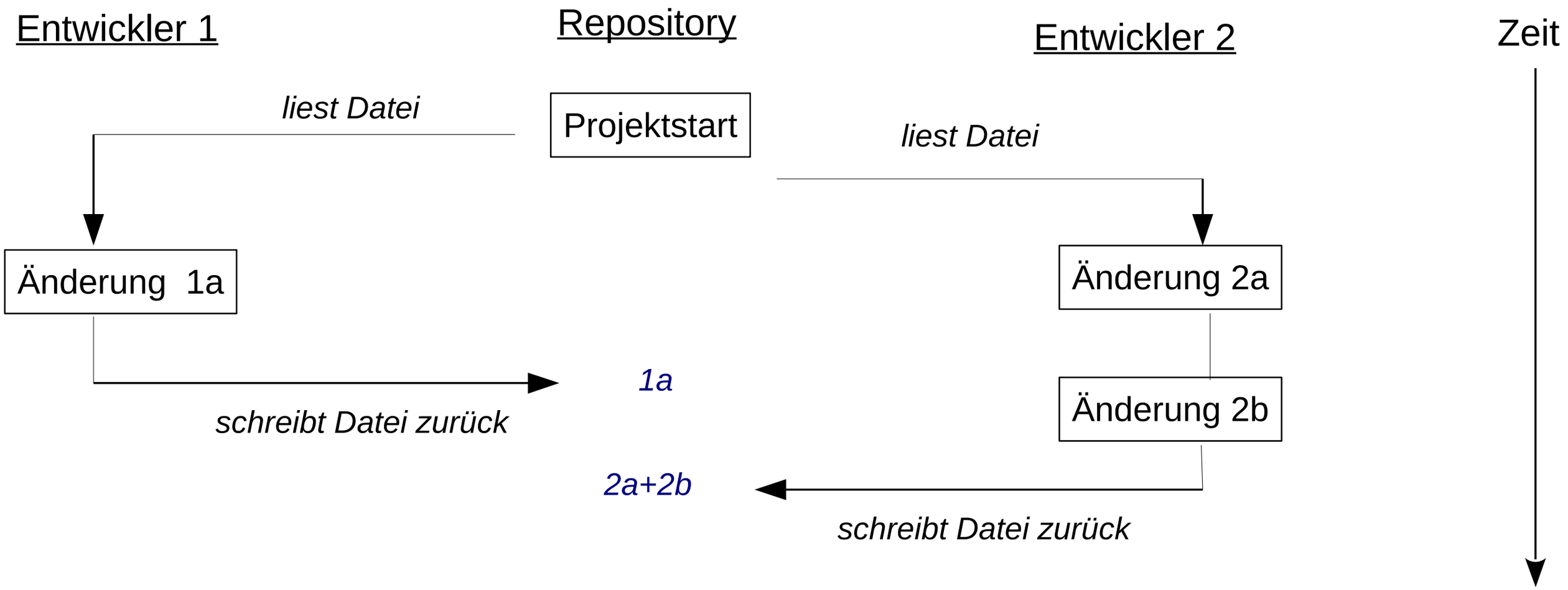
Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi/

8. Versionsverwaltung für mehrere Entwickler

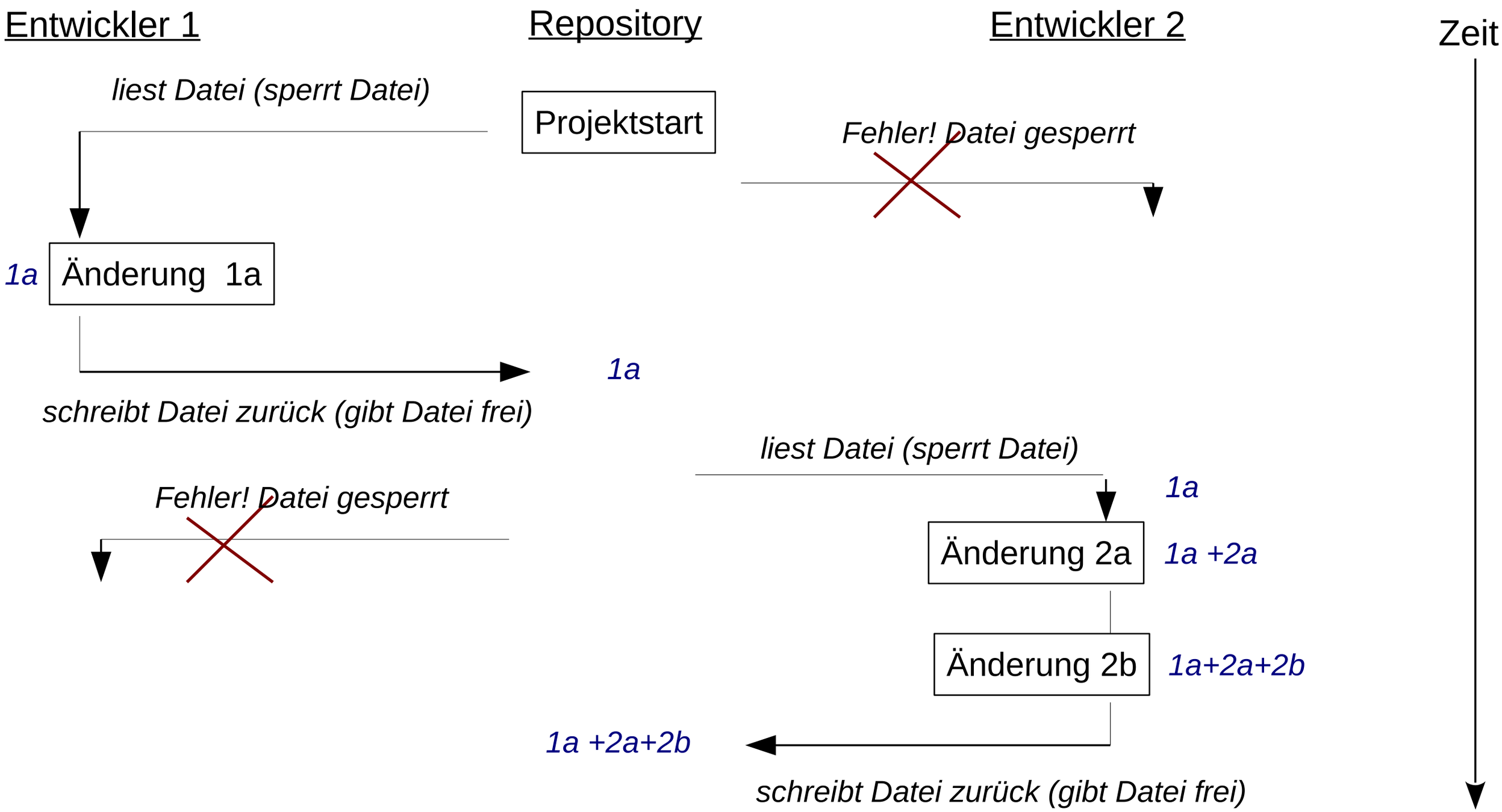
Grundproblematik der parallelen Entwicklung

Zwei Entwickler arbeiten am selben Projekt (an der selben Datei):



Problem:
Projekt (Datei) enthält immer nur die zuletzt zurückgeschriebene Änderungen, Änderungen anderer Entwickler werden überschrieben!

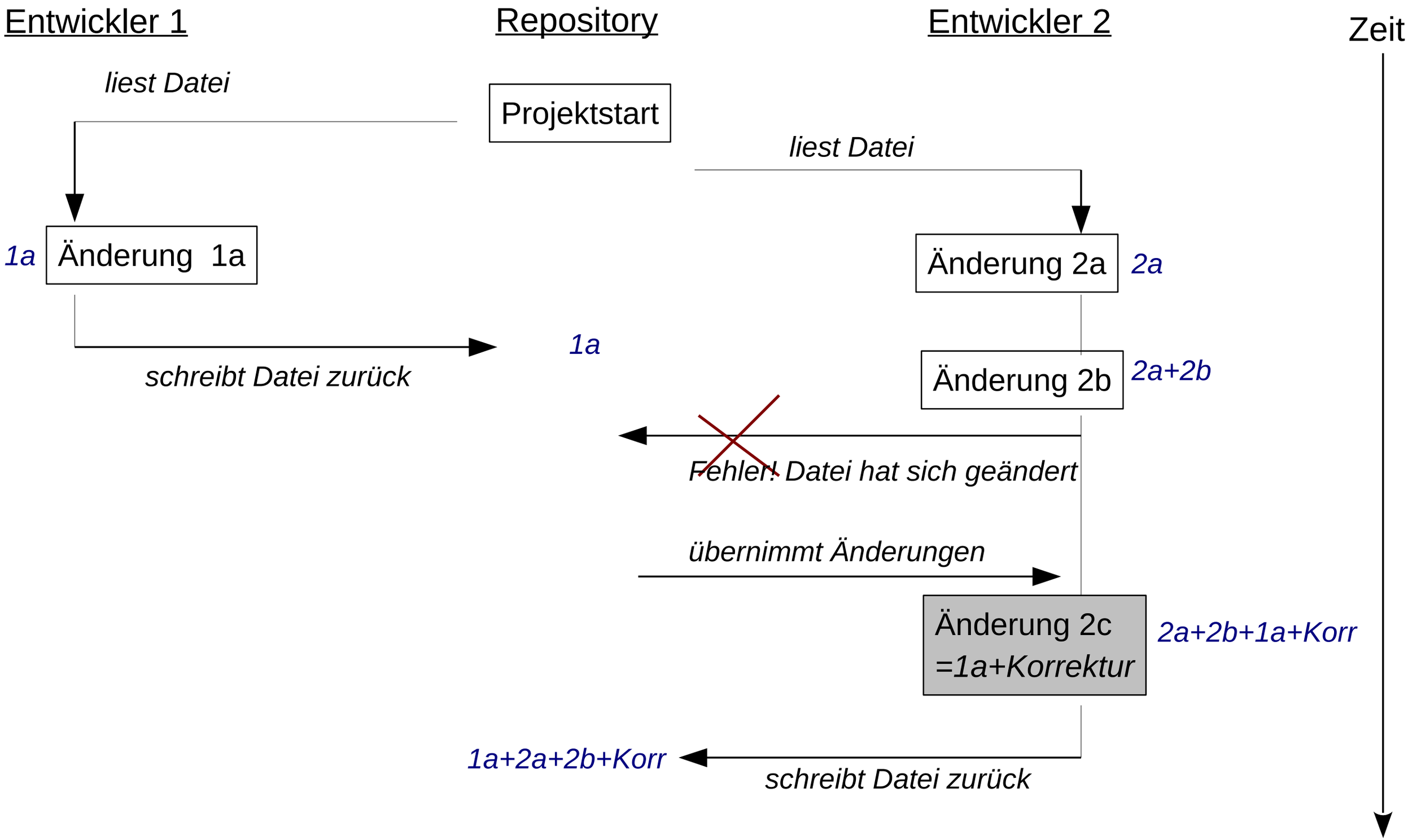
Sperren-Ändern-Freigeben (Zentralisiert)



Problem:

Gleichzeitig kann nur ein Entwickler an einer Datei arbeiten!

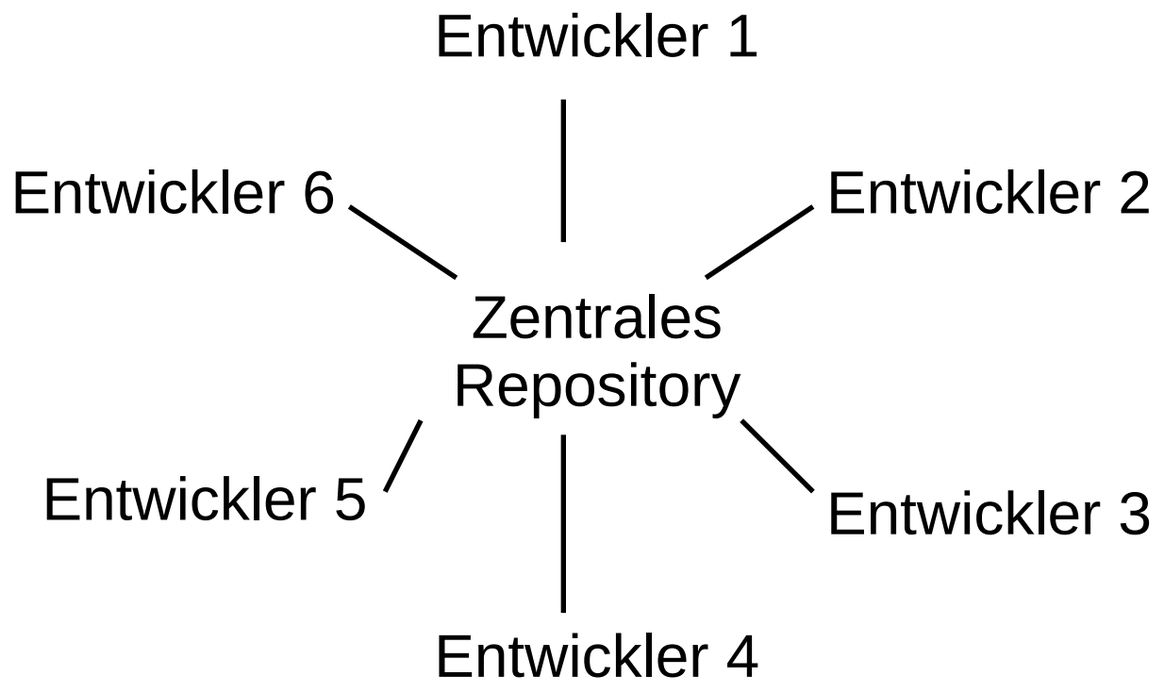
Kopieren-Ändern-Zusammenführen (Zentralisiert)



Mehrere Entwickler (Zentralisiert)

Vorteile:

- Offizielle Version des Codes im Repository
- Es wird unwahrscheinlicher, dass man aneinander vorbeiprogrammiert, da man alle Änderungen des zentralen Repository annehmen muss, bevor eigene Änderungen eingecheckt werden können.



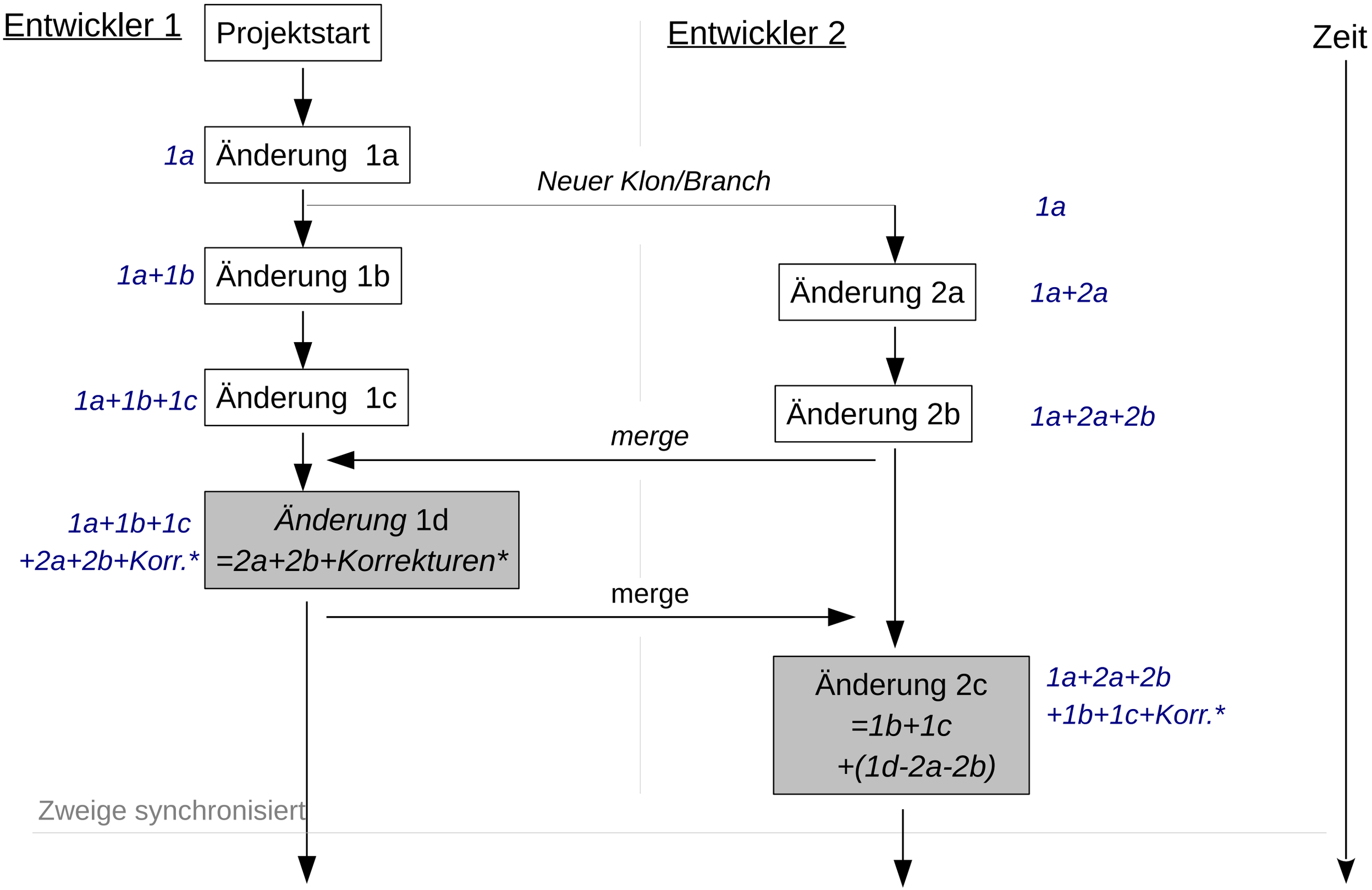
Nachteile:

- Es muss ein zentraler Server eingerichtet werden, mit Lese- und Schreibrechten für alle Entwickler
- Vor jedem Chekin muss eine Synchronisation mit zentralem Repository erfolgen, Programmieren ohne Netzwerkverbindung schwierig

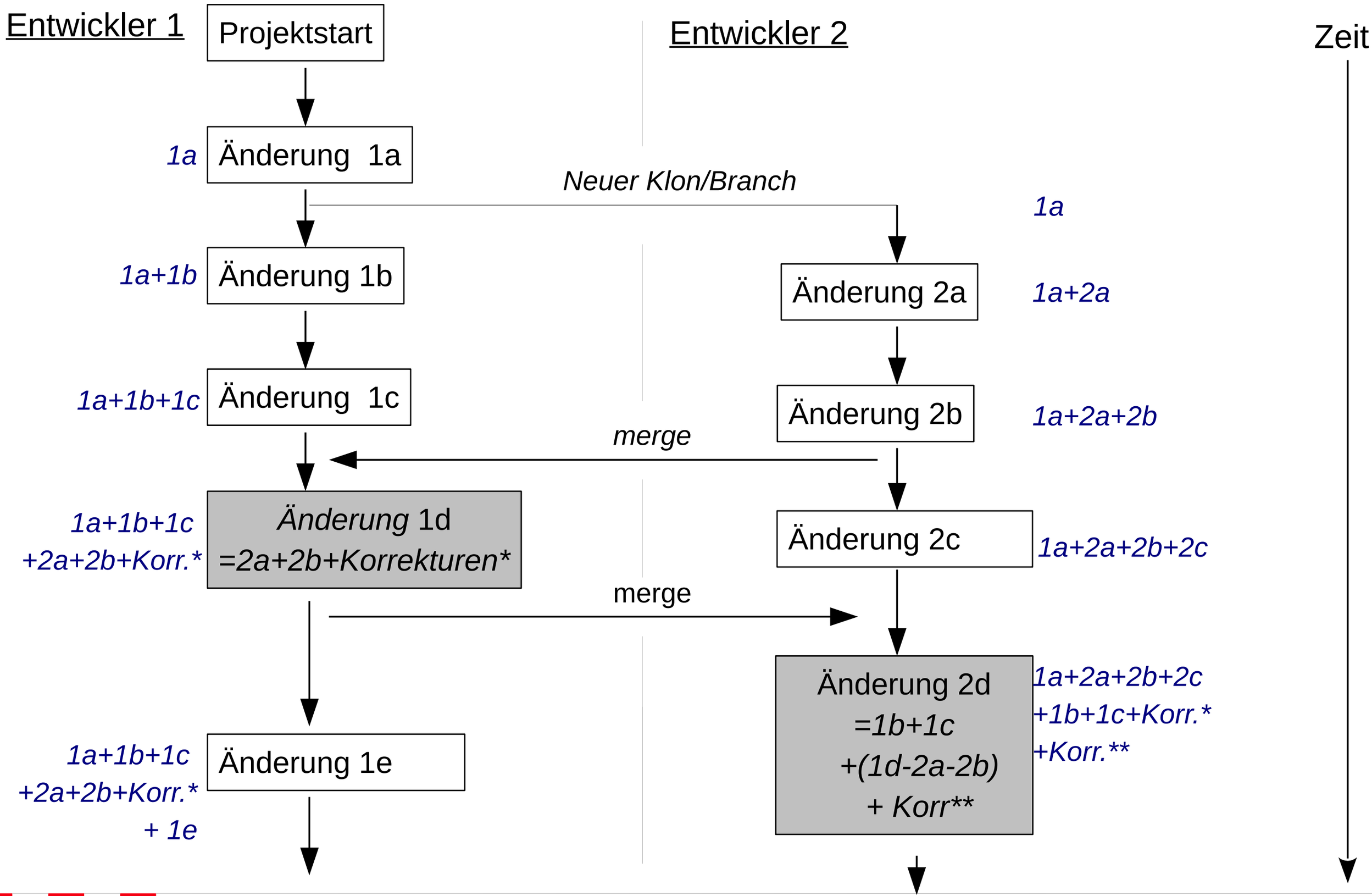
Entwicklungsablauf:

1. Synchronisieren mit z.R.
2. Änderungen machen.
3. Synchronisierung mit z.R.
4. Konflikte auflösen
5. Eigene änderungen einchecken

Zwei Entwickler (Dezentralisiert)



Zwei Entwickler (Dezentralisiert) (#2)



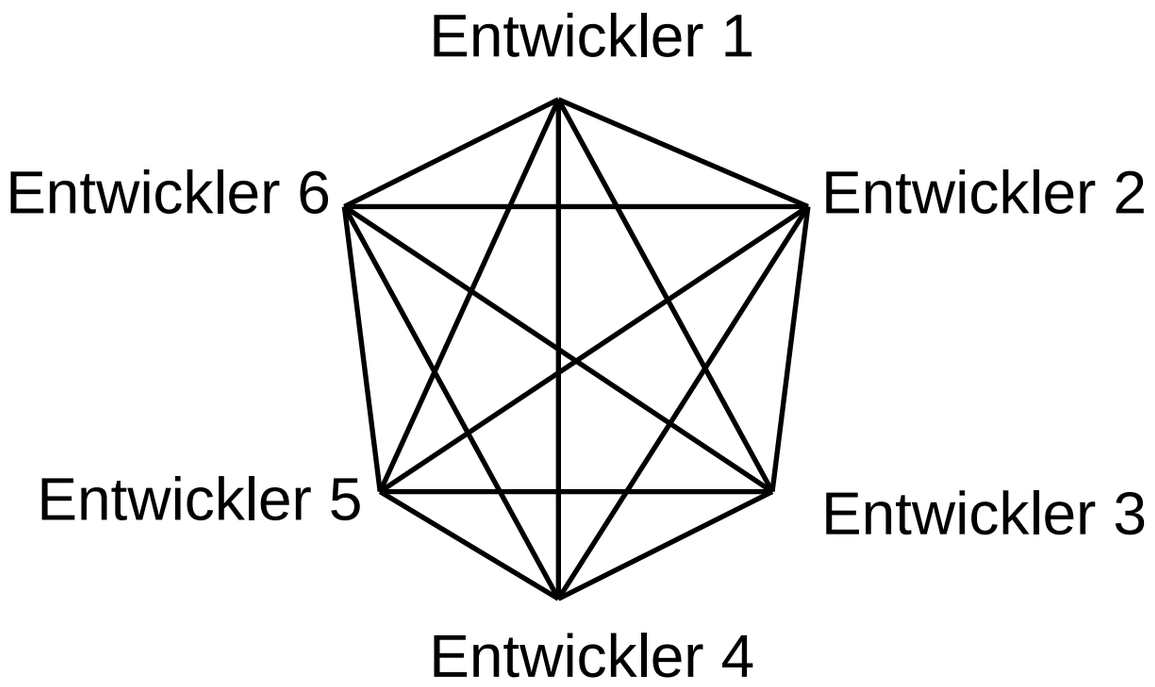
Mehrere Entwickler (Dezentralisiert)

Vorteile:

- Entwickler sind völlig unabhängig voneinander.
- Jeder Entwickler muss Repository von anderen nur lesen können.
- Jeder kann aussuchen von wem (und welche Änderungen) übernimmt.

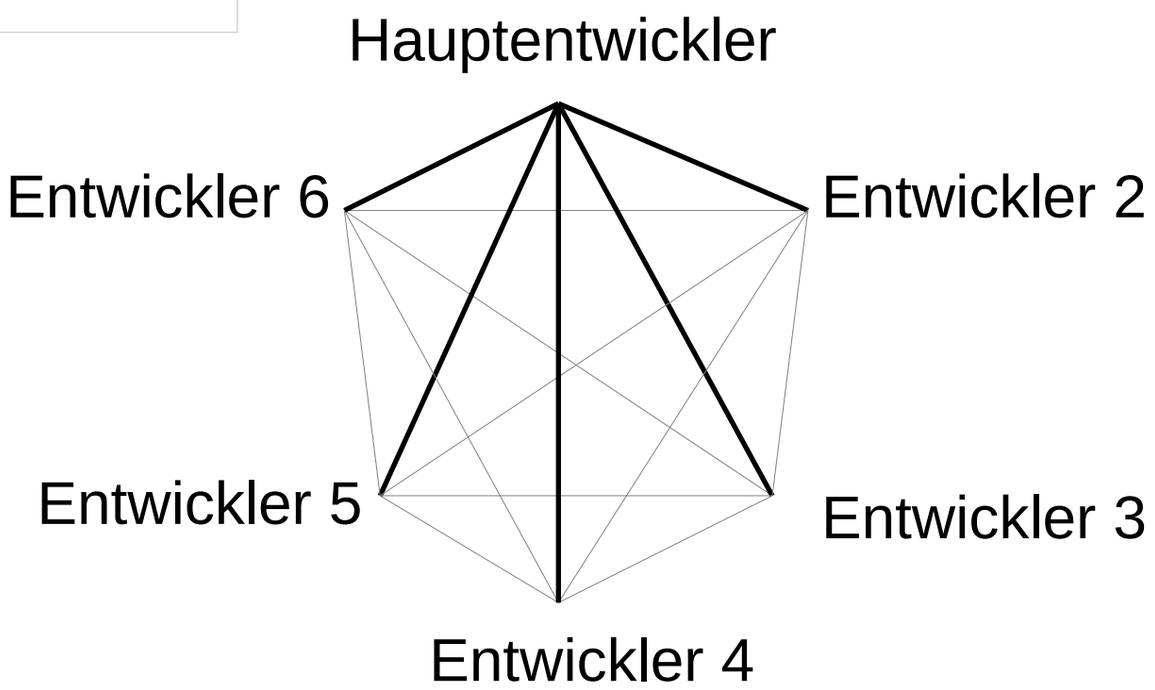
Nachteile:

- Kommunikationsintensiv
- Keine „offizielle“ Version des Programmes

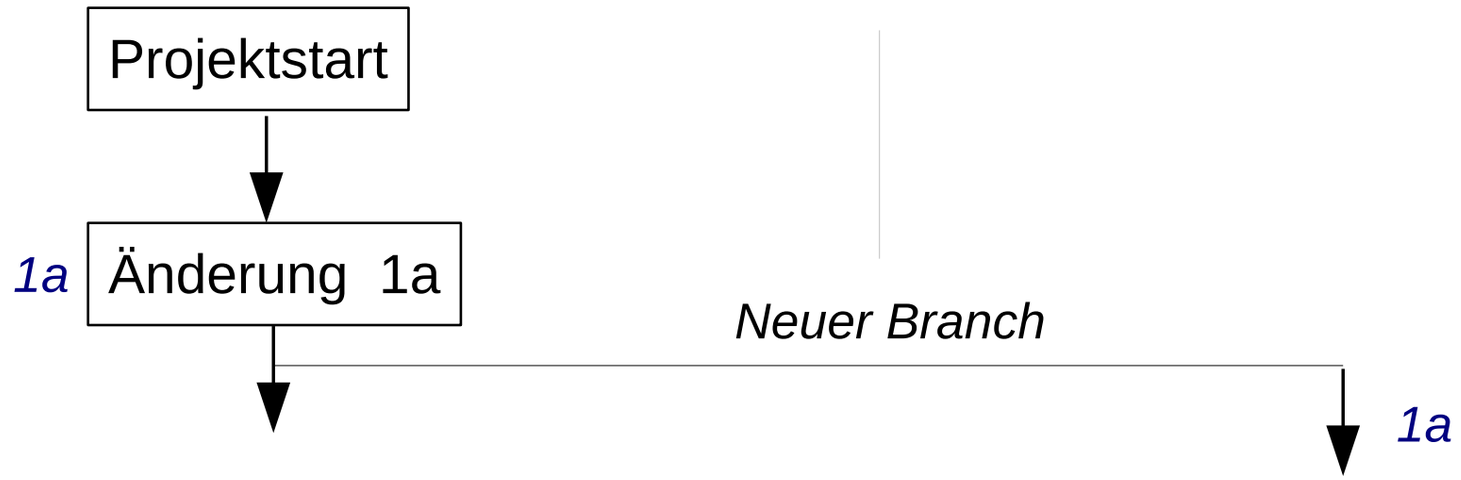


Vorteile:

- Wie oben.
- Weniger kommunikationsintensiv
- Offizielle Version beim Hauptentwickler



Versionsverwaltung mit Git – Demonstration (#1)




```
mkdir ~/developer1; cd ~/developer1
mkdir hello; cd hello
git init
```

```
emacs hello.f90
git add hello.f90
git commit
```

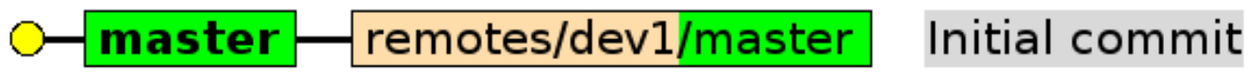
```
program hello
  implicit none
  print *, "Hello"
  ! Silly comment
end program hello
```


```
mkdir ~/developer2
cd ~/developer2
```

Nenne source dev1 (statt origin)

```
git clone -o dev1 ../developer1/hello
cd hello
cat hello.f90
git log
git remote -v
  dev1    ../developer1/hello/ (fetch)
  dev1    ../developer1/hello/ (push)
```

qgit, gitk, ...

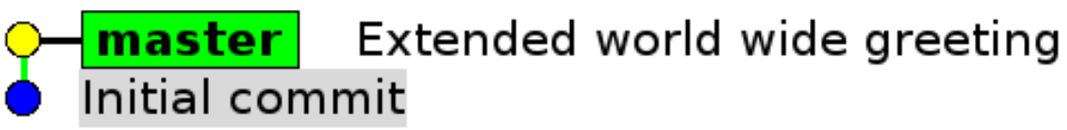


Versionsverwaltung mit Git – Demonstration (#2)

1a+1b Änderung 1b

```
emacs hello.f90
git status
git diff
git add -u
git commit
git log
```

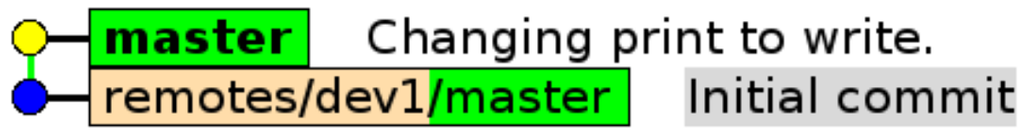
```
program hellow orld
  implicit none
  print *, "Hello World!"
  ! Silly comment
end program hellow orld
```



Änderung 2a *1a+2a*

```
emacs hello.f90
git st
git diff
git add -u
git commit
git log
```

```
program Hello
  implicit none
  write(*,*) "Hello"
  ! Silly comment
end program Hello
```

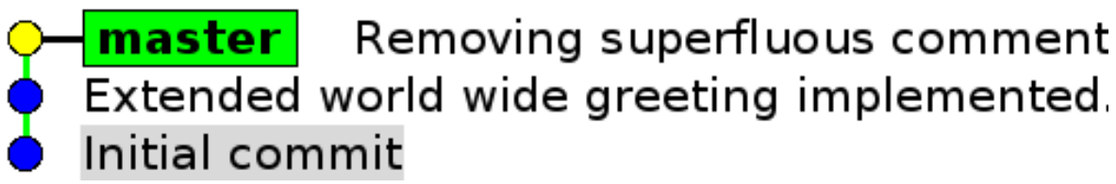


Versionsverwaltung mit Git – Demonstration (#3)

1a+1b+1c Änderung 1c

```
emacs hello.f90
git add -u
git st
git ci
git log
```

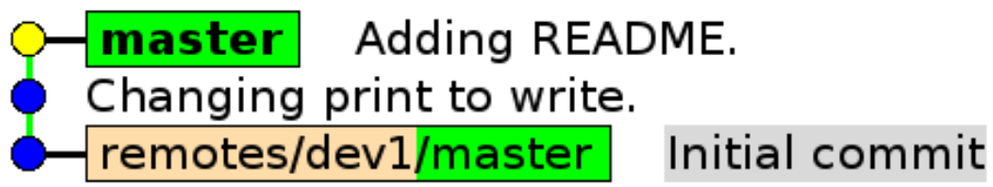
```
program helloworld
  implicit none
  print *, "Hello World!"
end program helloworld
```



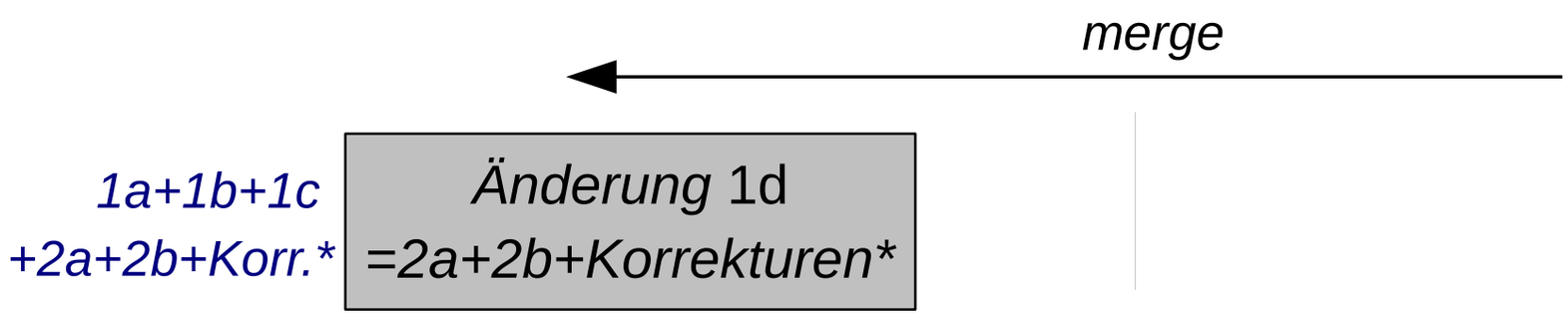
Änderung 2b *1a+2a+2b*

```
emacs README
git st
git add README
git ci
git log
```

```
Compiling:
gfortran hello.f90
```



Versionsverwaltung mit Git – Demonstration (#4)




```
git remote add dev2 ../../developer2/hello
git pull dev2 master
```

← Alle Änderungen von Entwickler 2 übernehmen

```
From ../../developer2/hello
 * branch            master     -> FETCH_HEAD
Auto-merging hello.f90
CONFLICT (content): Merge conflict in hello.f90
Automatic merge failed; fix conflicts and then commit the result.
```

Problem: hello.f90 hat sich seit dem Klonen sowohl bei Entwikckler1 als auch bei Entwickler2 geändert!

↓

Git versucht die beiden Versionen ineinander zu führen (**merge**).

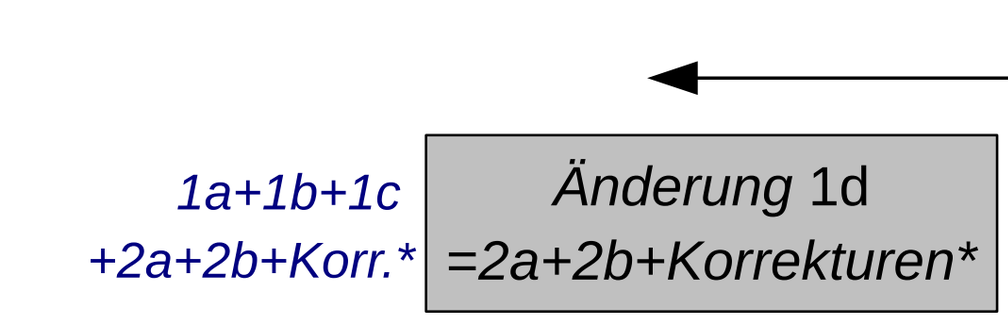
↓

Fehlgeschlagen, da die selben Zeilen parallel geändert wurden!

↓

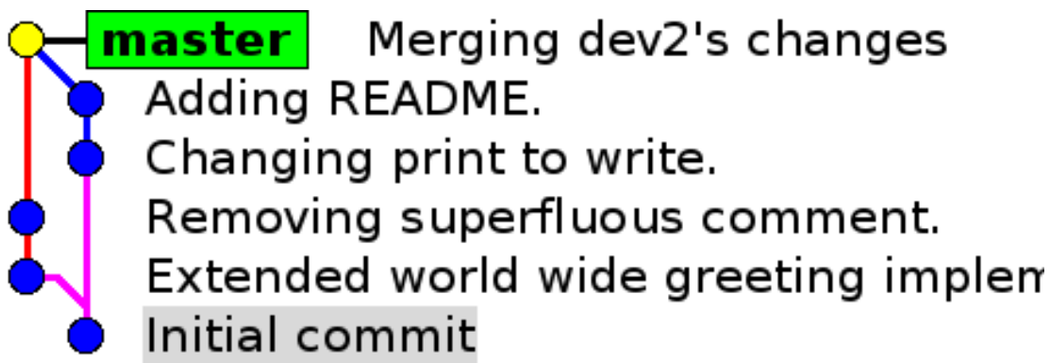
Konflikt muss per Hand aufgelöst werden.

Versionsverwaltung mit Git – Demonstration (#5)




```
emacs hello_world.f90
git status
# :
#  new file:   README
#
# Unmerged paths:
#   (use "git add/rm <file>..."...
#
#  both modified:      hello.f90
git add hello.f90
git commit -m “Merging dev2's changes”
```

README automatisch übernommen



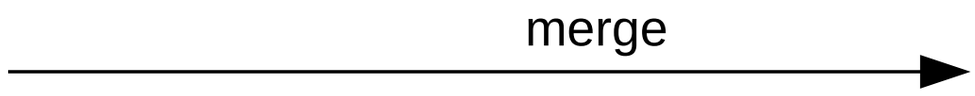

```
program helloworld
  implicit none
<<<<<<< HEAD
  print *, "Hello World!"
end program helloworld
=======
  write(*,*) "Hello"
  ! Silly comment
end program hello
>>>>>>> 24859aa039747d8c650...
```

lokale Version

Version von Entwickler2

```
program helloworld
  implicit none
  write(*,*) "Hello World!"
end program helloworld
```

Versionsverwaltung mit Git – Demonstration (#6)

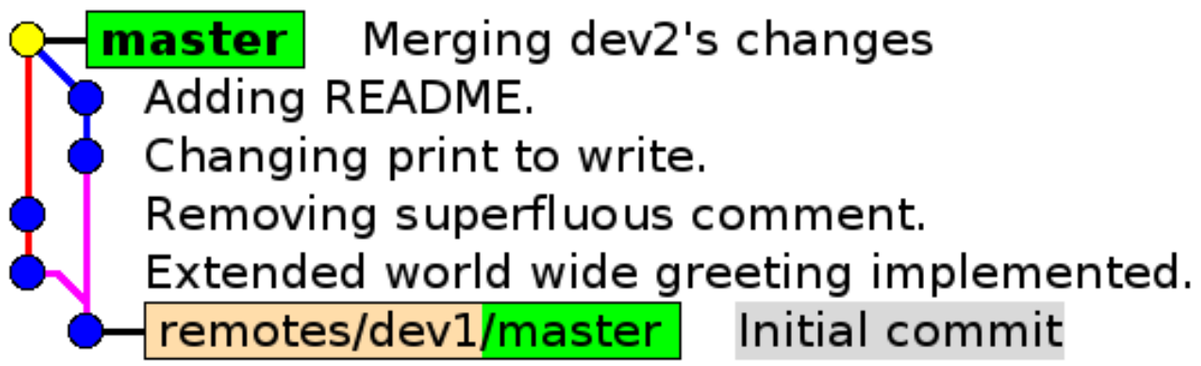


Änderung 2d
=1b+1c
+(1d-2a-2b)

1a+2a+2b
*+1b+1c+Korr.**

Zusammenführen klappte automatisch

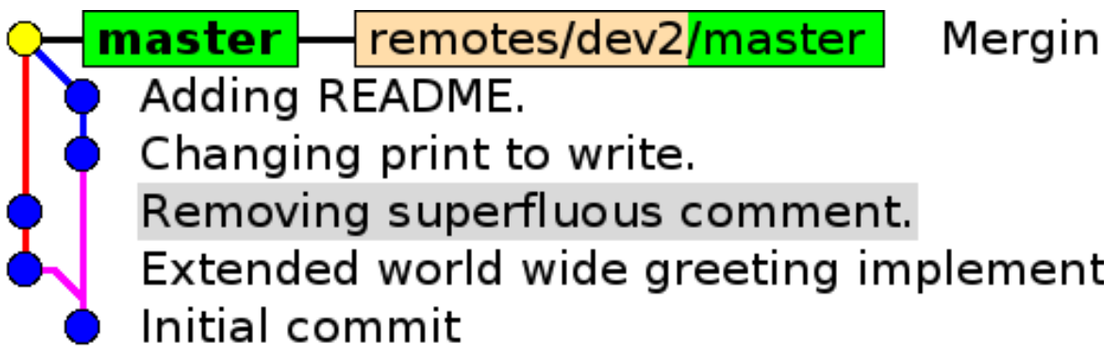
```
git pull dev1 master
:
Unpacking objects: 100% (9/9), done.
From /mnt/local/home/aradi/...
 * branch            master     -> FETCH_HEAD
Updating 24859aa..e17bbd7
Fast-forward
 hello.f90 |    5 ++---
 1 file changed, 2 insertions(+), 3 deletions(-)
```



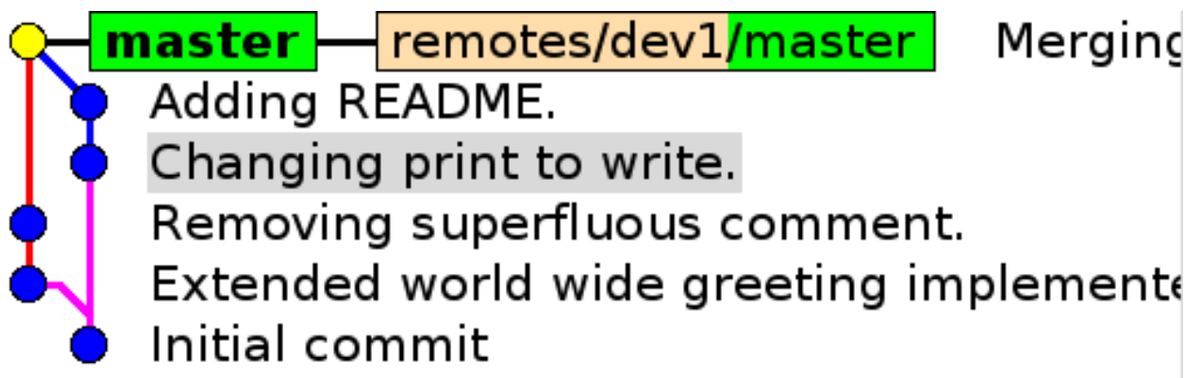
Versionsverwaltung mit Git – Demonstration (#7)

Zweige synchronisiert

```
cat hello.f90
program helloworld
  implicit none
  write(*,*) "Hello World!"
end program helloworld

cat README
Compiling:
gfortran hello.f90

git pull dev2 master
From ../../developer2/hello
:
Already up-to-date.
```




```
cat hello.f90
program helloworld
  implicit none
  write(*,*) "Hello World!"
end program helloworld

cat README
Compiling:
gfortran hello.f90

git pull dev1 master
From ../developer1/hello
:
Already up-to-date.
```



Allgemeine Bemerkungen

Allgemeiner Workflow:

- Entw1: Projekt starten, entwickeln, Repository veröffentlichen
- Entw2: Branch vom veröffentlichten Repository machen, eigene Entwicklungen machen:

:

- Entw: Eigenes Repository veröffentlichen, aus veröffentlichten Repositories anderer Entwickler neue Veränderungen übernehmen.

Veröffentlichung von Repositories (unter anderem):

- Repository im lokalen Dateisystem lesbar machen.
- Ins Netz stellen
  - Eigene Webseite
  - Öffentliche Webseiten mit Git-Hosting, z.B. http://github.com, http://bitbucket.org
- Das gesammte Repository per Mail schicken
- Repository per git-Server verfügbar machen

Kommunikation via Mail / USB-Stick

- Ablauf wie bis jetzt, vor clone und merge muss das Repository jedoch zusammengepackt und verschickt bzw. empfangen (übertragen) und ausgepackt werden.
- Z.B.: Entwickler 2 will aktuelle Änderungen von Entwickler 1 übernehmen:

Entwickler 1 (Maschine 1)

```
cd ~/mywork
tar -c -v -z -f hello.dev1.tgz hello
```

Mailprogramm (verschickt hello.dev1.tgz)

Entwickler 2 (Maschine 2)

Mailprogramm (speichert hello.dev1.tgz)

```
cd ~/dev1work
tar -x -v -z -f hello.dev1.tgz
cd ~/mywork/
git clone -o dev1 ../dev1work/hello
```

tar -c -v -z -f hello.dev1.tgz hello

Mailprogramm (verschickt hello.dev1.tgz)

Mailprogramm (speichert hello.dev1.tgz)

```
cd ~/dev1work
tar -x -v -z -f hello.dev1.tgz
cd ~/mywork/
git pull dev1 master
```

Veröffentlichung via git-daemon

Wenn ein Entwickler sein Repository veröffentlichen wird, startet er einen git-Daemon:

```
cd ~/work
git daemon --reuseaddr --export-all
  --base-path=/home/aradi/work /home/aradi/work/hello
```

git-Repository

Pfad wird bei Anfragen automatisch mit diesem Präfix ergänzt

Solange der Daemon läuft, können andere Entwickler das Repository lesen:

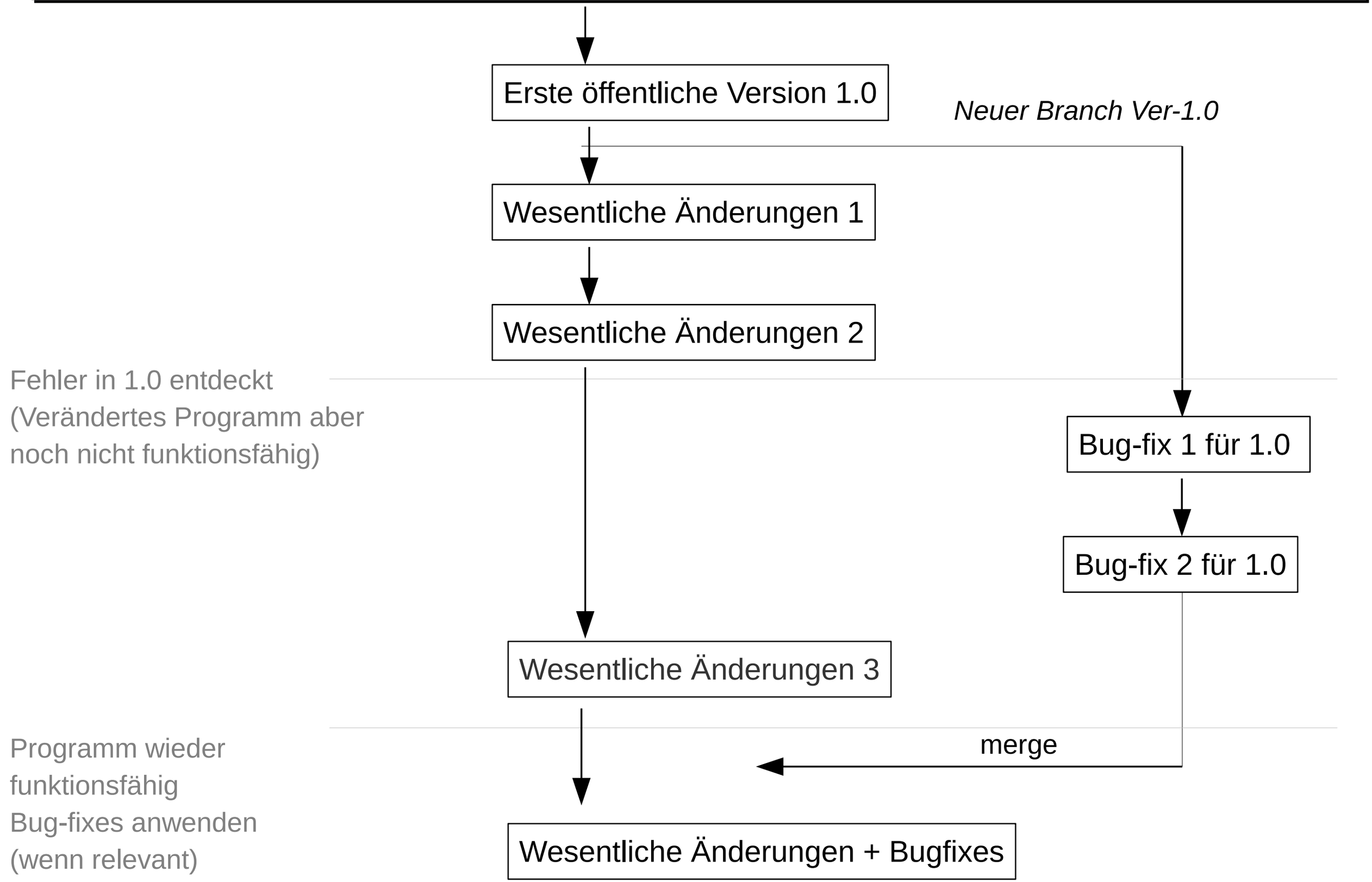
```
cd ~/devel/mywork/
git clone -o dev1 git://servermaschine/hello
cd hello
:
git pull
```

Pfad bezüglich base-path

Name oder IP-Addresse des Computers, auf dem der Git-Daemon läuft.

Vorsicht! Der Git-Daemon authentifiziert die Clients nicht, sodass jeder Teilnehmer im Netzwerk Lesezugriff auf das Repository hat, solange dieses via Git-Daemon exportiert wird.

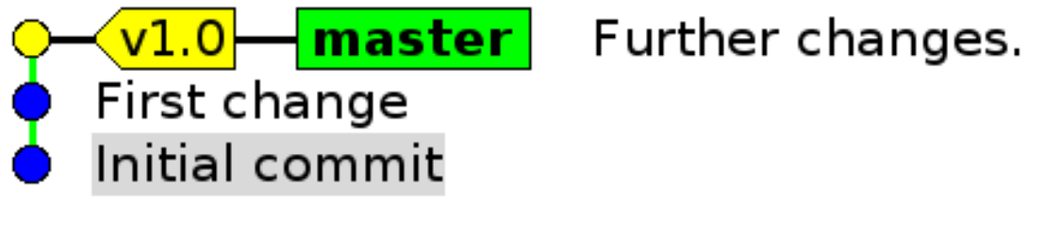
Zweige für Einzelentwickler



Zweige für Einzelentwickler (#2)

Erste öffentliche Version 1.0

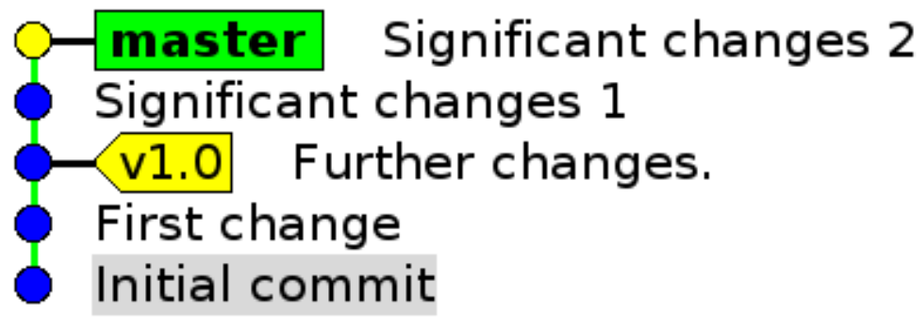
```
git init
git add …
git commit
:
git commit
git tag -a v1.0 -m “Tagging version 1.0”
```



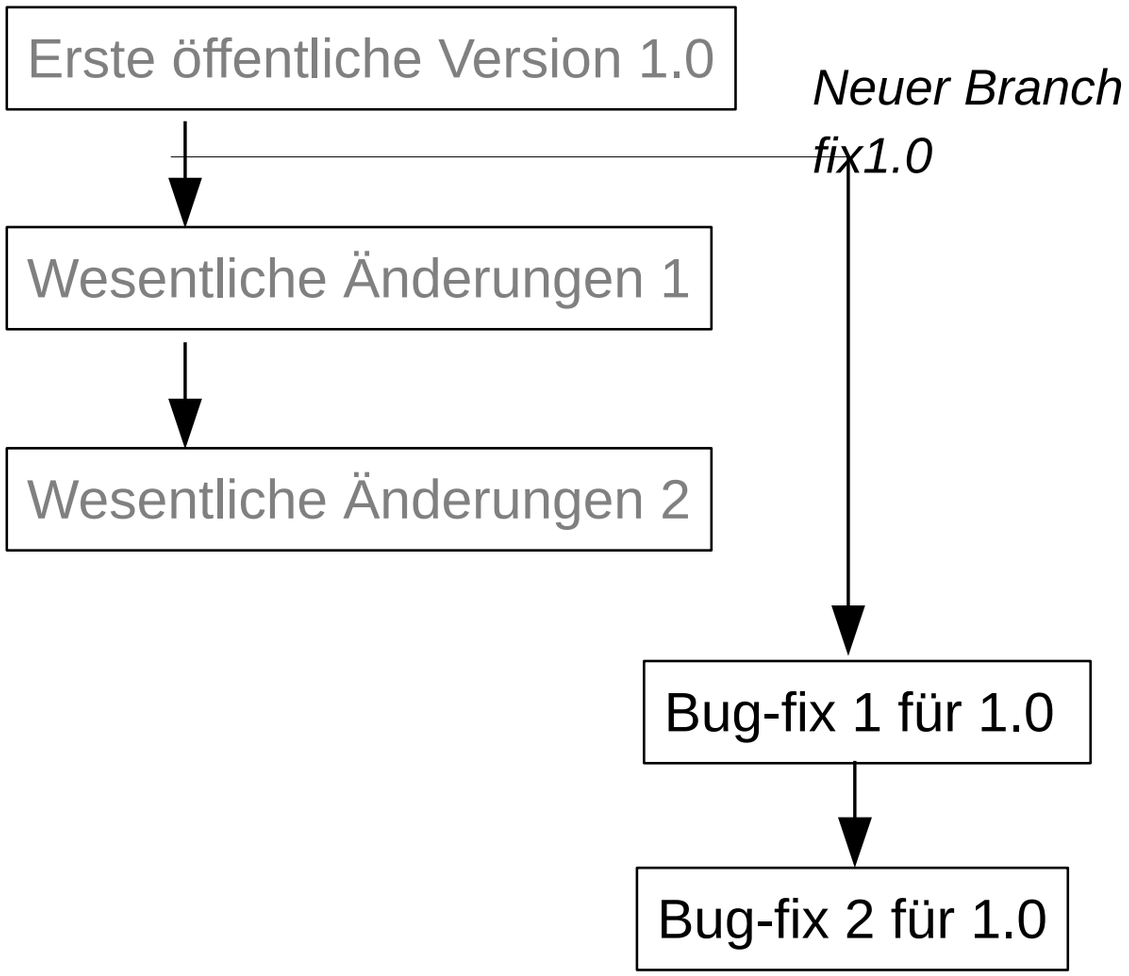
Wesentliche Änderungen 1

Wesentliche Änderungen 2

git commit
:
git commit



Zweige für Einzelentwickler (#3)

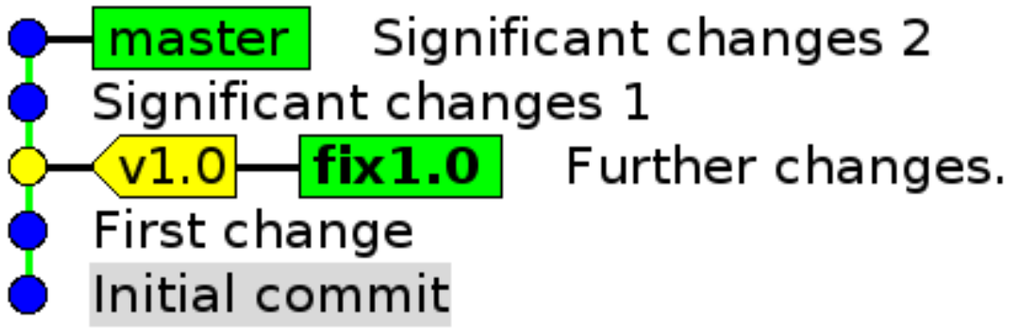



```
git branch
* master
git branch fix1.0 v1.0
git checkout fix1.0
Switched to branch 'fix1.0'
git branch
* fix1.0
  master
```

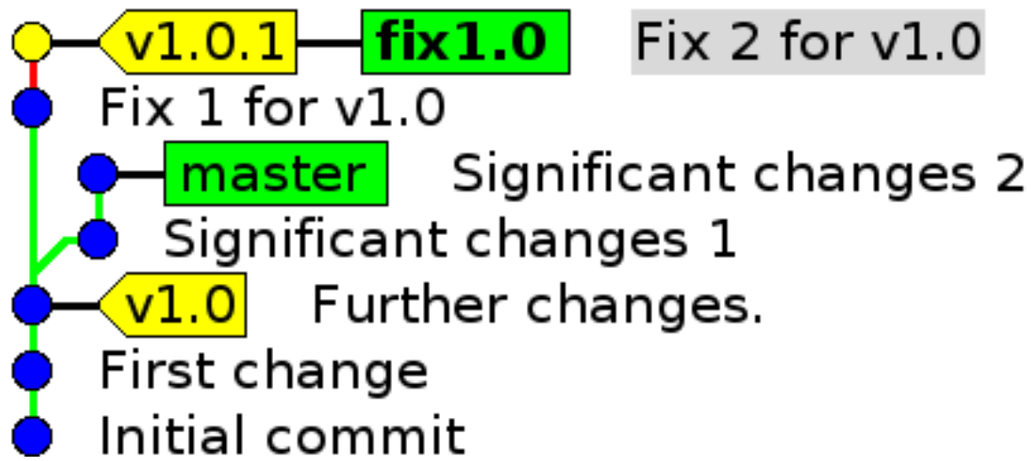
Branchname

Abzweigungspunkt (RevID oder Tag)

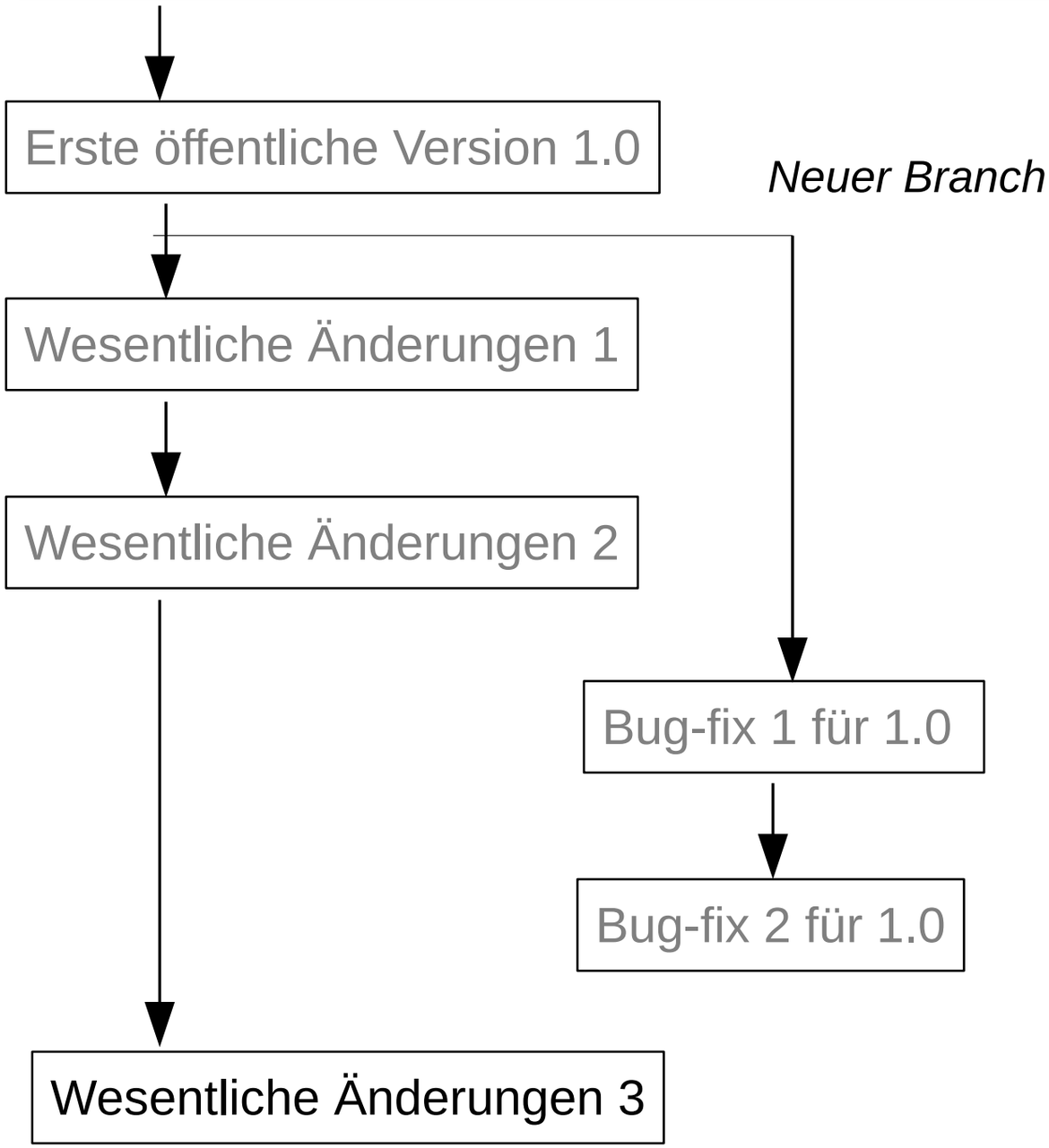
Arbeitskopie wie in Ver. 1.0




```
git commit -m “Fix 1 for v1.0“
git commit -m “Fix 2 for v1.0“
git tag -a v1.0.1 -m “Release 1.0.1”
```

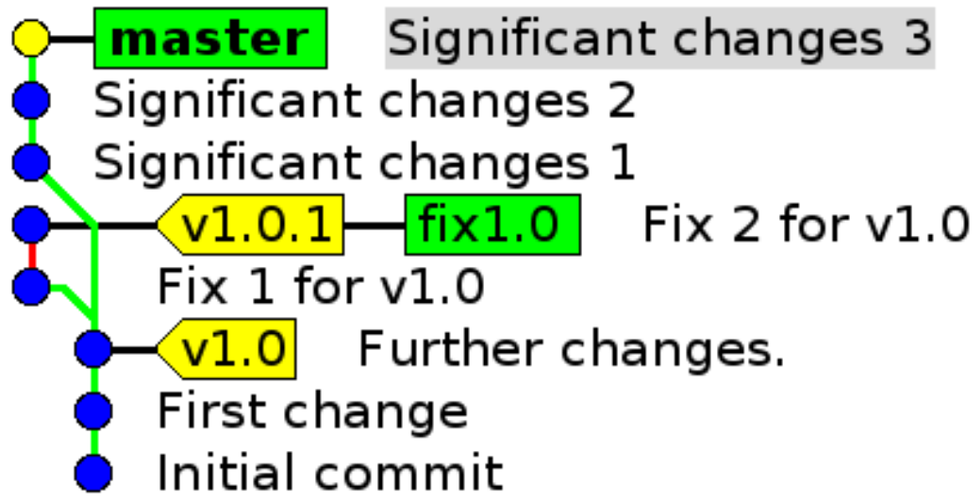


Zweige für Einzelentwickler (#4)

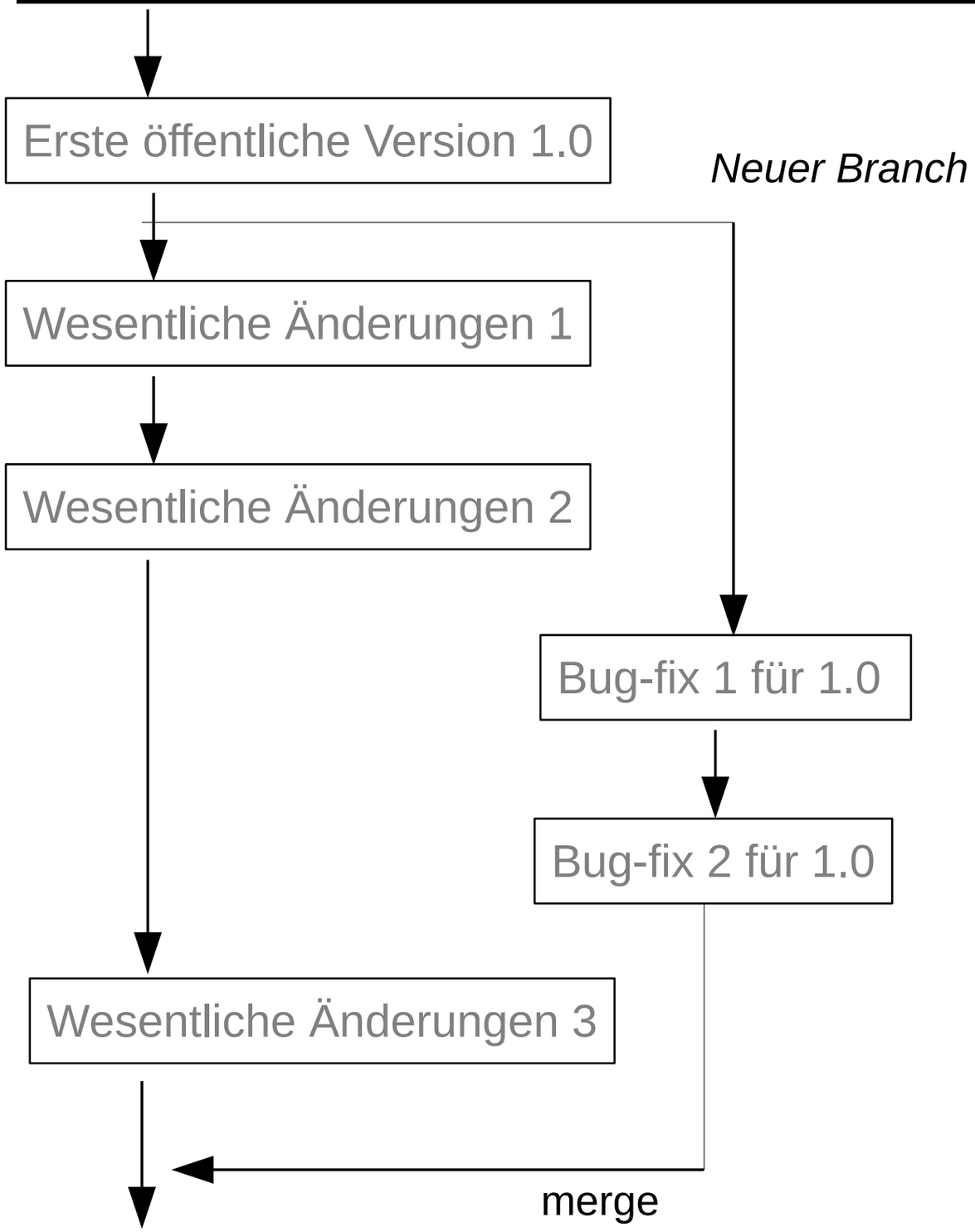



```
git checkout master
Switched to branch 'master'
git branch
  fix1.0
* master
:
git ci -m “Significant changes 3”
```

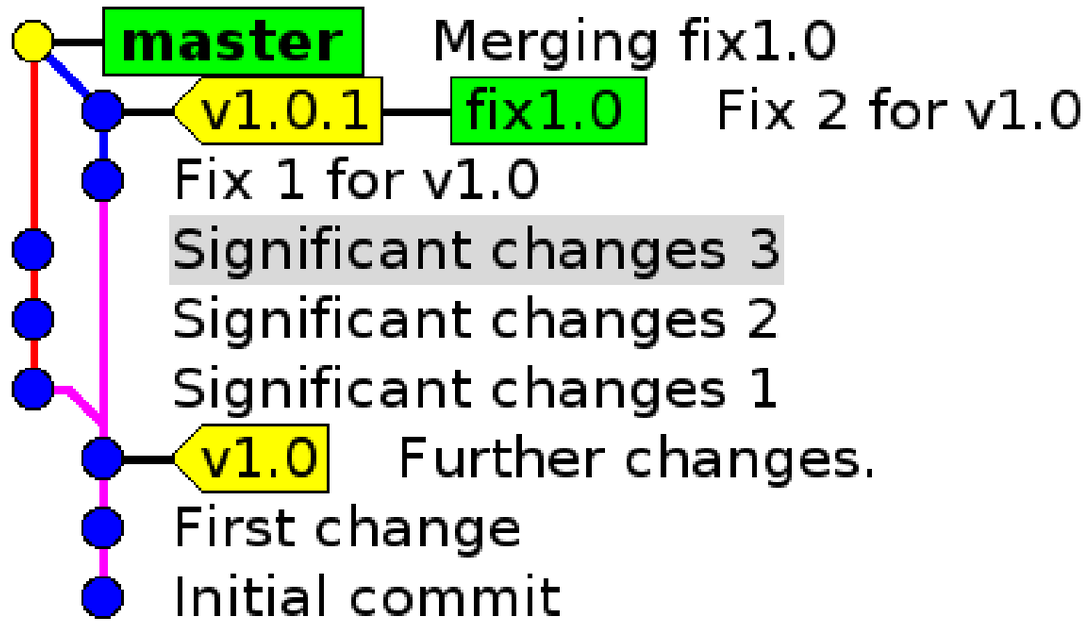
Arbeitskopie wie im master



Zweige für Einzelentwickler (#5)

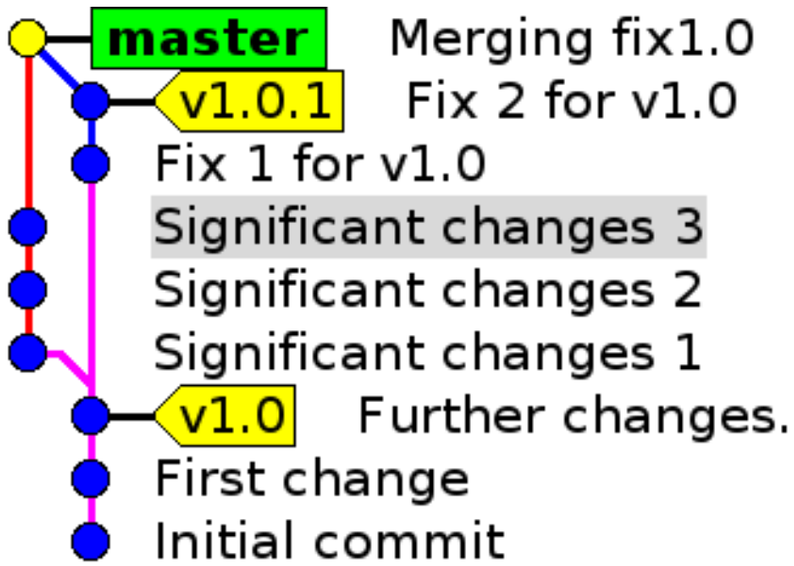



```
git merge fix1.0
Auto-merging test.dat
CONFLICT (content): Merge conflict
in test.dat
git add test.dat
git commit -m “Merging fix1.0”
```



Zweige für Einzelentwickler (#6)

- Zweige, die man nicht mehr benutzt, können bei Bedarf gelöscht werden. Wenn der Zweig in einen anderen Zweig übeführt wurde, bleiben die entsprechenden Commits in der Geschichte erhalten:

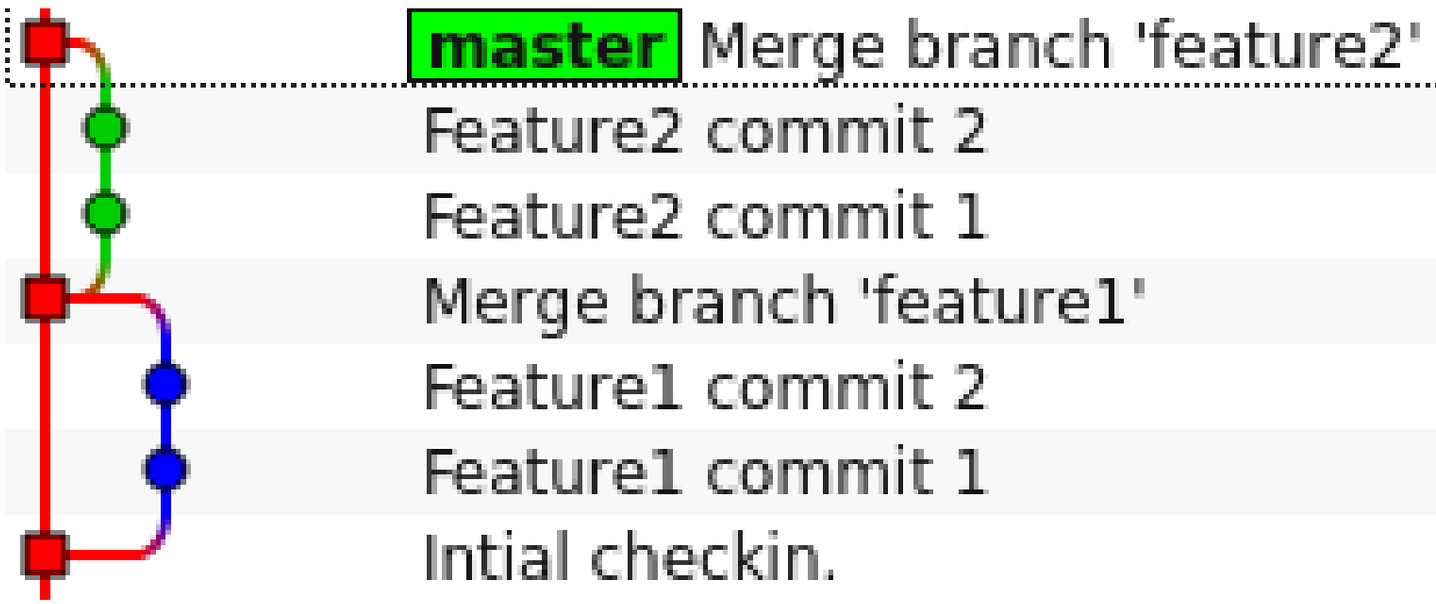



```
git branch -d fix1.0
Deleted branch fix1.0 (was 7ab2988).
```

Allgemeiner Workflow bei größeren Projekten

- Jedes größeres Projekt hat seinen allgemeinen Git-Branching-Workflow
- Sehr oft wird neue Funktionalität in Branches entwickelt, und erst wenn sie stabil ist, in die Hauptversion (master) übernommen:

```
git branch feature1
git checkout feature1
git commit
:
git commit

git checkout master
git merge --no-ff feature1
git branch -d feature1

git checkout -b feature2
git commit
:
git commit

git checkout master
git merge --no-ff feature2
git branch -d feature2
```




```
git branch feature2
git checkout feature2
```

Mache Merge explizit sichtbar

Siehe z.B.
http://nvie.com/posts/a-successful-git-branching-model/
für ein sehr praktisches Git-Workflow für Projekte kleiner und mittlerer Größe

Aufgabe 0

- Machen Sie die einzelnen Schritte der Git-Demonstration nach, um den Umgang mit git zu üben!

Aufgabe 1 (#1)

Zwei Entwickler (oder ein Entwickler in zwei separaten Zweigen) sollen das Gauss-Eliminationsprogramm (in getrennten Zweigen) mit folgender Funktionalität erweitern:

<u>Entwickler 1:</u>

- Im io-Module sollen statt *writeoutput()* zwei neue Routinen erstellt werden:
  - *writetoscreen*: Übernimmt den Lösungsvektor des Gleichungssystems und gibt diesen auf den Bildschirm aus (wie *writeoutput()* bis jetzt).
  - *writetofile*: Übernimmt den Lösungsvektor des Gleichungssystems und schreibt diesen in die Datei namens '*gauss.out*' aus. Es sollen nur die Zahlen des Vektors in die Datei ausgegeben werden (kein erklärender Text), allerdings mit allen signifikanten Stellen (z.B. Format ES23.15)
- Die Einleseroutine im Input-Modul soll so modifiziert werden, dass diese nach dem Einlesen der Daten (Matrix und rechte Seite) noch einen logischen Wert einliest (T/F). Ist dieser .true., (und nur dann) soll das Ergebnis in einer Datei gespeichert werden, ansonsten auf das Bildschirm ausgegeben werden.
- Die Ausgabe des Ergebnisses soll auschließlich über die Routinen *writetoscreen()* bzw. *writetofile()* erfolgen. (Keine write-Befehle sonst im Hauptprogramm).
- Makefile, Beispielinput (und evtl. README) soll entsprechend geändert werden.

Aufgabe 1 (#2)

Entwickler 2:

- Das Programm sollte so erweitert werden, dass es beliebige Anzahl von b-Vektoren einlesen kann. Das Gleichungssystem sollte für alle b-Vektoren gleichzeitig gelöst werden.
- Die Einleseroutine sollte so modifiziert werden, dass diese nach dem Einlesen der Matrix A zuerst die Anzahl der b-Vektoren und dann die entsprechende Anzahl von b-Vektoren einliest (zeilenweise: jede Zeile ein b-Vektor).
- Bildschirmausgabe soll entsprechend geändert werden.
- Beispielinput (und evtl. README) sollen entsprechend geändert werden.

Beispielinput:

```
3
2.0 4.0 4.0
1.0 2.0 -1.0
5.0 4.0 2.0
2
1.0 2.0 4.0
2.0 4.0 8.0
```

Aufgabe 1 (#3)

Zusammenführen der Zweige:

- Nachdem die Erweiterungen in den separaten Zweigen **vollständig** implementiert wurden (die Programme mit der implementierten Funktionalität funktionieren einwandfrei, die Dokumentation, Makefiles etc. wurden angepasst), sollten **die beiden Zweige** auf den gleichen Stand gebracht werden.
  (z.B.: Entw. 1 aktualisiert seinen Branch mit dem von Entw. 2, löst eventuelle Konflikte, macht die nötigen Änderungen damit seine Version beide Erweiterungen sinnvoll enthält und checkt ein. Danach aktualisiert Entw. 2 seinen Branch mit dem von Entw. 1, löst eventuelle Konflikte, checkt ein.)