NewElements

IT-Schulungen.com

GIT - Seminar

Dr.sc.nat. Michael J.M. Wagner*

Revision 26

*michael@wagnertech.de

Inhaltsverzeichnis

IT-Schulungen.com
Portfolio

IT-Schulungen.com ist eines der führenden, herstellerunabhängigen Seminarportale von Schulungen rund um die Informationstechnologie (IT) und das IT-Management. Seit über 15 Jahren ist IT-Schulungen.com eine anerkannte Anlaufstelle für viele Unternehmen und Behörden, wenn es um die Durchführung von DACH-weiten Schulungen geht.

- Applikationsserver / Middleware
- Business Intelligence
- Business-Skills und Führung
- Cloud
- CRM
- Datenbanken
- eBusiness
- ERP-Systeme
- IT Management
- IT-Recht / Lizenzierung
- ITIL
- Mobile
- Multimedia
- Office
- Open Source
- Portale
- SAP®
- Security
- Serversysteme
- Softwareentwicklung
- Systemmanagement

Zertifizierungen & Partnerschaften

1 Grundlagen

Konfigurationsmanagement

- Ein (meist) zentrales Repository für gemeinsamen Zugriff
- Koordination der gemeinsamen Arbeit in diesem Repository („einchecken", „auschecken")
- Dokumente werden versioniert
- Die Möglichkeit der Attributierung einzelner Dokumentversionen (Kommentar, Datum der Änderung, Autor, etc.)
- Die Rückverfolgbarkeit des Arbeitsverlaufs (Die Versionskette beantwortet die Frage: Wer hat was, wann verändert?).
- Bildung von Konfigurationen („Markieren" zusammengehöriger Dokumentversionen)

Marktübersicht

- OpenSource
  - RCS (Revision Control System)
    - 1981 an der Perdue University
    - Arbeitet auf einzelnen Dateien (pessimistisches Sperren)
  - cvs (concurrent versioning system)
    - 1989 aus RCS weiterentwickelt
    - Erweiterung auf Verzeichnisbäume (optimistisches Sperren)
    - Netzwerkfähig
  - svn (Subversion)
    - Entwicklung seit 2000
    - Atomarer Checkin von Verzeichnisbäume -> Revisionierung von Projekten statt Dateien
    - Webfähig
  - git[1]
    - 2005 von Linus Torwalds für die Linux-Kernelentwicklung entwickelt
    - Unterstützung verteilter Arbeitsabläufe

[1] https://de.wikipedia.org/wiki/Git (19.6.2018)

        * Sehr hohe Sicherheit gegen sowohl unbeabsichtigte als auch böswillige Verfälschung
        * Hohe Effizienz
        * Verteilte Repositories
- Proprietäre Software
    - Microsoft
      Visual Souce Safe (VSS) / Team Foundation Server
    - IBM
      Rational ClearCase
        * Entwicklung seit 1985
        * Multi Site: Verteilte Repositories
        * Build Tools integrierbar

Grundlagen der Versionskontrolle

Hauptpfad - Trunc	Hauptpfad der Entwicklung. Der *trunc* hat in Git den Namen `master` oder `main`.
Zweig - Branch	Seitenpfad: Nachträgliche Änderungen an älteren Softwareständen
Top	Jüngste Version in einem Pfad (Hauptpfad oder Zweig)
Head	Top-Version im Hauptpfad
Marke - Tag	Ausgezeichnete Version in einem Pfad

Zentral vs. verteilt

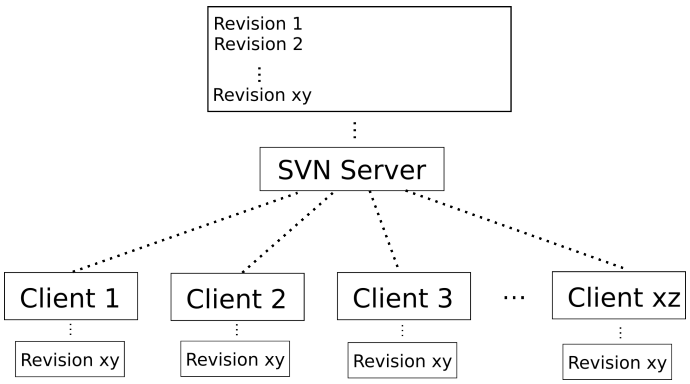

Abbildung 1: Zentrale Versionsverwaltung[2]

Bei einem zentralen Repository befindet sich dieses auf einem zentralen Server. Die Mitarbeiter eines Projekts holen sich aus diesem Repository die Dateien, modifizieren diese und speichern sie zurück. Die Versionsverwaltung regelt die Zugriffsrechte und überwacht Atomizität und Serialisierbarkeit (kein Überschreiben von konkurrierenden Modifikationen, Abb. 1).

[2] `https://de.wikipedia.org/wiki/Apache_Subversion` (8.9.2023)

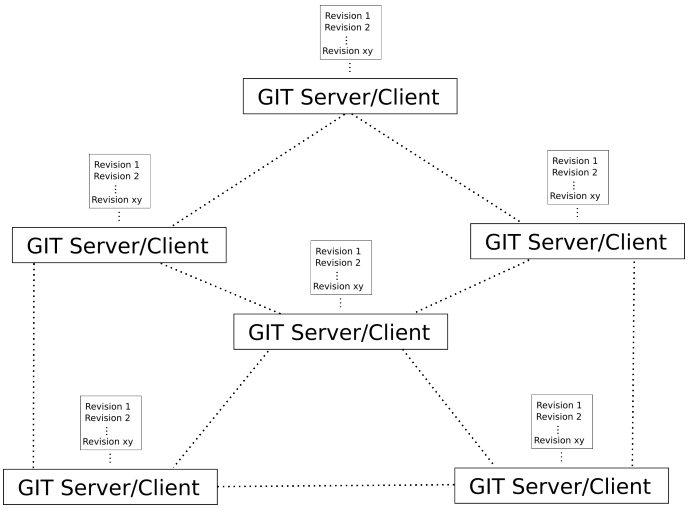

Abbildung 2: Verteilte Versionsverwaltung[3]

Bei einem verteilten Repository arbeitet jeder Mitarbeiter auf seinem eigenen Repository. Die Arbeitsstände müssen dann zwischen den Repositories synchronisiert werden (Abb. 2).

Faktisch hat sich auch bei git eine Architektur herausgebildet, bei der ein Projekt ein bis zwei Haupt-Repositories hat, über die alle Mitarbeiter ihre Arbeitsstände synchronisieren. Haupt-Repositories wie GitHub, GitLab, Bitbucket bieten über die reine Versionsverwaltung hinaus Funktionalität für Build und Test der Software.

Lokale Installation

Um sich mit git vertraut zu machen, kann erst mal eine lokale Installation verwendet werden. Wird in einem beliebigen Verzeichnis der Befehlt `git init` ausgeführt, so erhält man ein leeres Repository im Verzeichnis `.git`.

- Anlegen eines Projektarchivs (*repository*):
  `git init`
- Neue Dateien unter Versionskontrolle nehmen:
  `$ git add <file>`

  Anmerkung: Verzeichnisse allein können nicht hinzugefügt werden
- Änderungen einchecken:
  `$ git commit [-m "Kommentar"]`

Aufgabe:

- Legen Sie ein lokales Projektarchiv an.
- Fügen Sie Dateien hinzu.

[3] `https://de.wikipedia.org/wiki/Git` (8.9.2023)

- Mit `git status` können Sie immer abfragen, ob es noch Änderungen gibt, die noch nicht eingecheckt sind.

2 Arbeiten im lokalen Repository

Konfiguration

Jedes Repository hat eine Reihe von Konfigurationseinträgen. Diese können mit

```
git config --list
```

abgerufen werden.

Grundlegende Parameter können für alle Repositories übergreifend definiert werden. Diese werden mit der zusätzlichen Option `--global` bearbeitet.

Aufgabe:

- Betrachten Sie in Ihrem neu erstellen Repository die Einstellungen.
- Setzen Sie die folgenden globalen Einstellungen:

  ```
  git config --global user.name "Muster Mann"
  git config --global user.email "muster@eine-firma.de"
  ```

- Überprüfen Sie das Ergebnis.

Stagig-Area

Um Änderungen an bereits im Repository befindlichen Dateien festzuschreiben, müssen diese erst mit `git add DATEI` in die Staging-Area überführt werden. Mit `git status` können Sie das überprüfen.

Marken (tags)

Wie bei SVN sind Commits atomar, d.h.: Ist ein Commit erfolgreich werden alle Änderungen in das Repository übernommen. Gibt es Probleme beim Commit, bleibt das Repository unverändert. Während bei SVN Commit über ihre Commit-Nummer eindeutig sind, sind bei Git die Commits über sog. Hashwerte (z.B. `bb92a59411df457be7e8adb6a554562a4c92fb05`) eindeutig identifizierbar. Da diese HEX-Ungetüme wenig sprechend sind, können Sie Commits mit sprechenden Namen (Marken) versehen.

- `git tag MARKE` setzt eine Marke auf den aktuellen Arbeitsstand.
- Mit `git checkout MARKE` können Sie in Ihrem Arbeitsstand zu dem zur Marke gehörigen Softwarestand zurückkehren.

- `git branch` zeigt den Zustand Ihres Arbeitsstands an.
- `git tag` zeigt die im Repository befindlichen Marken an.
- `git checkout master|main` wechselt wieder auf *head*.
- Mit `git log` können Sie die zum Arbeitsstand passende Historie betrachten.
- `git tag -d MARKE` löscht eine Marke.

Aufgabe:

- Setzen Sie auf Ihren (eingecheckten) Arbeitsstand eine Marke.
- Führen Sie Änderungen durch und checken Sie diese ein.
- Wechseln Sie Ihren Arbeitsstand zwischen der Marke und *head*.
- Betrachten Sie sich die jeweiligen Historien.

Zweige (branches)

In Git müssen wir zwei Verzweigungswege unterscheiden: Wir können einerseits in weitere Repositories hinein verzweigen und die Repositories dann wieder synchronisieren, wie das in Abb. 2 angedeutet ist. Dies wird später im nächsten Kapitel näher betrachtet. Andererseits kann auch innerhalb eines Repositorys verzweigt werden. Dies soll nun näher betrachtet werden.

- `git branch BRANCH` legt einen neuen Zweig am aktuellen Arbeitsstand an.
- `git checkout BRANCH` wechselt den Arbeitsstand zwischen den Zweigen.
- `git branch` zeigt die Zweige im Repository an.
- `git checkout -b BRANCH MARKE` wechselt zu MARKE und erzeugt dort den Zweig BRANCH.
- `git branch -m Z1 Z2` benennt Zweig Z1 nach Z2 um.
- `git branch -d ZWEIG` löscht Zweig ZWEIG.
- `git merge ZWEIG` arbeitet Zweig ZWEIG in den aktuellen Arbeitsstand ein.

Aufgabe:

- Lassen Sie bei der zuvor gesetzen Marke einen Zweig beginnen.
- Führen Sie Änderungen im Zweig durch.
- Mergen Sie die Änderungen des *trunk* in den Zweig.
- Führen Sie weitere Änderungen im Zweig durch.
- Mergen Sie die Änderungen des Zweigs in den *trunk*.
- Provozieren Sie auch einen Konflikt. `git status` verrät auch, wie er aufzulösen ist.

3 Remote Repository

Wie in Abbildung 2 gezeigt, ist eine Git-Instanz eine Kombination aus Repository und Arbeitskopie. Wird eine Instanz nicht-interaktiv auf einem Server betrieben, kann es auch ohne Arbeitskopie angelegt werden. Dazu dient bei `init` oder `clone` die Option `--bare`.

Durch *continous integration* (CI)-Installationen wie Github/lab oder Bitbucket haben viele Arbeitsumgebungen doch wieder eine einzige Zentralinstanz bekommen.

In einer ersten Übung soll aber das bereits bestehende lokale Repository geklont werden.

- `git clone URL [DIR]` legt ein neues Repository im Verzeichnis `DIR` an. Es beinhaltet den im Stammrepository aktuellen Zweig mit seinen Marken. Ist `DIR` nicht gegeben, so heißt das Verzeichnis wie das Repository.
- `git remote -v` gibt Auskunft über das oder die verknüpften Repositorys.
- `git remote add NAME URL` verbindet mit einem weiteren Repository
- `git push NAME IDENT` synchronisiert Zweig oder Marke `IDENT` in das Repository `NAME`.
- `git fetch NAME IDENT` holt Zweig oder Marke `IDENT` in das lokale Repository. Dabei findet kein *merge* statt.
- `git pull NAME IDENT` holt Zweig oder Marke `IDENT` in das lokale Repository und arbeitet die Änderungen in den Arbeitsstand ein.

Anmerkung: Der Standardname für `NAME` ist `origin`, ist kein `IDENT` angegeben, wird der aktuelle Zweig verwendet.

Aufgabe:

- Klonen Sie das bestehende Repository.

  ```
  git clone REPOS REPOS1
  ```

- Prüfen Sie mit `git tag` und `git branch`, ob alle Marken und Zweige dabei sind.
- Rufen Sie in beiden Repositorys `git remote -v` auf.
- Machen Sie Änderungen im *trunc* des Stammrepositorys.
- Holen Sie sich die Änderungen einmal mit `git fetch origin master`.

  Die neuen Inhalte stehen nun unter den Bezeichnungen `FETCH_HEAD` und `origin/master` zur Verfügung.
- Mergen Sie die Änderungen in Ihre Arbeitskopie.
- Führen Sie erneut eine Änderung im Stammrepository durch.
- Holen Sie nun die Änderung über `git pull` ab.
- Führen Sie nun Änderungen im abgeleiteten Repository durch.

- Schieben Sie die Änderung mit `git push origin master` weiter.

  Anmerkung: Der Arbeitsstand des Stammrepositorys darf sich dabei nicht auf dem gleichen Zweig befinden, da Arbeitsstand und Repository sonst inkonsistent wären. Wenn Sie die entsprechende Fehlermeldung bekommen, können Sie mit `git checkout ZWEIG` den Arbeisstand des Stammrepositorys verändern.
- Falls vorhanden verbinden Sie Ihr abgeleitetes Repository mit einem zentralen Repository. Synchronisieren Sie sich auch mit diesem.

Zum Umgang mit Marken und Zweigen: Marken und Zweige werden nie implizit zwischen Repositorys synchronisiert. Die muss stets explizit über `fetch/push` geschehen. Hilfreich dabei:

- `git ls-remote --tags` zeigt die Tags im Stammrepository.
- `git fetch -t NAME ZWEIG` übernimmt auch alles Marken des genannten Zweigs.

Push vs. Backup

Beim Umgang mit einem Zentralrepository bestand (hoffentlich begründeter) Verlass, dass alles, was eingecheckt ist, auch sicher abgelegt ist. Datenverlust auf der eigenen Maschine konnte also nur die Änderungen seit dem letzen *checkin* betreffen.

Beim Umgang mit Git ist das grundlegend anders. Das Repository liegt auf der eigenen Maschine. Das hat zwar den Vorteil, dass es immer und überall verfügbar ist, hat aber den Nachteil, dass der Mitarbeiter[4] selbst für die Sicherung des Repositorys verantwortlich ist.

Eine Lösung dieses Problems kann in zwei Richtungen gehen:

- Ein „SVN-mäßige" Verwendung des Zentralrepositorys: Die Mitarbeiter *pushen* feingranular ihre Änderungen.
- Die Mitarbeiter haben eine unabhängige Möglichkeit zur Datensicherung. In das Zentralrepository gelangen nur solche Softwarestände, die auch für die Weitergabe im Team geeignet sind.

4 Git Vertiefung

4.1 Staging und Commit

Im Unterschied zu anderen Versionierungswerkzeugen kennt Git das *staging*. Dies ist ein Speicher, in dem sich die Dateien für einen *commit* befinden müssen. Dass es sich hierbei um einen echten Speicher handelt zeigt folgende Sequenz:

[4] m/f/d

```
echo "more text" >> file2
echo "123" > file3

git add file2 file3

echo " third line " >> file2
rm file3
```

Auf diese Weise wurde die Situation in Abb. 3 erzeugt.

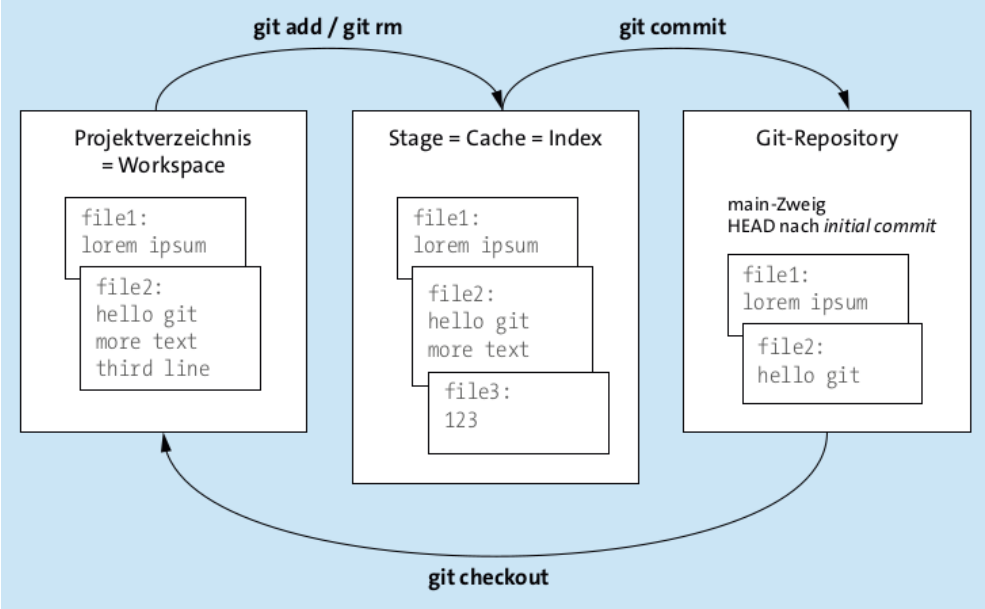

Abbildung 3: Projektverzeichnis, Stage und Repository[5]

Die einzelnen Inhalte lassen sich wie folgt abrufen:

```
cat file2           // Datei im Arbeitsverzeichnis
git show :file2     // Datei im staging
git show HEAD:file2 // HEAD-Version
```

Weitere Kommandos um das Commit:

- `git reset FILE` entfernt die Datei aus dem Stage-Bereich, ohne die Datei im Projektverzeichnis zu ändern.
- `git restore FILE` überschreibt die Arbeitskopie mit der *head*-Version ohne Rückfrage oder *undo*-Möglichkeit.
- `git restore -s HEAD~n FILE` überschreibt mit dem n-ten zurückliegenden Commit.
- `git commit --amend` lässt den letzten Commit-Eintrag nochmal editieren.

`restore + commit` lassen sich mit dem Kommando `git revert` zusammenfassen. Dieses Kommando versetzt die Arbeitskopie in den Zustand vor dem letzten Commit und checkt diesen erneut ein. Die Zwischenfassung geht also nicht verloren.

[5] Öggl 2022, S. 88.

Aufgabe:

Probieren Sie die genannten Befehle aus.

4.2 Weitere git-Operationen

`git mv` benennt eine bereits im Repository befindliche Datei um. Mit dem Kommando kann auch die Datei in ein anderes Verzeichnis verschoben werden. Beim nächsten Commit wird die Datei am neuen Ort gespeichert.

```
git mv OLDFILENAME NEWFILENAME
git mv FILE OTHER_DIR/
```

Es gibt kein vergleichbares Kommando, um eine Datei zu kopieren. Stattdessen wir die Datei mit den Mitteln des Betriebssystems kopiert und dann dem Repository hinzufefügt.

`git rm` löscht eine Datei sowohl aus dem Arbeitsverzeichnis als auch aus dem Repository.

Anzeigen und Vergleichen von Versionen aus dem Repository:

- `git show HEAD:FILE` zeigt die *head*-Version von `FILE`.
- `git show HEAD~n:FILE` zeigt die Version des n-ten zurückliegenden Commits.
- `git checkout HEAD~n -- FILE` setzt die Arbeitsversion auf die gewünschte Version zurück.
- `git diff FILE` vergleicht FILE mit seiner *head*-Version.
- `git diff HEAD~n FILE` vergleicht FILE mit einem entsprechend älteren Commit.

Aufgabe:

Probieren Sie die genannten Befehle aus.

4.3 Zweige - Merge

Zweige

> Sowohl die hohe Geschwindigkeit als auch die unkomplizierte Handhabung von Branches waren wichtige Entwicklungsziele für Git. Linus Torvalds wollte ein Werkzeug schaffen, das die Verwendung von Branches möglichst bequem und effizient macht und richtiggehend dazu einlädt, Branches zu nutzen. ...
>
> Git erlaubt auch, dass mehrere Mitarbeiter in ihren jeweils eigenen Repositories im gleichen Branch arbeiten. Beispielsweise verwenden die Entwickler A, B und C alle den Zweig develop , um neue Features zu programmieren. Durch git pull und durch git push werden die individuell durchgeführten Commits dann zusammengeführt. Die

> dabei ablaufenden Merge-Prozesse sind Git-intern dieselben, egal, ob zwischenzeitlich mehrere Zweige im Spiel waren oder nicht.[6]

Um das Arbeiten mit Zweigen möglichst einfach zu halten gibt es sie im Repository eigentlich gar nicht. Im Repository findet sich ein gerichteter Graph von Commits. Darüber hinaus gibt es ein kleines Verzeichnis welche Commits *heads* von aktiven Zweigen sind. Oder umgekehrt: Die Liste enthält die aktiven Zweige mit den Commits, die als *heads* dieser Zweige dienen.

Das Löschen eines Zweiges (`git branch -d`) löscht den entsprechenden Eintrag in dieser Liste. Die zugehörigen Commits bleiben selbstverständlich im System. Ausnahme: Verwaiste Commits. Das sind Commits, die weder einen Nachfolger im Commit-Graph haben, noch als *head* im genannten Verzeichnis geführt sind. Git betreibt hierfür einen *garbage collector*.

Merge

Wie das *merge* grundsätzliche aufgerufen wird, wurde schon gezeigt. Beim *merge* gibt es grundsätzlich zwei Varianten:

- *feature merge*: Der *merge* wird im Main-Zweig aufgerufen. Das implementierte *feature* wird in den Main-Zweig übernommen. Der *feature*-Zweig könnte im Anschluss gelöscht werden.
- *main merge*: Änderungen des Main-Zweigs werden in den *feature*-Zweig übernommen. Der Main-Zweig bleibt unverändert.

Desweiteren gibt es in Git die Möglichkeit gleich mehrere Zweige gleichzeitig zu mergen (*octopus merge*). Davon raten aber Autoren wie Bernd Öggl ab:

> Schon ein gewöhnlicher Merge-Prozess, der zwei Zweige zusammenführt, kann genug Probleme verursachen ... . Je mehr Zweige im Spiel sind, desto diffiziler wird die Behebung der anfallenden Konflikte oder Unklarheiten.
>
> Unser Tipp lautet: Führen Sie für jeden Zweig einen »einfachen« Merge durch! Der einzige Vorteil eines Octopus-Merges im Vergleich zu mehreren Einzel-Merge-Prozessen besteht darin, dass es – wenn alles gut geht – nur einen Merge-Commit gibt.
>
> Trotz all unserer Warnungen kommen Octopus-Merges in der Praxis tatsächlich vor. Der folgende Blogartikel hat das Git-Repository des Linux-Kernels durchsucht und etliche Octopus-Merges finden können, wobei einer beachtliche 65 Zweige zusammenführt. Aber nur weil Linus Torvalds und andere Kernel-Gurus nicht davor zurückschrecken, bedeutet das nicht, dass Sie auch so verfahren müssen![7]

Soll statt einem Zweig nur ein einzelner Commit eines anderen Zweiges in den aktuellen Zweig eingearbeitet werden, so nennt man dies *cherry picking*:

```
git cherry-pick HASH
```

[6] Öggl 2022, S. 102f.
[7] Öggl 2022, S. 114

Rebase

Rebase ist eine Variante des Merge, die die Abfolge der Commits, wie in Abbildung 4 gezeigt, künstlich umbaut. Während tatsächlich F1 und D1 auf *starting point* aufbauen, D2 auf D1 und F2 auf F1, sieht es nach einem Rebase so aus, dass nur D1 auf *starting point* aufbaut, F1 hingegen auf D2.

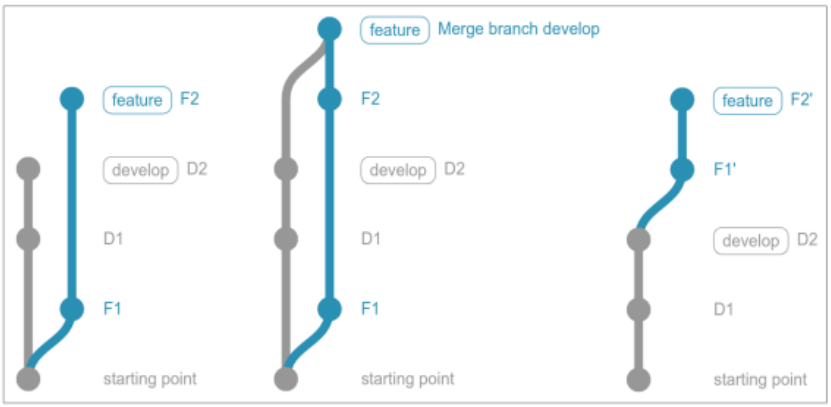

Abbildung 4: Änderung Commit-Graph bei Rebase[8]

Das kann natürlich unerwünschte Seiteneffekte haben: War beispielsweise der Codestand F1 *compile clean*, kann der künstlich erzeugte Stand F1' Compilefehler aufweisen.

Wie in Abschnitt 3 gezeigt, ist ein `pull` die Kombination aus `fetch` und `merge`. Will man beim `pull` einen `fetch` mit einem `rebase` kombinieren, kann dies auf zwei Weisen erfolgen:

- Einmalig mit dem Kommano `git pull --rebase`,
- immer durch das Setzten folgender Einstellung:
  `git config [--global] pull.rebase true`

Squash

Eine weitere Variante des Merge ist *squash* (Abb. 5. Hier wird der Commit-Graph dahingehend verändert, dass die Kante von B2 zum Mergepunkt in `main` nicht geschrieben wird. Dies hat folgende Seiteneffekte:

- Der *merge* ist im Graphen gar nicht als solcher erkennbar.
- Wird der Zweig *bug fix* gelöscht, entstehen mit B1 und B2 verwaiste Commits, die vom *garbage collector* aus dem System entfernt werden.

[8] Öggl 2022, S. 137.
[9] Öggl 2022, S. 142.

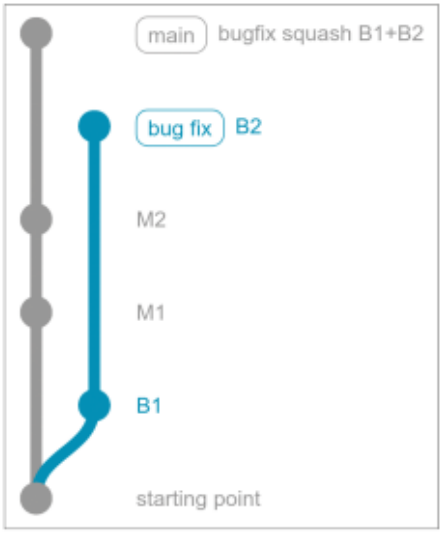

Abbildung 5: Änderung Commit-Graph bei Squash[9]

Konflikte

Bei einem Merge, welcher Art auch immer, kann es zu Konflikten kommen, wenn an einer Datei an derselben Stelle in beiden Zweigen was geändert wurde. Man kann aber Git zu Gute alten, dass es nicht so zickig wie SVN ist. Im Falle eines Konflikts legt Git drei Dateien an: Die Originaldatei, die Datei *to merge* und eine Version, die die Konflikte aufzeigt. Der Benutzer kann nun in letzterer den Konflikt auflösen und das Egebnis mit `git commit` festschreiben.

Alternativ kann man die Sache „rückabwickeln". Mit `git merge --abort` werden alle Änderungen rückgängig gemacht.

4.4 Stash

> Mit `git stash` können Sie aktuelle Änderungen im Arbeitsverzeichnis speichern, ohne einen Commit durchzuführen. »Wozu das?«, werden Sie vielleicht fragen. Schließlich speichert nur ein Commit Ihre Änderungen dauerhaft.
>
> *Stashing* (von *to stash*, verstauen, aufbewahren) ist dann praktisch, wenn Sie Ihre aktuelle Arbeit unterbrechen müssen und z. B. vorübergehend in einen anderen Zweig wechseln wollen, um dort rasch einen Bugfix durchzuführen. Ihr erst halbfertiges neues Feature wollen Sie aber nicht per Commit in das Repository aufnehmen, z. B. weil Sie befürchten, dass Ihre Änderungen bei den Teamkollegen Probleme verursachen könnten.[10]

Nach dem Rückwechseln lassen sich die Änderungen mit `git stash pop` wieder zurückholen.

Hier sieht man wieder, dass Git von Praktikern gemacht ist. Fehler im Vorgehen müssen nicht irgendwie umständlich und fehleranfällig gerade gezogen werden. Git gibt uns hier passende Werkzeuge. Zur Wiederholung:

[10] Öggl 2022, S. 116.

- Der Entwickler hat vergessen den Zweig zu wechseln: `git checkout` übernimmt bereits gemachte Änderungen in den Zielzweig.
- Sind die Änderungen zu tiefgreifend, so dass sich der `checkout` nicht durchführen lässt, können die Änderungen erst in den *stash* geschoben werden, dann wird der Zweig gewechselt, dann die Änderungen übernommen uns allfällige Konflikte behoben.
- Der Entwickler muss schnell mal ein einen anderen Zweig ohne die Änderungen mitzunehmen: `git stash` schiebt Änderungen auf einen Stapelspeicher. Anch der Rückkehr in den Ausgangszweig holt er sich die Änderungen wieder.

Eine Alternative für das letzte Beispiel ist es, das Repository erneut zu klonen und die gewünschten Änderungen in der weiteren Instanz durchzuführen. Wie schon zuvor angemerkt: Wir haben beim Arbeiten mit Zweigen immer die Möglichkeit in einem Repository zu verzweigen oder ein weiteres Repository als Zweig zu verwenden.

Aufgabe:

- Führen Sie einen *cherry pick* durch.
- Erzeugen Sie einen Konflikt und lösen Sie ihn wieder auf.
- Führen Sie ein Beispiele mit *stashing* durch (vergessen, den Zweig zu wechseln / temporäres Arbeiten in einem anderen Zweig)
- Führen Sie das letzte Beispiel durch Erstellung einer weiteren Repository-Instanz durch.

4.5 Hooks

> Ein *Hook*, vom englischen Wort für Haken, bezeichnet in Git ein Script, das ausgeführt wird, wenn ein gewisses Ereignis in Ihrem Git-Repository eingetreten ist.
>
> Hooks sind keine Erfindung von Git, auch andere Versionskontrollsysteme haben und hatten schon vor Git ein sehr ähnliches Konzept. Da Hooks auch ausgeführt werden können, bevor eine Aktion eintritt (zum Beispiel bevor ein Commit akzeptiert wird), werden sie gerne verwendet, um gewisse Richtlinien oder einen gewissen Stil einzufordern.[11]

In einem Repository findet sich unter dem `.get`-Verzeichnis das Verzeichnis `hooks`. Darin sind Beispiele für *hook scripts* für Linux-Systeme. In Windows muss erst eine geeignete Interpretersprache vorhanden sein (z.B. git-bash).

[11] Öggl 2022, S. 291.

4.6 Submodule und Subtrees

Submodule

In ein bestehendes Repository kann eine Referenz auf ein anderes Repository integriert werden. Dies geschieht mit dem Kommando

```
git submodule add URL DIR
```

Das mit `URL` referenzierte Repository wird in das Verzeichnis `DIR` geklont. Im Arbeitsbereich des äußeren Projekts wird eine Referenz auf genau diesen Commitstand angelegt.

Wird eine Änderung im Submodul gemacht, ...

- muss diese im Verzeichnis des Submoduls committet werden,
- ändert sich die Referenz im äußeren Projekt,
- muss auch diese Änderung des äußeren Projets committet werden.

Wird ein Repository mit einem Submodul geklont und soll dieses Submodul auch gleich geklont werden, erfolgt dies mit

```
git clone --recurse-submodules URL
```

Erkennt man erst nach einem Klon, dass man vergessen hat, die Submodule gleichfalls zu klonen, kann das Klonen des Submoduls nachträglich mit

```
git submodule update --init --recursive
```

im zuvor mitangelegten leeren Verzeichnis erfolgen.

Subtrees

Eine andere Variante ist die Verwendung von *subtrees*. `git subtree` ist kein Kern-Kommando, sondern eine Erweiterung, die in den meisten Git-Installationen enthalten ist. `git subtree` sorgt für eine nahtlosere Integration. So wird das im Subtree enthaltene Repository mit aus dem äußeren bedient. Ein Commit schreibt also Änderungen in beiden Repositorys fest. Das hat aber folgende Nebenwirkungen:

- Wird das Repository, das den Subtree enthält wieder geklont, so erkennt dieser Klon nicht mehr, woher die Daten eigentlich stammen.
- Änderungen in dem geklonten Repository landen bei einem `push` im Stammrepository des Hauptprojekts. Eine Verbindung zum Basisprojekts des *subtrees* gibt es nicht mehr.
- Der Code des Unterprojekts wird tatsächlich dem äußeren Repository hinzugefügt. Es handelt sich um eine Art *merge*.

Hier die Syntax:

- `git subtree add --prefix=DIR URL [--squash]` fügt das Repository aus `URL` in der Verzeichnis `DIR` ein.

- Die zusätzliche Option `--squash` bewirkt, dass nicht auch noch die Commit-Historie aus dem *tree*-Projekt in das äußere Projekt übernommen wird.
- `git subtree pull --prefix=DIR URL [--squash]` aktualisiert den *subtree*.
- `git subtree push` schiebt Änderungen ins *subtree*-Repository

5 Datenanalyse

Commits anzeigen

- `git log` zeigt die Commits mit Metadaten und Commit-Kommentar des aktuellen Zweiges an.
- `--all` erweitert die Anzeige auf alle Zweige.
- Mit der Option `--grep` lässt sich in den Meldungen suchen.
- Mit der Option `-i` findet die Suche *case insensitive* statt.
- `git log -- PFAD` zeigt die Commits der durch `PFAD` spezifizierten Dateien.
- `git log --oneline --all --graph` zeigt alle Commits mit Commit-Graph.

Differenzen anzeigen

- `git diff [PFAD]` zeigt die Unterschiede der durch `PFAD` spezifizierten Dateien zwischen Arbeitskopie und *top*.
- `git diff MARKE [PFAD]` zeigt die Unterschiede zwischen Arbeitskopie und `MARKE`.
- `git diff M1..M2 [PFAD]` zeigt die Unterschiede zwischen den Marken `M1` und `M2`.

Werden statt Marken die Namen von Zweigen angegeben, so wird mit der *top*-Version des angegebenen Zweiges verglichen.

Interessiert nur welche Dateien sich unterscheiden, kann zusätzlich die Option `--stat` angegeben werden.

Urheberschaft

Mit `git blame PFAD` werden alle Zeilen der durch `PFAD` spezifizierten Dateien zusammen mit Commit und Autor angezeigt.

6 Git-Plattformen

Wenn auch die Idee eines verteilten Konfigurationsmanagement-Systems ist, dass die einzelnen Instanzen, wie in Abb. 2 gezeigt, gleichwertig sind, hat sich durch Github et al. gezeigt, dass zentrale Instanzen auch ihre Vorteile haben. Während der Linux-Kernel immer noch auf einem durch Linus Torwalds betriebenen Server zusammen getragen wird, nutzen viele andere Projekte die Vorzüge der genannten Plattformen.

Diese Plattformen bieten neben der reinen Git-Funktionalität Dienste für Build, Test und Auslieferung. Um Code zu einem solchen Projekt beizutragen, unterscheiden sich die Wege, je nach dem, wie privilegiert der jeweilige Entwickler in dem Projekt ist.

Entwickler mit Vollzugriff legen einen Zweig an, bringen dort ihre Änderungen ein. Wenn diese Änderungen für gut befunden werden, werden diese in den *trunc* übernommen.

Entwickler, die zwar zu dem Projekt beitragen dürfen, aber keinen Zugriff auf den *trunc* haben (*committer*), legen für sich einen Zweig an und bringen dort ihre Änderungen ein. Danach stellen sie einen *pull request*. Werden die Änderungen für gut befunden, können diese von ersteren in den *trunc* übernommen werden.

Personen, die nicht zum engeren Projektkreis gehören, also im Grunde jedermann, kann das Projekt *forken*, d.h. innerhalb der Plattform eine zweite Instanz des Projekts anlegen. Hier ist nun der Entwickler selbst Herr über den Code und kann ggf. zusammen mit weiteren Entwicklern das Projekt im eigenen Sinne weiterentwickeln. Für Änderungen, die er in das Stammprojekt zurückfließen lassen will, kann er beim Stammprojekt einen *pull request* stellen. Die Entwickler des Stammprojekts können dann die Änderungen in einen Zweig nehmen, testen und falls die Änderungen für gut befunden werden, in den *trunc* übernehmen.

Bitbucket

> Mit Bitbucket gibt es noch einen weiteren wichtigen Player am Markt der Cloud-Git-Hosting-Lösungen. Bereits 2008 präsentierte Atlassian, die Firma hinter Bitbucket, die Software im Internet. Mit dem rasanten Aufstieg von GitHub verlor Bitbucket zusehends an Bedeutung, bleibt aber für Kunden, die ohnehin schon andere Produkte von Atlassian verwenden, eine gute Alternative. Hier werden vor allem Jira, eine sehr weitverbreitete Issue-Tracking-Software, und Confluence, eine Dokumentationssoftware auf Wiki-Basis, eine Rolle spielen.[12]

7 Quellen

Öggl 2022 Bernd Öggl, Michael Kofler, Git. Projektverwaltung für Entwickler und DevOps-Teams, Rheinwerk 2022

[12] Öggl 2022, S. 250.

Vielen Dank für Ihre Aufmerksamkeit

Ihr Referent und das Team von
IT-Schulungen.com

Fon +49 (0) 911 650 08 - 30
Fax +49 (0) 911 650 08 - 399
Mail info@it-schulungen.com
Web www.it-schulungen.com

Education Center der New Elements GmbH
Thurn-und-Taxis-Straße 10
90411 Nürnberg

www.newelements.de