Versionsverwaltung

Robert Wiesner

13. März 2011

Inhaltsverzeichnis

1 Inhaltsangabe

Versionsverwaltung soll allgemein, insbesondere aber auch bei der Softwareentwicklung in der Wissenschaft, helfen, den Entwicklungsprozess von Software zu sichern, verfügbar zu halten und zu strukturieren. Versionsverwaltung kann durch verschiedene *Versionsverwaltungssysteme* realisiert werden, welche, wie es der Name nahelegt, Versionen von Daten, also auch einer Software, verwalten. Alle Versionen, die zu einem aktuellen Entwicklungsstand hinführen, stellen eine *Historie* dar. Es kann mehrere Pfade, sogenannte *Branches* geben, auf welchen entwickelt wird, und viele Versionen auf einem Pfad. Alles zusammen ergibt ein *Repository*.

2 Einleitung

Bei der Softwareentwicklung stellt sich früher oder später die Frage, wie der Entwicklungsprozess gesichert werden soll. Sollte wegen eines Fehlers ein Teil der Software gelöscht werden (oder im schlimmsten Fall das gesamte Projekt) ist dies zumindest ärgerlich. Wenn es einen Auftraggeber gibt und Termine zu halten sind kann dies im schlimmsten Fall sogar zu einem Scheitern des Projektes führen. Dies ist nur ein mögliches Szenario, das für eine Versionsverwaltung spricht.
Nicht viel anders stellst sich die Situation bei der Softwareentwicklung in der Wissenschaft dar: Sollte beispielsweise der Datensatz zu einer gegebenen Problemstellung durch eine Manipulation fehlerhaft werden, kann es notwendig werden diesen neu einzuspielen, sollte sich die Manipulation nicht rückgangig machen lassen. Dies kann sehr aufwendig werden. Wenn die Ergebnisse veröffentlicht werden sollen und dies in der Zwischenzeit einer anderen Institution gelingt, wäre die gesamte Arbeit umsonst gewesen. Versionsverwaltung kann Änderungen an Daten sichern und auch Rückgängig machen, ein korrupter Datensatz ließe sich hier korrigieren.
Im Kontext von Versionsverwaltung sind viele Begriffe für ein System, das diese Aufgabe übernimmt, zu finden: Versionsverwaltungssytem, Versionierungssystem, Revisionssytem und viele andere mehr. All diese Begriffe werden im Folgenden unter dem Begriff *Versionsverwaltungsystem* (VVS) zusammengefasst werden. Ein VVS hat die Aufgabe, den Entwicklungsprozess einer Software zu verwalten. Dazu werden Entwicklungszustände als *Versionen* gespeichert. Alle Versionen auf einem Entwicklungspfad sind zeitlich geordnet. Die meisten VVS unterstützen mehrere Pfade, auf denen entwickelt werden kann, sogenannte *Branches*. Dabei werden Versionen immer einem Branch zugeordnet. Versionen auf unterschiedlichen Branches stehen nicht in Relation zueinander.
Die Datenbank, in der die Versionen gespeichert werden, wird *Repository* genannt. In einem Repository wird verwaltet, auf welchen Branches sich welche Versionen in welcher Reihenfolge befinden. Die Versionen, die zu einem bestimmten Entwicklungsstand hinführen, werden *Historie* genannt. Der Datensatz, der zur Zeit bearbeitet wird, ist die *Arbeitskopie*.
Da Versionen Entwicklungszustände sind unterstützt ein VVS ideal das inkrementelle

Entwickeln; beispielsweise ließe sich bei einem zyklischen Softwareentwicklungsprozess, bei welchem nach jedem Zyklus ein Prototyp entsteht, für jeden Prototyp eine Version anlegen (natürlich ließen sich auch weitere Versionen erzeugen). Durch die Historie sind auch ältere Prototypen jederzeit verfügbar, sollte der Bedarf entstehen. Auch wenn parallel durch mehrere Teams eine Software entwickelt werden soll ist dies durch Branches gut möglich. Beim Zusammenführen, dem *Merging* dieser Branches, müssen allerdings, je nach Synchronisationsmechanismus, gegebenenfalls Konflikte aufgelöst werden.
Aus den bisher umrissenen Eigenschaften leiten sich einige Anforderungen ab, die an ein VVS gestellt werden, sollte es in einer professionellen Softwareentwicklungsumgebung, also auch bei der Softwareentwicklung in der Wissenschaft, eingesetzt werden:

- Verfügbarkeit der Historie
- Parallele Entwicklung mit Branches
- Integration von Versionen vieler Entwickler
- Autoreninformationen bei Änderungen
- Koordination der Entwicklung
- Effiziente Fehlerbehandlung und -suche

In den folgenden Abschnitten werden diese Anhand von den wesentlichen Konzepten der Versionsverwaltung und -ssysteme erläutert. Es wird betrachtet, wie grundsätzlich mit VVS gearbeitet wird (Abschnitt 3). Unterschiedliche Architekturen von VVS werden präsentiert (Abschnitt 4). Beispielhaft werden einige VVS vorgestellt und verglichen (Abschnitt 5). Stellvertretend wird anhand von *Git* Theorie und Praxis der VVS vorgeführt (Abschnitt 6). Abschließend werden die behandelten Inhalte zusammengefaßt (Abschnitt 7).

3 Grundsätze beim Arbeiten mit Versionsverwaltungssystemen

Die Arbeit mit VVS lässt sich abstrakt durch drei Schritte beschreiben:

- Initialisieren / Klonen eines Repositories
- Modifikation der Daten in der Arbeitskopie
- Aktualisieren / Synchronisation des Repositories

Der erste Schritt wird einmalig durchgeführt, danach wechseln sich Schritt zwei und drei kontinuierlich ab.
Soll ein Repository angelegt werden muss der Entwickler sich als erstes für ein System seiner Wahl entscheiden. Alternativ kann ein bestehendes Repository geklont werden, in diesem Fall muss zumindest ein kompatibles VVS gewählt werden. Dabei sollte die Wahl davon abhängen, welches VVS den Entwicklungsanforderungen entspricht. Ein exemplarischer Vergleich einiger VVS findet sich im Abschnitt 4.
Modifizierte Daten müssen im Repository aktualisiert werden. Ein solches Aktualisieren wird ein *Commit* genannt. Dabei werden die Änderungen als neue Version persistent im Repository hinterlegt.

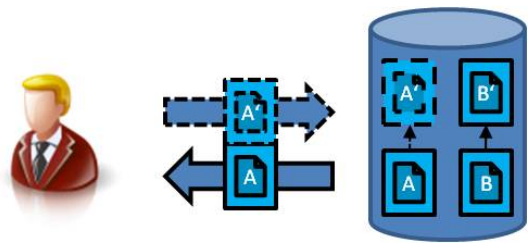

Abbildung 1: Konzept der Arbeitsweise

Sollten mehrere Entwickler an einer Software entwickeln, könnte es beim Commit dazu kommen, dass ein Konflikt entsteht. Daher müssen Konflikte entweder vermieden oder aufgelöst werden. VVS übernehmen diese Aufgabe, in dem sie eine von zwei Strategien verfolgen: *Lock-Modify-Write* oder *Copy-Modify-Merge*. Lock-Modify-Merge vermeidet Konflikte, indem es Bereiche oder Dateien sperrt, Copy-Modify-Merge erfordert eine Konfliktbereinigung, sollte ein Konflikt beim Merging entstehen. Beide Varianten besitzen ihre Vor- und Nachteile. Lock-Modify-Write garantiert, dass keine Konflikte entstehen, was einen potentiellen zusätzlichen Aufwand vermeidet. Jedoch kann an jedem Abschnitt nur ein Entwickler arbeiten und es muss festgelegt werden, welche Bereiche gesperrt werden. Copy-Modify-Merge wiederum ermöglicht auch paralleles Entwicklen an einem Bereich. Ein Sperren ist nicht notwendig. Sollten jedoch zwei oder mehr Entwickler überschneidend Programmieren muss beim Merging ein Konflikt aufgelöst werden. Werden

für den Entwicklungsprozess Schnittstellen definiert, welche einzelne Entwicklerteams voneinander abgrenzen, sollten nur wenige Konflikte entstehen.

4 Architekturen von Versionsverwaltungssystemen

VVS können drei Arten von Versionsverwaltungsarchitekturen nutzen:

- Lokale Versionsverwaltung
- Zentrale Versionsverwaltung
- Verteilte Versionsverwaltung

Dabei kann ein VVS mehrere Architekturen unterstützen.
Lokale Versionsverwaltung ist begrenzt auf einen Arbeitsplatz. Softwareentwicklung in der Wissenschaft findet in Teams statt, ein lokales VVS ist somit eher nutzlos. In Büroanwendungen, wo Änderungen an einzelnen Dokumenten archiviert werden sollen kann eine lokales Versionsverwaltungssystem sinnvoll sein.

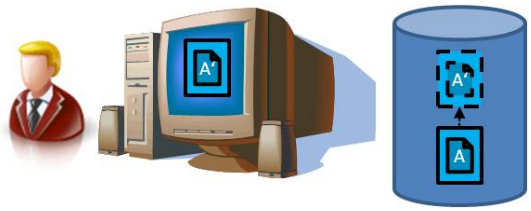

Abbildung 2: Lokale VVS

Zentrale Versionsverwaltung bietet eine vernetzte Entwicklungsumgebung. Somit kann an vielen Arbeitsplätzen entwickelt werden. Jeder besitzt eine Arbeitskopie, an welcher er die Entwicklung vorantreiben kann. Es können beliebig viele Arbeitskopien erzeugt werden. Allerdings existiert nur ein einziges Repository, aus welchem die Arbeitskopien entnommen und in welches sie wieder eingestellt werden. Eine solche Architektur wird auch Client-Server-Architektur genannt. Durch die Zentralisierung wird selbst bei sehr vielen Arbeitskopien der Entwicklungsaufwand vereinfacht. Denn so müssen nur an einer Stelle Konflikte bereinigt werden und können nicht durch mehrere verteilte Repositories entstehen. Die Arbeitskopien stellen eine Momentaufnahme des Repositories da, die zentrale Versionsverwaltung kann als Snapshot-Verfahren angesehen werden.
Durch eine Rechteverwaltung kann eine Zugriffsregelung eingerichtet werden. Die Koordination kann dadurch erleichtert werden. Außerdem kann auch unberechtigter Zugriff verhindert werden.
Bei der *verteilten Versionsverwaltung* gibt es keinen Server mit einem Repository mehr. Jeder Entwickler besitzt sein eigenes Repository, mit welchem er arbeitet. Diese müssen, soll die Entwicklung zusammengeführt werden, miteinander abgeglichen werden. Da es mehrere Repositories gibt, kann die Lock-Modify-Write-Strategie nicht angewendet werden, sondern nur Copy-Modify-Merge.
Sowohl bei der zentralen als auch bei der verteilten Versionsverwaltung kann die Copy-Modify-Merge-Strategie verwendet und viele Entwickler eingesetzt werden. Die Entwickler müssen folglich zwangsläufig mit Konflikten rechnen. Im Idealfall sollte der Umgang

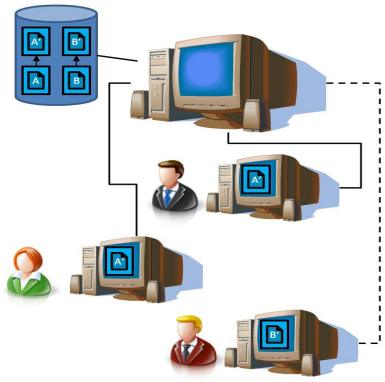

Abbildung 3: Zentrale VVS

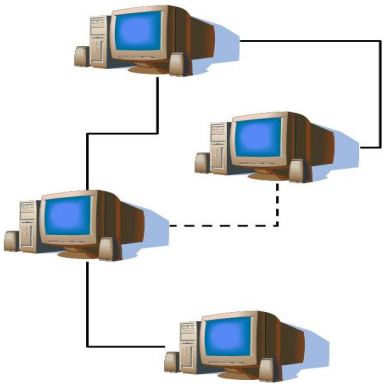

Abbildung 4: Verteilte VVS

mit Konflikten bevor sie auftreten geregelt sein. Auch weniger informationstechnik-affine Wissenschaftler sollten zumindest wissen, wer in einem solchen Fall unterstützen kann oder selbst Konflikte beheben können. Der verteilte Ansatz besitzt dabei den Vorteil, dass eine Entwickler, wenn er einziger Nutzer seines (Teil-)Repositories ist, seine Änderungen commiten kann und für Merges zwischen den Repositories VVS-Experten hinzugezogen werden können. Dies kann ein Beitrag für die Akzeptanz von VVS sein falls Entwickler von konfliktreichen Merges abgeschreckt sein sollten.

5 Beispiele und Vergleiche von Versionsverwaltungssystemen

Um einen auszugartigen Überblick über verschieden VVS zu erhalten werden einige in diesem Kapitel betrachtet und verglichen. Die Auswahl besitzt dabei keinen Anspruch auf vollständigkeit sondern soll einige Vertreter beleuchten.
Source Code Control System (SCCS) und *Revision Control System (RCS)* sind lokale VVS. SCCS wird als das älteste Versionsverwaltungsprogramm bezeichnet. [1] Es wurde 1972 von Marc J. Rochkind geschrieben. Es verwaltet nur einzelne Dateien und ist für größere Projekte nicht geeignet. Es werden jeweils nur die Änderungen an einem Dokument gespeichert, wodurch der Speicherbedarf reduziert wird. RCS verwaltet ebenfalls einzelne Dateien und kann mit SCCS verglichen werden. [3] Es speichert die aktuelle Version und die Differenzen zu den Vorgängerversionen. RCS empfindet sich als "Verbesserung seines Vorgängers SCCS"[2] Auf RCS baut beispielsweise das unter zentralen VVS aufgeführte CVS auf.
Als zentrale VVS lassen sich das *Concurrent Version System (CVS)* und das darauf aufbauende *Subversion (SVN)* nennen. CVS ist 1986 in Form von Skripten veröffentlicht worden und 1990 zu C portiert.[BAE05] Es ist frei erhältlich und auf Windows, Unix und Mac einsetzbar. CVS unterstützt keine atomare Commits, es kann also passieren, dass das Repository in einem inkonsistenten Zustand hinterlässt. Subversion wurde 2004 geschrieben und soll in Anlehnung an CVS dessen Schwächen vermeiden. So implementiert es im Gegensatz zu CVS *Change Sets*, das heißt es werden Änderungen logisch zusammengefasst und registriert werden. [BAE05] Es ist ebenfalls auf den gängigen Plattformen nutzbar und es gibt für viele Entwicklungstools Plugins für SVN.
Bei den verteilten VVS lassen sich das bereits erwähnte *Visual Source Safe*, *BitKeeper*, *Git*, *Monotone*, und *Mercurial (hg)* nennen. Visual Source Safe ist von Microsoft entwickelt und kommerziell. Allerdings gibt es für andere Plattformen Entwicklungen, welche kompatibel zu Visual Source Safe sind. Visual Source Safe basiert auf Source Safe, einem lokalen VVS, welches von Microsoft aufgelauft wurde. Auch BitKeeper ist ein kommerzielles Produkt. Ursprünglich frei, wurde es für die Entwicklung des Linux-Kernels eingesetzt. Als es kommerziell wurde, entwickelte Linus Torvald *Git*. Näheres zu Git und seine Anwendung wird im Kapitel 6 mit erläutert. Monotone diente Git als Vorlage [4] und verwendet Hashes zur Identifikation von einzelner Dateien anstelle von Versionsnummern. Darüberhinaus komprimiert es Dateien im Repository mittels gzip. Mercurial wurde in etwa zei leich mit Git entwickelt und verfolgt ähnliche Ziele.

Eigenschaften	SCCS	RCS	CVS	SVN	VSS	Bk	Git	Mt	hg
Lizenz	GPL	GPL	GPL	Apache	Propr.	Propr.	GPL	GPL	GPL
Vers. von Verz.	X	X	Nein	Ja	Ja	Ja	Ja	Ja	Ja
Changesets	X	X	Nein	Ja	Nein	Ja	Ja	Ja	Ja
Atom. Commits	X	X	Nein	Ja	Ja	Ja	Ja	Ja	Ja

6 Git

6.1 Überblick

Git ist ein verteiltes Versionierungssystem. Viele Entwickler können parallel eine Software entwickeln und haben dabei jeweils ein eigenes Repository. Diese können zusammengeführt werden. Rein physikalisch besteht somit keine Hierarchie, allerdings können logische Hierarchien definiert werden, welche von den Entwicklern einzuhalten sind. Durch eine Hierarchie erleichtert sich der Koordinationsaufwand, da an zentraler Stelle aktuelle Entwicklungsstände abgeglichen werden können. So verringert sich die Gefahr, dass ein Entwickler an einer veralteten Version arbeitet.
Weitere Eigenschaften von Git sind[5]:

- Gute Unterstützung nicht linearer Entwicklung. Einerseits durch die Verteilung an und für sich, des weiteren durch zahlreiche Werkzeuge, welche Navigation und Visualisierung einer nicht linearen Historie unterstützen.
- Effiziente Handhabung großer Projekte. Es nutzt ein effizientes Komprimierungsformat und arbeitet schneller als die meisten anderen VVS. Dies kann bei der Softwareentwicklung in der Wissenschaft die Akzeptanz erhöhen, insbesondere wenn sehr viele und große Dateien verwaltet werden müssen und ein VVS keinen deutlichen Mehraufwandan Platz und Zeit bedeutet.

Jeder Entwickler bei Git hat sein eigenes Repository. Jedoch wird nicht im Repository selbst gearbeitet, sondern, wie in Kapitel 3 beschrieben, in der Arbeitskopie. Ein Git Repository besteht aus Arbeitskopie und Repository. Diese beiden Komponenten stellen die Grobarchitektur (siehe Abbildung 5) von Git da. Das Repository lässt sich in zwei weitere Elemente zerlegen: Den *Index*, welcher als Schicht zwischen den Versionen und der Arbeitskopie angesehen werden kann, und der *Object Store*. Der Index, auch *Staging Area* genannt, ist die Schnittstelle, über welche die Arbeitskopie als Version in das Repository gestellt werden kann. Jedoch enthält der Index nicht selbst die Version, sondern enthält die Änderungen an der aktuellen Version aus dem Object Store, welche beim nächsten Commit als Version gespeichert werden sollen. Es findet ein "Staging" der Änderungen statt, welche commitet werden sollen. Jedoch ist der Index nicht notwendig, wenn eine bestimmte Version als Arbeitskopie verfügbar gemacht wird. Die Gesamtheit aller Versionen und mehr befinden sich im Object Store. Näheres dazu im Abschnitt 6.2. Arbeitskopie, Index und Object Store werden im Folgenden als Feinarchitektur (siehe Abbildung 6) bezeichnet.

6.2 Repository

Wie in Abschnitt 6.1 beschrieben, besteht das Repository aus dem Index und dem Object Store. Der Index enthält die Änderunge, welche durch einen Commit als neue Version angelegt werden sollen. Der Object Store enthält mehrere Objekte:

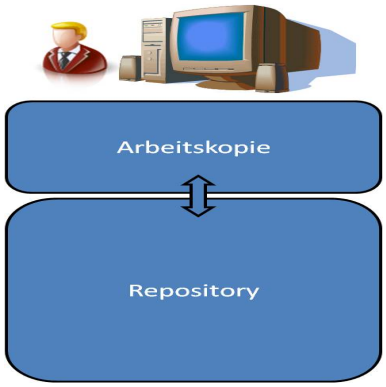

Abbildung 5: Grobarchitektur

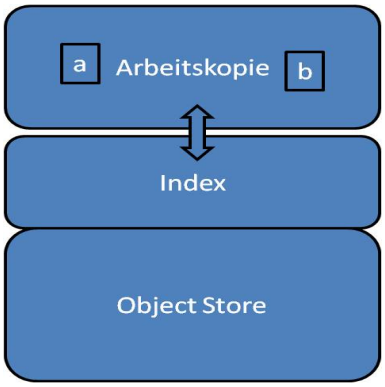

Abbildung 6: Feinarchitektur

- Blobs
- Trees
- Commits
- Tags

Die folgenden Absätze dieses Abschnittes erläutern diese Objekte [LOE09]: *Blob* ist eine Abkürzung und steht für Binary Large Objects. Die Blobs stehen für die verschiedenen Dateiinhalte. Allerdings enthalten sie keinerlei Metadaten über die Dateien oder die Dateinamen.
Ein *Tree* steht für eine Ebene von Verzeichnisinformation. Trees enthalten Pfadnamen, Dateinamen und dazugehörige Blobidentitäten. Sie können rekursiv andere Trees referenzieren. Durch Trees können vollständige Verzeichnis- und Dateihierarchien dargestellt werden.
Die *Commits* enthalten Metadaten. Dies sind Informationen über den Author, das Commitdatum und ein Log, welches Anmerkungen zu dem Commit enthält. Jeder Commit verweist auf einen Baum, welcher die Vollständigen Daten und Metadaten direkt oder über weitere Objekte enthält. Der erste Commit hat keinen Vorfahren, jeder folgende Commit mindestens einen.
Tags können Objekten, meistens Commits, zugewiesen werden um zusätzliche Informationen zu speichern.
Einer der wesentlichen Vorteile von Git ist, dass Commits (Datei-)Inhalte und nicht Dateien speichert. Enthalten zwei unterschiedliche Commits gleiche Inhalte, so werden für diese die selben Objekte referenziert. Ändern sich beispielsweise in zwei Commits ein Vereichnis und die enthaltenen Dateien nicht, so werden die entsprechenden Blobs und der Tree, von beiden Commits referenziert. Es gibt also keine redundante Speicherung von Daten. Ein Objekt kann folglich auch erst gelöscht werden, sobald es von keinem anderen Objekt mehr referenziert wird.

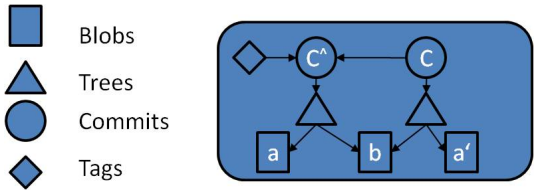

Abbildung 7: Objekte des Object Store

Zur Identifikation von Objekten berechnet Git für jedes Objekt einen SHA1 (oder Synonym: Hash Code oder Objekt-Identifikator). Dieser ist für gleiche Inhalte immer gleich. Somit kann allein durch Vergleich des SHA1 festgestellt werden, ob zum Beispiel aus einem anderen Repository, welches nur über das Internet erreichbar ist, Dateien übertragen werden müssen, wenn man ein Version abgleichen will. Nur wenn der SHA1 unterschiedlich ist, muss übertragen werden.

6.3 Arbeiten mit Git

Um mit Git zu Arbeiten, muss es als erstes installiert sein. Danach wird mit Befehlen auf Konsolenebene gearbeitet. Natürlich gibt es auch zahllose Erweiterungen und Plugins für verschiedene Zwecke und IDEs. Im Folgenden wird nur auf das Arbeiten auf der Konsolenebene eingegangen.
Die Syntax der Git-Anweisungen ist:
$git\ \{\cdots\}\ <COMMAND>\ \{<ARGS>\}$
Im Abschnitt 3 wurden drei Schritte genannt, welche durchgeführt werden, wenn man mit VVS arbeitet. In Git läuft ein typischer, einfacher Arbeitsablauf wie folgt ab:

- Initialisieren / Klonen eines Repositories
  $git\ init\ [<Ziel>]$
  Erzeugt ein neues Repository, entweder im gegenwärtigen Verzeichnis, oder aber im Verzeichnis <Ziel>, falls dieses angegeben wird,
  $git\ clone\ <Quelle>\ [<Ziel>]$
  Erzeugt ein neues Repository als Kopie von <Quelle> im gegenwärtigen Verzeichnis, oder aber im Verzeichnis <Ziel>, falls dieses angegeben wird,

- Modifikation der Daten in der Arbeitskopie Es wird an der Arbeitskopie wie in einem normalen Verzeichnis gearbeitet,

- Aktualisieren / Synchronisation des Repositories
  $git\ add\ (<Datei>\{,\ Datei\}|.)$
  Aktualisiert den Index, indem alle angegebenen <Datei>-en im Index aufgenommen werden, oder alle Dateien in allen Unterverzeichnissen des Repositories, falls ein "." verwendet wurde,
  $git\ commit\ [-a]$

Erzeugt einen neuen Commit im Object Store, sämtliche im Index aufgenommenen Dateien werden aktualisiert. Wird der Parameter "-a" verwendet, werden alle Dateien aus der Version entfernt, welche in der Arbeitskopie gelöscht wurden.

Die entstehende Historie kann als Graph aufgefasst werde, bei welchem die Commits Knoten darstellen, welche jeweils mit ihrem Vorgänger und Nachfolger verbunden sind. Um in einem Repository parallel zu Entwickeln werden die in der Einleitung erwähnten Branches genutzt. $gitbranch < Name >$ erzeugt einen neuen Branch, welcher identisch zu dem gegenwärtigen Commit ist. Betrachtet man das Repository erneut als Graphen, kommt mit dem Branching ein neuer Knoten hinzu, welcher mit dem aktuellen Commit verbunden ist. Durch ein Merging, mittels dem Befehl $git\ merge\ < Quellbranch > [< Zielbranch >]$, wird vom Branch mit dem Namen <Quellbranch> mit dem aktuellen Branch (oder dem angegebenen Branch <Zielbranch>) eine zusammengeführte Version erzeugt. Dies wäre ein neuer Knoten, welcher als Vorgänger die letzten Knoten der am Merge beteiligten Branches hat. Sind von den ursprünglich gemeinsamen Daten nur unterschiedliche Bereiche geändert worden, geht dies problemlos. Sind gleiche Bereiche geändert worden, kommt es zu einem Konflikt, welcher behoben werden muss. Auf das beheben von Konflikten wird hier nicht eingegangen.

7 Zusammenfassung

VVS sind ein gutes Mittel in der Softwareentwicklung. In der Softwareentwicklung in der Wissenschaft ist es nicht nur gut, sondern sogar wichtig. Denn es leistet viele, teils unverzichtbare Dienste. Durch ein VVS kann eine Datensicherung realisiert werden, gibt es wichtige Daten (aus Messungen oder anderen Quellen) kann sichergestellt werden, dass diese immer wieder verfügbar sind. Es sind sämtliche erfassten Entwicklungsstände verfügbar, wurden nicht behebbare Fehler gemacht kann zu einer älteren Version zurückgegangen werden, welche diesen Fehler nicht aufweist. Sollen Entwicklungsergebnisse ausgetauscht werden, kann dies durch ein VVS problemlos realisiert werden, indem ein anderer Entwickler sich die aktuelle Version holt.
Je nach Arbeitsumgebung kann ein passendes VVS gewählt werden. Git ist ein flexibles, performantes, relativ einfach zu bedienendes, gut dokumentiertes VVS, welches sowohl zentrale als auch verteilte VVS unterstützt. Es bietet somit auch für Wissenschaftler mit wenig Informatikhintergrund ein brauchbares Werkzeug. Da jeder Entwickler sein eigenes Repository besitzt, kann das Merging beispielsweise durch einen versierten Mitarbeiter durchgeführt werden, welcher die jeweils aktuelle Version der anderen Mitarbeiter abgleicht. Die Wissenschaftler können sich somit auf das konzentrieren, was im Mittelpunkt stehen sollte: Die Entwicklung.

8 Quellenangaben

[1] http://de.wikipedia.org/wiki/SCCS
[2] http://www.gnu.org/software/rcs
[3] http://de.wikipedia.org/wiki/Revision_Control_System
[4] http://de.wikipedia.org/wiki/Monotone
[5] http://git-scm.com/about
[*BAE*05] Stefan Baerisch vom Juni 2005 , "Versionskontrollsysteme in der Softwareentwicklung"
[*LOE*09] Jon Loeliger vom Mai 2009, "Git"