

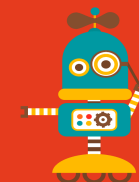


# git history & branches

DI Reinhold Buchinger

Creative Commons-Lizenz CC BY-NC-SA 4.0 AT.

Höhere Abteilung für Mechatronik  
Höhere Abteilung für Informationstechnologie  
Fachschule für Informationstechnik



# Ziele

- » git branches verstehen und anwenden können.
- » git branches zusammenführen und löschen können
- » git "detached head" verstehen

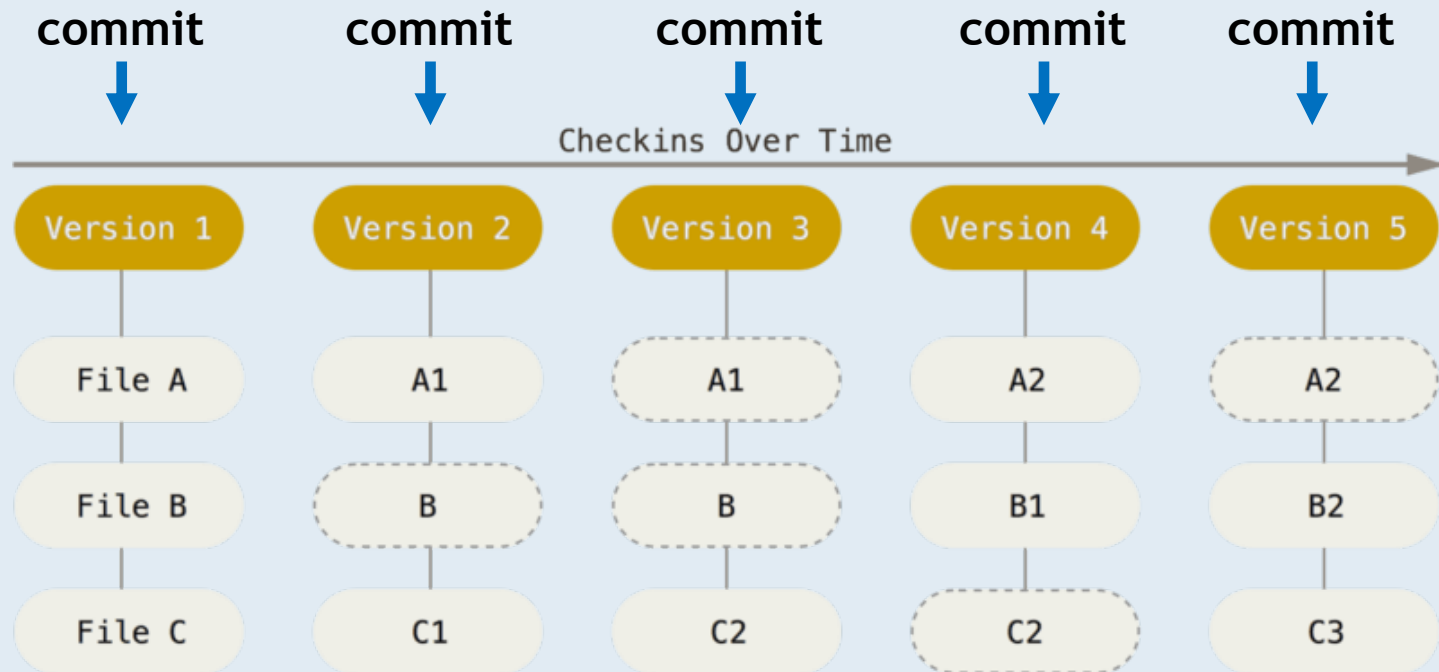
# Vorbereitung

- » Lege ein neues leeres Repository an.
- » Erstelle eine Textdatei mit beliebigen Inhalt.
- » Add & commit diese Datei.
- » Ändere die Datei.
- » Add & commit diese Datei.

# Commit-Historie

# Commits

» Bei jedem Commit speichert git ein Abbild<sup>1</sup> von allen Dateien wie sie gerade aussehen und speichert einen Verweis zu diesen Snapshot.



<sup>1</sup> Um Speicherplatz zu sparen wird für unveränderte Files nur ein Link zum bereits vorhandenen unveränderten File gespeichert.

# Commit-History

» Können uns die Liste von Commits mit dem Befehl `git log` ansehen.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

commit 085bb3bcb608ele8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test
```

SHA-1 Prüfsumme

Commit Nachricht

# git log

- » Der `git log` Befehl ist sehr mächtig und hat viele mögliche Optionen.
- » Beispiele:
  - » `git log -p` → zeigt die Änderungen (Diff) an
  - » `git log --stat` → ein paar Statistiken zu jedem Commit
  - » `git log -2` → zeigt nur die letzten beiden Commits an
  - » `git log --since=2.weeks` → Commits der letzten beiden Wochen
  - » ...
- » Siehe <https://www.git-scm.com/docs/git-log>

# Probiere es selbst aus!





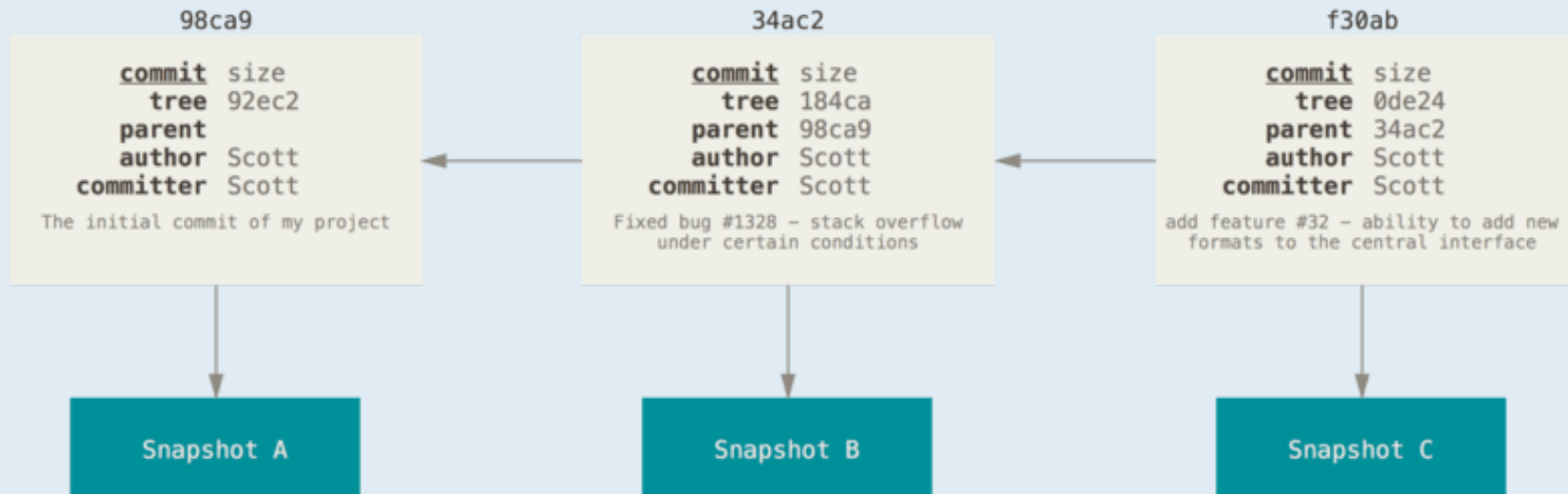
# Branch

# Problemstellung

- » Ein Team möchte parallel an unterschiedlichen Features arbeiten, ohne sich ständig in die Quere zu kommen.
- » Man entwickelt ein neues Feature und ist mitten im Entwickeln. Das Programm ist gerade nicht lauffähig. Plötzlich meldet sich der Kunde mit einem kritischen Bug, der dringend behoben werden muss.
- » Man hat eine Idee, wie eine Lösung evt. verbessert werden kann. Man will es gerne ausprobieren, sich aber den aktuellen Status nicht zerstören.
- » ...

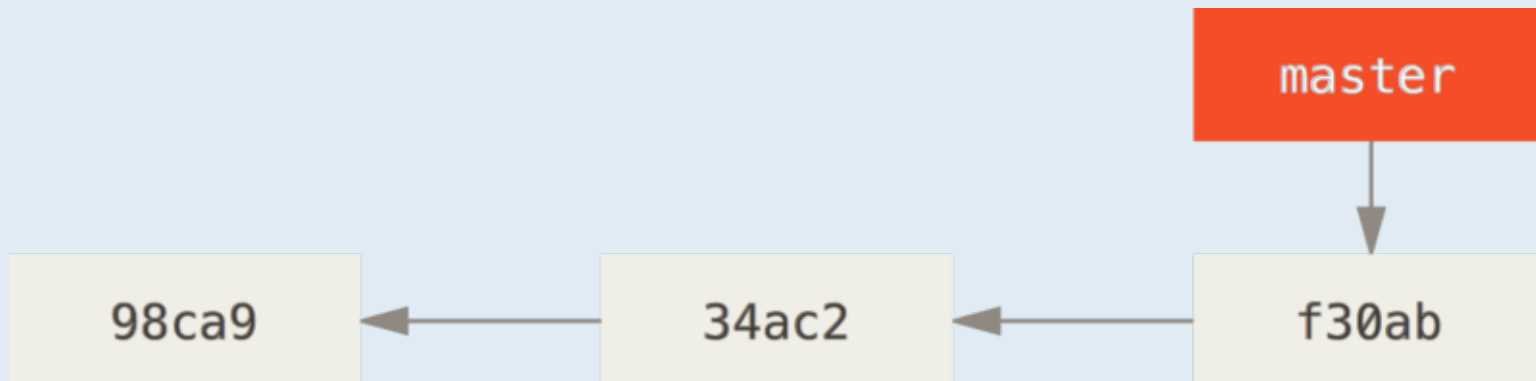
# Commit & Parents

» Jeder Commit (bis auf den ersten) speichert auch einen Pointer, der auf den vorherigen Commit verweist.



# Branch

- » Ein **Branch** ist ein beweglicher Pointer zu einem Commit.
- » Der Name des Default-Branch ist **master**.
- » Mit jedem neuen Commit wandert auch der Pointer weiter.



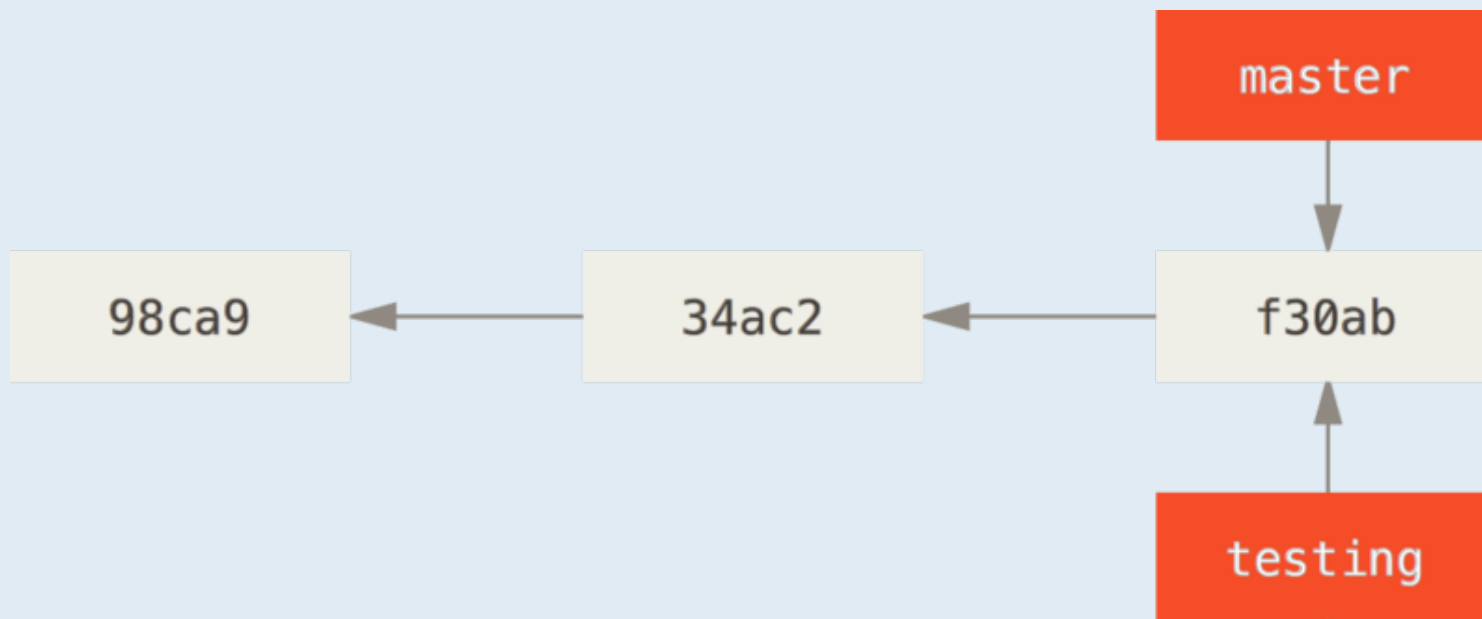
# Anmerkung

» Der Default-Branch heißt aktuell **main** (und nicht mehr **master**). Die Folien müssen noch entsprechend umgebaut werden.

# git branch

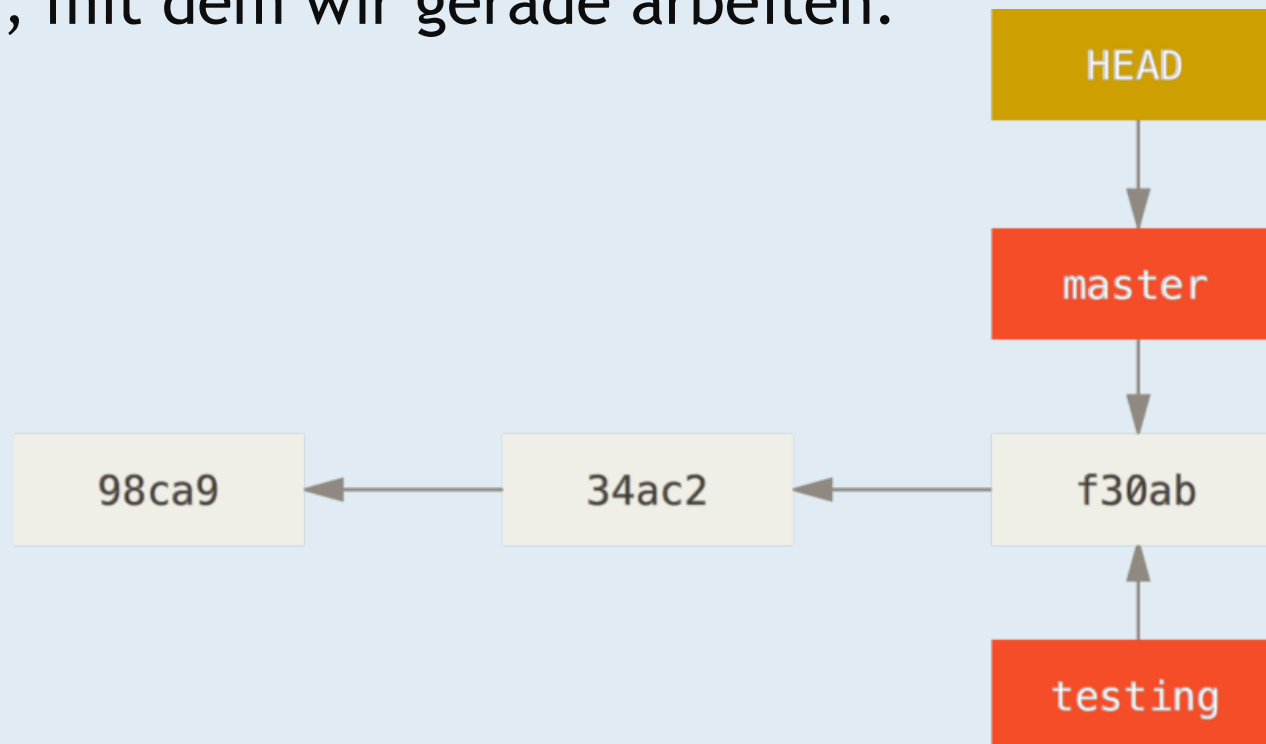
» Mit dem Befehl `git branch` erzeugt man einen neuen Branch ("Pointer").

```
$ git branch testing
```



# HEAD

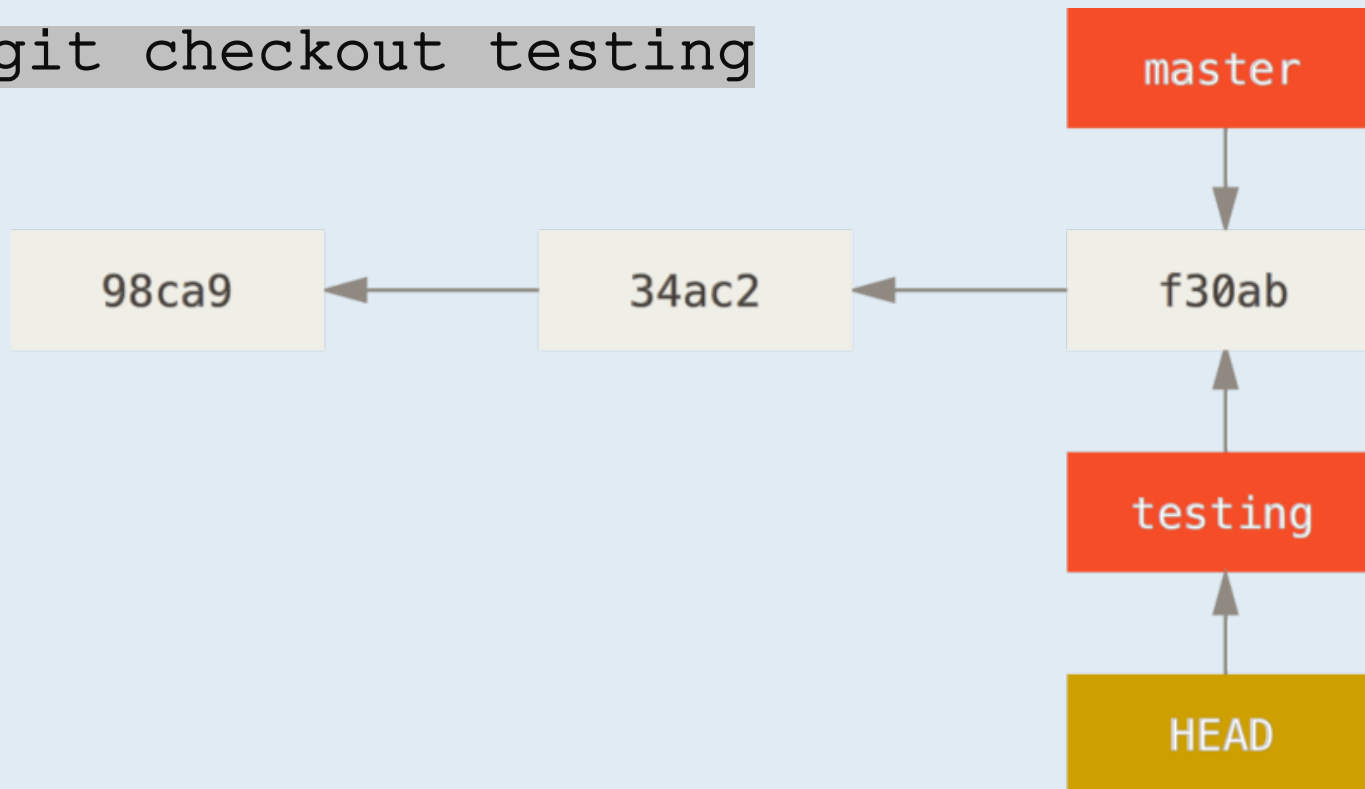
- » Wie weiß git, mit welchem Branch wir gerade arbeiten?
- » Der spezielle Pointer namens "HEAD" zeigt immer auf den Branch, mit dem wir gerade arbeiten.



# Branch wechseln

» Um einen Branch zu wechseln, nutzen wir den Befehl `git checkout`

```
$ git checkout testing
```





# Aktueller Branch

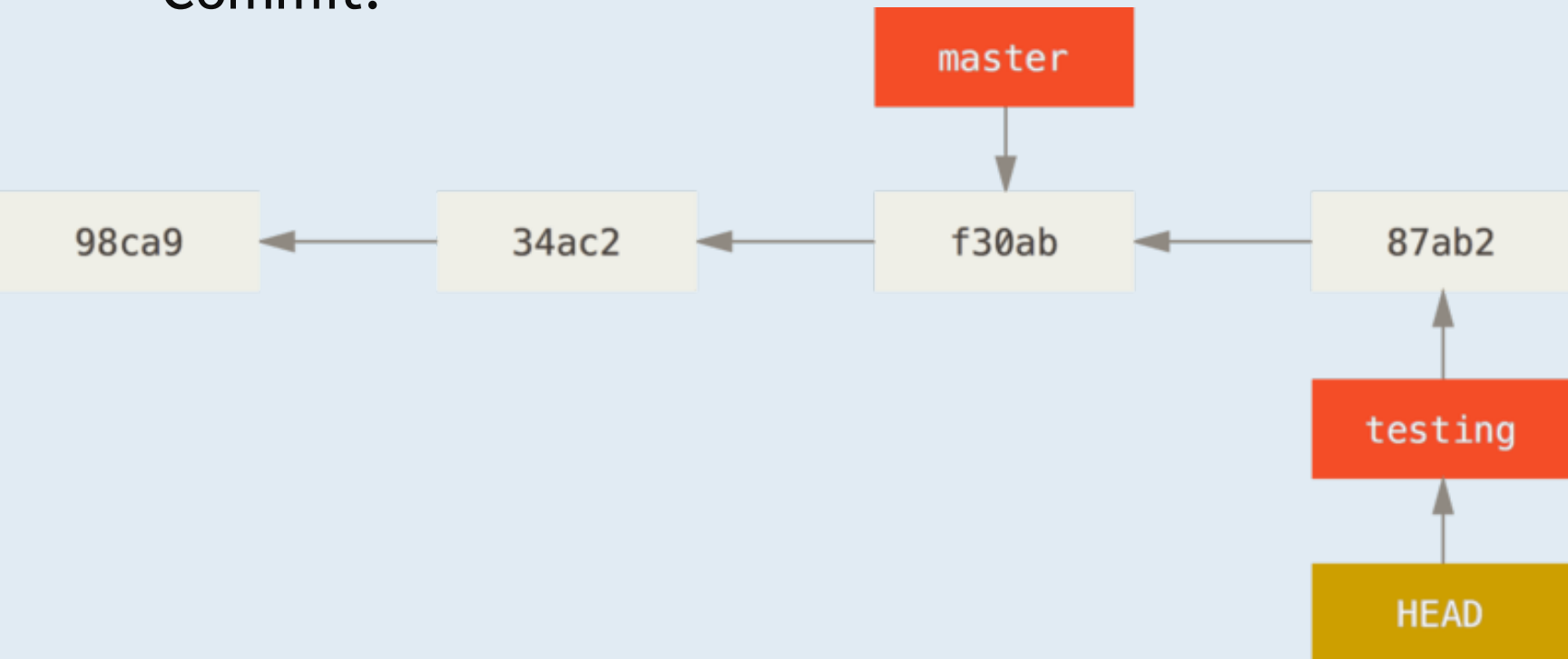
- » `git branch` ohne weitere Angaben listet alle Branches auf.
- » Vor dem aktuellen Branch (worauf HEAD gerade zeigt), steht ein Stern (\*)

```
$ git branch
```

```
master  
* testing
```

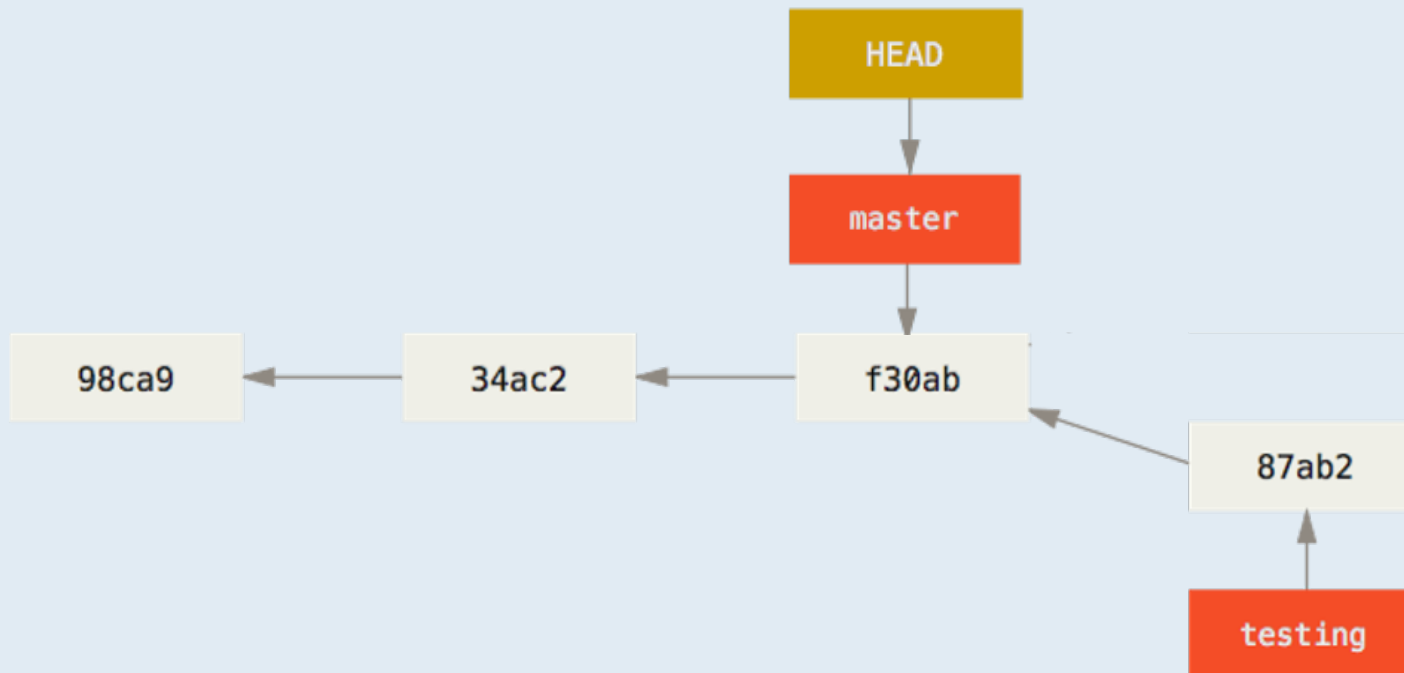
# Branch & Commit

- » Wir arbeiten jetzt am aktuellen Branch weiter und machen ein neues Commit.
- » Der Pointer des aktuellen Branch rückt mit jedem Commit automatisch vor und zeigt immer auf den letzten Commit.



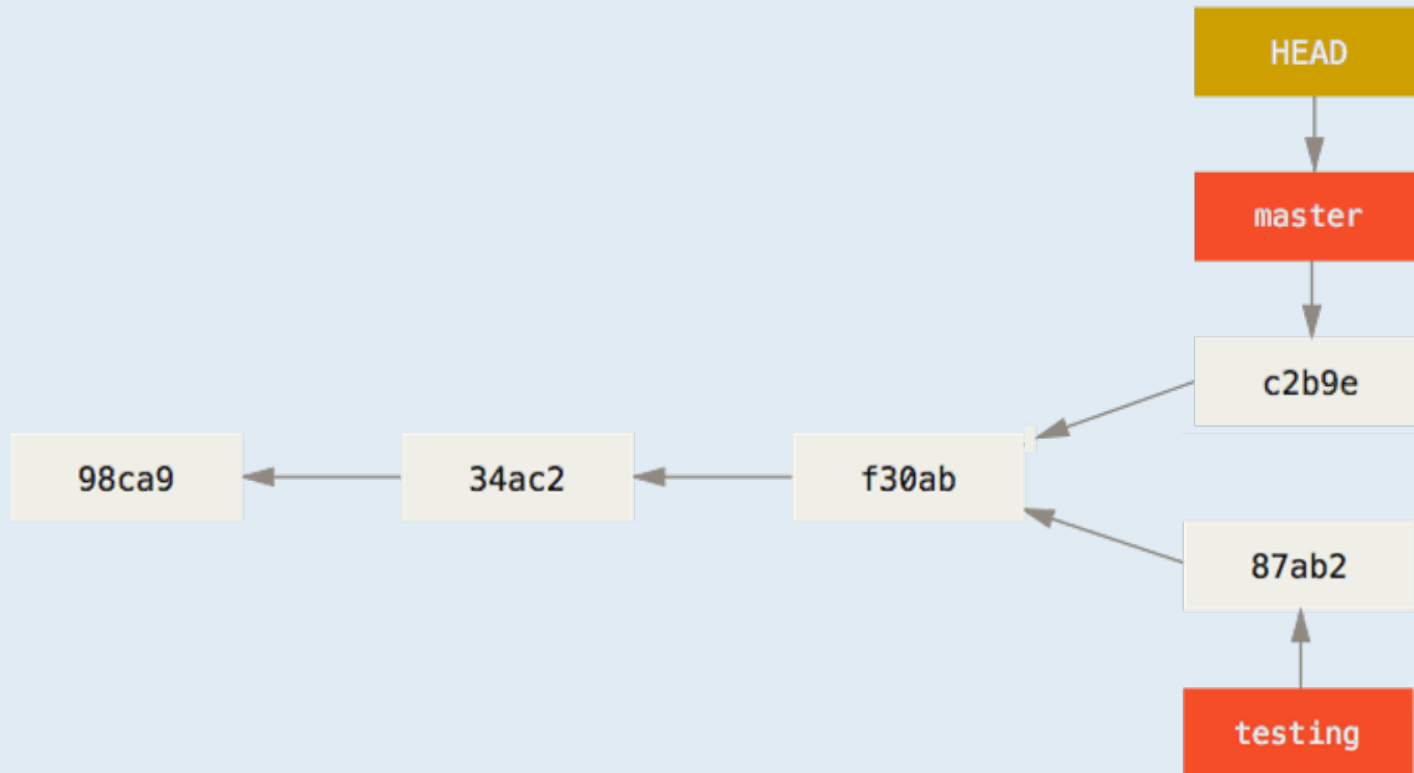
# Divergent history

- » Wir wechseln mittels `git checkout` zurück zum master branch
- » Alle Dateien werden auf den Status des letzten Commits am master (f30ab) zurück gesetzt!



# Divergent history

- » Wir erzeugen am master neue Commits.
- » Dadurch geht unsere git history auseinander und wir erhalten zwei Pfade.



# Divergent history

» Mittels `git log` kann man sich die Abspaltung darstellen lassen.

```
$ git log --decorate --graph --all
```

```
* commit 286a1ccf98ab72bbe245db4fdfeb19cdbe49501b (HEAD -> master)
| Author: Reinhold Buchinger <buc@htl.rennweg.at>
| Date: Mon Jan 11 17:51:27 2021 +0100
|
| update content
|
| * commit a981561975892a4b7f8ae20e1afc5e5d8338dc34 (testing)
|/ Author: Reinhold Buchinger <buc@htl.rennweg.at>
| Date: Mon Jan 11 17:25:38 2021 +0100
|
| added more items
|
| * commit ad11dd8945d54507a0b27b3254f7b4c1e8efe1a1
| Author: Reinhold Buchinger <buc@htl.rennweg.at>
| Date: Tue Dec 15 13:41:51 2020 +0100
|
| write semmel
|
| * commit 46fbc38980a021438302a52e15b2aa1d769945fc (origin/master)
| Author: Reinhold Buchinger <buc@htl.rennweg.at>
| Date: Tue Nov 10 11:27:43 2020 +0100
|
| say hallo,du
```

# Probiere es selbst aus!

- » Erzeuge einen zweiten Branch.
- » Wechsle dorthin und führe mind. ein Commit durch.
- » Wechsle zum master branch zurück und führe dort ebenfalls ein Commit durch.
- » Lass dir die git history mit allen Branches anzeigen.

```
$ git branch ...
```

```
$ git checkout ...
```

```
$ git log --decorate --graph --all
```

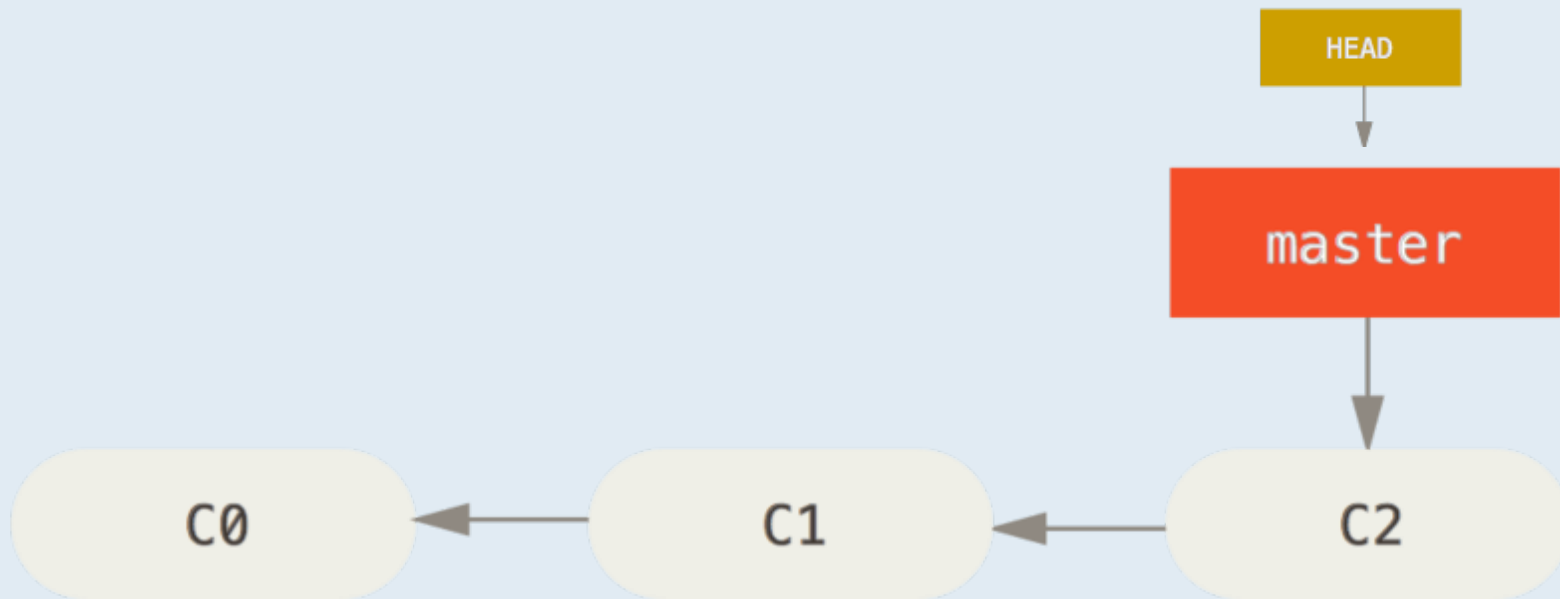
## Tipp

Erzeugt einen Branch und wechselt dorthin in einem Befehl:

```
git checkout -b <newbranchname>.
```

# Ein Beispielszenario

» Du arbeitest an einem Projekt und hast bereits ein paar Commits gemacht.

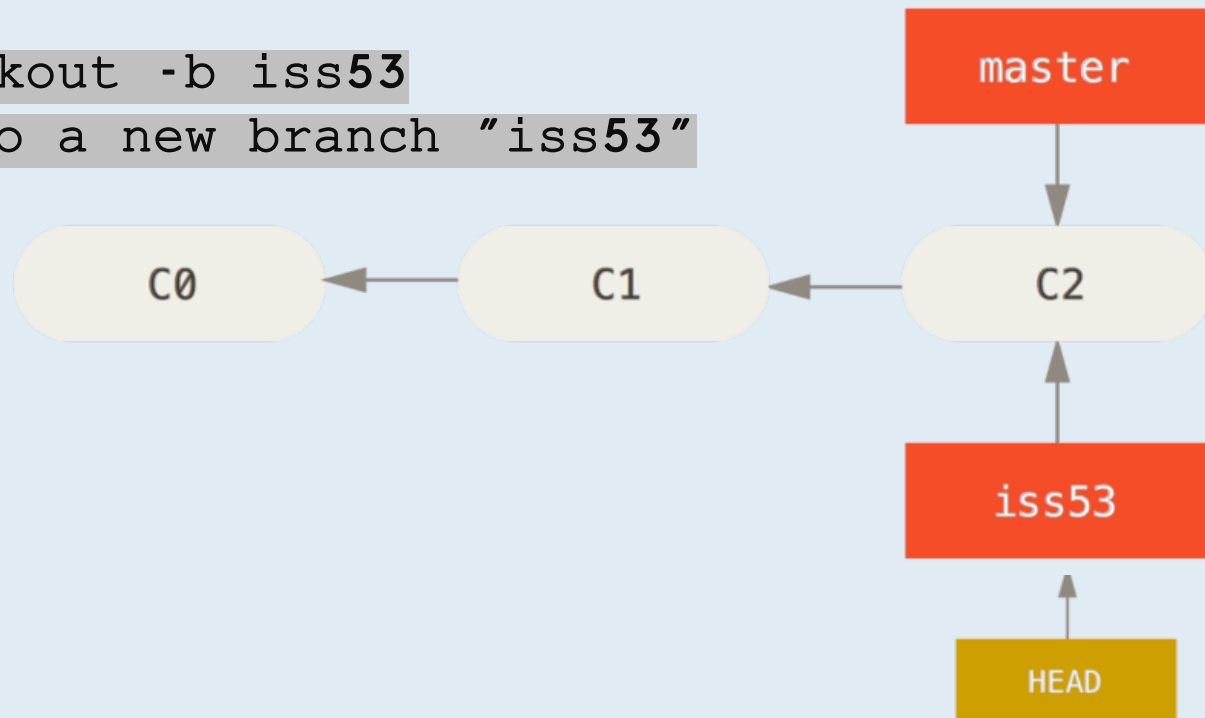




# Ein Beispielszenario

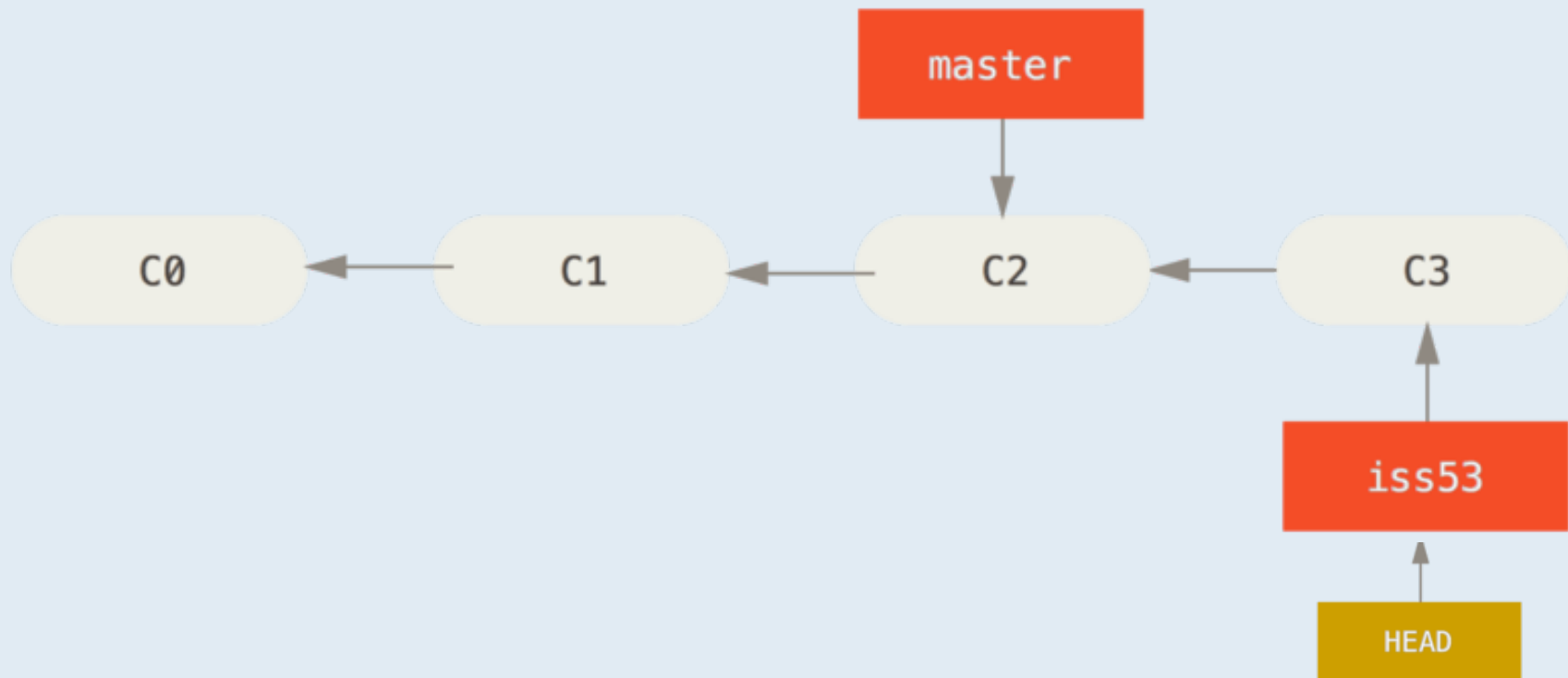
- » Du möchtest als nächstes ein kleines Problem beheben, das vor einiger Zeit gemeldet worden ist. Es ist als "issue #53" in deinem issue-tracking-System abgespeichert.
- » Dazu erzeugst du einen neuen Branch und wechselst dorthin.

```
$ git checkout -b iss53  
Switched to a new branch "iss53"
```



# Ein Beispielszenario

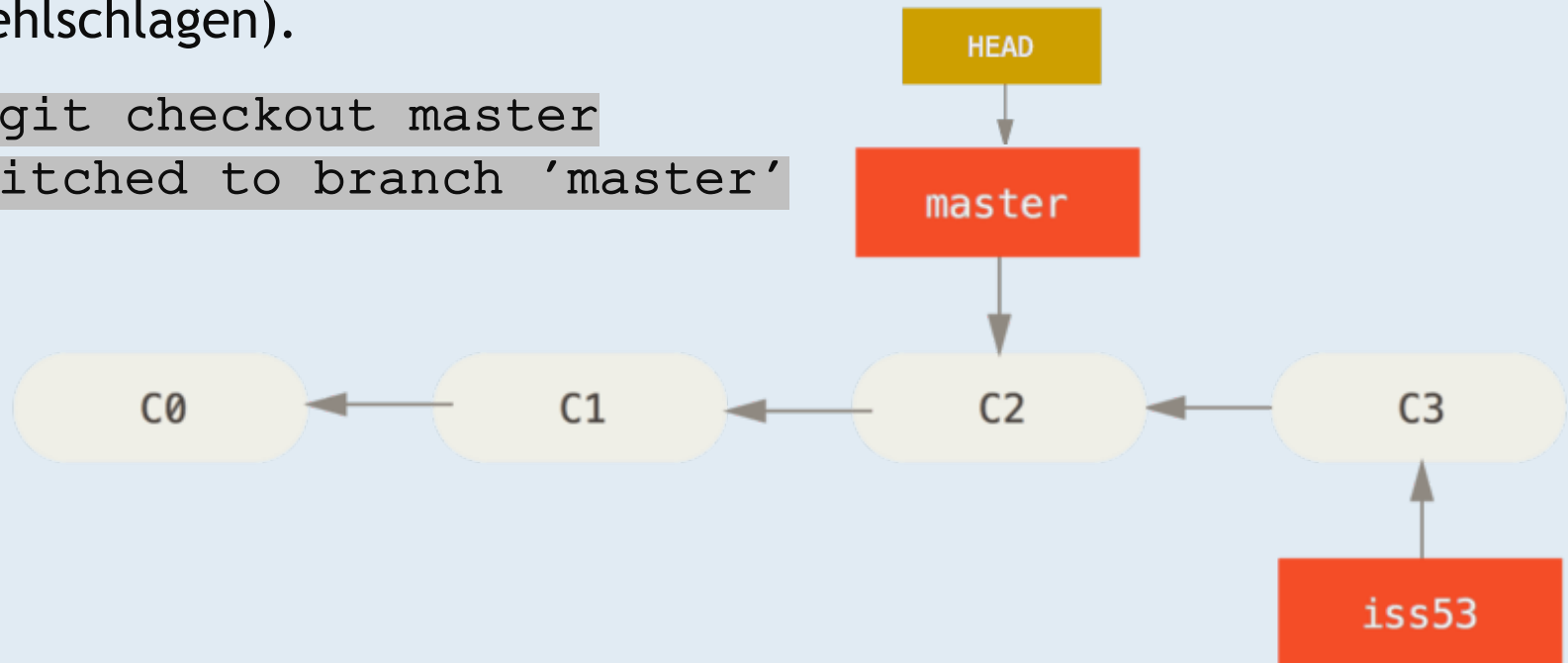
- » Du arbeitest an diesem Issue und machst dabei commits.
- » Weil dein aktueller branch "iss53" ist (dein "HEAD") wandert dieser Pointer mit jedem commit weiter.



# Ein Beispielszenario

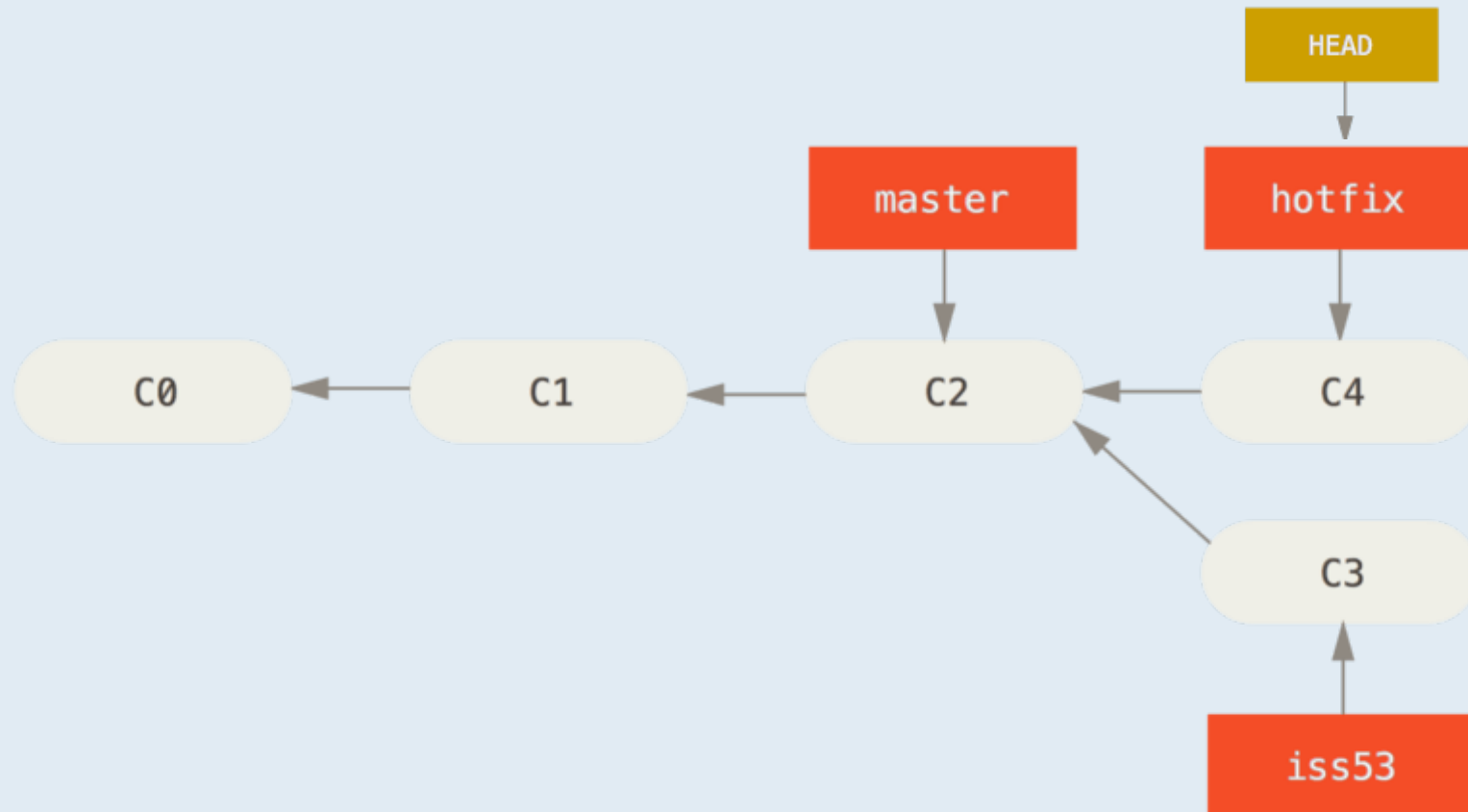
- » Plötzlich wird ein Problem gemeldet, das sofort behoben werden muss.
- » Dazu musst du zum Codestand wechseln, der momentan beim Kunden läuft. Du wechselst zurück zum master branch.
- » Bevor du wechselst, machst du ein letztes Commit am iss53 branch um einen sauberen Zustand zu haben (der Wechsel könnte sonst fehlschlagen).

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



# Ein Beispielszenario

- » Du erzeugst einen neuen Branch "hotfix", in dem du arbeitest und deine Änderungen committest, bis du das Problem gelöst hast.

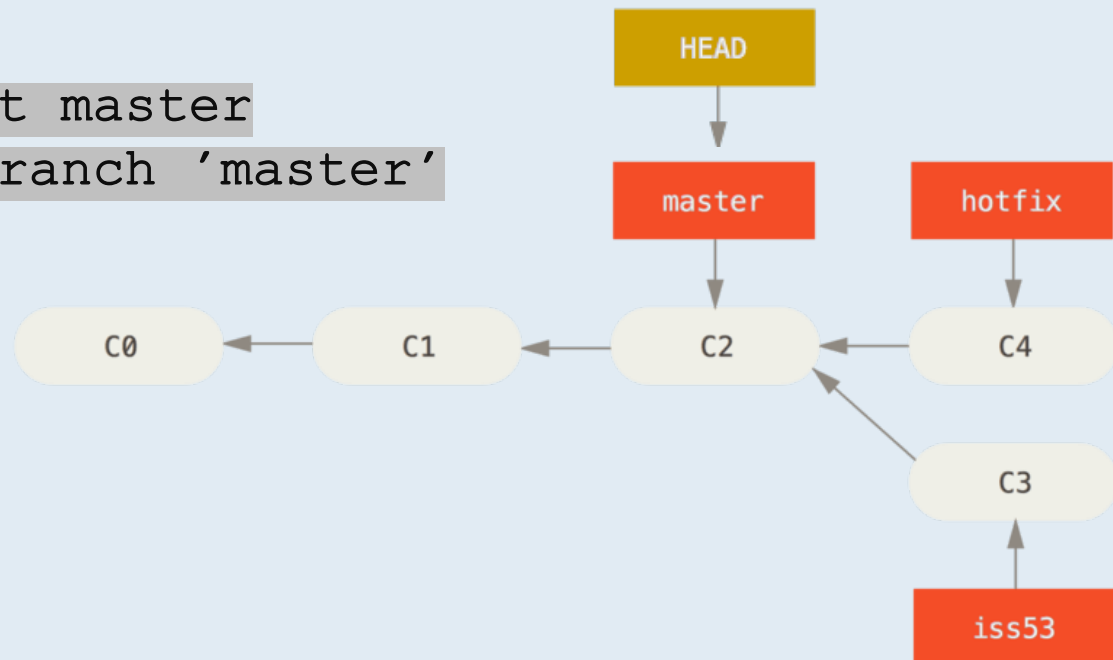


# Merge

# Ein Beispielszenario - Merge

- » Du hast die Arbeit am hotfix beendet und möchtest sie mit der Arbeit am master branch zusammenführen.
- » Die Zusammenführung zweier Branches nennt man **Merge**.
- » Dazu wechselst du in den Branch, der die Änderungen übernehmen soll ("das Ziel ist"). In unserem Fall der master branch.

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



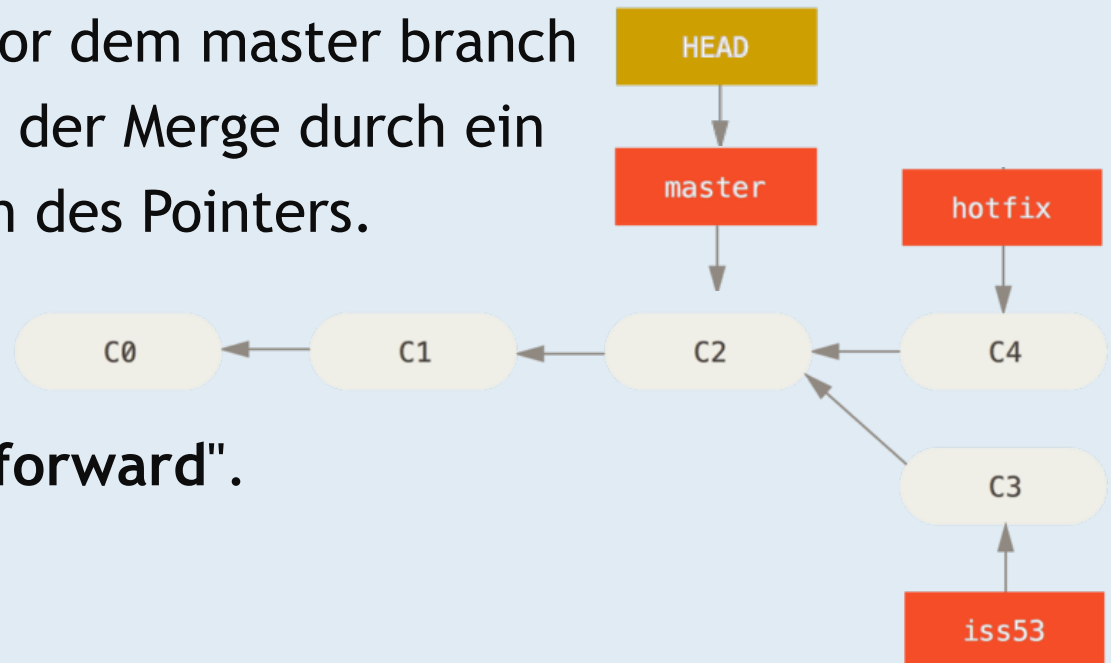
# Merge

» Mittels `git merge` integrieren wir den hotfix branch in den master branch

```
$ git merge hotfix  
Updating f42c576..3a0874c
```

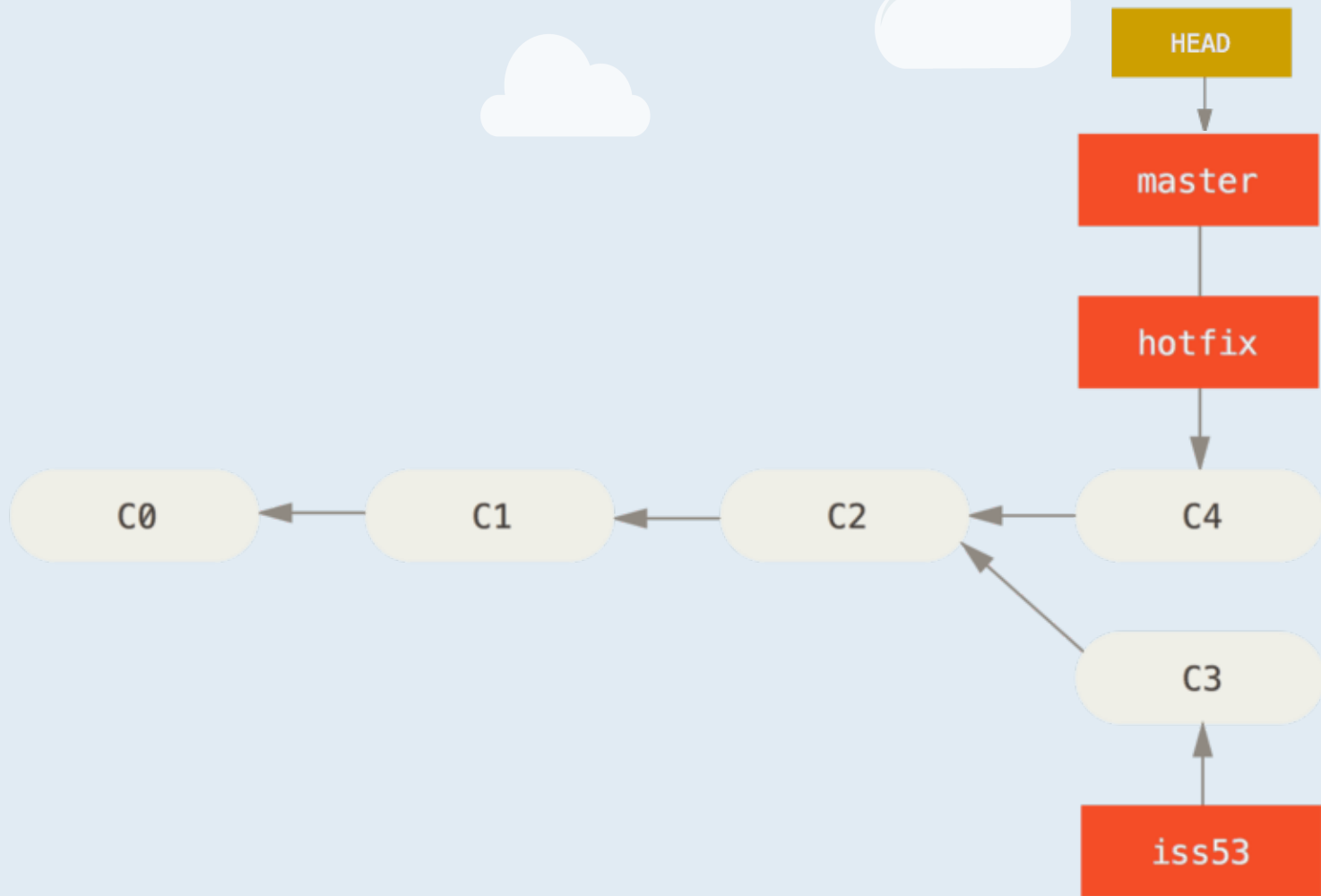
→ Fast-forward  
index.html | 2 ++  
1 file changed, 2  
insertions(+)

» Weil hotfix direkt vor dem master branch gestanden ist, erfolgt der Merge durch ein einfaches Verschieben des Pointers.



Man nennt dies "fast-forward".

# Ergebnis fast forward Merge





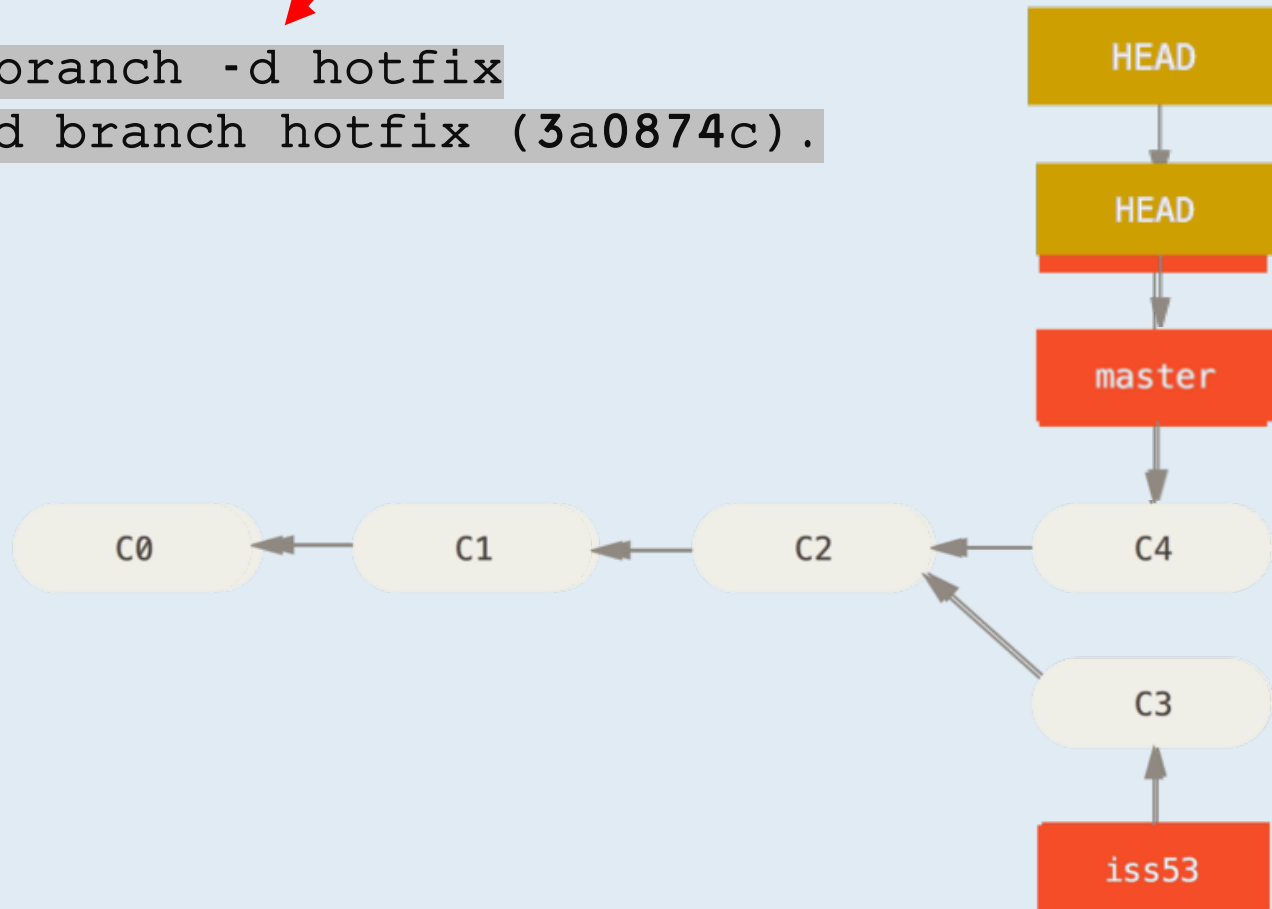
# Branch löschen

» Da wir den Branch hotfix nicht mehr benötigen, können wir ihn löschen.

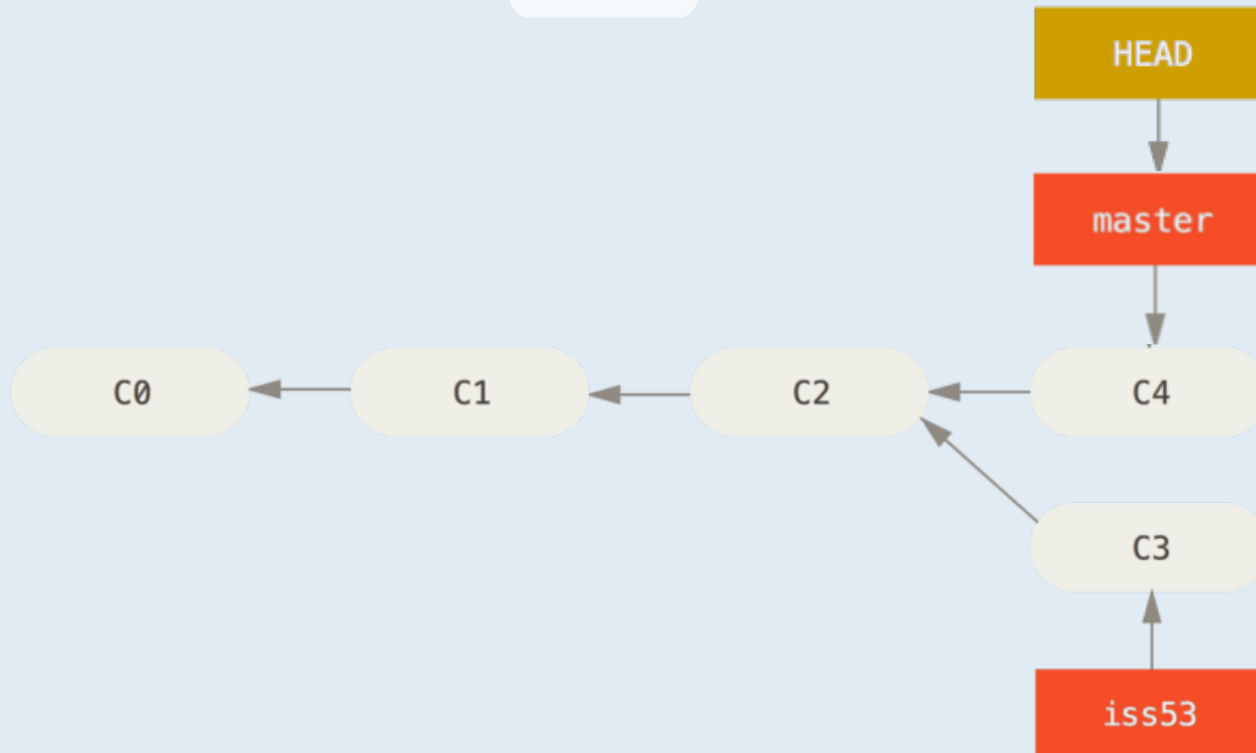
delete  
↙

```
$ git branch -d hotfix
```

```
Deleted branch hotfix (3a0874c).
```

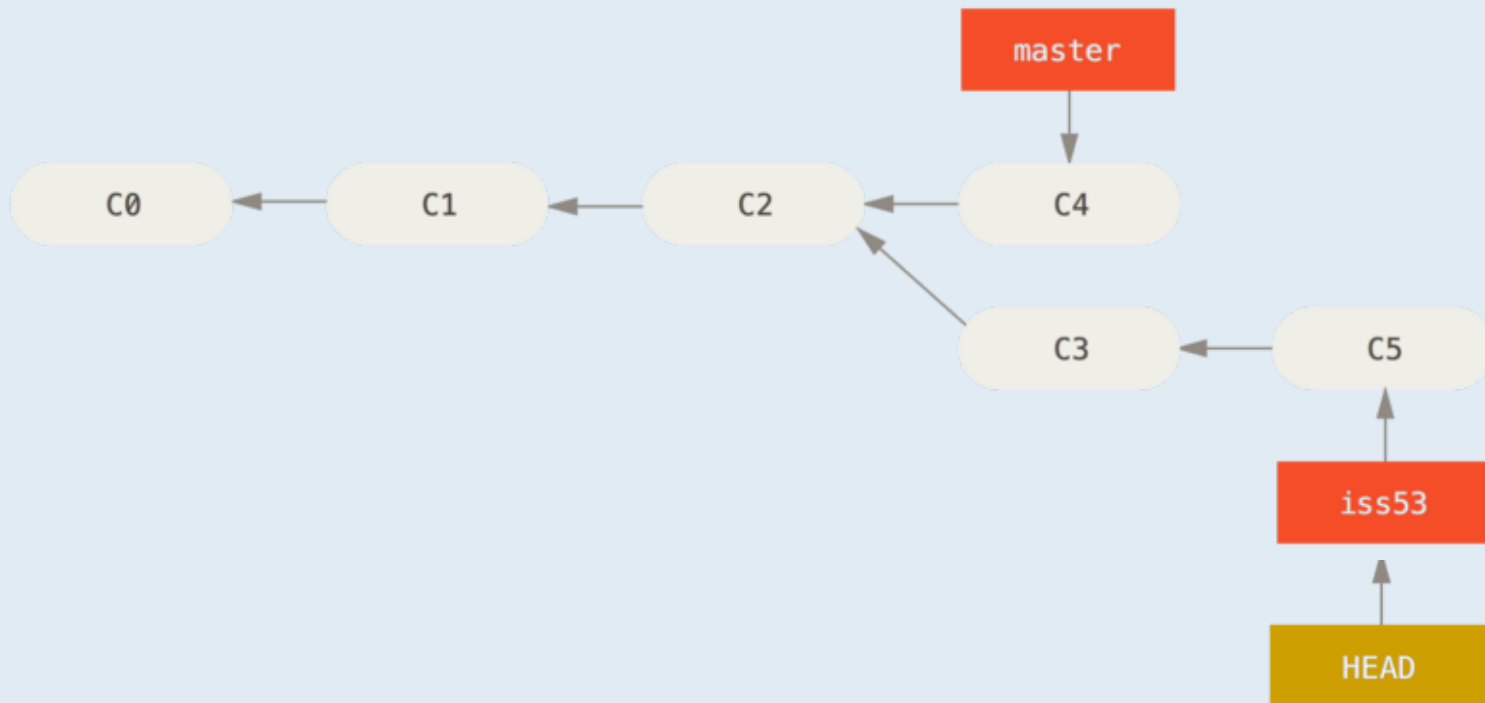


# Ergebnis Branch löschen



# Ein Beispielszenario

- » Du möchtest am "Issue #53" weiter arbeiten.
- » Du wechselst also zu diesem Branch und führst ein paar weitere Commits durch.



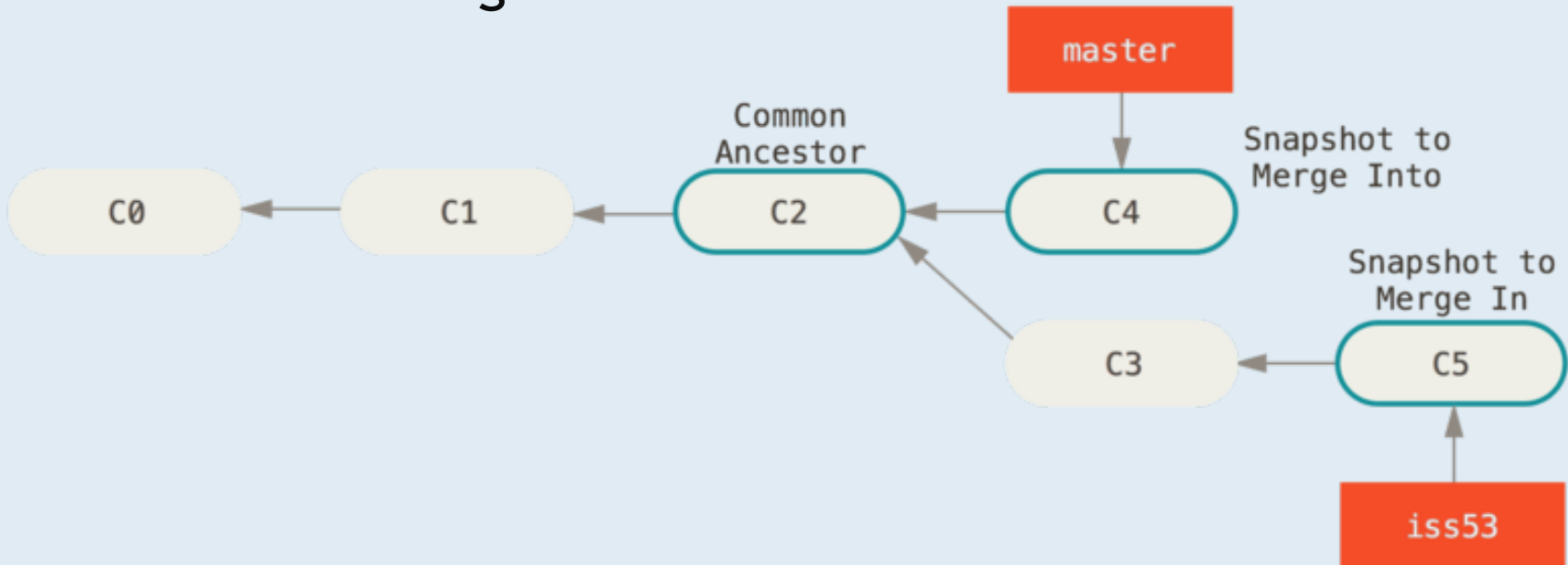
# Merge

- » Du bist mit der Arbeit am "Issue #53" fertig und möchtest die Arbeit in deinen master branch übernehmen.
- » Du wechselst also in den master branch und führst ein merge durch.

```
$ git checkout master  
Switched to branch 'master'  
$ git merge iss53  
Merge made by the 'recursive'  
strategy. index.html | 1 +  
1 file changed, 1 insertion(+)
```

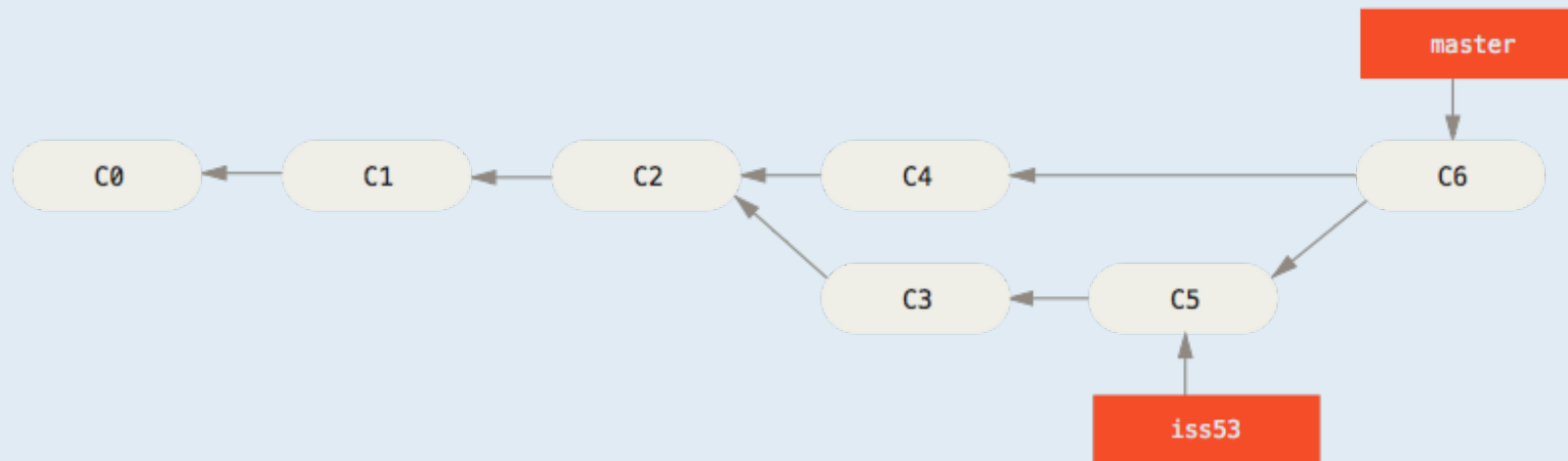
# Merge

- » Im Gegensatz zum letzten Merge, ist der master branch aber kein direkter Vorgänger des branch, der integriert werden soll.
- » Ein einfaches Verschieben des Pointers ("fast-forward") ist also nicht möglich.



# Merge Commits

- » git führt die Snapshots der Commits C2, C4 und C5 zusammen ("three-way merge") und erzeugt einen neuen Snapshot.
- » Dabei wird automatisch ein neuer Commit C6 erstellt.
- » Solche Commits nennt man "**merge commits**".
- » "Merge Commits" haben mehr als einen Eltern-Commit.



# Probiere es selbst aus!

- » Erzeuge einen dritten Branch und führe mind. ein Commit durch.
- » Merge die beiden neuen Branches in den master Branch.
- » Lösche die beiden neuen Branches.

# Merge Conflicts



# Merge Conflicts

- » In manchen Situationen benötigt git manuelle Unterstützung um die Änderungen von verschiedenen Branches zusammen zu führen.
- » Die gleiche Stelle wurde verändert. Git weiß nicht welche Änderung übernommen werden soll.

```
$ git merge iss53
```

```
Auto-merging index.html
```

```
CONFLICT (content): Merge conflict in index.html
```

```
Automatic merge failed; fix conflicts and then commit  
the result.
```

# Unmerged Paths

- » Git erzeugt in diesem Fall keinen automatischen Merge Commit, sondern pausiert den Prozess.
- » `git status` zeigt ebenfalls welche Dateien nicht zusammengeführt werden konnten.

```
$ git status
```

```
On branch master
```

```
You have unmerged paths. (
```

```
fix conflicts and run "git commit")
```

```
Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)
```

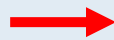
```
both modified: index.html
```

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

# Konfliktmarkierungen

» git fügt für jeden Konflikt Markierungen in die Datei(en) ein.

Trennlinie zwischen  
den beiden Versionen



```
<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=====
<div id="footer"> please contact us at
support@github.com </div>
>>>>>>> iss53:index.html
```

# Manuelle Konfliktbehebung

- » Man kann die Dateien manuell verändern.
- » Hat man alle Konflikte gelöst, schließt man den Merge mit einem Commit ab.
- » Beispiel 1: Ich möchte die Änderungen von HEAD übernehmen (Branch, der das Ziel des Merge ist)

```

<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=====
<div id="footer"> please contact us at
support@github.com </div>
>>>>>>> iss53:index.html
  
```

Um einen Konflikt zu lösen, müssen zumindest die Zeilen, die mit <<<<<<<, =====, und >>>>>>> beginnen, gelöscht werden.

# Manuelle Konfliktbehebung

- » Man kann die Dateien manuell verändern.
- » Hat man alle Konflikte gelöst, schließt man den Merge mit einem Commit ab.
- » Beispiel 2: Ich möchte die Änderungen von Branch iss53 übernehmen (Branch, der das Ziel des Merge ist)

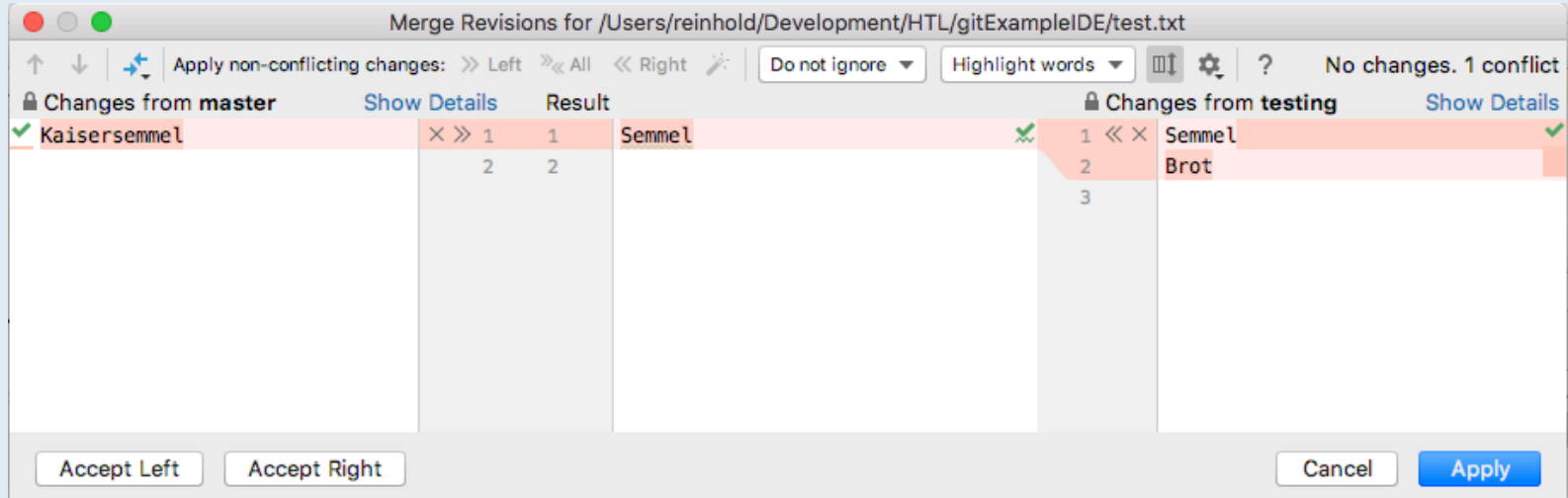
```

<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=====
<div id="footer"> please contact us at
support@github.com </div>
>>>>>>> iss53:index.html

```

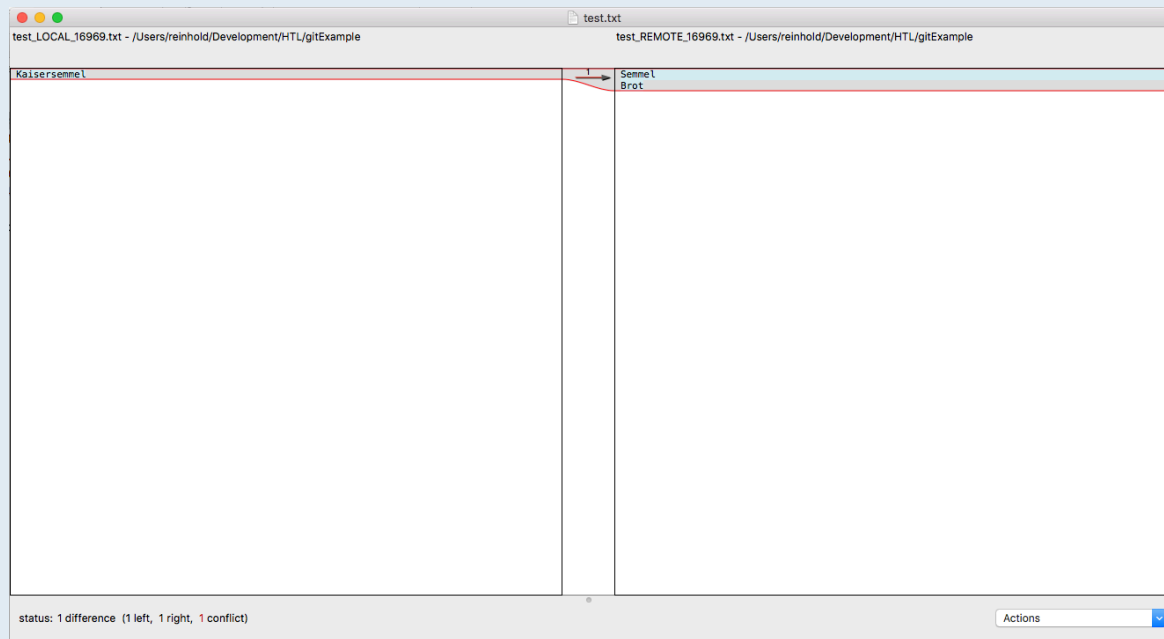
# Grafische Tools

- » Es gibt jede Menge grafische Tools, die beim Auflösen von Konflikten helfen.
- » z.B. integriert in IntelliJ



# Grafische Tools

- » `git mergetool` startet ein grafisches Tool für die Konfliktlösung
- » Kann je nach Betriebssystem unterschiedlich sein



# Merge Commit

- » Nachdem alle Konflikte gelöst worden sind, muss mit einem Commit der Merge abgeschlossen werden.
  - » Wie gewohnt add & commit
- » git schlägt allerdings bereits eine commit message vor, die man ergänzen kann.

```
Merge branch 'iss53'
```

```
Conflicts:
```

```
index.html
```

```
#
```

```
# It looks like you may be committing a merge.
```

```
# If this is not correct, please remove the file
```

```
# .git/MERGE_HEAD
```

```
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
```

```
# All conflicts fixed but you are still merging.
```

```
#
```

```
# Changes to be committed:
```

```
#       modified:   index.html
```

```
#
```



# Probiere es selbst aus!

- » Versuche einen Konflikt beim Mergen zweier Branches zu erzeugen und ihn zu lösen.

# Detached Head

# Checkout eines Commits

- » Mittels `git checkout` wechselt man unter Angabe eines Branch-Namens zwischen den Branches.

```
$ git checkout iss53
```

- » Man kann damit auch zum Zustand jeden beliebigen Commits wechseln.
- » Dazu gibt man statt des Branch-Namen den *SHA1 hash* des Commits an.

```
$ git checkout 56a4e5c08
```

```
Note: checking out '56a4e5c08'.
```

```
You are in 'detached HEAD' state...
```

# SHA1 hash

» Wie bekomme ich den SHA1 hash eines Commits?

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

← SHA-1 Prüfsumme

Change version number

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

Remove unnecessary test

» Es reicht, die ersten paar Zeichen (min. 4) des Hash anzugeben (bis er eindeutig ist)

» siehe z.B. Ausgabe von `git log --oneline`

# Detached Head

- » Checkt man ein spezifisches Commit aus, zeigt HEAD auf den aktuellen Commit, der aber nicht mit einem branch verknüpft ist -> "detached head"
- » Um Änderungen an diesem Zustand speichern zu können, muss der Zustand in einem neuen Branch gespeichert werden.
  - » `$ git checkout -b <new-branch-name>`
- » Dieser Branch kann dann weiter verändert und mit anderen Branches gemerged werden.

# Änderungen "rückgängig" machen

# Herangehensweise

- » Vermeide Situationen, bei denen du Änderungen rückgängig machen musst!
  - » Commit A
  - » ....ich habe eine Idee und möchte etwas ausprobieren
  - » Commit B
  - » Commit C
  - » ....war keine gute Idee. Ich möchte zu Commit A zurück.
- » Das ist möglich, aber eine andere Herangehensweise macht die Angelegenheit viel einfacher.

# Arbeite mit Branches!

- » Branches können in git schnell erstellt, vereinigt und gelöscht werden. Nütze diese Möglichkeiten!
- » Commit A am Branch "master"
- » ...ich habe eine Idee und möchte etwas ausprobieren
- » ...erstelle einen neuen Branch "idea-recursive2"
- » Commit B am Branch "idea-recursive2"
- » Commit C am Branch "idea-recursive2"
- » ...war keine gute Idee. Ich möchte zu Commit A zurück.
- » ...du kannst einfach zum master-Branch wechseln und den Branch "idea-recursive2" löschen (Commit B und C gehen verloren)



# Änderungen rückgängig machen

- » Es gibt git Befehle, die Änderungen "rückgängig" machen:
  - » `git revert`, `git reset`,...
- » Experimentiere gerne mit ihnen herum, aber **nicht am master branch!**<sup>1</sup>
- » Warum? **Manche** dieser Befehle ändern die Commit-History.
  - » Deine Commits am master werden aber mit dem master branch des remote-Repositories synchronisiert.
  - » Es können Konflikte zwischen der Commit-History des lokalen und remote Repositories entstehen.


<sup>1</sup>Außer du weißt genau, was du tust ;)

# Wunschliste git

Welche Möglichkeiten würden dich  
aktuell beim Arbeiten an den  
Abgaben unterstützen?

# Online-Ressourcen

- » <https://git-scm.com/book/en/v2>
  - » Auch in Deutsch verfügbar, wenngleich manche Übersetzungen verwirrend sein können.
- » <https://github.com/git-guides/>
- » <https://www.atlassian.com/git/tutorials>
  
- » Du hast ein spannendes git Feature entdeckt, das deinen Workflow unterstützt? Bitte erzähle mir davon! 😊



Fragen?  
Anregungen?  
Bemerkungen?