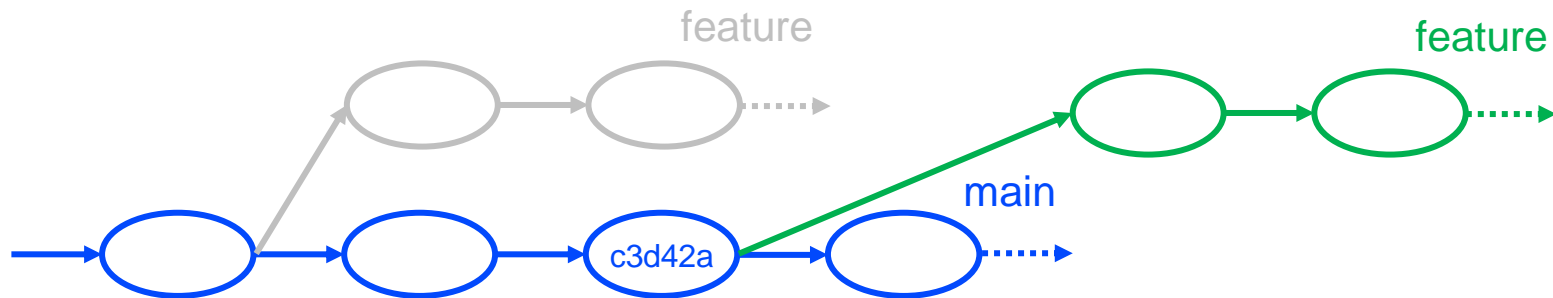
- Rebased auf den aktuellen HEAD des angegebenen Branches

- Beispiel: **feature** auf den HEAD des **main** rebasen

```
$ git checkout feature      oder      $ git rebase main feature  
$ git rebase main
```



- `<upstream>` und `<branch>` können Branches sowie einzelne Commits-IDs sein
- Beispiel: `feature` auf den `c3d42a` Commit des `main` rebasen
\$ git checkout feature
\$ git rebase c3d42a



Rebase Mode Options

- Anwenden mittels `git rebase <option> ...`
- Werden während eines Rebase verwendet und können nicht mit anderen Optionen kombiniert werden

<code>--continue</code>	Setzt Rebase nach Auflösen von Merge Konflikten fort
<code>--skip</code>	Überspringt aktuelle Ändeuerung und setzt Rebase fort
<code>--abort</code>	Bricht Rebase ab und setzt HEAD zurück
<code>--quit</code>	Bricht Rebase ab, HEAD und Workspace werden nicht resetet

- | | |
|-----------------------------------|---|
| <code>--edit-todo</code> | Fügt während interaktivem Rebase einen Eintrag zur ToDo Liste hinzu |
| <code>--show-current-patch</code> | Zeigt aktuelle Änderungen während interaktivem Modus oder beim pausiertem Rebase an |

Rebase Optionen

- Anwenden mittels `git rebase <option> ...`
- Werden beim Aufruf angegeben

`--onto <newbase>`

Gibt Startpunkt `<newbase>` an von dem neue Commits aus erstellt werden

`--keep-base`

Behält alten Startpunkt beim Rebase bei

`--apply`

Nutzt *applying strategies* während des Rebases

`--empty=(drop | keep | stop)`

Legt Verhalten für Commits fest, die beim Start Änderungen enthalten, welche beim Rebase aber durch gleiche Änderungen vom Upstream leer werden

<code>--no-keep-empty</code> <code>--keep-empty</code>	Bestimmt, ob Commits, die zu Beginn des Rebase leer waren, beibehalten oder verworfen werden (Default ist <code>--keep-empty</code>)
<code>--reapply-cherry-picks</code> <code>--no-reapply-cherry-picks</code>	Bestimmt, ob Commits, welche als Cherry-Pick bereits im Branch vorhanden waren beim Rebase behalten oder verworfen werden (Default ist verwerfen)
<code>--merge</code>	Merging Strategie wird beim Rebase zum Anwenden der Commits genutzt. (Default, nicht kompatibel zu <code>--apply</code>)
<code>-s <strategy></code>	Merge-Strategie die verwendet wird (Default <code>ort</code>)
<code>-X<strategy-option></code>	Merge-Strategie Optionen

<code>--no-ff</code> <code>--force-rebase</code> <code>-f</code>	Stellt sicher, dass alle Commits explizit einzeln angewendet werden. Die Historie des rebased Branches besteht damit ausschließlich aus neuen Commits
<code>--fork-point</code> <code>--no-fork-point</code>	Nutzt Reflog, um einen besseren gemeinsamen Vorfahren zwischen <code><upstream></code> und <code><branch></code> zu finden. Vorfahre wird zur Bestimmung der von <code><branch></code> eingefügten Commits benutzt
<code>--interactive</code> <code>-i</code>	Startet einen interaktiven Rebase
<code>--autosquash</code> <code>--no-autosquash</code>	Squashed Commits mit bestimmten Commit-Nachrichten

--autostash
--no-autostash

Stashed zu Beginn eine temporäre Kopie des Workspaces und stellt diese nach Abschluss wieder her.

Ermöglicht Rebase mit ungespeicherten Änderungen im Workspace.

Kann trotzdem nach Abschluss des Rebases zu Konflikten führen.

Rebase – Merge Strategien auswählen

- Mit der Option `-s` lässt sich eine Merge-Strategie übergeben, mit der die Änderungen aus den zu rebasenden Commits mit den Änderungen aus dem Upstream zusammengefügt werden
- Alle beim Merging thematisierte Strategien können verwendet werden
- Default bei Rebase ebenfalls `ort`
- `ours` und `theirs` sind beim Rebase vertauscht
- `ours` bezieht sich auf die Änderungen von `main`, `theirs` auf Änderungen von `feature`
- Dadurch ergibt ein Rebase mit Strategie `ours` keinen Sinn, da nur Änderungen von `main` übernommen werden
- Mit `-X<option>` können Optionen zur Strategie angegeben werden

Option --onto

- --onto ermöglicht feinere Einstellung

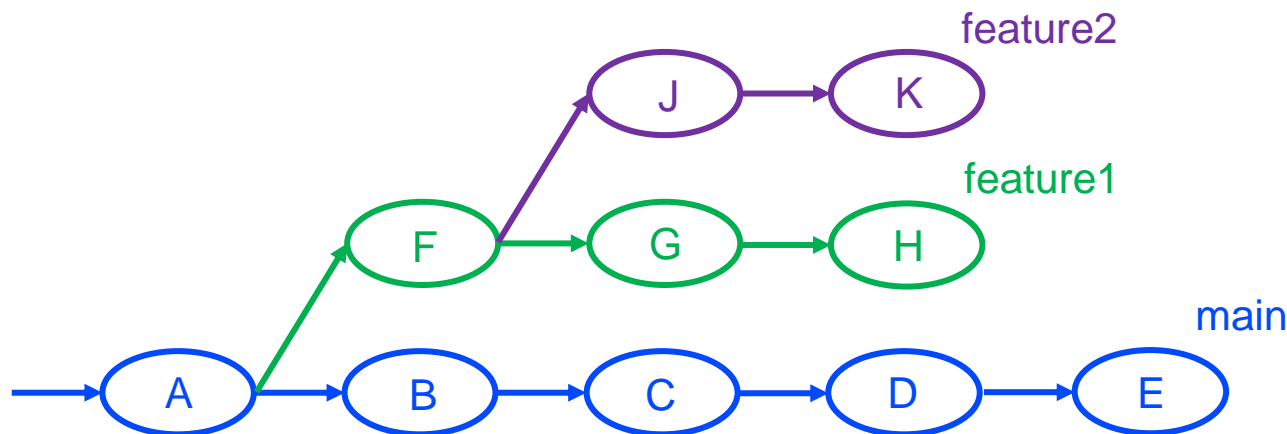
`git rebase --onto <newbase> <upstream> <branch>`

- <newbase> ist neuer Startpunkt
- <upstream> ist Branch oder der Commit, ab dem Commits verschoben werden
- <branch> ist Branch oder Commits, die verschoben werden.
Betrachtung rückwärts bis zum Erreichen des in <upstream> definierten Ausgangspunkt

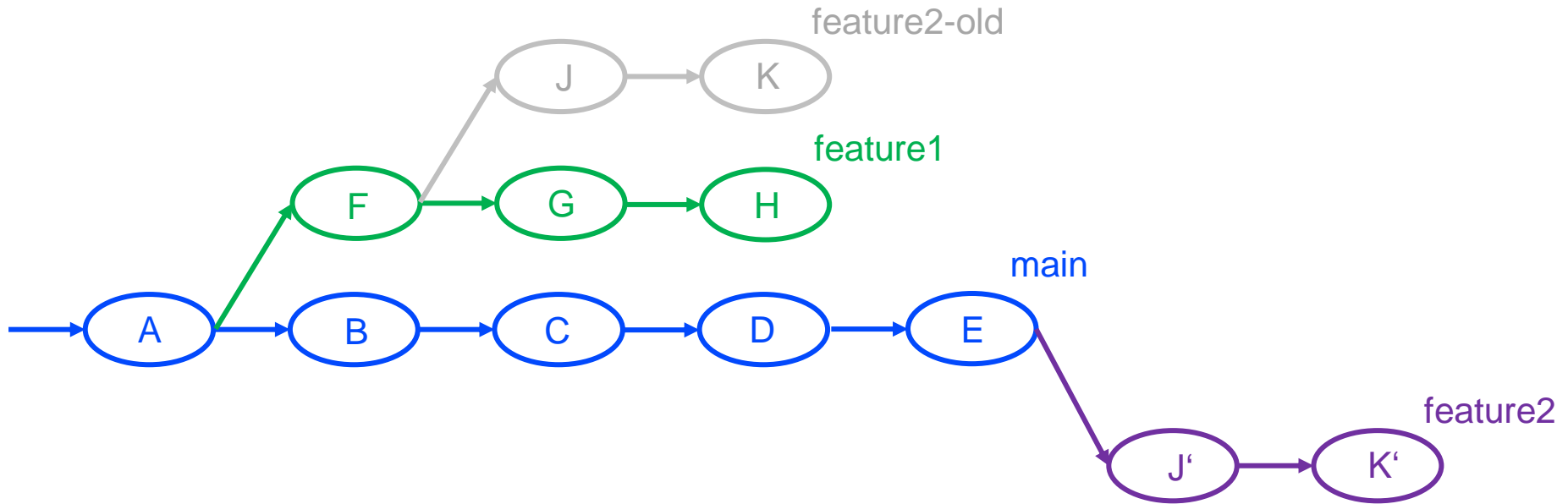
Beispiel: feature2 auf main rebasen

git rebase --onto main feature1 feature2

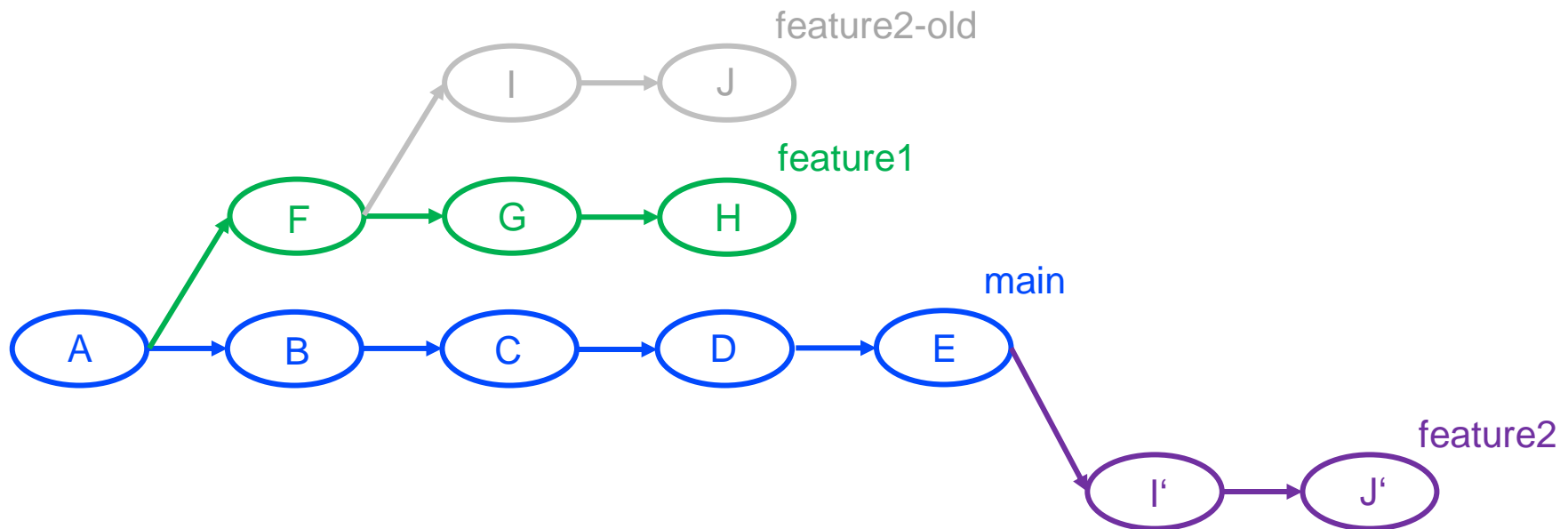
- main ist der neue Startpunkt, in diesem Fall der HEAD
- feature1 ist der Punkt, ab dem die Commits rebased werden.
- feature2 sind die Commits, die neu angewendet werden sollen



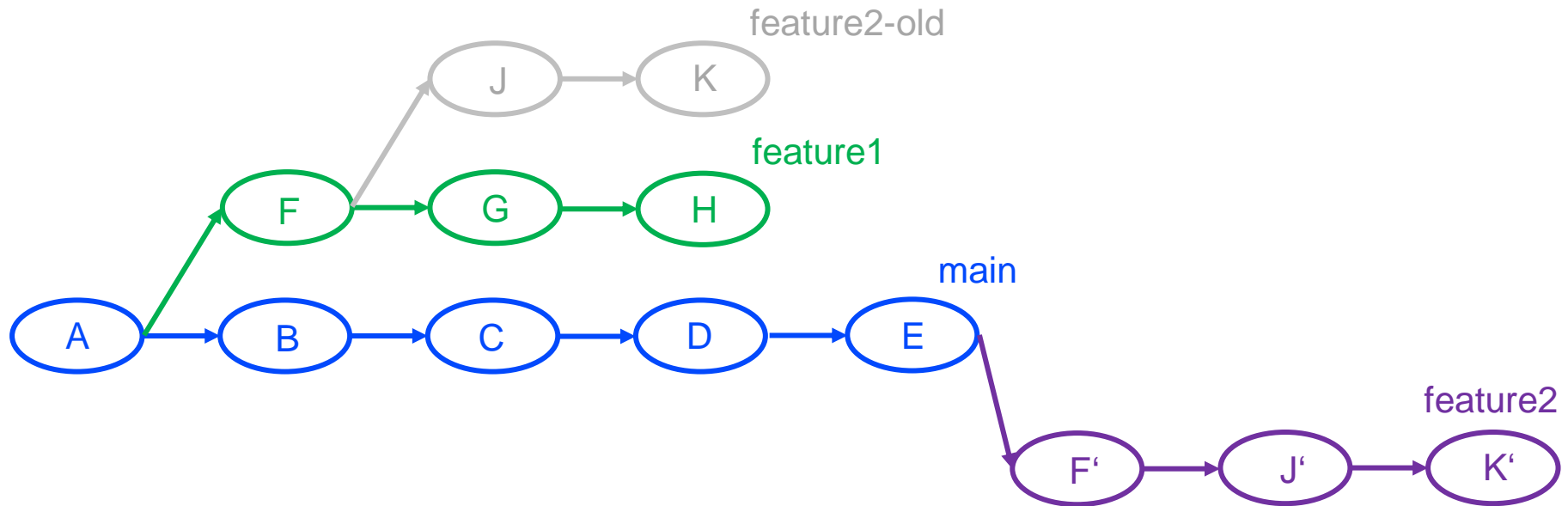
Ergebnis nach Rebase



- **feature2** basiert jetzt auf dem letzten Stand von **main**
- Commit **F** wurde explizit durch die Angabe von **feature1** als Beginn der zu rebasenden Commits ausgeschlossen
- Möglicher Use Case: **feature2** wurde in Abhängigkeit zu **feature1** geplant, stellt sich aber doch als unabhängig zu **feature1** heraus und soll daher unabhängig von **main** abzweigen

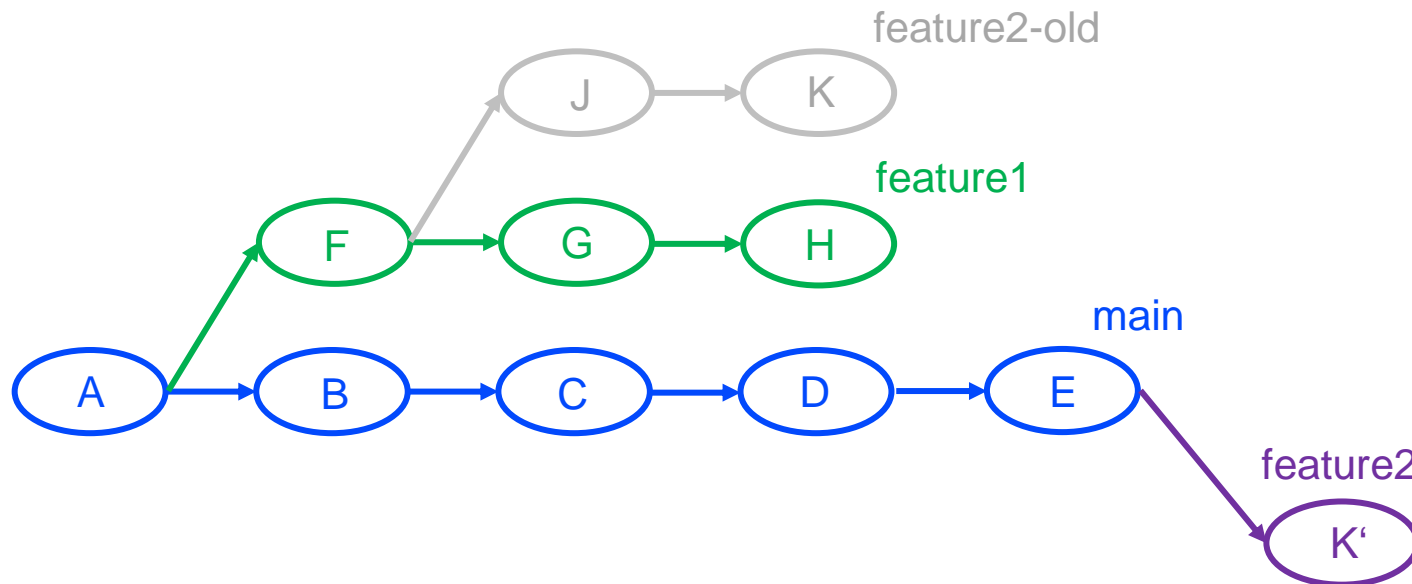


- `git checkout feature2` && `git rebase main` würde zu einem anderen Ergebnis führen, da die Commits ab `main` bis zum HEAD von `feature2` gerebased werden



- Ersetzt man **feature1** durch die Commit-ID von Commit J, so wird nur Commit K in den Rebase übernommen
- Commit J bildet hier den Beginn, ab dem die Commits gerebased werden

git rebase --onto **main** J **feature2**



Interactive Rebase

- Beim interaktiven Rebase kann man über einen Texteditor den Rebase steuern
- Starten eines interaktiven Rebase mittels Option `--interactive` oder `-i`
- Im Editor kann man die Commit-IDs der einzelnen Commits mit einem Command versehen, um Git mitzuteilen, wie mit diesem Commit umgegangen werden soll
- Commits können ebenfalls in ihrer Reihenfolge vertauscht werden
- Zum Übernehmen im Editor abspeichern und diesen schließen


```
pick b9898cd E
pick b059bc6 F
pick ce204eb G
pick 7b96b56 H
```

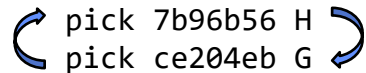
```
# Rebase 95e57be..7b96b56 onto 95e57be (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which case
#                               keep only this commit's message; -c is same as -C but
#                               opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was specified); use -c <commit> to
# reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

- Betrachtung der folgenden 4 Commits

```
pick b9898cd E
pick b059bc6 F
pick ce204eb G
pick 7b96b56 H
```

- Vertauschen der Reihenfolge möglich

```
pick b9898cd E
pick b059bc6 F
pick 7b96b56 H
pick ce204eb G
```



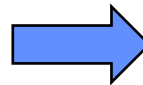
- Sqaushen von Commit E und G möglich

```
pick b9898cd E
squash ce204eb G
pick b059bc6 F
pick 7b96b56 H
```

Interactive Rebase – Fixup

- Ein Fixup beschreibt eine Situation, bei der ein Commit einen Fehler in einem anderen Commit behebt
- Rückwirkend ist es schön, diese Commits zusammenzufügen
- Commit-Nachricht von Fixup-Commit wird verworfen, außer anders spezifiziert
- Beispiel: Commit H ist Fixup von E

```
pick b9898cd E  
pick b059bc6 F  
pick ce204eb G  
pick 7b96b56 H
```



```
pick b9898cd E  
fixup 7b96b56 H  
pick b059bc6 F  
pick ce204eb G
```

--autosquash

- Funktioniert sehr ähnlich zur Variante im interaktiven Rebase
- Commits werden bei speziellen Commit Nachrichten gesquashed
- Mögliche Befehle für Commit Nachrichten sind squash, fixup oder amend
- Beispiel
 - Es existiert ein Commit mit der ID c0ffee und der Commit Message „My parent commit“
 - Existiert ein weiterer Commit mit der Commit Message „fixup! My parent commit“ oder „fixup! c0ffee“ so wird dieser automatisch mit seinem Parent Commit zusammengefügt

Übungsaufgabe 12: Rebase

Im Folgenden werden wir einen häufigen Use Cases eines Rebase betrachten, indem wir per Rebase Änderungen des **main** Branches in unseren Feature Branch übernehmen

1. Erstellen Sie einen Branch **feature5** und wechseln Sie in diesen.
2. Legen Sie im Ordner **features** eine Datei **feature5_file1.txt** an und committen Sie diese. Legen Sie anschließend eine weitere Datei **feature5_file2.txt** und committen diese ebenfalls.
3. Wechseln Sie nun zurück auf den **main** Branch, legen dort eine Datei **important_changes.txt** an und committen diese.

Übungsaufgabe 12: Rebase

Unser Ziel ist es nun, diese **important_changes.txt** auch im **feature5** Branch zur Verfügung zu haben. Man könnte nun den **main** Branch in unser Feature mergen, solange man aber alleine auf dem Feature Branch arbeitet ist ein Rebase eine elegantere und saubere Lösung.

4. Wechseln Sie in den **feature5** Branch, da wir diesen nun verändern möchten.
5. Lassen Sie sich mittels `git log` die Commit-IDs der beiden **feature5** Commits anzeigen und notieren Sie sich die IDs.
6. Rebasen Sie nun **feature5** auf den HEAD des **main** Branches.
7. Verifizieren Sie, ob die Datei **important_changes.txt** nun existiert.
8. Vergleichen Sie die Commit-IDs der aktuellen Ausgabe von `git log` mit der vorherigen Ausgabe. Unterscheiden sich die IDs?

Git

Konflikte beim Rebase

- Treten ähnlich zu Mergekonflikten auf, wenn Dateinhalte auf beiden Branches unterschiedlich verändert werden
- Rebase wird pausiert und Konflikte müssen analog zu Mergekonflikten manuell behoben werden
- Nach Auflösen des Konflikts müssen betroffene Dateien mit `git add <file>` hinzugefügt werden
- Danach mittels `git rebase --continue` den Rebase fortsetzen
- Da jeder Commit einzeln angewendet wird können Konflikte bei jedem Commit auftreten, daher muss man ggf. mehrfach Konflikte beheben
- Schwierigkeit eines Rebases steigt mit „Unordnung“ in Commit-Historie

Beispiel

```
$ git branch feature
$ echo "hello world from main" > conflict_file.txt
$ git add conflict_file.txt
$ git commit -m "Add file from main"
[main 722eaf0] Add file from main
 1 file changed, 1 insertion(+)
 create mode 100644 conflict_file.txt

$ git checkout feature
Switched to branch 'feature'

$ echo "hello world from feature" > conflict_file.txt
$ git add conflict_file.txt
$ git commit -m "Add file from feature"
[feature ce692c7] Add file from feature
 1 file changed, 1 insertion(+)
 create mode 100644 conflict_file.txt
```

```
$ git rebase main
Auto-merging conflict_file.txt
CONFLICT (add/add): Merge conflict in conflict_file.txt
error: could not apply ce692c7... Add file from feature
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply ce692c7... Add file from feature

$ nano conflict_file.txt
$ git add conflict_file.txt
$ git rebase --continue
Successfully rebased and updated refs/heads/feature.
```

Git – Rebase

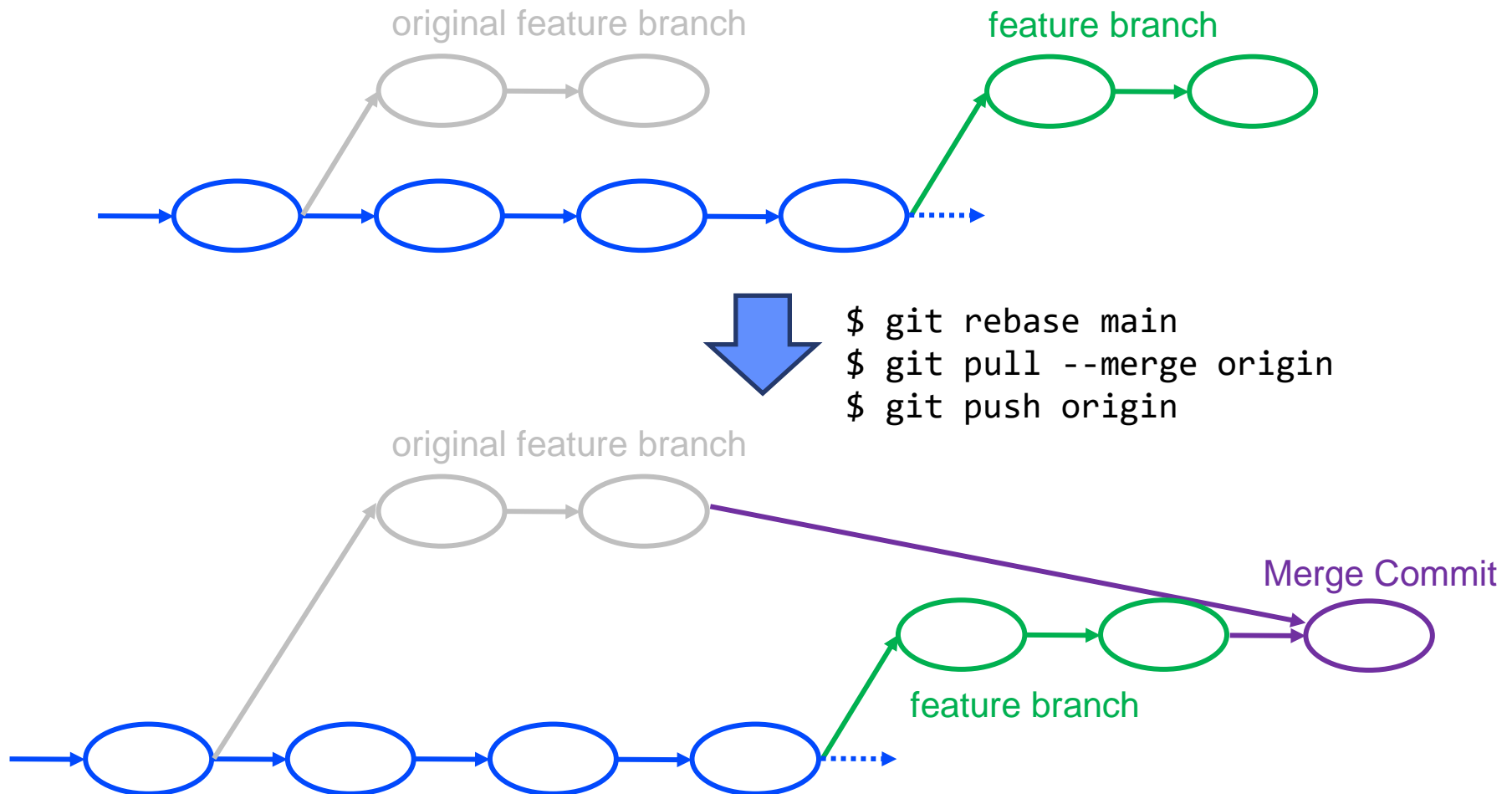
Auswirkungen

Auswirkungen auf Commit Historie

- Ein Rebase verändert die Commit-Historie (im Gegensatz zum Merge)
 - Kann zu Konflikten mit einem Remote-Repository führen
 - Branch folgt bei einem push nicht der Commit-Historie vom Remote Repository
 - Dadurch lehnt Remote den Push ab
 - Muss mittels `git push --force` (oder `-f`) ausgeführt werden, um alten Branch im Remote Repository zu überschreiben
 - Andere Entwickler, welche lokal mit dem Branch arbeiten, können diesen nicht weiter verwenden
- ➔ Auf öffentlichen oder geteilten Branches Rebase unbedingt vermeiden

- Ein Branch sollte nach einem Rebase immer mit `--force` gepushed werden, andere Varianten können zu ungewollten Erscheinungen führen
- Gelegentlich sieht man ein nach einem Rebase ein `git pull --merge` vom Remote mit anschließendem `git push` ohne `--force`
- Erzeugt zwar kein Konflikt, dafür aber ein zusätzlichen Commit auf dem rebased Branch

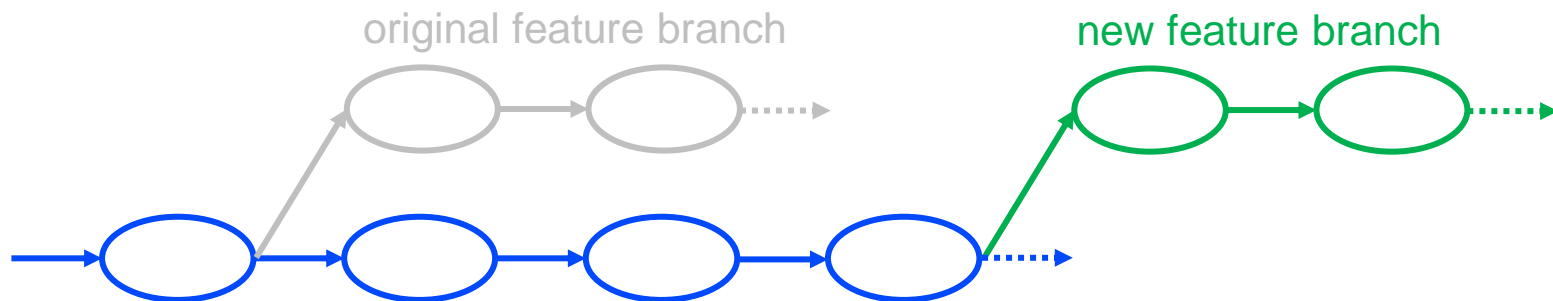
- **Beispiel** zu beschriebenem Szenario (oben Stand nach git rebase)



Git – Rebase

Use Cases

- Ein Rebase kann in vielen Situationen verwendet werden. Einige beispielhafte Situationen wären
 - Entwickler arbeitet auf Feature Branch und möchte neue Änderungen vom Upstream in seinen Branch übernehmen
 - Feature wird für ein bestimmten Release nicht fertig und soll als Feature von einem späteren Release eingefügt werden
 - Ein Feature zweigt von einem anderen Feature ab, stellt sich in der Umsetzung aber doch als unabhängig heraus. Daher Rebase auf main
 - Generelle Strukturänderungen im Repository, teilweise sogar nur einen Branch betreffend (Squashing, verändern von Commitreihenfolge, ...)



Nutzung zur nachträglichen Änderung von Commits

- Nachträgliche Änderung von Commit-Historie auf aktuellem Branch durch interactive Rebase
- Startpunkt des Branches wird nicht verschoben
- Änderungen der letzten n Commits möglich mittels

```
git rebase -i HEAD~n
```

- Letzten n Commits des aktuellen Branches werden gerebased und damit neu geschrieben
- Möchte man alle Commits des aktuellen Branches neu schreiben, so kann man die Option --keep-base verwenden

```
git rebase --keep-base -i main
```

- Andere Möglichkeit, alle Commits des Branches neu anzuwenden mittels

```
git merge-base feature main
```

um den aktuellen Abzweigungspunkt vom feature branch herausfinden.

Danach kann man dessen ID als Rebase Punkt angeben.

- Ermöglicht nachträgliches „Aufräumen“
- Nur auf lokalen, privaten Branches anwenden (mehr dazu in Kürze)

Beispiel

- Auf dem aktuellen Branch existieren 4 Commits

```
$ git log --oneline
50f7452 (HEAD -> feature) Fix feature_file1
568ede5 Add feature_files
21f72fe Add content to feature_file1
f26ab7a Add feature_file1
722eaf0 (main) Add file from main
c61ef14 Initial commit
```

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu. Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe. Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

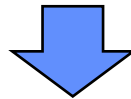
Rebase – Use Cases

```
$ git merge-base feature main  
722eaf04b1fe83e7c238852ca5857449202301ff
```

```
$ git rebase -i 722eaf
```

Rebase Editor öffnet sich

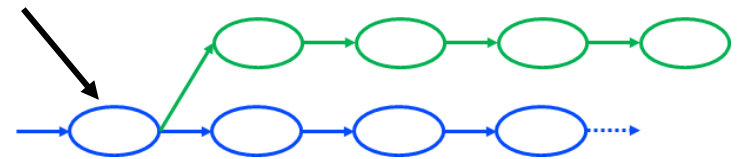
```
pick f26ab7a Add feature_file1  
pick 21f72fe Add content to feature_file1  
pick 568ede5 Add feature_files  
pick 50f7452 Fix feature_file1  
...
```



```
pick f26ab7a Add feature_file1  
squash 21f72fe Add content to feature_file1  
fixup 50f7452 Fix feature_file1  
edit 568ede5 Add feature_files  
...
```

Speichern und schließen

merge-base



- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu. Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe. Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

Commit-Nachricht Editor öffnet sich

```
# This is a combination of 3 commits.  
# This is the 1st commit message:  
  
Add feature_file1  
  
# This is the commit message #2:  
  
Add content to feature_file1  
  
# The commit message #3 will be skipped:  
  
# Fix feature_file1  
  
...
```

Ggf. Ändern, Speichern und
schließen

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu.
Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe.
Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

```
$ git merge-base feature main  
722eaf04b1fe83e7c238852ca5857449202301ff
```

```
$ git rebase -i 722eaf  
[detached HEAD ca0713e] Add feature_file1  
Date: Sun May 26 12:44:05 2024 +0200  
1 file changed, 1 insertion(+)  
create mode 100644 feature_file1.txt  
Stopped at 568ede5... Add feature_files  
You can amend the commit now, with
```

```
git commit --amend
```

Once you are satisfied with your changes, run

```
git rebase --continue
```

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu.
Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe.
Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

```
$ git reset --mixed HEAD^
```

```
$ git status
```

```
interactive rebase in progress; onto 722eaf0
```

```
Last commands done (4 commands done):
```

```
    fixup 50f7452 Fix feature_file1
```

```
    edit 568ede5 Add feature_files
```

```
(see more in file .git/rebase-merge/done)
```

```
No commands remaining.
```

```
You are currently editing a commit while rebasing  
branch 'feature' on '722eaf0'.
```

```
(use "git commit --amend" to amend the current  
commit)
```

```
(use "git rebase --continue" once you are  
satisfied with your changes)
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will  
be committed)
```

```
    feature_file2.txt
```

```
    feature_file3.txt
```

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

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu. Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe. Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

```
$ git add feature_file2.txt
$ git commit -m "Add feature_file2"
[detached HEAD 949a354] Add feature_file2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 feature_file2.txt
```

```
$ git add feature_file3.txt
$ git commit -m "Add feature_file3"
[detached HEAD 3beb3c4] Add feature_file3
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 feature_file3.txt
```

```
$ git rebase --continue
Successfully rebased and updated refs/heads/feature.
```

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu. Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe. Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

Resultat

```
$ git log --oneline
3beb3c4 (HEAD -> feature) Add feature_file3
949a354 Add feature_file2
ca0713e Add feature_file1
722eaf0 (main) Add file from main
c61ef14 Initial commit
```

- Commit f26ab7a soll mit 21f72fe zusammengefügt werden
- Commit 568ede5 fügt feature_file2.txt und feature_file3.txt hinzu.
Nachträglich wollen wir für bei jeweils einen Commit haben
- Commit 50f7452 ist ein Fix für Commit 21f72fe.
Rückblickend möchten wir diesen Commit nicht in der Historie haben und legen ihn mit f26ab7a zusammen

Übungsaufgabe 13: Rebase

Im Folgenden werden wir einen weiteren Use Cases eines Rebase betrachten, indem wir per Rebase die Commits unseres aktuellen Branches „aufräumen“.

1. Sie sollten sich noch auf dem **feature5** Branch befinden.
2. Füllen Sie die Datei **feature5_file1.txt** mit dem Inhalt „Hallo aus Datei 1 aus feature5 “ und committen Sie Ihre Änderungen.
3. Füllen Sie die Datei **feature5_file2.txt** mit dem Inhalt „Hello from file 2 in feature5 “ und committen Sie Ihre Änderungen.
4. Erstellen Sie einen weiteren Commit, in dem Sie den Inhalt von **feature5_file1.txt** umändern, sodass dieser ebenfalls in Englisch ist.

Übungsaufgabe 13: Rebase

Nun haben Sie auf Ihrem Branch relativ viele einzelne Commits, von denen einer nur eine Reparatur einer vorherigen Änderung ist.

Da Sie die einzige Person sind, die auf dem Branch arbeitet, können Sie die Commits neu schreiben. Dabei soll der Startpunkt des Branches nicht verändert werden.

5. Starten Sie einen interaktiven Rebase.

feature5 soll auf **main** gerebased werden, jedoch ohne die Merge-Base zu verändern.

Sie können hierzu zum einen mittels `git merge-base` den Abzweig-Commit von **feature5** herausfinden und auf diesen Rebasen oder den Rebase mit der Option `--keep-base` starten.

Übungsaufgabe 13: Rebase

6. Ordnen Sie im Editor die angegebenen Commits anders an. Fügen Sie den Commit zum Erstellen von file1 mit dem Commit zum Hinzufügen des Inhalts zu file1 mittels squash zusammen.
7. Squashen Sie ebenfalls die beiden Commits von file2.
8. Fügen Sie den Commit zum Reparieren des Inhalts von file1 als fixup an den ursprünglichen Commit hinzu.

Ihr Editor sollte nun ungefähr so aussehen

```
pick 8dafe28 add feature5_file1
squash 3d45a2 add content to feature5_file1
fixup af422a8 change content language to english
pick 2de828 add feature5_file2
squash bd7f22e add content to feature5_file2
```

9. Speichern Sie Ihre Änderungen ab und passen Sie im nächsten Fenster die Commit Nachrichten entsprechend an.

Übungsaufgabe 13: Rebase

6. Speichern Sie ebenfalls die Änderungen an den Commit Nachrichten. Der interaktive Rebase sollte nun erfolgreich abgeschlossen sein.
7. Verifizieren Sie über `git log`, dass Ihre Änderungen am **feature5** Branch erfolgreich waren.

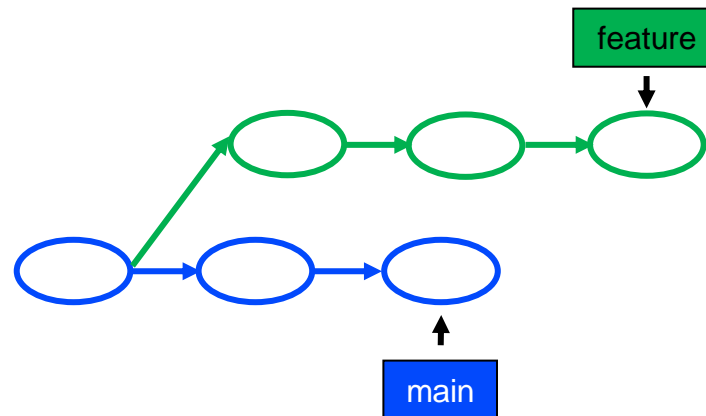
Git – Rebase

Use Cases – Merge vs Rebase

- Merge und Rebase haben hauptsächlich die gleiche Aufgabe, Änderungen in einen Branch einzupflegen
 - Rebase ändert aktiven Branch und kann dabei Änderungen vom Upstream übernehmen
 - Merge zieht Änderungen von anderen Branches in den aktiven Branch
- Daher die Frage „Wann benutze ich Merge, wann Rebase?“
 - Da Merging die Commit-Historie fortführt und nicht ändert, bietet es weniger Konfliktpotenzial als Rebasing
 - Rebasing erzeugt saubere, lineare aber veränderte Commit-Historie
- Um lokal Änderung aus dem Upstream in einen Branch einzupflegen, ist Rebase oft die bessere Option
 - Mergen vom Upstream in Feature kann zu komplizierten Szenarien führen, wenn man später das Feature wieder in den Upstream merged
 - Erzeugt lineare Historie

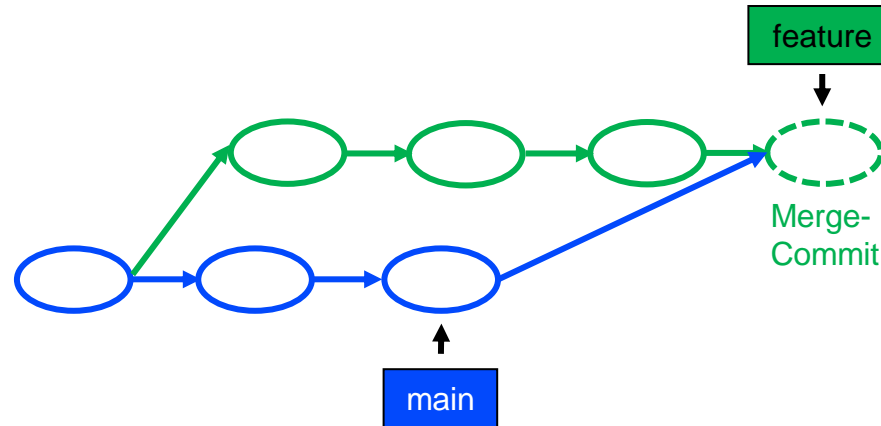
Merge vs Rebase

- Beispiel: Einfügen von Änderungen von **main** in **feature** Branch

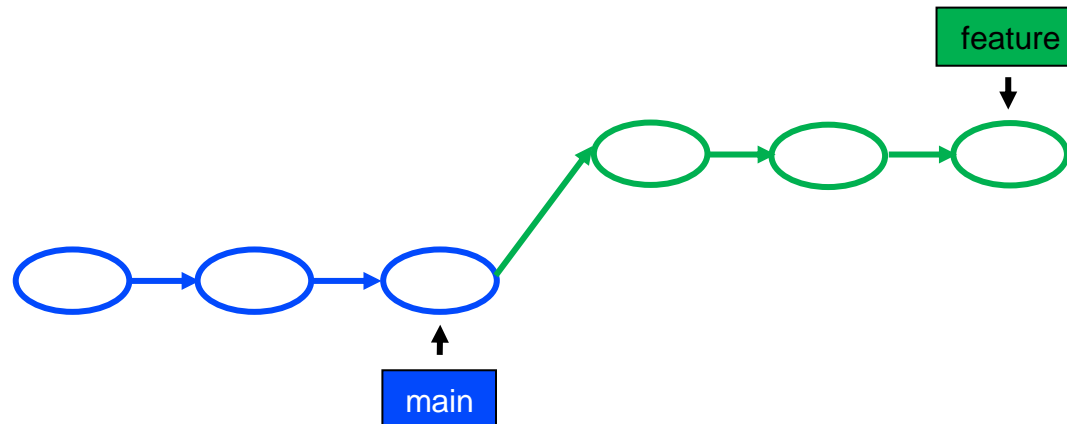


Merge vs Rebase

Merge



Rebase



- Um einen Feature Branch in seinen Upstream zurückzuführen, verwendet man Merge
 - Upstream soll weiter fortgeführt werden, Feature Branch oftmals nicht
 - Upstream meistens mit anderen Entwicklern geteilt
 - Ggf. kann ein Rebase des Feature Branches auf den HEAD des Upstream vor einem Merge sinnvoll sein. Dadurch erhält man automatisch einen konfliktfreien fast-forward Merge

