Versionsverwaltung

In der Praxis werden Daten oder Files ständig geändert. Die Kontrolle über die geänderte Files ist besonders bei der Entwicklung von Software Projekten sehr wichtig. Software kann heute aus Millionen von Zeilen Code bestehen. Dazu einige Beispiele bekannter Programme:

- Windows 7 besteht aus ca. 40 – 50 Millionen Zeilen Code
- Facebook besteht aus mehr als 60 Millionen Zeilen Code
- Max OS X besteht aus ca. 85 Millionen Zeilen Code
- Microsoft Office besteht aus ca. 50 Millionen Zeilen Code
- Android besteht aus mehr als 10 Millionen Zeilen Code
- grosse Web-Sites können ebenfalls Millionen von Zeilen HTML Code enthalten

Dieser Code wird physisch in Files gespeichert, irgendwann zu einem Release zusammengebaut, getestet und nachdem die Abnahmetests erfolgreich sind auch an die Kunden verteilt wird. Basierend auf der Grösse dieser Projekt und der Komplexität der Software arbeiten dabei unzählige Personen in diesen Software-Projekten.

Da sich die Software ändern kann (Korrekturen, Fixes, neue Anforderungen, neue Features, etc.) muss der Code laufend angepasst und erweitert werden. Es muss also irgendwie sichergestellt werden, dass beim „Bau“ der Software die richtigen Versionen der Dateien verwendet werden, dies parallel zum laufenden Betrieb bei dem ständig Dateien geändert werden. Zudem muss sichergestellt werden, dass die verschiedenen Projektmitarbeiter sich nicht gegenseitig Dateien überschreiben.

Das Speichern der Dateien in unterschiedlichen Directories ist hier sicher keine Lösung. Backups im klassischen Sinn (Inkrementell, Differenziell oder Full) helfen hier auch nicht, da Dateien mehrfach pro Tag ändern können. Die Snapshot Technologien können zwar jeden Zustand einer Datei aufzeichnen, bieten jedoch keine Möglichkeit um festzuhalten, dass eine Datei *X in der Version 27* zusammen mit der Datei *Y in der Version 15* verwendet werden muss oder dass lediglich eine einzelne Datei z.B. auf die Version 26 zurückgestellt werden soll

Für die Verwaltung von Source-Code, Konfigurationen oder irgendwelchen Files, wird also eine Tool benötigt, das Versionen und Zustände von Files verwalten kann. Diese Programme werden als **Versionsverwaltungs-Tools** bezeichnet.

Wikipedia definiert **Versionsverwaltung** wie folgt::

> Eine **Versionsverwaltung** ist ein System, das zur **Erfassung** von ***Änderungen an Dokumenten oder Dateien*** verwendet wird. Alle Versionen werden in einem **Archiv** mit Zeitstempel und Benutzerkennung gesichert und können später wiederhergestellt werden.

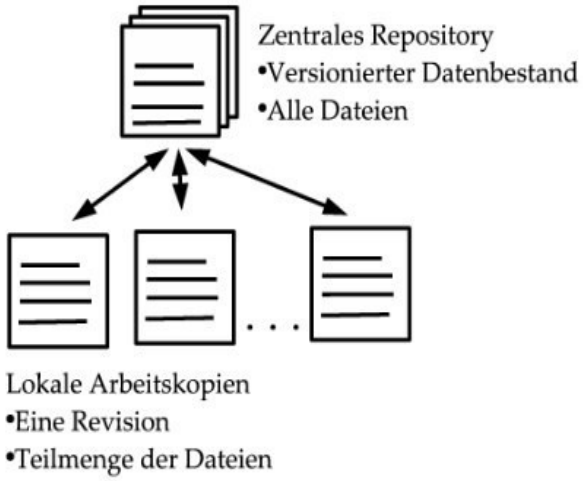
Die Hauptaufgaben einer Versionsverwaltung sind dabei:

- **Protokollierung** der Änderungen: Es kann jederzeit nachvollzogen werden, wer wann was geändert hat.
- **Wiederherstellung** von alten Ständen einzelner Dateien: Somit können versehentliche Änderungen jederzeit wieder rückgängig gemacht werden.
- **Archivierung** der einzelnen Stände eines Projektes: Dadurch ist es jederzeit möglich, auf alle Versionen zuzugreifen.
- **Koordinierung** des gemeinsamen **Zugriffs** von mehreren Entwicklern auf die Dateien.
- **Gleichzeitige Entwicklung** mehrerer Entwicklungszweige (engl. Branches) eines Projektes.

Funktionsweise einer Versionsverwaltung

Ein Versionskontrollsystem besteht aus einem **Server** und einem **Client** und es wird zwischen mindestens zwei unterschiedlichen Datenbeständen unterschieden:

- Der **Server** ist für die ***Verwaltung des Repositories*** verantwortlich und kooperiert mit einer oder mehreren Instanzen des Clients.
- Der **Client** ist für die Verwaltung der lokalen Arbeitskopien und die Koordination mit dem Server verantwortlich. Zudem stellt er dem Benutzer die Funktionen des Versionskontrollsystems zur Verfügung.



- Das **Repository** (eine Art Datenbank) ist für die zentrale Speicherung und Bereitstellung aller Revisionen zuständig. Das **Repository** umfasst alle Dateien des Projekts mit allen Revisionen (Versionen).
- Bei der **Arbeitskopie** handelt es sich um alle Projektdateien in einer Revision mit den Veränderungen des Benutzers und ergänzt um Arbeitsinformationen des Versionskontrollsystems. Arbeitskopien werden in der Regel lokal (auf dem System des Benutzers) gespeichert.

- Sollen Dokumente oder Files der Kontrolle des Versionskontrollsystems übergeben werden, muss als erster Schritt ein Projekt mit allen relevanten Dateien beim Versionskontrollsystem angelegt werden. Bei der Erstellung eines Projektes wird unter anderem festgelegt, wo das Repository des Versionskontrollsystems angelegt werden soll; weiterhin werden für das Projekt gültige Einstellungen (z.B. Zugriffsrechte) bestimmt.
- Nachdem ein Projekt erstellt wurde, werden in einem weiteren Schritt, dem sogenannten **Import**, die zu verwaltenden Dateien eingestellt. Es wird also vereinfacht ausgedrückt eine Kopie der Projektdateien im Repository erstellt.
- Soll nun mit den in das Projekt eingestellten Dateien gearbeitet werden, ist ein so genannter **Checkout** notwendig. Hierbei wird eine Kopie der Projektdateien aus dem Repository entnommen. Es wird also eine Arbeitskopie für den Anwender erstellt. Bei einem solchen **Checkout** können prinzipiell beliebige Revisionen der Dateien des Projektes aus dem Repository entnommen werden, aber meist wird die aktuellste Revision verwendet.
- Ein Benutzer kann nun mit der lokalen Kopie der Dateien arbeiten. Wenn die angestellten Veränderungen als neue Revision aufgefasst werden sollen, ist ein entsprechender, als **Commit** bezeichneter Aufruf des Versionskontrollsystems notwendig. Bei diesem **Commit** werden durch das Versionskontrollsystem alle Unterschiede erfasst, die der Benutzer an seiner lokalen Kopie seit dem Checkout vorgenommen hat.
- Alle Versionskontrollsysteme bieten die Möglichkeit, zu einem bestehenden Repository neue Dateien hinzuzufügen. Dabei wird nicht von einem Import, sondern von einem **Add** gesprochen. Ansonsten werden keine Unterschiede zwischen einer beim Import einstellten und einer später hinzugefügten Datei gemacht.
- Wird eine Revision des Repositories abgefragt, die vor dem Add datiert, sind entsprechende Dateien nicht Teil des Checkouts.

- Werden die Dateien eines Projektes unter Versionskontrolle durch mehrere Personen zur gleichen Zeit bearbeitet, verwendet jeder Anwender eine lokale Kopie. Diese kann vom Repository und von den lokalen Kopien der anderen Anwender abweichen. Möchte ein Benutzer seinen Bestand von Dateien mit dem Repository synchronisieren, so führt er ein so genanntes **Update** durch.

  Bei einem **Update** wird wie bei einem Commit ein Vergleich zwischen lokaler Kopie und Repository durchgeführt, jedoch hier mit dem Ziel, die Arbeitskopie auf den aktuellen Stand im Repository zu aktualisieren.

- Die **aktuellste Version** im Repository wird dabei als **Head** bezeichnet, die ursprüngliche Arbeitskopie, auf der alle Arbeiten des Benutzers stattfanden, als **Base**.

  Hat ein anderer Benutzer zwischenzeitlich ein Commit durchgeführt, d.h. entspricht also die ***Head Version*** nicht mehr der ***Base Version***, werden die Unterschiede zwischen Head und Base Version an den Nutzer übertragen. Während bei einem Commit das Repository auf den Stand der Arbeitskopie aktualisiert wird, werden also beim Update neuere Änderungen aus dem Repository in die Arbeitskopie übertragen.

- Die vorangegangene Schilderung setzt voraus, dass keine beteiligten Benutzer zu einem gegebenen Zeitpunkt mit der gleichen Datei arbeiten, dies ist jedoch kein Regelfall. Wenn mehrere Benutzer ein Projekt gemeinsam bearbeiten, ergibt sich hieraus das Risiko von Konflikten. Als **Konflikt** wird in diesem Zusammenhang ein Zustand bezeichnet, in dem sich die Änderungen, die unterschiedliche Nutzer im gleichen Zeitraum an einer Datei vorgenommen haben, widersprechen und ohne Intervention des Versionskontrollsystems eine Änderung durch die jeweils nachfolgende überschrieben worden wäre.

  Der folgende Vorgang stellt ein einfaches Beispiel für einen Konflikt dar: Zwei Benutzer entnehmen mittels Checkout die gleiche Datei aus dem Repository:

  - Beide Benutzer (User1 und User2) verändern die jeweilige lokale Version der Datei
  - Benutzer User1 übergibt seine Änderungen mittels **Commit** ins Repository.
  - Der zweite Benutzer (User2) übergibt seine Version der Datei ebenfalls mittels Commit ins Repository.
  - Das Versionskontrollsystem greift ein, da ohne Eingriff die späteren Änderungen von User2 die vorhergehenden von User1 überschreiben würden.

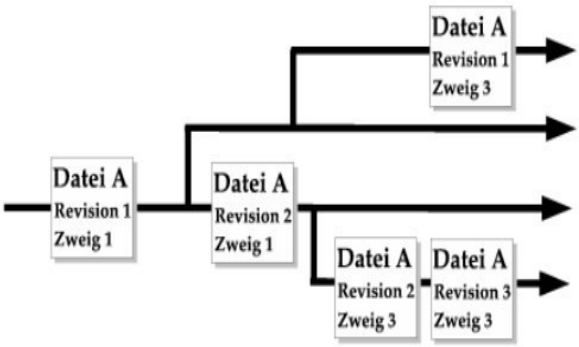
  Der einfachste Ansatz zur Vermeidung einer solchen Situation besteht darin, jede Datei für einen bestimmten Zeitraum einem Benutzer zuzuordnen, der sie exklusiv bearbeiten kann, und alleine Schreibrechte auf diese Datei im Repository hat. Erst wenn dieser Benutzer seine Bearbeitung beendet und einen Commit durchgeführt hat, erhalten andere Benutzer die Gelegenheit, die Datei in ihrer nun vorliegenden Version zu verändern. Dieses Vorgehen hat sich in der Praxis jedoch als zu restriktiv herausgestellt. Bei einem potentiellen Konflikt erzwingt das System einen Update, d.h. theoretisch würden damit die Modifikationen von User2 in der lokalen Arbeitskopie durch die Version des Benutzers User1 überschrieben.

  Überwiegend wird daher durch das Versionskontrollsystem versucht, die unterschiedlichen Änderungen verschiedener Nutzer an einer Datei automatisch zu integrieren oder einen menschlichen Benutzer bei dieser Integration zu unterstützen.

  Die modernen Versionskontrollsysteme können heute solche Konflikte meist automatisch beheben. Sie vergleichen dabei die beiden Dateien und führen den Code der beiden Benutzer automatisch zusammen. Dar automatische Zusammenführen gelingt in der Regel immer, wenn die Benutzer die Files in unterschiedlichen Bereichen geändert haben, d.h. es keine Überschneidungen gibt

  - User1 macht eine Anpassung auf Zeile 20 (z.B. MeinWert = 17 auf MeinWert = 15
  - User2 ändert Zeilen 50 bis 60 und fügt zusätzliche neue Zeilen nach Zeile 60 ein

In den bisherigen Szenarien wurde stets davon ausgegangen, dass es bei der Bearbeitung des Datenbestandes unter der Kontrolle des Repositories nur einen Zweig des Datenbestandes unter aktiver Entwicklung gibt, in den alle Änderungen einfliessen. Dabei blieben Aufspaltungen in der Entwicklung, so genannte **Branches**, unberücksichtigt.



Einen Branch anzulegen ist dann sinnvoll, wenn die Bearbeitung der Dateien des Projektes für einen bestimmten Zeitraum in getrennten Linien parallel nebeneinander fortgeführt werden soll.

Eine Darstellung der zeitgleichen Existenz einer Datei in mehreren **Branches** zeigt nebenstehende Abbildung.

Dieser Fall kann auftreten, wenn das bearbeitete Projekt veröffentlicht werden soll, gleichzeitig aber neue Bestandteile hinzugefügt werden. Es liegen nun zwei Äste des Projektes vor, die sich weitgehend getrennt voneinander weiterentwickeln.

Während in die zur Veröffentlichung bestimmte Version in erster Linie nur Fehlerkorrekturen einfliessen, werden weitergehende Neuerungen in den anderen Ast eingefügt.

Die explizite Unterstützung der Erstellung und Pflege von Ästen, also das **Branching**, ist daher Bestandteil aller verbreiteten Versionskontrollsysteme. Um einen Ast anzulegen, legt ein Benutzer dabei eine Version im Versionskontrollsystem als Ursprung des Astes fest. Wie dies geschieht, und wie der Ast angesprochen werden kann, unterscheidet sich zwischen unterschiedlichen Versionskontrollsystemen stark. Es ist nun möglich, einen neuen Branch zu beginnen und ohne Konflikte mit anderen Ästen am eigenen Ast zu arbeiten. Aus der Sicht der Benutzer handelt es sich bei den Ästen um allein stehende Projekte.

Sollen Äste wieder vereinigt werden, oder sollen Veränderungen aus einem Ast in einen anderen Ast einfliessen, so werden alle Veränderungen in den zu vereinigenden Ästen auf Vereinbarkeit verglichen, sich widersprechende Veränderungen werden als Konflikte behandelt. Das Zusammenführen von Veränderungen aus unterschiedlichen Ästen wird dabei nicht als Update, sondern als **Merge** bezeichnet, da hier nicht eine Aktualisierung stattfindet, sondern das Verschmelzen eigenständig geführter Entwicklungslinien.

Zusammenfassung

- Um für die Dateien eines Projektes ein Versionskontrollsystem nutzen zu können, muss dieses in das Repository eingestellt werden; dieser Vorgang wird als **Import** bezeichnet.
- Der Begriff **Checkout** bezeichnet den Vorgang, eine komplette Kopie eines Projekts zur lokalen Bearbeitung aus dem Versionskontrollsystem zu entnehmen.
- Als **Commit** wird die Übergabe von neuen Änderungen an der lokalen Kopie an das Versionskontrollsystem bezeichnet.
- Ein **Add** ist das Hinzufügen neuer Dateien in ein bestehendes Repository.
- Jeder Benutzer erhält eine eigene Kopie der Dateien des Projektes zur Bearbeitung.
- Ein **Update** bezeichnet das Aktualisieren der lokalen Kopie durch zwischenzeitlich von anderen Benutzern in das Repository eingestellte Änderungen.
- Die aktuellste Revision im Repository wird als **Head** bezeichnet.
- Die Revision, die ein Benutzer zuletzt aus dem Repository entnommen hat wird als **Base** bezeichnet.
- Konflikte können entstehen, wenn die gleichen Dateien im gleichen Zeitraum durch mehrere Personen bearbeitet werden.
- Liegen potentielle Konflikte vor, so müssen diese mit einem **Update vor** einem **Commit** beseitigt werden, hierdurch werden die Änderungen aus dem Repository in den lokalen Datenbestand aufgenommen.
- Die Integration von lokalen Änderungen in das Repository kann automatisiert erfolgen, wenn die Bearbeitungen in unterschiedlichen Bereichen einer Datei stattfanden, ist dies nicht der Fall, muss ein Benutzer entscheiden, welche Änderungen übernommen und welche verworfen werden sollen.
- Als **Branching** wird das Abspalten einer parallelen Entwicklungslinie bezeichnet.
- Vor der Erstellung eines Branches wird der Ausgangspunkt der Veränderung markiert, eine solche Markierung wird als **Tag** bezeichnet.
- Die neue entstandene Entwicklungslinie wird dabei als **Branch** bezeichnet.
- Das zusammenführen verschiedener Branches wird als **Merge** bezeichnet.

Versionsverwaltungssysteme

Bei den verschiedenen Versionsverwaltungssystemen wird zwischen den beiden folgenden Konzepten unterschieden:

- **Zentrale Versionsverwaltung**

  Diese Art ist als Client-Server-System aufgebaut, sodass der Zugriff auf ein Repository auch über Netzwerk erfolgen kann. Durch eine Rechteverwaltung wird dafür gesorgt, dass nur berechtigte Personen neue Versionen in das Archiv legen können. Die Versionsgeschichte ist hierbei nur im Repository vorhanden. Dieses Konzept wurde vom Open-Source-Projekt **CVS** (Concurrent Versions System) populär gemacht, mit **Subversion** (SVN) neu implementiert und von vielen kommerziellen Anbietern verwendet.

  Beispiele von bekannten Versionsverwaltungssystemen dieser Kategorie::

  - Open-Source Produkte: **CVS**, **Subversion** (SVN), **RCS**
  - Kommerzielle Produkte: **ClearCase**, **Visual Source Save**, **Perforce**

- **Verteilte Versionsverwaltung**

- Die verteilte Versionsverwaltung (DVCS, distributed VCS) verwendet kein zentrales Repository mehr. Jeder, der an dem verwalteten Projekt arbeitet, hat sein eigenes Repository und kann dieses mit jedem beliebigen anderen Repository abgleichen. Die Versionsgeschichte ist dadurch genauso verteilt. Änderungen können lokal verfolgt werden, ohne eine Verbindung zu einem Server aufbauen zu müssen.

- Im Gegensatz zur zentralen Versionsverwaltung kommt es nicht zu einem Konflikt, wenn mehrere Benutzer dieselbe Version einer Datei ändern. Die sich widersprechenden Versionen existieren zunächst parallel und können weiter geändert werden. Sie können später in eine neue Version zusammengeführt werden. Dadurch entsteht ein gerichteter azyklischer Graph (Polyhierarchie) anstatt einer Kette von Versionen. In der Praxis werden bei der Verwendung in der Softwareentwicklung meist einzelne Features oder Gruppen von Features in separaten Versionen entwickelt und diese bei grösseren Projekten von Personen mit einer Integrator-Rolle überprüft und zusammengeführt.

- Systembedingt bieten verteilte Versionsverwaltungen keine Locks. Da wegen der höheren Zugriffsgeschwindigkeit die Granularität der gespeicherten Änderungen viel kleiner sein kann, können sie sehr leistungsfähige, weitgehend automatische Merge-Mechanismen zur Verfügung stellen.

- Eine Unterart der Versionsverwaltung bieten einfachere Patchverwaltungssysteme, die Änderungen nur in eine Richtung in Produktivsysteme einspeisen.

- Obwohl konzeptionell nicht unbedingt notwendig, existiert in verteilten Versionsverwaltungsszenarien üblicherweise ein offizielles Repository. Das offizielle Repository wird von neuen Projektbeteiligten zu Beginn ihrer Arbeit geklont, d.h. auf das lokale System kopiert.

  Beispiele von bekannten Versionsverwaltungssystemen dieser Kategorie:

  - Open-Source Produkte: **BitKeeper**, **Mercurial**, **Git**
  - Kommerzielle Produkte: **Rational Team Concert**

Welches Versionsverwaltungssystem

Es gibt verschiedene Versionsverwaltungssysteme mit unterschiedlichen Zielen sowie Vor- und Nachteilen.

Die bei der Softwareentwicklung heutzutage am meisten verbreiteten sind ***Subversion***, ***Git*** und ***Mercurial***. Subversion gilt dabei als veraltet, vor allem da es auf eine zentrale Funktionsweise setzt, was das Offline-Arbeiten umständlicher macht. ***Git*** und ***Mercurial*** sind dagegen dezentrale Versionskontrollsysteme, die das getrennte Arbeiten und Verwalten von Projekten durch den Einsatz lokaler Arbeitskopien deutlich verbessern.

Funktional sind *Git* und *Mercurial* miteinander durchaus vergleichbar, jedoch stand hinter Git (vor allem aus historischen Gründen) stets eine deutlich grössere Community, weshalb es mit der Zeit immer mehr zum dominierenden Versionskontrollsystem unserer Zeit aufgestiegen ist.

Die starke Verbreitung, die vielen verfügbaren Werkzeuge und die Unterstützung auf vielen Diensten zur Projektverwaltung macht die Empfehlung leicht: ***An Git kommt man derzeit kaum vorbei***.

Ihr Auftrag:

- Sie haben in diesem Theorie-Block eine generelle Einführung über Versionsverwaltung erhalten, d.h. Sie sollten das Grundkonzept verstanden haben (falls NEIN, lesen Sie nochmals die Theorie durch oder erarbeiten Sie die Grundlagen selbst mit Hilfe des Internets).
- Suchen Sie im Internet eine „Einführung in Git“ - es gibt genügend Tutorials, Online-Dokumentes und Online-Videos zu Git via denen Sie sich mit diesem Tool vertraut machen können.
- Installieren Sie auf einem virtuellen Server (VirtualBox Image) alle Komponenten, die für ein Git-Repository notwendig sind. Es ist Ihnen überlassen welches Betriebssystem Sie als Basis verwenden, Git gibt es für alle Betriebsysteme (OsX, Windows und Linux). Ob Sie Git ab Scratch aufsetzen oder via Docker Image, ist ebenfalls Ihr Entscheid. Am Ende benötigen Sie einfach ein lokales Git Repository, mit dem Sie arbeiten können.
- Erstellen Sie auf dem Server ein Git Repository, in das anschliessend ein Source-Projekt importiert werden kann (das Projekt wird Ihnen zur Verfügung gestellt). Ihr Git Repository soll nicht mit einem existierenden Online Repository (wie GitHub oder GitLab) verknüpft werden.
- Git ist grundsätzlich ein Command-Line Tool, es gibt jedoch auch verschiedene GUI Clients und Plugins (z.B. für Notepad++) via denen die Commands über eine grafische Oberfläche ausgeführt werden können.
- Die Installation ist zu dokumentieren und muss abgegeben werden. Sie werden Ihre Git Installation demonstrieren resp. vorweisen müssen (der Auftrag wird benotet).
- Sie werden basierend aud Ihrem Git Repository auch verschiedene Übungsaufgaben machen. D.h. es ist zwingend, dass Sie ein Git Repository bereitstellen.