

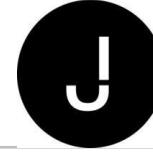


JAVACREAM

*Training
Consulting
Projectmanagement*

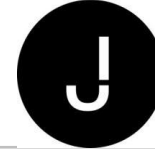
GIT

- Name
- Rolle im Unternehmen
- Themenbezogene Vorkenntnisse
- Aktuelle Problemstellung
- Konkrete individuelle Zielsetzung

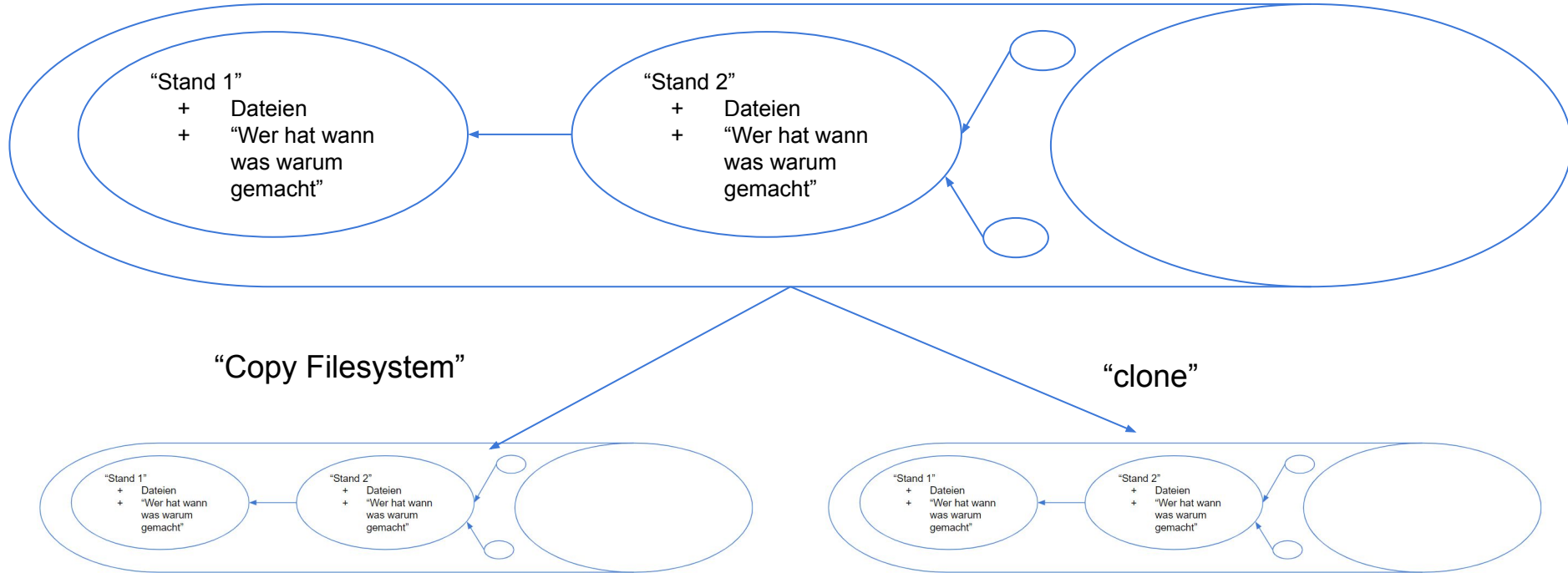


Git - Ein verteiltes Versionsverwaltungssystem

- Das git-Kommando implementiert eine vollständige Umgebung
 - Ich brauche keinen Git-Server (!)
- Synchronisation mit einem Server ist möglich, aber nicht permanent nötig
- Authentifizierung / Autorisierung ist nicht vorgesehen
 - `git config --global user.name = "..."`
 - `git config --global user.email = "..."`
 - Diese Informationen werden in ihrem User-Profil abgelegt `.gitconfig`
- Team-Zusammenarbeit ist mit dem eigentlichen Git nicht vorgesehen



- Produkte
 - GitHub (Microsoft)
 - Bei Ihnen **GitHub Enterprise** gehosted auf Server der FI
 - GitLab (GitLab)
 - Bitbucket (Atlassian)
- Aufgaben
 - Echte Authentifizierung und Autorisierung
 - Zentrale Ablage der Informationen
 - Team-Zusammenarbeit
 - Team-Git-Flows zum effizienten Arbeiten
 - “Pull Request”



- Es muss gewährleistet sein, dass die Informationsablage im Repository konsistent (“fälschungssicher”) abgelegt ist
- Lösung
 - Unabhängig von Git
 - Merkle Trees
 - Aus Informationen wird ein Hashwert berechnet
 - Historie von Informationen wird dadurch berücksichtigt, dass der Hashwert des Vorgänger-Standes in die Information mit aufgenommen wird
 - Blockchain-Technologie
- Git arbeitet seit Version 1.0 mit Blockchain

- Ist die Fälschungssicherheit garantiert
 - Nein
- 1000 Mitarbeitende, die pro Tag 100 Stände neu definieren und das über mindestens 100 Jahre -> Wahrscheinlichkeit einer Kollision im unteren Prozentbereich
 - Für reale Softwareentwicklung absurd Unwahrscheinlich

Git First Contact

Erzeugen eines Repositories

Developer



clone

GitHub Enterprise

- + (Administratives) Anlegen eines Repositories
- + Definition des Teams

sauber



Developer

```
MINGW64/c/Users/Rainer Sawitzki
Rainer Sawitzki@LAPTOP-GVSFDDCT MINGW64 ~
$ git --version
git version 2.25.0.windows.1
Rainer Sawitzki@LAPTOP-GVSFDDCT MINGW64 ~
$ |
```

```
mkdir git_first_contact
cd git_first_contact
git init
```

training

~~Directory~~ Git Project Directory

git init

.git
= Git-Repository

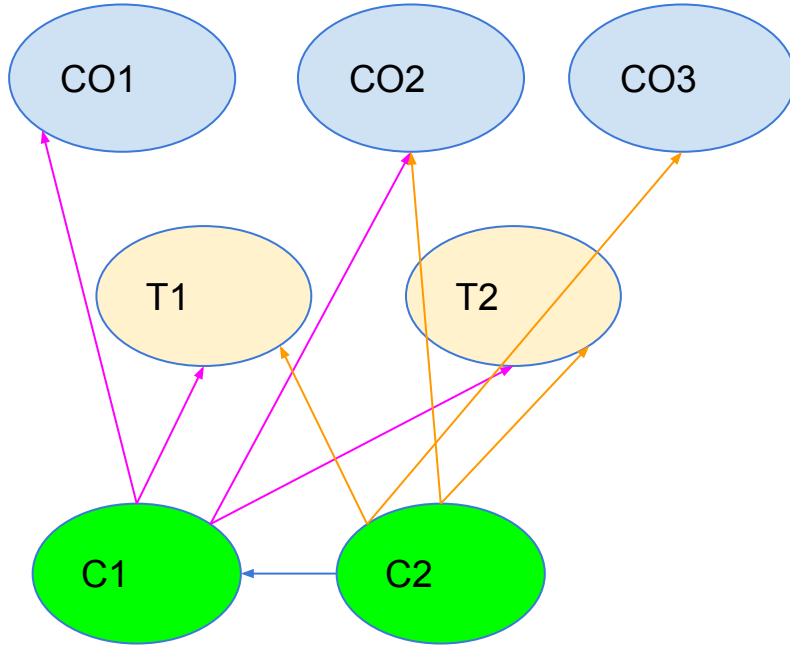
- Verzeichnisse und Dateien
 - "Workspace"

Wichtig

Eine Git-Installation enthält keinerlei Hintergrund-Dienste, Domain-Prozesse oder ähnliches

Versionsverwaltungs-Funktionen laufen nur, solange das Git-Kommando aktiv ist

- Anlegen / Ändern einer Datei im Workspace
 - `echo Hello > content.txt`
 - CHECK: `ls -> content.txt`
- Hinzufügen zu Git
 - `git add .`
 - CHECK: `objects/e9/...`
- Definieren eines neuen Standes
 - “Wer hat wann was warum gemacht”
 - `git commit -m “”`
 - CHECK
 - `objects/xy, objects/ab`
 - `git log -> Ausgabe des commit-Hashwertes`



Content-Objekte (BLOB)

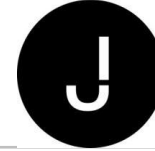
- + Datei-Inhalt
- + git add

Tree-Objekte

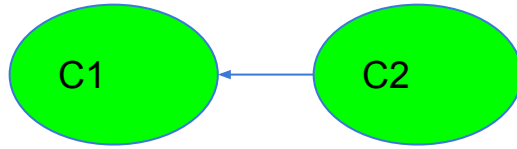
- + Pfad und Datei-Attribute
- + git commit

Commit-Objekt

- + Meta-Daten
- + Referenzen



- Git arbeitet NIE mit Deltas!
 - Alle Objekte in Git sind immer atomar vollständig
- Konsequenzen
 - Braucht mehr Speicherplatz
 - Interpretation eines Objektes ist einfach bestimmbar



- git commit definiert immer ein neues Commit-Objekt mit einem Vorgänger-Commit
- Commits werden durch den Hashwert eindeutig identifiziert
- Die Information “auf welchem Stand bin ich gerade” wird durch die Angabe des Hashwertes beantwortet
 - Benutzer: “Nerd-Modus”



- Statt Nerd-Modus sprechende Namen
- 2 Situationen
 - Ein bestimmter Stand ist fix
 - v1.1
 - savepoint
 - “Heute Morgen um 9:00”
 - Eine gerade laufende Aktion
 - implement/feature1
 - development
 - Ticket-Nummer
 - “experimentiere”

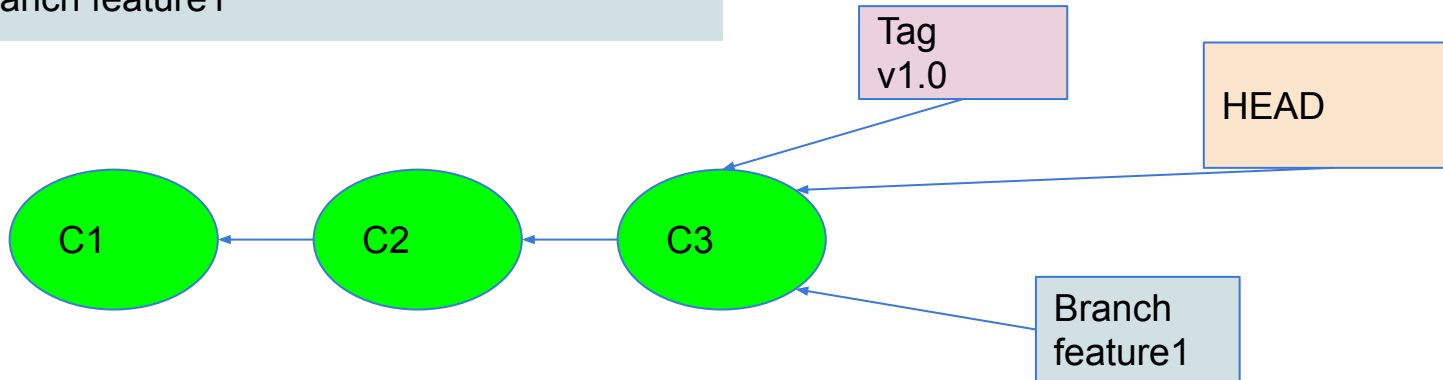
Alles folgende sind und bleiben ausschließlich Alias-Namen

Alias-Name für die aktuelle Position: HEAD

Tags und Branches sind
total leichtgewichtig

```
git tag v1.0
```

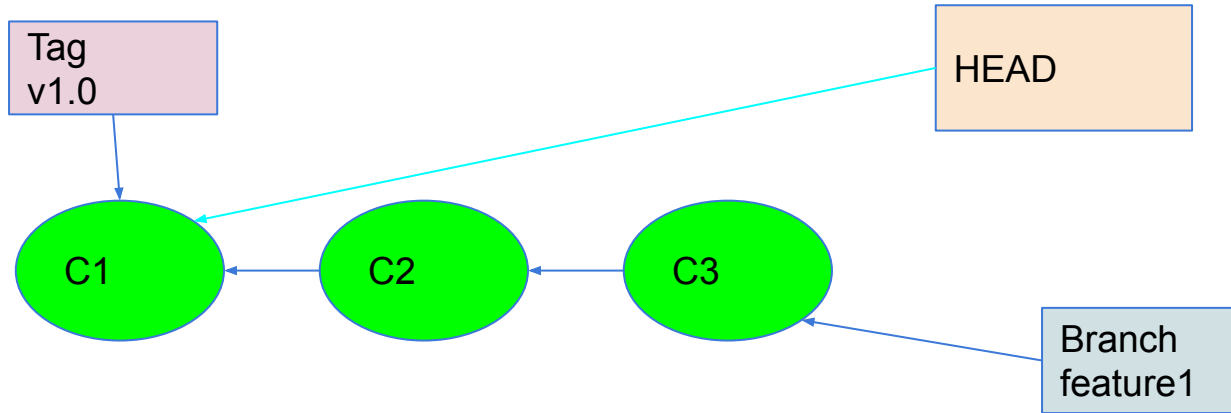
```
git branch feature1
```



Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout C1

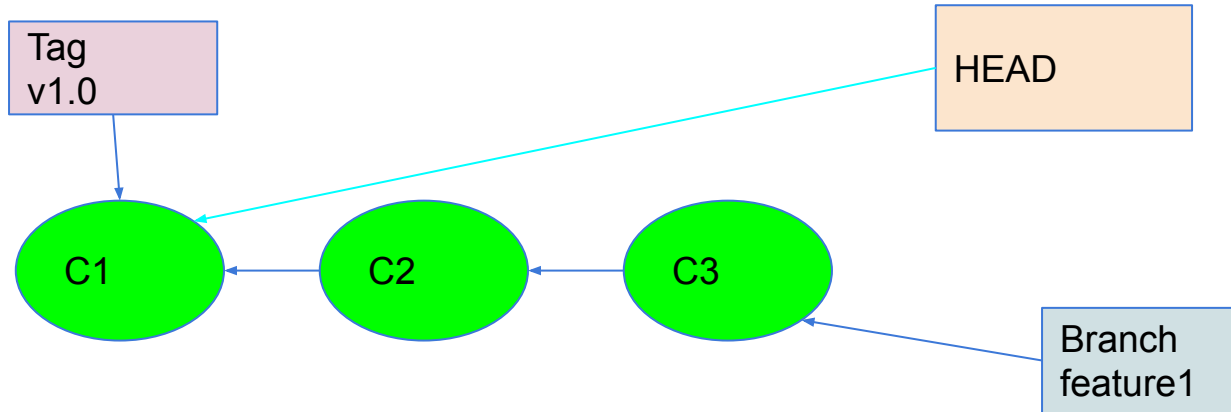


git status
+ DETACHED HEAD

Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout v1.0

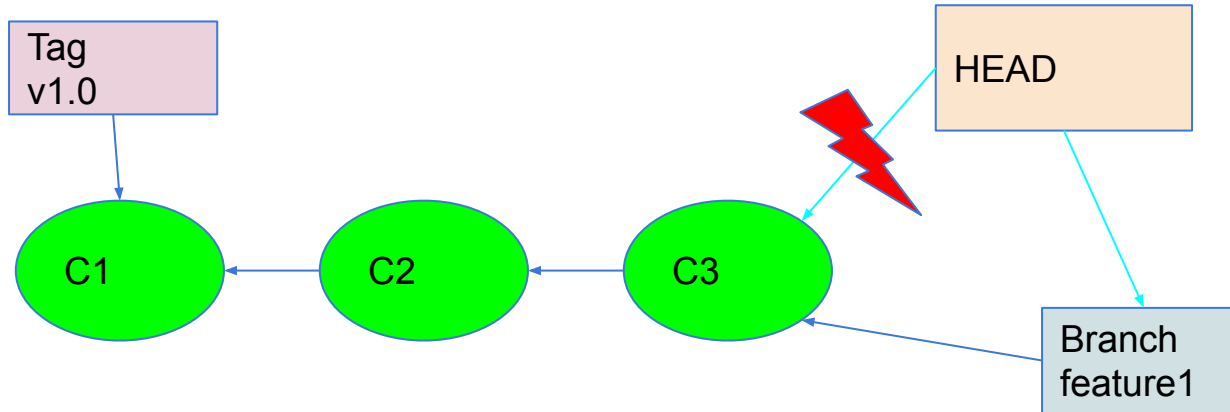


git status
+ DETACHED HEAD

Navigieren im Graphen der Commit-Objekte

- “HEAD = git checkout <HASH>”
- git checkout <HASH>

git checkout feature1



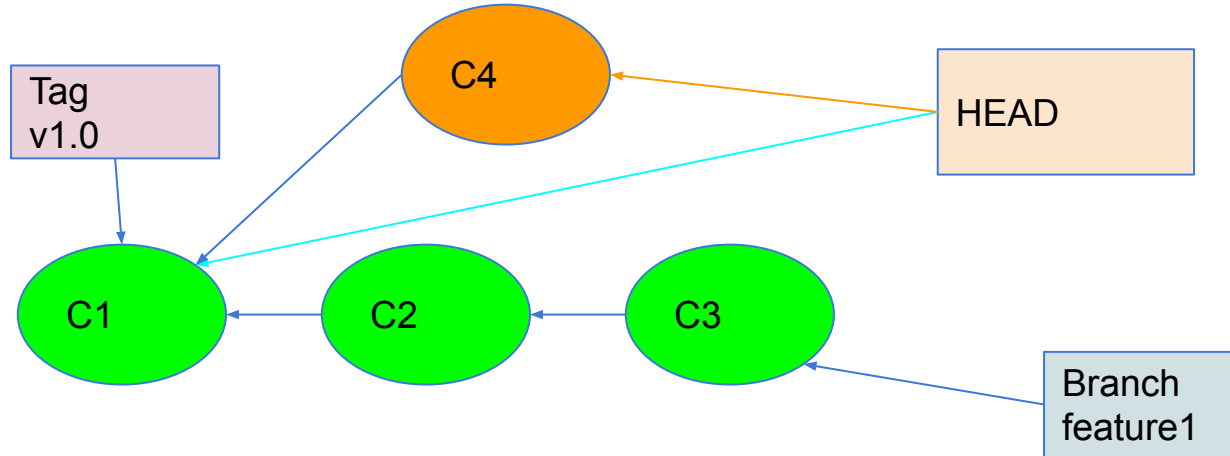
git status
+ Unauffällig
+ ATTACHED HEAD

- Ein checkout sollte nur gemacht werden, wenn git status unauffällig ist
 - Anlegen eines neuen Branches und commit
 - add + stashing -> später, Git.pdf, Online-Dokumentation
- `git checkout -b new_branch ...`

Commit im Detached HEAD

git checkout v1.0 oder C1

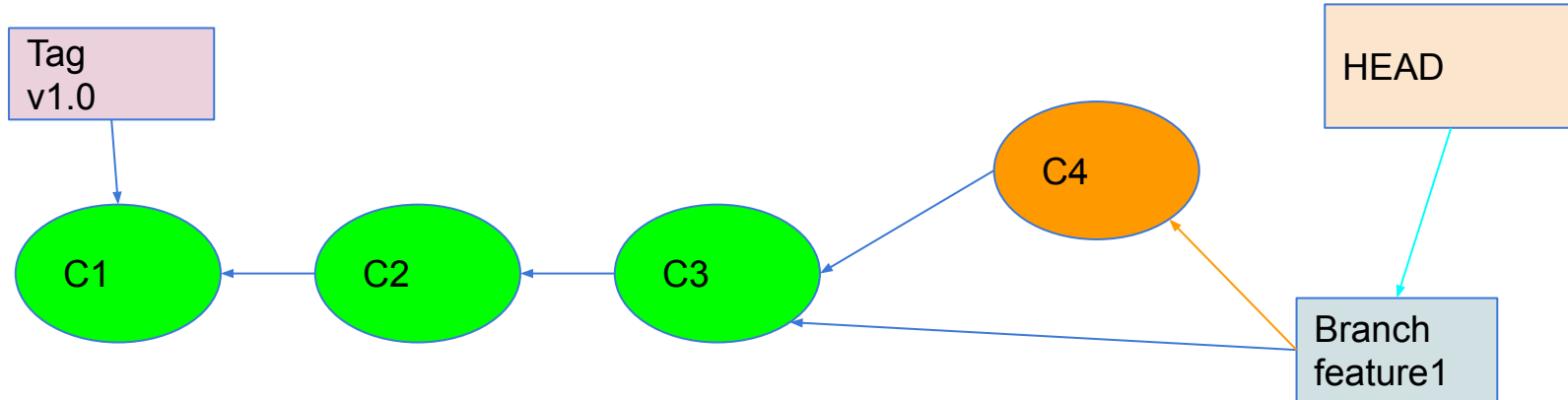
```
//Änderungen  
git add .  
git commit -m ""
```



Commit im Attached HEAD

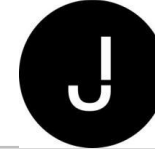
git checkout feature1

```
//Änderungen  
git add .  
git commit -m ""
```

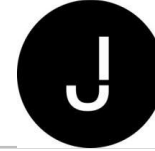




- Ein “Dangling Object” ist ein Objekt, das nicht über den HEAD, einen Branch oder ein Tag direkt oder indirekt referenzierbar ist
- Anzeigen
 - `git fsck --unreachable --no-reflogs`
- Garbage Collection
 - `git reflog expire --expire-unreachable=now --all`
 - `git gc --prune=now`



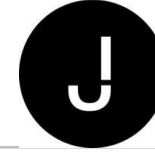
Konsolidieren von Ständen / Commit-Objekte



- Merge
 - fast-forward
 - recursive
- Rebase
- Interactive Rebasing
- Cherry Pick
 - Aktuell in der Git Community eher als “deprecated” betrachtet
- Patching
 - Problemsituation: Internes Repo -> Kunden -> Kunden-Repo
 - Erstellen eines Patches und Einspielen in der Zielumgebung

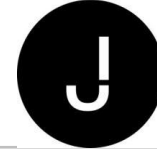
- `setup-branches.sh` auf Referenten-GitHub

- Ziel
 - Integration von feature1 und feature2 in den master und damit ein Fortschritt der allgemeinen Entwicklung im Projekt
 - Es muss alles so nachvollziehbar sein, wie es geschehen ist
- Plan (subjektiv)
 - feature2 wird konsolidiert mit den beiden parts
 - anschließend kommt feature1 dazu
 - zum schluss dann der master



- muss unauffällig

git checkout feature2



```
* 6550e54 (feature2_part1) change content-feature2, part1
| * b02e4a3 (feature2_part2) change content-feature2, part2
|/
* 7c75225 (HEAD -> feature2) add content-feature2
| * 64f7bb8 (feature1) add content-feature1
|/
* bbefa32 (master) change content
* 176197d add content
* 283c53a setup project
```

git merge feature2_part1



```
Updating 7c75225..6550e54
Fast-forward
 content-feature2.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
* 6550e54 (HEAD -> feature2, feature2_part1) change content-feature2, part1
| * b02e4a3 (feature2_part2) change content-feature2, part2
|/
* 7c75225 add content-feature2
| * 64f7bb8 (feature1) add content-feature1
|/
* bbefa32 (master) change content
* 176197d add content
* 283c53a setup project
```


git merge feature2_part2

```
$ git merge feature2_part2
Auto-merging content-feature2.txt
CONFLICT (content): Merge conflict in content-feature2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

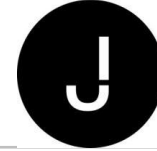
```
* 70f93b5 (HEAD -> feature2) Merge branch 'feature2_part2' into feature2
|
| * b02e4a3 (feature2_part2) change content-feature2, part2
| * | 6550e54 (feature2_part1) change content-feature2, part1
|/
|
| * 7c75225 add content-feature2
| * 64f7bb8 (feature1) add content-feature1
|/
|
| * bbefa32 (master) change content
| * 176197d add content
| * 283c53a setup project
```

git merge feature1



```
*    ba092be (HEAD -> feature2) Merge branch 'feature1' into feature2
|
| *    64f7bb8 (feature1) add content-feature1
| *    70f93b5 Merge branch 'feature2_part2' into feature2
|
| *    b02e4a3 (feature2_part2) change content-feature2, part2
| *    6550e54 (feature2_part1) change content-feature2, part1
|
| *    7c75225 add content-feature2
|
| *    bbefa32 (master) change content
| *    176197d add content
| *    283c53a setup project
```

git checkout master git merge --no-ff feature2



```
* 9eec2ba (HEAD -> master) Merge branch 'feature2'
* ba092be (feature2) Merge branch 'feature1' into feature2
* 64f7bb8 (feature1) add content-feature1
* 70f93b5 Merge branch 'feature2_part2' into feature2
* b02e4a3 (feature2_part2) change content-feature2, part2
* 6550e54 (feature2_part1) change content-feature2, part1
* 7c75225 add content-feature2
* bbefa32 change content
* 176197d add content
* 283c53a setup project
```

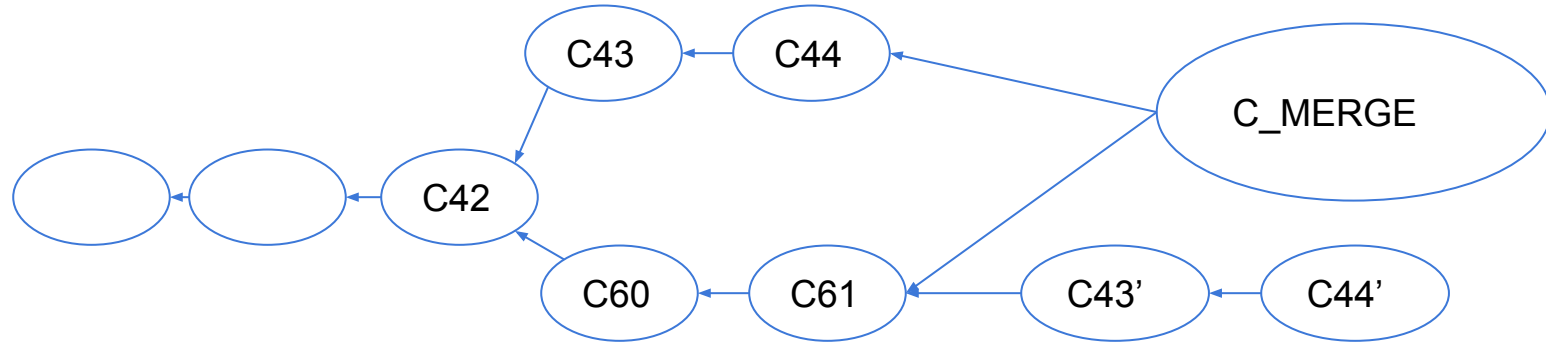
- Ich persönlich würde diesen merge-Plan etwas modifizieren, und das merging auf einem temporären Branch durchführen
- merge immer mit no-ff
 - Empfehlung der Git-Community
- Es fehlte im Beispiel das Arbeiten mit Tags
- Cleanup fehlt
 - feature-Branches löschen
 - Ersetzen durch Tags
 - feature1-Branch -> finish/feature1

- Ziel
 - Integration von feature1 und feature2 in den master und damit ein Fortschritt der allgemeinen Entwicklung im Projekt
 - ~~Es muss alles so nachvollziehbar sein, wie es geschehen ist~~
 - es soll eine stringente, nachvollziehbare sequentielle Historie entstehen
 - Geschichtsfälschung
- Plan (subjektiv)
 - feature2 wird konsolidiert mit den beiden parts
 - anschließend kommt feature1 dazu
 - zum schluss dann der master

- git checkout feature2
- git merge feature2_part1
 - fast forward
- git rebase feature2_part2

```
* 77023cd (HEAD -> feature2, feature2_part1) change content
| * 6c1281c (feature2_part2) change content-feature2
|/
* 1147bcb add content-feature2
| * 4e6647c (feature1) add content-feature1
|/
* 9ef0fb6 (master) change content
* 89698b9 add content
* bdace3c setup project
```

```
* c6b5952 (HEAD -> feature2) change content
| * 6c1281c (feature2_part2) change content
| * 77023cd (feature2_part1) change content
|/
* 1147bcb add content-feature2
| * 4e6647c (feature1) add content-feature1
|/
* 9ef0fb6 (master) change content
* 89698b9 add content
* bdace3c setup project
```





- git rebase feature1
- git checkout master
- git merge feature2
- Cleanup

```
$ git log --oneline --graph --all
* 5f205d5 (HEAD -> master) change content-feature2, part1
* ad7c40d change content-feature2, part2
* 3d89739 add content-feature2
* 4e6647c add content-feature1
* 9ef0fb6 change content
* 89698b9 add content
* bdace3c setup project
```