

Versionsverwaltung - GIT

SYP (PRE)

5AHIFS

2018/2019

Julian BLASCHKE

Armin HARACIC

Inhaltsverzeichnis

[1 Allgemein 3](#_Toc535415577)

[2 Verteiltes Versionskontrollsystem - Git 5](#_Toc535415578)

[2.1 Entstehung 5](#_Toc535415579)

[2.2 Name 5](#_Toc535415580)

[2.3 Speicherung von Daten 6](#_Toc535415581)

[2.4 Git - Operationen 7](#_Toc535415582)

[2.5 Datenintegrität 7](#_Toc535415583)

[2.6 Dateizustände und Arbeitsbereiche 7](#_Toc535415584)

[2.7 Arbeitsprozess 8](#_Toc535415585)

[2.8 Branching 9](#_Toc535415586)

[2.9 Merging 9](#_Toc535415587)

[2.10 Workflow 10](#_Toc535415588)

[2.11 Best Practice 11](#_Toc535415589)

[2.12 Wichtigste Befehle 11](#_Toc535415590)

[3 Git Server 13](#_Toc535415591)

[4 Example 14](#_Toc535415592)

[4.1 Task 14](#_Toc535415593)

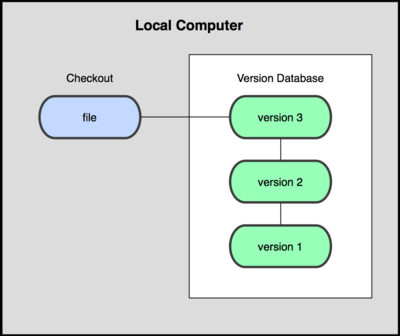
[4.2 Solution 14](#_Toc535415594)

# Allgemein

**Versionskontrollsysteme** (**VCS**) protokollieren Änderungen an einer oder mehreren Dateien über einen bestimmten Zeitraum hinweg. Vor allem im Bereich der Softwareentwicklung ergeben sich dadurch viele Vorteile und daher wird dieses Konzept sehr verbreitet eingesetzt. Ein wesentlicher Vorteil ist, dass man zu jedem Zeitpunkt auf vorherige Versionen der Dateien zugreifen kann und dadurch Änderungen rückgängig machen kann. Besonders bei Problemen mit dem Softwarequellcode ist es essentiell auf einen funktionierenden Stand zurück kommen zu können.

Es gibt mehrere Möglichkeiten eine Versionskontrolle durchzuführen. Die einfachste und zugleich fehleranfälligste ist, die Dateien lokal zu kopieren und dadurch zu sichern. Viele Leute verwenden diese Methode, wovon jedoch abzuraten ist. Es ist umständlich und bei Unachtsamkeit kann es zu Verlusten kommen. Auch die Zusammenarbeit von mehreren Personen an einem Projekt ist nicht problemlos möglich und daher sollte man ein **Versionskontrollsystem** verwenden. Grundsätzlich gibt es **lokale Versionskontrollsysteme, zentralisierte Versionskontrollsysteme** (**CVCS**) und **verteilte Versionskontrollsysteme** (**DVCS**).

**Lokale Versionskontrollsysteme** wurden damals entwickelt, um die Arbeit mit Dateiversionen zu erleichtern und sicherer zu machen. Dabei werden die Änderungen der jeweiligen Dateiversion in einer lokalen Datenbank gesichert. Das **Revision Control System** (**RCS**) war ein sehr beliebtes **lokales Versionskontrollsystem** und wird heutzutage immer noch auf vielen **UNIX**-basierten Systemen ausgeliefert.



**Zentralisierte Versionskontrollsysteme** (z.B. **Subversion - SVN**) basieren auf einem zentralen Server, der alle Dateien verwaltet. Dieses Konzept bietet eine umfangreichere Kontrolle über die Arbeit der Benutzer, da das System zentral administriert wird. Einen erheblichen Nachteil stellt der „**Single Point of Failure**“ durch den zentralen Server dar. Bei einem Ausfall des Servers kommt es zum Stillstand, da kein Datenaustausch zwischen Client und Server stattfinden kann. Im schlimmsten Fall droht der Datenverlust.



**Verteilte Versionskontrollsysteme** (z.B. **Git** oder **Mercurial)** arbeiten ähnlich wie ein **zentralisiertes Versionskontrollsystem**. Der Unterschied ist jedoch, dass jeder Benutzer eine Kopie des gesamten Repositorys besitzt. Sollte der Server wiederhergestellt werden müssen, so kann jede beliebige lokale Kopie des Repositorys dazu verwendet werden.



# Verteiltes Versionskontrollsystem - Git https://git-scm.com/images/logos/downloads/Git-Logo-Black.png

## Entstehung

**Git** ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die von **Linus Torvalds** veranlasst wurde, da das vorher genutzte **BitKeeper**–System durch eine Lizenzänderung nicht mehr kostenlos genutzt werden konnte. Die Entwicklung der neuen **Quellcode-Management-Software** begann im **April 2005**.

Torvalds Anforderung an das neue verteilte System waren:

* Unterstützung verteilter Arbeitsabläufe
* Sehr hohe Sicherheit gegen Verfälschung
* Hohe Effizienz

Torvalds hatte die Möglichkeit das Projekt **Monotone** an seine Anforderungen anzupassen, da dieses bereits zwei von drei Anforderungen entsprach. Jedoch entschied sich Torvalds gegen diese Idee und begann ein eigenes System zu entwickeln. Mit folgender Aussage argumentierte er seine Entscheidung.

„Ich habe nur ein wirkliches konzeptionelles Problem mit ‚monotone‘: Ich mag die Arbeitsweise, aber sie erschwert die Nutzung von Wegwerf-Bäumen durch die scheinbare Arbeitsmethode ‚eine Datenbank je Entwickler‘ statt ‚eine Datenbank je Baum‘. Man muss zwar nicht diesem Model folgen, aber das Setup scheint darauf ausgerichtet zu sein. Zusammen mit ihren ‚Zweigen‘ befürchte ich ein schnelles Verdrecken der monotone-Datenbank. Ein anderes, hoffentlich nicht zu grundlegendes Problem, ist die derzeitige Leistungsfähigkeit von monotone.“

Die Gestaltung von **Git** bedient sich an den Ideen von **Monotone** und **BitKeeper**, jedoch nicht an deren Quellcodes, da es ein eigenständiges Versionsverwaltungssystem werden sollte.

## Name

Der Name „**Git**“ bedeutet in der britischen Umgangssprache so viel wie „Blödmann“. Linus Torvalds erklärte seine Wahl des ungewöhnlichen Namens mit einem Witz, sowie damit, dass das Wort praktisch ist und in der Softwarewelt noch weitgehend unbenutzt war:

„Ich bin ein egoistisches Arschloch und ich benenne all meine Projekte nach mir. Zuerst ‚Linux‘, jetzt eben ‚Git‘.“

„Der Witz ‚Ich benenne alle meine Projekte nach mir, zuerst Linux, nun eben Git‘ war einfach zu gut, um ihn nicht zu machen. Aber es (der Befehl) ist auch kurz, einfach auszusprechen und zu schreiben auf einer Standardtastatur, dazu einigermaßen einzigartig und kein gewöhnliches Standardkommando – sehr ungewöhnlich.“

## Speicherung von Daten

Die Schnittstelle von **Git** ähnelt den Schnittstellen anderer Versionskontrollsysteme, jedoch unterscheidet sich die interne Dateiverwaltung grundlegend. Anstatt eine fortlaufende Liste von Änderungen („**Diff“**) zu speichern werden sämtliche Daten zum Zeitpunkt eines **Commits** gespeichert („**Snapshot**“). Um bei diesem Vorgehen effizient zu bleiben wird für unveränderte Dateien lediglich eine Verknüpfung auf die vorherige Version der Datei erzeugt.





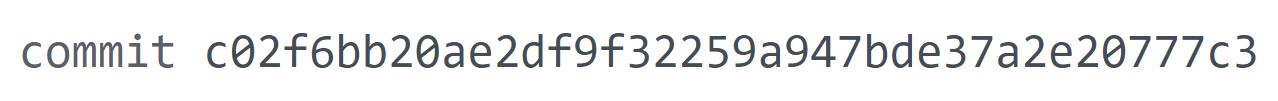
## Git - Operationen

Die meisten Operationen laufen lokal auf dem eigenen Rechner ab und durch den Entfall der Netzwerkwartezeiten bietet dies einen großen Geschwindigkeitsvorteil gegenüber einem **zentralisieren Versionskontrollsystem**. Um zum Beispiel die Historie eines Projektes durchzusuchen, braucht Git keinen externen Server, sondern verwendet die lokalen Ressourcen, wobei nur sehr geringe Verzögerungen entstehen. Dies gilt auch beim Suchen von Dateien oder bestimmen von Unterschieden. Dadurch ist Git sehr flexibel und nicht netzwerkabhängig. Bei anderen Systemen (z.B. **Subversion**) kann man zwar Dateien ohne Verbindung zum Server lokal ändern, jedoch kann man die Änderungen nicht sichern.

Fast alle Git - Operationen, fügen jeweils nur Daten zur internen **Git - Datenbank** hinzu. Im Vergleich zu anderen Systemen ist es in Git fast unmöglich Daten zu verlieren oder unbrauchbar zu machen. Einen einmal gespeicherten Snapshot zu verlieren ist auch sehr unwahrscheinlich.

## Datenintegrität

Git stellt die **Datenintegrität** sicher, indem vor dem sichern Checksummen der Änderungen gebildet werden. Die Änderungen werden anschließend anhand der Prüfsummen referenziert. Dies bietet den Vorteil, dass Git jegliche Änderung in der Dateistruktur bemerkt und dadurch keine Informationen bei der Übertragung unbemerkt verloren gehen können oder unwissentlich beschädigte Dateien verwendet werden. Zur Bildung einer Checksumme wird der **SHA-1** Algorithmus verwendet, welcher eine 40 Zeichen lange Zeichenkette, bestehend aus Hexadezimalzeichen, bildet.



## Dateizustände und Arbeitsbereiche

Eine Datei innerhalb eines Git-Systems kann einen der drei folgenden Zustände annehmen.

* **Modified**: Die Datei enthält nicht gesicherte Änderungen
* **Staged:** Die Datei wurde für den nächsten Commit vorgemerkt
* **Committed**: Die Datei wurde in der lokalen Git-Datenbank gesichert

Zusätzlich zu den Zuständen einer Datei gibt es drei Arbeitsbereiche in denen sich diese befinden.

* Git-Verzeichnis:

Dieser Bereich ist der wichtigste innerhalb des Git-Systems und wird beim Klonen eines Repositorys kopiert. Hier befinden sich die lokale Git-Datenbank und die Metadaten.

* Arbeitsverzeichnis:

In diesem Bereich befindet sich ein Abbild einer spezifischen Version eines Projektes. Die Daten werden aus der komprimierten Datenbank geholt und in einer modifizierbaren Form bereitgestellt.

* Staging Area:

Dieser Bereich wird auch als Index bezeichnet und hier befinden sich die vorgemerkten Dateien für den nächsten Commit.

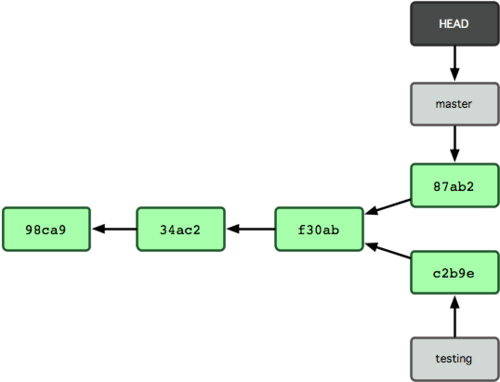


## Arbeitsprozess

Der grundlegende Arbeitsprozess mit Git beginnt mit der Bearbeitung der Dateien im Arbeitsverzeichnis. Nach Abschluss der Arbeit an einer Datei wird sie in der „Staging Area“ für den nächsten Commit vorgemerkt. Abschließend werden die vorgemerkten Dateien mittels Commit in der lokalen Git-Datenbank gesichert.

## Branching

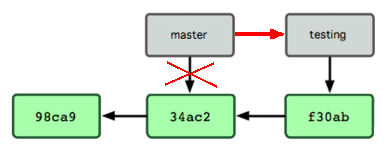
Branching bedeutet, dass vom Hauptentwicklungszweig eine Abzweigung erstellt wird, damit unabhängig von dem Hauptzweig weitergearbeitet werden kann. Zweige können einfach nach Belieben erstellt und entfernt werden, da sie nur kleine Dateien sind. Bei jedem Commit wird ein sogenanntes Commit-Objekt gespeichert, welches einen Zeiger, der sich auf den Schnappschuss, die Commit-Metadaten und den Elternteil des Commits richtet, enthält.



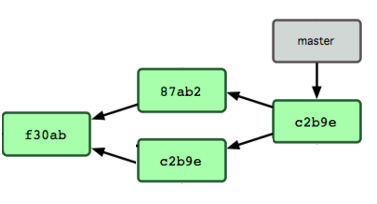
## Merging

Bei der Zusammenführung zweier Zweige, spricht man von einem sogenannten Merge. Davon existieren zwei verschiedene Arten:

* **Fast-Forward Merge:** Dies geschieht, wenn der neue Commit direkt vom ursprünglichen Commit ausgeht. Dabei wird von Git nur der Zeiger weiterbewegt.



* **3-Wege Merge:** Dies wird verwendet, wenn die Entwicklungshistorie beider Zweige auseinanderläuft. Anstatt einfach den Zeiger weiterzubewegen, wird ein neuer Snapshot erstellt, der aus dem Merge resultiert und einen neuen Commit erzeugt. Sollte dieselbe Datei in unterschiedlichen Zweigen geändert worden sein, so können Merge-Konflikte entstehen. In diesem Fall kann Git die Zweige nicht automatisch zusammenführen und der Konflikt muss manuell aufgelöst werden.



## Workflow

* Langfristige Branches:

Neben dem „Master“-Branch, auf welchen sich nur stabiler Code oder Code der veröffentlicht werden kann befindet, gibt es andere Branches zum Arbeiten oder Testen. Erreicht der parallele Zweig einen stabilen Status, kann er mit dem „Master“ zusammengeführt werden.

* Themen-Branches:

Ein Themen-Branch ist ein kurzlebiger Ast der für eine spezielle Aufgabe erstellt und benutzt wird. Diese Branches werden nach dem Zusammenführen mit anderen Zweigen wieder gelöscht. Diese Technik erlaubt es einem schnell und vollständig den Kontext zu wechseln. Da die Arbeit in verschiedene Depots aufgeteilt ist, in denen alle Änderungen unter die Thematik dieses Branches fallen, ist es leichter nachzuvollziehen was bei Code-Überprüfungen und ähnlichem geschehen ist.

## Best Practice

* **Commit für zusammengehörige Änderungen**: Man sollte kleine Commits machen damit die Teammitglieder die Änderungen einfacher verstehen und diese bei Fehlern einfacher rückgängig machen können. Zum Beispiel sollten für zwei verschieden Bugs zwei separate Commits durchgeführt werden.
* **Regelmäßige Commits**: Die Commits sollen klein und verständlich bleiben. Dadurch werden auch Merge-Konflikte vermieden.
* **Keine unfertigen Commits**: Dabei soll nur funktionsfähiger und getesteter Code commited werden. Falls eine saubere Arbeitskopie benötigt wird, kann man seine Änderungen am Stash festhalten.
* **Sinnvolle Commit-Nachrichten**: Diese sollten eine Kurzfassung und eine genaue Beschreibung der Änderungen beinhalten. Es soll der Imperativ und die Gegenwart benutzt werden.
* **Kein Backup-System**: Ein Backup ist nur ein Seiteneffekt einer Versionsverwaltung.
* **Workflow**: Das Team sollte sich auf einen Workflow einigen.

## Wichtigste Befehle

Git kann über jede Kommandozeilenanwendung ausgeführt werden. Dies sind wichtigsten Befehle die man benötigt um Git benutzen zu können:

**Erstellen** (Repository wird im aktuellen Verzeichnis erstellt)**:**

git init

**Klonen** (z.B. von GitHub)**:**

git clone https://github.com/REPOSITORY

**Status Anzeigen:**

git status

**Zweige Anzeigen:**

git branch

**Zweig erstellen:**

git branch „newBranch“

**Zweig wechseln:**

git checkout „newBranch“

**Zweig löschen:**

git branch -d „newBranch“

**Änderungen festhalten:**

git add „Datei“ oder git add \*

**Änderungen festschreiben:**

git commit oder git commit -m “Commit message“

**Änderungen von Repository laden** (falls ein remote Repository genutzt wird):

git pull

**Änderungen an Repository senden** (falls ein remote Repository genutzt wird):

git push

# Git Server

Um eine Zusammenarbeit zu ermöglichen benötigt man ein gemeinsam genutztes Repository. Es ist technisch möglich, direkt mit dem lokalen Repository anderer Personen zu arbeiten, was aber nicht empfehlenswert ist. Daher bietet sich ein Server, auf dem ein externes Repository für alle Benutzer existiert, an. Für das hosten von Git Repositorys sind nur minimale Ressourcen notwendig und in den meisten Fällen ist kein eigener dezidierter Server notwendig. Es ist auch möglich Git Repositorys von externen Anbietern hosten zu lassen, wobei dies die Einrichtung und den Betrieb eines eigenen Servers hinfällig macht.

Es gibt mehrere externe Hosting-Anbieter wie zum Beispiel **Bitbucket**, **GitHub** oder **GitLab**. GitHub ist die mit Abstand größte Open-Source Git Hosting Plattform und bietet als eine der wenigen, sowohl öffentliche als auch private Hosting-Optionen. Daher kann man Open-Source Code und proprietären Code auf einer einzelnen Plattform verwalten. Die Projekte werden anders als bei anderen Anbietern nach Benutzern gruppiert. Wenn ein Anwender ein geschütztes privates Repository benötigt so ist dies kostenpflichtig, wodurch sich die Plattform finanziert. Jeder Benutzer kann unlimitiert kostenlose, öffentliche Repositorys besitzen.

|  |  |  |
| --- | --- | --- |
| C:\Users\Marco\Downloads\Bitbucket@2x-blue.png | C:\Users\Marco\Downloads\wm_no_bg.png | C:\Users\Marco\AppData\Local\Temp\7zE42DBFD66\GitHub_Logo.png |

# Example

## Task

You work at a software company and Mr Santner is your boss. Some days ago he visited a software engineering conference and heard about the Version Control System Git. He was fascinated by the features provided by Git and now wants you to learn how to use Git properly.

He already created a GitHub repository at <https://github.com/mrsantner/myfirstrepository> and wants you to do the next steps. Clone the repository, create a branch called “Development” and check out this branch. Furthermore, add a file to the repository, commit the changes, merge the new branch into master and push it to GitHub.

## Solution

|  |  |
| --- | --- |
| git clone <https://github.com/mrsantner/myfirstrepository> | Clones the repository from GitHub |
| git branch development | Creates a branch called “development” |
| git checkout development | Switches to the branch “development” |
| git add helloworld.java | Adds the file “helloworld.java” to the staging area |
| git commit -m "add helloworld.java" | Commits the files in the staging area including the specified message |
| git merge development | Merges the “development” branch into “master” |
| git push | Pushes the commits to the repository on GitHub |