**Git** ist eines der bekanntesten Software-Tools zur **verteilten Versionsverwaltung**. Mit einer Versionsverwaltung können Teams nicht nur gemeinsam an einem Softwareprojekt, sondern auch einzelne Softwareentwickler an verschiedenen Orten zur selben Zeit an demselben Quelltext arbeiten. Die vollständige Dokumentation zu Git und einige Videos findet man online unter https://git-scm.com/doc. Wenn man tiefer in die Materie einsteigen und weitere Funktionen von Git kennenlernen möchte, findet man unter https://git-scm.com/book/ eine Online-Version des englischsprachigen Buchs „Pro Git“ (Autoren: Scott Chacon und Ben Straub) sowie eine deutsche Übersetzung davon.
Die Firma Atlassian (Anbieter der Plattform Bitbucket) stellt außerdem verschiedene englischsprachige Tutorials bereit (unter https://www.atlassian.com/git/tutorials).

**Info-Block**

Git ist eines der bekanntesten Software-Tools zur verteilten Versionsverwaltung. Mit einer Versionsverwaltung kann man gemeinsam im Team an einem Softwareprojekt arbeiten. Die Quelltexte werden dazu in einem Repository abgelegt. Ein Repository ist ein Verzeichnis in einem speziellen Format. Es enthält den Quelltext und einige zusätzliche Informationen, mit denen der Versionsverlauf – also die Geschichte des Quelltextes – gespeichert ist.

So kann man genau nachverfolgen, wer wann welche Änderungen an den Dateien des Projekts durchgeführt hat, auf einzelne Zwischenstände der Software zugreifen und (versehentliche) Änderungen wieder rückgängig machen. Mit verschiedenen Tools erleichtern Versionskontrollsysteme auch das Zusammenführen von Änderungen – teilweise sogar automatisiert – und bieten Unterstützung für das parallele Arbeiten in unterschiedlichen Entwicklungszweigen.

Im Laufe der Zeit haben sich drei verschiedene Typen von Versionsverwaltungssystemen (engl. Version Control Systems, VCS) etabliert:

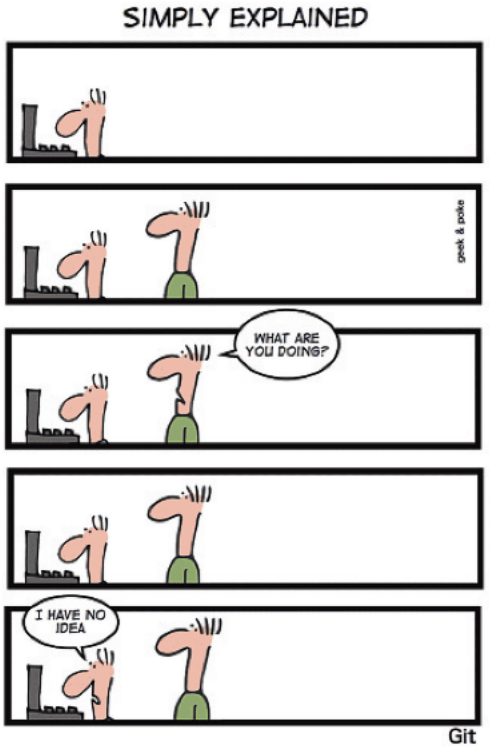


*Abb. 7.1: Git ist ganz einfach?!*

- Ein **lokales Versionsverwaltungssystem** speichert verschiedene Versionen der Quelltexte zwischen, allerdings nur auf dem lokalen Datenträger. Alle Benutzer verwenden eigene Versionsstände (s. Abb. 7.2). So können (versehentliche) Änderungen rückgängig gemacht werden – allerdings eignen sich diese Systeme nicht für die gemeinsame Arbeit mehrerer Personen und werden für die Teamarbeit nicht mehr eingesetzt.


*Abb. 7.2: Lokales Versionskontrollsystem*

- Bei einem **zentralen Versionsverwaltungssystem** liegen alle Quelltexte und Versionsstände auf einem zentralen Server, auf den die Entwickler Zugriff benötigen. Ein Entwickler checkt eine Arbeitskopie auf den lokalen Arbeitsplatz aus (lädt also einen bestimmten Versionsstand herunter) und checkt die geänderte Datei nach dem Bearbeiten wieder ein. In der Arbeitskopie ist jedoch nur ein Versionsstand gespeichert. Zum Sichern des Versionsstands ist also immer eine Verbindung zum zentralen Repository erforderlich, und auch die Historie ist nur über das zentrale Repository einsehbar. Bei einem zentralen Versionsverwaltungssystem werden somit die Versionsstände vom Server verwaltet, auf dem lokalen Arbeitsplatz befindet sich jeweils eine bestimmte Version als Arbeitskopie (s. Abb. 7.3).

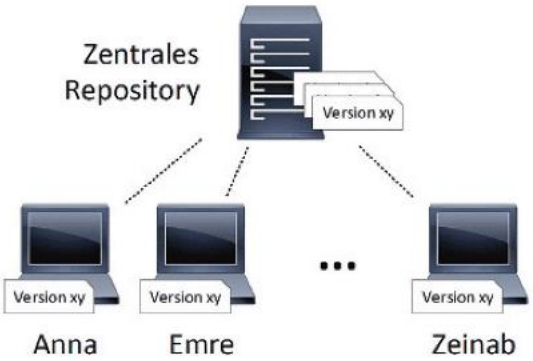


*Abb. 7.3: Zentrales Versionskontrollsystem*

- Ein **verteiltes Versionskontrollsystem**, engl. Distributed Version Control System (**DVCS**), verzichtet auf ein zentrales Repository. Jeder Entwickler hat ein eigenes Repository auf dem lokalen Arbeitsplatz, mit der vollständigen Historie des gesamten Projekts. So können auch offline verschiedene Versionsstände eingesehen und Änderungen gepflegt werden. Um dennoch einen organisierten Austausch zwischen den Entwicklern zu ermöglichen, wird oft ein gemeinsames Online-Repository im Firmennetzwerk oder bei einem spezialisierten Anbieter gepflegt. Der größte Vorteil eines verteilten Versionskontrollsystems ist, dass beim täglichen Arbeiten die verschiedenen Programmfunktionen unabhängig von anderen Entwicklern implementiert werden können. Alle Entwickler aus dem Projektteam können eigene Zwischenversionen verwalten und die Änderung dann wieder in die Hauptversion einfließen lassen. Erst wenn eine neue Funktion vollständig getestet und implementiert ist, wird sie zurück in das gemeinsame Online-Repository gespielt und in die aktuelle Programmversion integriert. So können evtl. auftauchende Fehler in anderen Funktionen den eigenen Entwicklungsprozess nicht stören.
Bei einem verteilten Versionskontrollsystem befindet sich somit auf jedem Arbeitsplatz eine vollständige Kopie aller Versionsstände. Zum vereinfachten Austausch wird üblicherweise ein gemeinsames Online-Repository verwendet (s. Abb. 7.4).

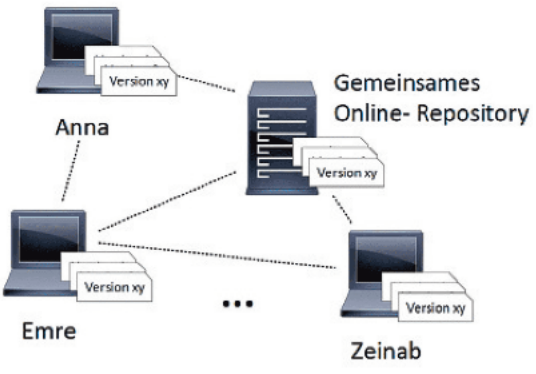


*Abb. 7.4: Verteiltes Versionskontrollsystem (DVCS)*

Besonders im Zusammenhang mit agilen Entwurfsmethoden kann ein verteiltes Versionskontrollsystem seine Stärken ausspielen. Die bekanntesten Vertreter der verteilten Versionskontrollsysteme sind u. a. Git, mercurial und bazaar. Viele bekannte Open-Source-Projekte verwenden eines dieser drei Systeme. Nicht zuletzt durch die populäre Plattform Github[1] hat sich aber Git als am weitesten verbreitetes System etabliert. Auch weitere Anbieter wie Gitlab[2] oder Atlassian BitBucket[3] setzen Git als zugrunde liegende Technologie für ihre Plattformen ein.

### 7.1.1 Git-Überblick und Grundbegriffe

Die Daten liegen bei einem typischen Workflow mit Git auf einem gemeinsamen Repository, auf das alle beteiligten Entwickler Zugriff haben. Dazu ist es für alle Beteiligten entweder über das Internet oder das lokale Firmennetz erreichbar. Außerdem hat jeder Entwickler eine lokale Kopie, das lokale Repository, auf dem eigenen Rechner abgelegt. Dass die Versionsstände bei den Entwicklern nicht immer identisch sein müssen, macht die Stärke eines verteilten Versionskontrollsystems wie Git aus: Alle Änderungen werden erst lokal gespeichert und verwaltet und dann bei Bedarf auf das gemeinsame Online-Repository zurück übertragen. Damit sind die Entwickler bei der täglichen Arbeit nicht auf das gemeinsame Online-Repository angewiesen. Sie müssen nur in regelmäßigen Abständen die eigenen Fortschritte darauf übertragen und Zwischenstände der anderen Entwickler zurück in das eigene Repository spiegeln (s. Abb. 7.5).

1 https://github.com/
2 https://about.gitlab.com/
3 https://bitbucket.org/

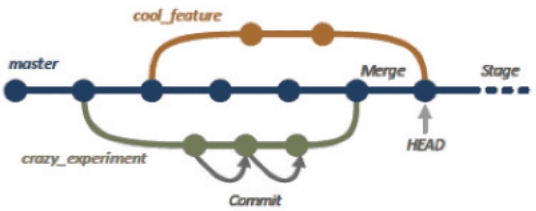


*Abb. 7.5: Beispiel-Repository mit zwei Branches*

Dazu wird für ein Feature, also eine neue Funktion, ein neuer Branch (dt. Zweig) erstellt.

- Ein Branch ist die Kopie des ganzen Programms. Mit der Kopie kann man unabhängig von anderen Entwicklern arbeiten, die neue Funktion implementieren und testen.
- Sobald ein Teil eines Features funktioniert, kann man mit einem Commit den Zwischenstand im eigenen Branch abspeichern. Jeder Entwickler hat also immer eine funktionierende Zwischenlösung, z. B. zur Vorführung für den Kunden.
- Der Zustand zwischen zwei Commits, also eine nicht gesicherte Zwischenversion, heißt in der Git-Welt Stage.
- Zu einem Commit gehört immer ein Comment: Was hat man geändert, und warum?
- Mit einem Push landet der eigene Zwischenstand dann auf dem Git-Server. Ansonsten bleiben alle Änderungen nur auf dem eigenen Rechner.
- Wenn man ein Feature fertiggestellt hat, kann man es mit einem Merge zurück in den Hauptzweig – den Master-Branch – schieben. Damit ist das Feature fertig.

Will man auch die Änderungen anderer Anwender aus dem gemeinsamen Repository laden, kann man mit einem Pull die Änderungen auf den eigenen Rechner holen.

Für viele Entwicklungsoberflächen wie Visual Studio, Eclipse oder IntelliJ stehen auch Git-Anbindungen mit grafischer Oberfläche zur Verfügung. Im folgenden Übungsprojekt wird jedoch die Git-Kommandozeile verwendet, die man für diverse Betriebssysteme auf der Webseite von Git[1] herunterladen und auf dem eigenen Rechner installieren kann.

Im weiteren Verlauf dieses Kapitels kann man anhand eines kleinen Übungsprojekts die Bedienung von Git als Beispiel für ein verteiltes Versionskontrollsystem kennenlernen und Schritt für Schritt die wichtigsten Befehle für die tägliche Entwicklungsarbeit nutzen und ausprobieren. Innerhalb des Übungsprojekts werden die wichtigsten Git-Kommandos kurz vorgestellt und beispielhaft erläutert. Am Ende des Kapitels findet man eine kurze Übersicht zum Nachschlagen und ein Glossar der wichtigsten Begriffe aus der Git-Welt.

### 7.1.2 Git in der Praxis: Übungsprojekt „Taschenrechner“

Jetzt ist es endlich soweit: Anna und Emre dürfen ihr erstes eigenes Git-Projekt realisieren.

In den nächsten Unterabschnitten werden einige für die Softwareentwicklung wichtige Git-Abläufe beispielhaft vorgestellt. Für eine ausführliche Beschreibung der Funktionalität von Git sei an dieser Stelle auf die Git-Dokumentation verwiesen.[2]

---

1 *https://git-scm.com/*

2 *https://git-scm.com/book/*

Der Quelltext des Beispielprogramms kann dabei jeweils mit einer beliebigen Entwicklungsumgebung bearbeitet werden. Zum Verwalten des Quelltextes werden die Git-Kommandozeilenbefehle eingesetzt.

Die folgende Abb. 7.6 stellt die Ausgabe des zu implementierenden Programms „Taschenrechner“ dar.

```
============================
|     Taschenrechner       |
============================
| Options:                 |
|        1. Addieren       |
|        2. Subtrahieren   |
|        3. Multiplizieren |
|        4. Dividieren     |
|        5. Exit           |
============================
Select option: 1
1. Zahl: 123
2. Zahl: 17

Ergebnis: 123.0 + 17.0 = 140.0
```

*Abb. 7.6: TUI des Programms „Taschenrechner“*

Die folgende Abb. 7.7 stellt die beiden Klassen des zu implementierenden Taschenrechners als UML-Klassendiagramm dar.

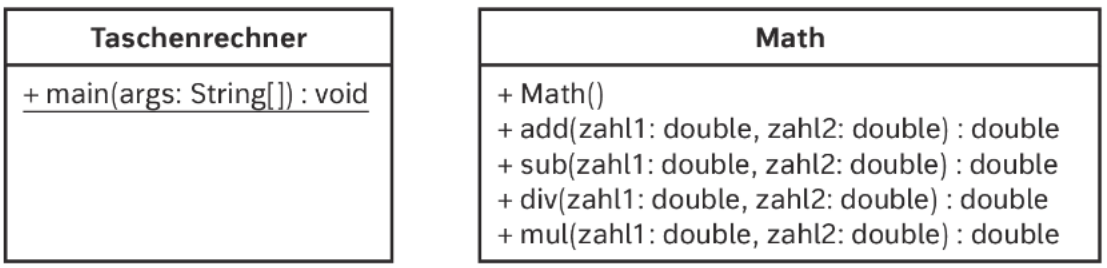


*Abb. 7.7: UML-Klassendiagramm für das Programm „Taschenrechner“*

Bevor überhaupt mit der Entwicklung begonnen wird, wird in der Regel ein gemeinsames Online-Repository für die Verwaltung der Software erstellt („taschenrechner_rep.git“). Dazu nutzt man einen Online-Git-Anbieter – das könnten z. B. BitBucket, Github oder Gitlab sein[1] – oder man verwendet einen selbst bereitgestellten Server.

Nach der Erstellung des Online-Repositorys sollen alle beteiligten Entwickler ihren eigenen Account mit den entsprechenden Rechten erhalten (Lese- und Schreibrechte).

## Erstellung einer Kopie des Online-Repositorys (Klonen)

Jeder Entwickler erstellt eine Kopie des aktuellen Online-Repositorys auf seinem lokalen Arbeitsrechner und benutzt dabei seine eigenen Zugangsdaten.

Hierzu kann der Entwickler z. B. in sein Home-Verzeichnis wechseln und dort das Verzeichnis „reps“ für alle lokalen Repositorys anlegen. Anschließend wechselt er in das Verzeichnis „reps“ und klont das gemeinsame Online-Repository.

1 *https://bitbucket.org/, https://github.com, https://gitlab.com*

```
$ cd ~
$ mkdir reps
$ cd reps/
$ git clone https://emre@supergit.org/anna/taschenrechner_rep.git
 Cloning into 'taschenrechner_rep'...
 warning: You appear to have cloned an empty repository.
$ ls
 taschenrechner_rep/
$ cd taschenrechner_rep/
```

### Eine neue Datei unter Git-Verwaltung stellen und versionieren

Ein Entwickler erstellt die Klasse `Math` („Math.java“) und implementiert die erste Methode dieser Klasse.

```
public class Math {
  public double add(double zahl1, double zahl2) {
    return zahl1 + zahl2;
  }
}
```

Damit eine Datei von Git verwaltet werden kann, muss sie zuerst unter die Git-Verwaltung gestellt werden. Dies geschieht mithilfe des Kommandos `add`. Erst danach kann der aktuelle Stand der Datei versioniert werden. Hierzu verwendet man das Kommando `commit` und übergibt dabei in der Regel eine kurze Log Message.

```
$ git add Math.java
$ git commit -m "initial version with add function"
 [master (root-commit) 2edc3f3] initial version with add function
  1 file changed, 5 insertions(+)
  create mode 100644 Math.java
```

### Übertragung des aktuellen Stands des lokalen Repositorys in das Online-Repository

Vor der Übertragung der Änderungen in das gemeinsame Online-Repository muss der aktuelle Stand des gemeinsamen Repositorys geholt und dieser mit dem aktuellen Stand der Software auf dem lokalen Repository zusammengeführt werden (`merge`). Anschließend soll der neue Stand erneut getestet und ggf. korrigiert und versioniert werden.

Erst nach einem erfolgreichen Test darf die neue Version in das gemeinsame Online-Repository übertragen werden.

Diese Vorgehensweise dient dazu, dass die auf dem gemeinsamen Repository gespeicherte Software möglichst konflikt- bzw. fehlerfrei bleibt. Somit werden die Entwickler nicht unnötig mit Fehlern der anderen belastet.

```
$ git pull origin
$ git push origin master
 Counting objects: 3, done.
 Delta compression using up to 8 threads.
 Compressing objects: 100% (2/2), done.
 Writing objects: 100% (3/3), 304 bytes | 304.00 KiB/s, done.
 Total 3 (delta 0), reused 0 (delta 0)
 To https://supergit.org/anna/taschenrechner_rep.git
  * [new branch]      master -> master
```

## Entwickeln mit dem Git-Branch

Git unterstützt wie fast alle anderen Versionsverwaltungstools das Branching-Konzept. Im Gegensatz zu anderen Versionsverwaltungstools ist Branching bei Git sehr effizient umgesetzt. Das heißt, das Erstellen bzw. Löschen von Branches geschieht sehr schnell und kostet dabei wenig Ressourcen. Branching bedeutet, dass man sich von der Hauptentwicklungslinie trennt. Dies kann aus verschiedenen Gründen geschehen, z. B. wenn die Software unabhängig voneinander in verschiedene Richtungen entwickelt werden soll oder wenn man erst nach vollständiger Implementierung bestimmter Features diese in die Hauptentwicklungslinie integrieren und somit den anderen Entwicklern zur Verfügung stellen möchte.

**Beispiel:**
Ein Entwickler will an der Oberfläche des Taschenrechners arbeiten und erst nach Abschluss seiner Arbeit den Teammitgliedern sein Ergebnis zur Verfügung stellen. Hierzu erstellt er den neuen Branch „impl_taschenrechner“ und wechselt dort hin.

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

Im neuen Branch wird eine erste Version (Addition der "hardcodierten" Zahlen 4 und 5) der Klasse `Taschenrechner` implementiert, getestet und versioniert. Dies geschieht so lange, bis die Klasse vollständig implementiert ist.

```
import java.util.Scanner;
public class Taschenrechner {

  public static void main(String[] args) {

    Scanner myScanner = new Scanner(System.in);
    Math aMath = new Math();
    char option;
```

```
    // Display menu
    System.out.println("============================");
    System.out.println("| MENU SELECTION DEMO      |");
    System.out.println("============================");
    System.out.println("| Options:                 |");
    System.out.println("|        1. Addieren       |");
    System.out.println("|        2. Subtrahieren   |");
    System.out.println("|        2. Multiplizieren |");
    System.out.println("|        3. Dividieren     |");
    System.out.println("|        4. Exit           |");
    System.out.println("============================");
    System.out.print(" Select option: ");
    option = myScanner.next().charAt(0);

    // Switch construct
    switch (option) {
    case '1':
      System.out.println("4 + 5 = " + aMath.add(4, 5));
      break;
    // ...
    default:
      System.out.println("Invalid selection");
    }
  }
}
```

```
$ git add Taschenrechner.java
$ git commit -m "file Taschenrechner.java added"
 [impl_taschenrechner 7718bfc] file Taschenrechner.java added
  1 file changed, 34 insertions(+)
  create mode 100644 Taschenrechner.java
```

Um die neuen Entwicklungen in die Hauptentwicklungslinie zu übernehmen, wechselt der Entwickler zum Master-Branch und führt das Kommando `merge` aus.

```
$ git checkout master
 Switched to branch 'master'
 Your branch is up-to-date with 'origin/master'.
$ git merge impl_taschenrechner
 Updating 2edc3f3..7718bfc
 Fast-forward
  Taschenrechner.java | 34 ++++++++++++++++++++++++++++++++++
  1 file changed, 34 insertions(+)
  create mode 100644 Taschenrechner.java
```

Um sicherzustellen, dass sich keine Fehler eingeschlichen haben, wird im Master-Branch alles nach der Zusammenführung noch mal getestet. Sind alle neuen Teile erfolgreich integriert und getestet, kann der erstellte Branch wieder entfernt werden.

Jetzt muss nur noch der aktuelle Stand auf das gemeinsame Online-Repository übertragen werden. Wie diese Übertragung mithilfe von Git umgesetzt werden kann, wurde bereits im vorherigen Abschnitt beschrieben.

```
$ git branch -d impl_taschenrechner
 Deleted branch impl_taschenrechner (was 7718bfc).
$ git pull origin
 Already up-to-date.
$ git push origin master
 Counting objects: 3, done.
 Delta compression using up to 8 threads.
 Compressing objects: 100% (3/3), done.
 Writing objects: 100% (3/3), 697 bytes | 697.00 KiB/s, done.
 Total 3 (delta 0), reused 0 (delta 0)
 To https://supergit.org/anna/taschenrechner_rep.git
    2edc3f3..7718bfc  master -> master
```

Es ist empfehlenswert, dass die neuen Features immer in einem neuen Branch entwickelt werden. Erst wenn die Entwicklung vollständig abgeschlossen und erfolgreich getestet wurde, soll sie in den Master-Branch übernommen und dort noch mal getestet werden. Zum Schluss sollten diese Entwicklungs-Branches der Übersichtlichkeit halber wieder entfernt werden.

Durch diese Vorgehensweise stellt man sicher, dass während der Entwicklungsphase immer eine lauffähige Version der Software im Master-Branch vorhanden ist.

### Überblick über die Versionen

Mithilfe des Log-Befehls erhält man einen Überblick über die erstellten Versionen (Commits) im aktuellen Branch.

```
$ git log
commit b7b605fdbc2353d868bc490876624452663404fc
Author: Emre <emre@tortoisetools.io>
Date:   Thu Aug 10 12:44:46 2017 +0200

    function sub implemented

commit 7718bfc7d0ee5b0bce4f48fa1bc9ebba39625d4b
Author: Anna <anna@tortoisetools.io>
Date:   Thu Aug 10 12:37:08 2017 +0200

    file Taschenrechner.java added

commit 2edc3f3543d9fa5643074e24a08620f201be88b9
Author: Emre <emre@tortoisetools.io>
Date:   Thu Aug 10 12:28:47 2017 +0200

    initial version with add function
```

### Was ist ein Konflikt in der Git-Welt?

Ein Konflikt entsteht immer dann, wenn eine Datei an zwei Stellen – in verschiedenen Branches oder in verschiedenen lokalen Repositorys – in einer neueren, geänderten Version vorliegt und deswegen nicht klar ist, welche Version übernommen werden soll. Bei der Entwicklung des Taschenrechners stellt z. B. ein Entwickler während des Testens einen Fehler in einem von ihm nicht implementierten Bereich fest und beseitigt diesen Fehler. Durch das oben beschriebene Szenario kann eine Stelle in der gleichen Datei von mehreren Leuten unabhängig voneinander in ihrem lokalen Repository geändert werden. Wenn man diese verschiedenen Versionen zusammenführen möchte, werden diese von Git als Konflikte erkannt.

```
$ git add Taschenrechner.java
$ git commit -m "sub() method added and tested"
 [master 2198ad8] sub() method added and tested
  1 file changed, 3 insertions(+)
$ git pull
 remote: Counting objects: 9, done
 remote: Finding sources: 100% (6/6)
 remote: Getting sizes: 100% (6/6)
 remote: Total 6 (delta 1), reused 6 (delta 1)
 Unpacking objects: 100% (6/6), done.
 From https://supergit.org/anna/taschenrechner_rep.git
    7718bfc..dfc3a2a master -> origin/master
 Auto-merging Taschenrechner.java
 CONFLICT (content): Merge conflict in Taschenrechner.java
 Automatic merge failed; fix conflicts and then commit the result.
```

In manchen Fällen kann Git die Konflikte automatisch lösen, z. B. wenn verschiedene Zeilen geändert wurden oder eine Datei zwischenzeitlich umbenannt wurde. Manchmal hilft aber nur „Handarbeit“. Im letzteren Fall erzeugt Git einen sog. „Diff“ in der ursprünglichen Datei. Dabei werden alle Änderungen gekennzeichnet und man kann selbst von Änderung zu Änderung entscheiden, was übernommen werden soll.

Ein Beheben der Konflikte ist mit grafischen Tools allerdings meist einfacher als mit einem Texteditor. Hier ein Auszug aus „Taschenrechner.java“, der Konflikt ist orange markiert. Die lokale Version ist der obere Abschnitt – er beginnt mit „HEAD“.

```
...
 case '2':
<<<<<<< HEAD
          System.out.println("4 - 5 = " + aMath.sub(4, 5));
          break;
=======
          System.out.println("5 - 4 = " + aMath.sub(5, 4));
          break;
>>>>>>> dfc3a2a42d75ee3aee95e820bdd1b82a4dcac4e1
 default:
...
```

Sobald man den Konflikt behoben hat, kann man mit einem `add` und dem anschließenden `commit` eine neue, konfliktfreie Version erstellen.

```
$ git add Taschenrechner.java
$ git commit -m „Conflict resolved :-)“
 [master 684d872] Conflict resolved :-)
$ git pull origin
 Already up-to-date.
$ git push origin
 Counting objects: 4, done.
 Delta compression using up to 4 threads.
 Compressing objects: 100% (4/4), done.
 Writing objects: 100% (4/4), 564 bytes | 0 bytes/s, done.
 Total 4 (delta 1), reused 0 (delta 0)
 remote: Resolving deltas: 100% (1/1)
 remote: Updating references: 100% (1/1)
 To https://supergit.org/anna/taschenrechner_rep.git
    dfc3a2a..684d872  master -> master
```

## Wo speichert Git seine Daten?

Im Repository-Verzeichnis findet man ein verstecktes Verzeichnis mit dem Namen „.git“. Hier befindet sich die lokale Kopie (s. Abb. 7.8) des vollständigen Repositorys mit allen Commits, Branches und Verwaltungsdaten. Außerdem speichert Git dort alle Änderungen zwischen, die man mit `git add` für einen Commit vorgemerkt hat.

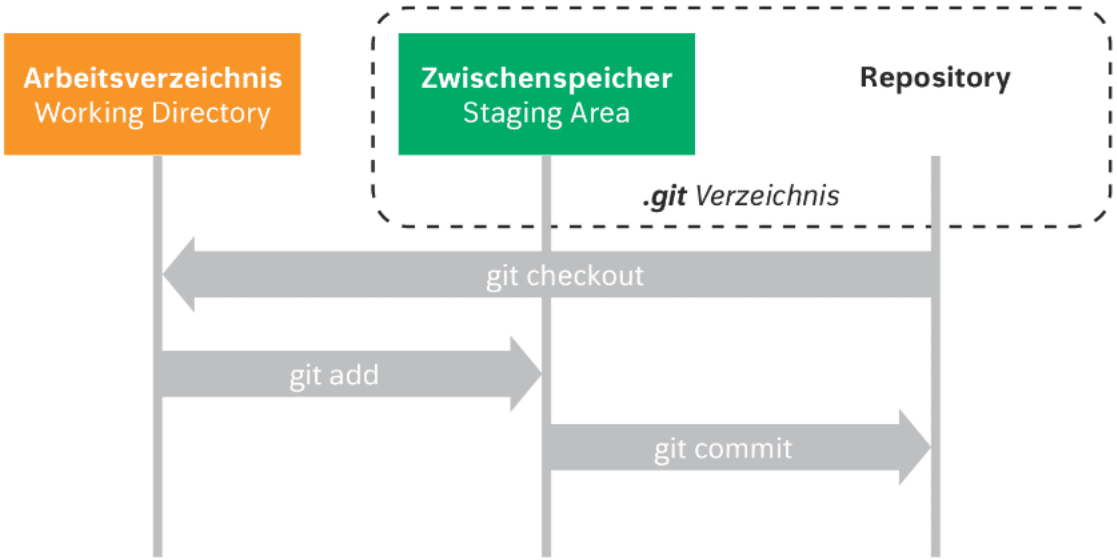


*Abb. 7.8: Aufbau eines lokalen Repositorys*

Der Zwischenspeicher lässt sich auch mit dem Befehl `git stash` sichern. So kann man halb fertige Arbeiten behalten und trotzdem z. B. den Branch wechseln. Mit `git stash list` kann man sich den sog. Stack anzeigen lassen und ihn mit `git stash apply` wieder zurück ins Arbeitsverzeichnis holen.

### 7.1.3 Zusammenfassung, Git-Kommandos im Überblick und Glossar

Neben kleinen Projekten, wie beim Übungsprojekt „Taschenrechner“, wird Git in der Regel bei viel größeren Projekten (sowohl kommerziell als auch bei Community-Projekten) eingesetzt. Unter https://github.com/torvalds/linux findet man z. B. das Git-Repository des Linux-Kernels.

Die vollständige Dokumentation zu Git und einige Videos findet man online unter https://git-scm.com/doc. Wenn man tiefer in die Materie einsteigen und weitere Funktionen von Git kennenlernen möchte – z. B. den Stash – findet man unter https://git-scm.com/book/ eine Online-Version des englischsprachigen Buchs „Pro Git“ von Scott Chacon und Ben Straub sowie eine deutsche Übersetzung davon. Die Firma Atlassian, Anbieter der Plattform Bitbucket, stellt außerdem verschiedene englischsprachige Tutorials unter https://www.atlassian.com/git/tutorials bereit.

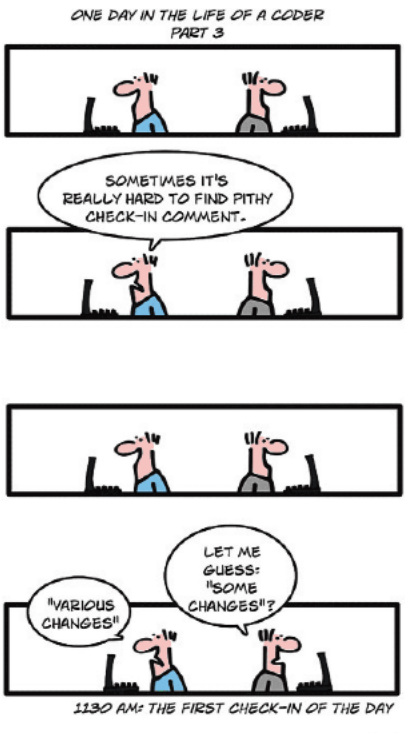


*Abb. 7.9: Git Kommentare – Dein Freund und Helfer?!*

Im Folgenden werden die wichtigsten Git-Kommandos und ihre Bedeutung aufgelistet. Von den meisten Kommandos gibt es noch viele weitere Varianten und Optionen. Mit dem Parameter `-h` kann man sich in der Git-Kommandozeile zu jedem Kommando eine Kurzhilfe anzeigen lassen, z. B. `git add -h`. Mit dem Aufruf von `git help add` kann man eine ausführliche Hilfe aufrufen.

1. Repositories erstellen oder clonen
   - `git init [Projektname]` erzeugt ein neues lokales Repository mit dem Namen `Projektname`.

- `git clone [URL]` lädt ein entferntes Repository von der Adresse `URL` mit der gesamten Versions-Historie herunter und stellt es als lokales Repository zur Verfügung.

2. Änderungen an Quelltexten verwalten
   - `git status` zeigt alle neuen oder geänderten Dateien an, die noch commited oder geaddet werden müssen.
   - `git add [Datei]` fügt dem Index eine **Änderung** hinzu und merkt sie damit für den nächsten Commit vor.
   - `git commit` bestätigt alle Änderungen und erstellt einen neuen Versionsstand im lokalen Repository (aber noch nicht im entfernten).
   - `git diff` zeigt die Änderungen zwischen zwei Versionsständen, z. B. zwischen der aktuellen Arbeitsversion und dem letzten Commit, aber auch zwischen zwei Branches.
   - `git revert <commit>` erstellt einen neuen Commit, der alle Änderungen seit einem bestimmten Commit rückgängig macht – eine sichere Variante, um ungewünschte, aber bereits von einem Commit erfasste Änderungen rückgängig zu machen.
   - `git rm` löscht eine Datei aus der Arbeitskopie und merkt die Löschung für den nächsten Commit vor.
3. Arbeiten mit Branches
   - `git branch` zeigt alle Branches im Repository an.
   - `git branch <Branch>` erzeugt einen neuen Branch mit dem Namen `<Branch>`.
   - `git branch -d <Branch>` löscht einen nicht mehr benötigten Branch mit dem Namen `<Branch>`.
   - `git checkout <Branch>` wechselt in den Branch `<Branch>` und aktualisiert die Arbeitskopie.
   - `git merge <Branch>` pflegt die Änderungen **aus** dem Branch `<Branch>` **in** den aktuellen Branch ein und warnt bei evtl. auftretenden Konflikten, die dann von Hand gelöst werden müssen.
4. Synchronisieren mit entfernten Repositorys
   - `git fetch <Remote>` aktualisiert das lokale Repository: Es lädt die neusten Änderungen vom entfernten Repository `<Remote>` herunter, jedoch ohne Aktualisieren des Arbeitsverzeichnisses. Üblicherweise wird das ursprüngliche Repository mit `git fetch origin` verwendet.
   - `git pull <Remote>` aktualisiert das lokale Repository: Es lädt die neusten Änderungen vom entfernten Repository `<Remote>` herunter (`git fetch <Remote>`) und führt sie mit dem aktuellen Stand der Arbeitskopie zusammen (`git merge`).
   - `git push <Remote> <Branch>` lädt die Änderungen aus einem lokalen Branch `<Branch>` zu einem entfernten Repository `<Remote>` hoch – typischerweise wird dabei der Master-Branch zum ursprünglichen Repository gepusht: `git push origin master`.

Die folgende Tabelle fasst noch einmal einige wichtige Begriffe als Glossar zusammen.

| Arbeits-verzeichnis | Bereich des (lokalen) Repositorys, in dem Änderungen vorgenommen werden |
|---|---|
| **Branch** | Nebenzweig im → Repository, technisch eine Sammlung von → Commits. Wird oft verwendet, um neue → Features in das Programm einzubauen. |
| **Clean** | Gegenteil von dirty – bezeichnet ein Arbeitsverzeichnis, in dem keine Änderungen im Vergleich zum letzten (lokalen) Commit vorgenommen wurden. |
| **Clone** | Erstellen einer Kopie eines → Repository als neues, lokales Repository. Dabei wird die gesamte Historie (alle → Commits) des Projekts mitkopiert. |
| **Conflict** | Er entsteht, wenn zwei unterschiedliche Versionen einer Datei (z.B. aus zwei verschiedenen Branches oder zwei verschiedenen → Repositories) zusammengeführt werden sollen und an der gleichen Stelle Änderungen vorgenommen wurden. Schlägt ein automatischer → Merge fehl und die Änderungen müssen von Hand zusammengeführt werden, ist ein → Conflict entstanden. |
| **Comment** | Wird auch Log Message genannt. Kurzzusammenfassung der Inhalte eines → Commits, steht als Änderungsprotokoll zur Verfügung. Viele Versionskontrollsysteme erfordern einen Comment. |
| **Commit** | Ablegen von Änderungen im Verzeichnis. Zu diesen Änderungen kann immer wieder zurückgesprungen werden. |
| **Dirty** | Bezeichnet ein Arbeitsverzeichnis, in dem lokale Änderungen vorgenommen wurden, die noch nicht von einem Commit erfasst wurden – z.B., weil eine Datei geändert oder neu hinzugefügt wurde. |
| **Feature** | Kleiner Teil der entwickelten Software, z.B. neue Funktion im Dialogfenster. |
| **Fork** | Aus einem vorhandenen → Repository durch Kopie erstelltes Repository – nicht zu verwechseln mit einem → Clone. |
| **Head** | Zeigt auf den neuesten Commit (also den aktuellen Stand) im verwendeten → Branch. |
| **Index** | Andere Bezeichnung für → Staging Area. |
| **Log Message** | s. → Comment. |
| **Master** | Der Haupt-→ Branch. Er sollte immer einen lauffähigen Zwischenstand der entwickelten Software enthalten. |
| **Merge** | Zusammenführen von zwei Branches, insbesondere auch beim Einpflegen eines neuen → Features in den → Master. |
| **Pull** | Abrufen von aktuellen Änderungen aus einem entfernten → Repository, also z.B. einem Git-Server im Internet. |
| **Push** | Senden von Änderungen vom lokalen Repository auf den Git-Server. |
| **Repository** | Datenbank, die Informationen über die Historie aller Dateien eines Softwareprojekts enthält. |
| **Resolve** | Auflösen eines → Conflicts durch manuellen Eingriff. Er muss anschließend, wie normale Änderungen auch, mit einem Commit „bestätigt" werden. |
| **Stage / Staging Area** | Aktueller „Stand" des Projekts zwischen zwei → Commits. In der Staging Area werden alle Änderungen gesammelt, die für den nächsten Commit vorgemerkt wurden. Wird manchmal auch als Index bezeichnet. |