Ausarbeitung zum Thema "Versionsverwaltung"

Benjamin Schott

27. Mai 2008

Inhaltsverzeichnis

1 Einleitung

Um überhaupt erst einmal klar zu machen, wozu Versionsverwaltungen gebraucht werden, beginne ich mit einer kleinen Einleitung. In dieser stelle ich die Abläufe und möglichen Probleme in Projekten dar, die zwar Dokumente erstellen und bearbeiten, aber gänzlich auf Versionsverwaltungen verzichten. Da sich die Nachteile und Probleme der gleich von mir erläuterten Fälle überschneiden, trenne ich die Fallvorstellung und die Probleme strikt in zwei Abschnitte. So wird zuerst ganz allgemein der jeweilige Ablauf geschildert, um danach zusammenhängend sowohl fallspezifische als auch allgemeine Probleme zu erläutern.

1.1 Arbeiten ohne Versionsverwaltung

Schauen wir uns der Einfachheit halber nur einen kleinen Teil eines beliebig großen Projektes an. Genau genommen picken wir uns lediglich zwei Projekt-Mitglieder heraus, die neben vielen anderen Dokumenten auch am gleichen Dokument arbeiten. Beide Mitglieder, wir nennen sie Harry und Sally, arbeiten auch nicht zwangsläufig zusammen an diesem Dokument, sondern müssen lediglich für ihr Teilprojekt Änderungen an diesem Dokument vornehmen.

1.1.1 Keine zentrale Dokumentenablage

Sehen wir uns zuerst an, wie Harry und Sally an einem Dokument arbeiten, wenn diese jeweils bei ihrem jeweiligen Ersteller abgelegt sind. Wenn Harry nun der Ersteller von Dokument A ist, kann er dieses beliebig bearbeiten, da er es ja direkt bei sich zu liegen hat. Sobald er mit seinen Bearbeitungen fertig ist, legt er A einfach wieder bei sich ab. Jetzt möchte Sally jedoch einige Änderungen an A vornehmen, wofür sie dieses erst zu sich bringen muss. Hier gibt es nun zwei denkbare Situationen:

1. Sally darf ohne Nachfrage Dokumente von Harry zu sich holen
2. Sally muss erst bei Harry nachfragen, ob er ihr A geben möchte

Ersteres ist der unkompliziertere Fall. Sally nimmt sich einfach Dokument A, bearbeitet es wie benötigt und legt es am Ende wieder zurück. Sie kann A also so bearbeiten, als wäre es ihr eigenes Dokument.

Schwieriger wird es da schon, wenn wir die zweite Situation haben. Sally hat in diesem Fall keine Befugnis, ohne Harrys Wissen Dokumente von ihm zu nehmen und zu bearbeiten. Also muss sie vorher mit Harry kommunizieren. Wenn Harry einverstanden ist, was innerhalb eines Projektes jedoch im Allgemeinen angenommen werden kann, wird er Dokument A nehmen und es Sally übergeben. Nun kann sie A endlich bearbeiten und muss es am Ende nur noch an Harry zurückgeben, schließlich bleibt er der Ersteller von A. Da das gerade die Richtlinie für den Ablageort war, darf Sally A nicht selbst behalten.

1.1.2 Zentrale Dokumentenablage

Eine weitere Variante ist die Dokumente eines Projektes alle an einem zentralen Ort abzulegen. Dies ähnelt im Ablauf der ersten Situation in Abschnitt 1.1.1 . Da alle Dokumente am gleichen Ort liegen ist dieser auch für jeden zugänglich. Sally kann sich Dokument A also nehmen, es bearbeiten und wieder zurücklegen. Im Unterschied zum Fall aus 1.1.1 kann sich Sally nun jedoch auch sofort mehrere Dokumente von verschiedenen Urhebern gleichzeitig

heranziehen. Im vorherigen Fall müsste sie dafür erst von Urheber zu Urheber gehen und sich alle gewünschten Dokumente zusammensuchen.

1.2 Probleme

Die im Abschnitt 1.1 geschilderten Vorgehensweisen sind natürlich nicht problemlos. Wären sie dies, gäbe es keine Notwendigkeit für Versionsverwaltungen. Zuerst möchte ich auf die spezifischen Probleme beider Vorgehensweisen eingehen, bevor ich aufzeige, welche Nachteile die oben beschriebenen Abläufe allgemein haben.

1.2.1 Spezifische Probleme

Im ersten Fall von Abschnitt 1.1.1 bekommen wir ein Problem, wenn Sally gleich mehrere Dokumente von verschiedenen Personen bekommen möchte. Dafür muss sie erst zu vielen verschiedenen Orten und ist so erst lange damit beschäftigt, alle Dokumente zu bekommen, bevor sie wirklich produktiv arbeiten kann. Wir haben jedoch im Abschnitt 1.1.2 bereits eine mögliche Lösung für dieses Problem gesehen, da dort alle Dokumente an einem Ort liegen. Dadurch entfallen lange Weg zu verschiedenen Orten.

Im zweiten Fall in 1.1.1 kommt ebenfalls wieder der Zeitfaktor zum Tragen. Grundsätzlich hat Sally natürlich schon mal den gleichen Zeitaufwand wie eben beschrieben, wenn sie Dokumente von verschiedenen Personen haben möchte. Nun kommt jedoch noch die Kommunikations- und Dokumentenübergabedauer hinzu. Anstatt sich ein Dokument ohne weitere Verzögerung zu nehmen, wenn sie den Ablageort erst einmal erreicht hat, muss sie nun noch Zeit investieren, um den Urheber nach einem Dokument zu fragen.

Beide dargestellten Probleme lassen schon vermuten, dass die zentrale Dokumentablage der dezentralen vorzuziehen ist. Doch natürlich hat auch das zentrale Modell seine möglichen Probleme. Diese sind jedoch auch im dezentralen Modell vorzufinden, bzw. dort teilweise noch etwas größer. Deswegen werden diese Probleme im allgemeinen Problem-Abschnitt(1.2.2) behandelt.

1.2.2 Allgemeine Probleme

Die vorgestellten Abläufe gehen implizit davon aus, dass der Mensch nahezu perfekt ist. Damit ist gemeint, dass er nichts vergisst und alles wirklich genauso tut, wie es beschrieben wurde. Leider entspricht diese Idealvorstellung nicht der Realität. Es könnten zum Beispiel folgende Situationen auftreten:

- Sally nimmt Dokument A, aber vergisst es zurück zu geben
- Harry weiß zwar, dass Sally A hat, aber Sally wird krank, ist im Urlaub oder anderweitig unerreichbar
- Harry weiß gar nicht wer A hat

In allen Situationen können weder Harry noch andere Personen an Dokument A arbeiten. Hier fehlt es also an einer guten Synchronisierung, so dass jeder weiß, wer wann und wo an welchem Dokument arbeitet.

Unter Synchronisierung kann man natürlich auch die parallele Arbeit an einem Dokument verstehen. Im bisher erläuterten Vorgehensmodell ist diese unmöglich, denn von einem Dokument gibt es immer nur ein Exemplar. Nur an diesem Originaldokument wird gearbeitet. Ergo

kann niemand anderes gleichzeitig daran arbeiten, wenn es gerade einer Änderung unterzogen wird.

Aufgrund dieses „Nur-Originaldokument“-Prinzips ergibt sich ein weiteres Problem. Änderungen können im Allgemeinen nicht 1:1 zurückgenommen werden. Liegt eine Änderung vielleicht sogar schon Monate zurück, ist es nahezu unmöglich, diese 100%-ig zu reproduzieren. Als einfachstes Beispiel kann man das Löschen von Dokumentteilen betrachten. Wenn nach einiger Zeit erkannt wird, dass der gelöschte Teil doch ins Dokument gehört, muss man ihn zwangsläufig komplett neu verfassen. Dabei könnte man wiederum kleine, aber wichtige, Details des ursprünglichen Teils vergessen.

Ein Problem, was all diese Probleme gleichsam implizieren, ist die höhere Arbeitszeit, wenn solch ein Problem auftritt. Im schlechtesten Fall ist Harry längere Zeit untätig, weil alle Dokumente, an denen er arbeiten könnte oder müsste, für ihn nicht zugänglich sind. Oder er verfasst zumindest keine wirklich neuen Inhalte, weil er Änderungen von Hand wieder rückgängig machen muss. Und das, obwohl er stundenlang arbeitet.

1.3 Lösungen (und neue Probleme)

Die oben genannten Probleme haben im Grunde sehr triviale Lösungen, auf die jeder sicher auch ohne diesen Abschnitt kommen würde. Der Vollständigkeit halber führe ich diese aber nun noch einmal auf. Das erleichtert auch die darauf folgende Betrachtung der neuen Probleme, die durch die Lösung der alten auftreten.

1.3.1 Lösungen

Wie in 1.2.1 bereits erwähnt, können die spezifischen Probleme der dezentralen Dokumentenablage durch Einsatz einer zentralen Ablage behoben werden. Doch die allgemeinen Probleme bleiben dann natürlich bestehen.

In 1.2.2 wurde beispielsweise gesagt, dass Harry ja wissen könnte, bei wem ein Dokument gerade liegt. Selbst wenn Harry vorher gefragt werden muss, kann er sich nicht ohne weiteres merken, wer welches Dokument von ihm hat. Hier müsste also immer in eine Liste eingetragen werden, wer sich welches Dokument genommen hat. So kann man jederzeit einsehen, wo ein Dokument momentan liegt.

Damit ist aber noch nicht gelöst, wie man an einem Dokument arbeiten kann, wenn dieses noch bei jemand anderem in Bearbeitung ist oder diese Person nicht zu erreichen ist. Hier müsste man vom „Nur-Originaldokument“-Prinzip weggehen. Statt dessen sollte das Originaldokument immer an seinem ursprünglichen Ort verbleiben. Wenn Sally jetzt damit arbeiten möchte, holt sie sich lediglich eine Kopie des Dokumentes. Das können auch alle anderen machen, selbst wenn Sally gerade an dem Dokument arbeitet, denn das Original ist ja immer verfügbar.

Das Problem der komplizierten Änderungs-Rücksetzung wird von der Kopie-Methode grundsätzlich auch gelöst. Jedoch muss man diese hierfür noch etwas erweitern. So darf das aktuelle Originaldokument nicht einfach mit der bearbeiteten Kopie ausgetauscht werden. Vielmehr sollte das Ausgangsdokument für eine Kopie erhalten bleiben und die neue Version wird einfach „darüber“ gelegt und allen Personen danach als aktuellstes Dokument präsentiert. So kann man einfach alle älteren Dokumentversionen durchsuchen, um den ursprünglichen Zustand eines Dokumentteils zu finden, den man wiederherstellen möchte.

1.3.2 Neue Probleme

Sowohl das Aufschreiben, wenn man sich ein Dokument holt, als auch das Kopieren des aktuell gültigen Dokuments ziehen zwei neue Probleme nach sich. Zum einen wird der Arbeitsprozess um einen weiteren Vorgang, entweder das Eintragen oder das Kopieren, verlängert. Das mag im ersten Moment noch marginal erscheinen, doch wenn man mehrere Dokumente gleichzeitig haben möchte, ist dieser Zwischenschritt nicht mehr vernachlässigbar klein.

Das Kopieren hat zudem einen weiteren negativen Faktor. Wenn Harry und Sally am gleichen Dokument arbeiten und Harry legt seine Kopie zuerst zurück, darf Sally ihre nicht einfach darüber legen. So würden Harrys Änderungen alle verloren gehen. Sally muss hier zuerst gucken, ob der Dokumentzustand, den sie sich kopiert hat, noch immer der aktuellste ist. Da dies im Beispiel nicht der Fall ist, muss sie ihre Kopie mit dem aktuellen Zustand vergleichen und beide Versionen zusammenfügen.

Auch bei den vorgestellten Lösungen gibt es ein gemeinsames Problem. Alle Lösungsvorschläge ziehen ein erhöhtes Datenaufkommen nach sich. Mit jedem neuen Dokument, dass zur Bearbeitung geholt wird, wird ein neuer Eintrag in der jeweiligen Liste der aktuellen Besitzer getätigt. Durch das Kopierverfahren werden diese Listen zwar nicht mehr benötigt, doch es werden noch mehr Daten erstellt. Eine einfach Rechnung hierzu: Es existieren 10 Dokumente. An jedem arbeiten 9 Personen gleichzeitig. Wenn jemand mit seiner Änderung fertig ist, wird seine Kopie ja zur aktuellen Version, während das oder die vorher bereits aktuell gewesenen Dokumentzustände nicht gelöscht werden. Sind alle Personen fertig, haben wir das Quelldokument und 9 Versionen. Das macht in kürzester Zeit eine Bruttoanzahl von 10 mal 10, also 100 Dokumenten, obwohl es effektiv nur 10 Dokumente gibt. Wenn man sich größere Projekte vorstellt, wird sich dieses Verhältnis noch schneller verschlechtern.

2 Grundlagen

Nachdem wir in Kapitel 1 einen Einblick bekommen haben, was für Probleme und Lösungsvorschläge für diese bei manueller Dokumentenverwaltung[1] auftauchen, kommen wir nun zum eigentlichen Kernthema. Mit dem immer größer werdenden Einsatz von EDV-Systemen in nahezu allen Bereichen werden natürlich auch die zuvor beschriebenen Szenarien immer mehr auf digitale Weise realisiert. Da liegt es nahe, die Arbeitsabläufe und Probleme ebenfalls digital, sprich durch Software, zu lösen. Und da Software im Allgemeinen ebenfalls in Gruppen entwickelt wird, war es natürlich ein trivialer Schritt, dass solche Dokumentenverwaltungssoftware programmiert wird. Denn der Mensch neigt ja bekanntlich dazu, immer wiederkehrende Abläufe zu automatisieren oder zumindest zu vereinfachen. Da Software-Entwickler eben selbst davon profitieren, musste man ihnen diese Arbeit nicht einmal extra schmackhaft machen. Somit möchte ich nun einige Grundlagen solcher Software erläutern und dabei auch zwei grundlegende Vorgehensweisen erklären. Man wird schnell merken, dass es sich hauptsächlich um die Realisierung der in Kapitel 1 beschriebenen Abläufe handelt. In diesem Kapitel wird deswegen auch in erster Linie darauf eingegangen, wie diese Szenarien technisch gesehen in Verwaltungssoftware allgemein abgebildet werden und wie die Fachtermini

[1] Obwohl das Thema Versionsverwaltung lautet, wird hier von Dokumentenverwaltung gesprochen. Gemeint ist dasselbe, nur dass der Begriff der Version im genannten Kontext schon zu speziell gewesen wäre. Da der Kern solcher Software jedoch die Speicherung der verschiedenen Zustände, also Versionen, und deren Nutzung ist, hat sich der Begriff der Versionsverwaltung durchgesetzt. Diese Dokumenten-***History*** wird jedoch noch extra in Kapitel 5 behandelt.

für bestimmte Tätigkeiten innerhalb solcher Systeme heißen.

2.1 Grundlegende Architektur

Es wurde ja schon deutlich, dass die zentrale Dokumentenablage bereits einige spezifische Probleme der dezentralen Dokumentenablage löst. Deswegen haben auch Verwaltungssoftwaresysteme dieses Prinzip als Grundlage. Es gibt also einen zentralen Ablageort. Dieser wird ***Repository*** genannt. Hier lagern alle Dokumente, die für jeden, der auf dieses ***Repository*** zugreifen kann, sichtbar sein sollen. Wie in Abbildung 1 zu sehen, können alle diese Teilnehmer(Clients) auf das ***Repository*** schreiben und von ihm lesen. So also in unseren

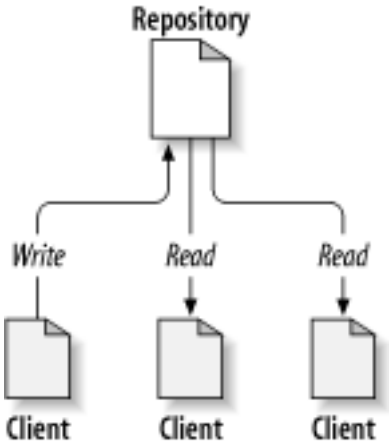

Abbildung 1: Mitarbeiter(Clients) greifen aufs ***Repository*** zu

Beispielen auch Harry und Sally. Die beiden können sich nun natürlich diese Dokumente aus dem ***Repository*** nehmen. Allerdings wird hier auch das Prinzip der Arbeit mit Kopien angewandt. Deswegen haben Harry und Sally dann bei sich nur eine ***Working-Copy***. Diese ist jedoch nicht auf einzelne Dokumente beschränkt. Zu Beginn der Arbeit, wenn beide noch keine Dokumentkopien bei sich haben, müssen Harry und Sally einen ***Check-Out*** machen. Das bedeutet, sie holen sich eine ***Working-Copy*** der aktuellen Versionen aller oder nur bestimmter Dokumente aus dem ***Repository***. Dann können sie mit diesen Kopien arbeiten, ohne dass irgendjemand sonst von ihren Änderungen erfährt oder sie von den Änderungen anderer etwas mitbekommen.

Hat Harry nun alle Änderungen vorgenommen, muss er seine veränderte Version nur wieder zurück in das ***Repository*** schreiben, um sie publik zu machen. Dies wird ***Commit*** genannt. Wenn Harry seine neue Version von Dokument A ***commit***tet kennt Sally diese Version aber im Allgemeinen gar nicht, denn sie hatte bereits vorher ihren ***Check-Out*** gemacht.

Im einfachsten Fall hat sie jedoch in der Zwischenzeit noch keine Änderungen an ihrer, nun alten, Version von A getätigt. Vielleicht hat sie A nur direkt in den ***Check-Out*** mit aufgenommen, weil sie wusste, dass sie irgendwann daran arbeiten will oder weil sie schlicht das ganze ***Repository*** ausgecheckt hat. Nun ist es jedoch ungünstig, wenn sie auf der alten Version weiterarbeitet. Deswegen kann Sally vorher durch ein so genanntes ***Update*** nachsehen, ob es neue Versionen im ***Repository*** gibt. Solch ein ***Update*** bemerkt natürlich auch, wenn ganz neue Dokumente im ***Repository*** hinzugekommen sind. Durch das ***Update*** sieht

Sally aber nicht nur Veränderungen, sondern holt diese direkt in ihre ***Working-Copy***. Das ***Update*** ist also nur ein verkleinerter ***Check-Out***, weil ausschließlich neue und geänderte Dokumente geholt werden und nicht alle vorhandenen. Deswegen macht man nach dem initialen ***Check-Out*** auch nur noch ***Update***s.

Der kompliziertere Fall ist, dass Sally bereits selbst Änderungen an ihrer Kopie von Dokument A vorgenommen hat. Sie möchte jetzt ebenfalls ihre Änderungen ***commit***ten, doch Harry kam ihr bekanntlich zuvor. Dieses Szenario sehen wir in Abbildung 2. Wie bereits in

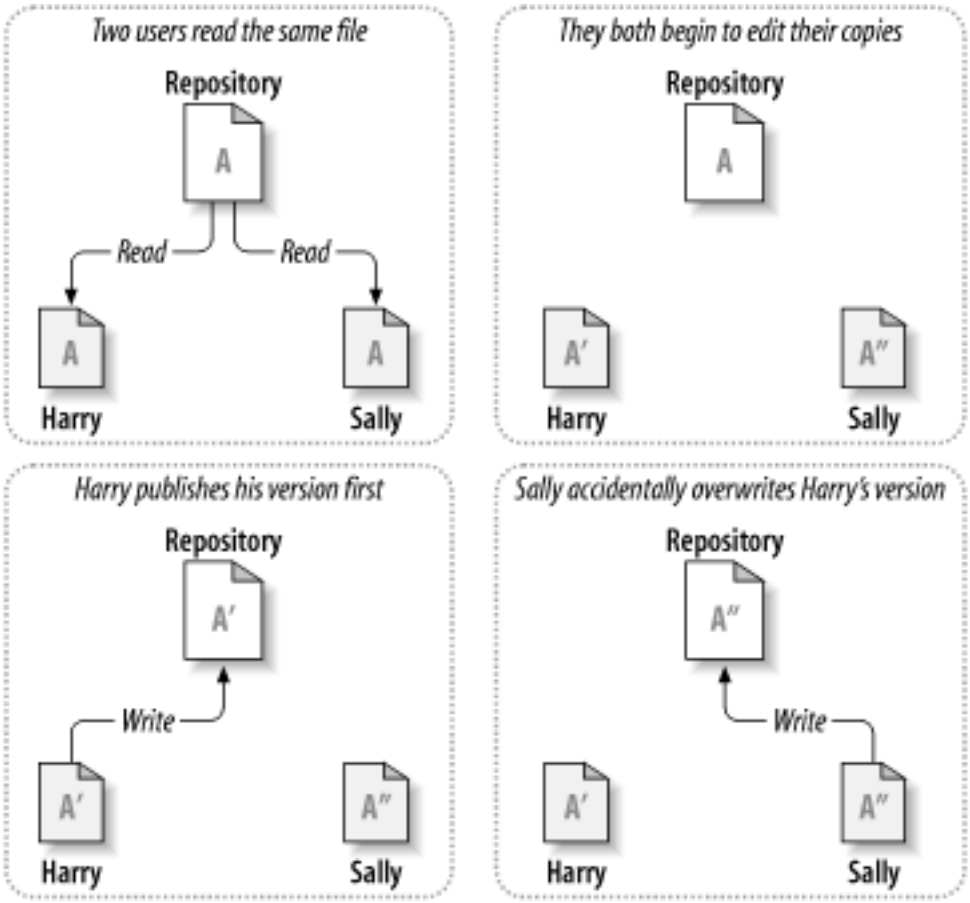

Abbildung 2: Harry und Sally arbeiten unkontrolliert am gleichen Dokument

1.3.2 gesehen wäre ein einfaches Überschreiben der nun aktuellen Version katastrophal, denn Harrys Änderungen würden einfach verschwinden[2]. Um das zu Umgehen gibt es in Versionsverwaltungssystemen zwei Methoden, die nun vorgestellt werden.

2.2 Lock-Modify-Unlock

Dies ist im abstrakteren Sinne das gleiche Bearbeitungsmodell als würden wir ohne Kopien arbeiten. Möchte Harry an A arbeiten sperrt er es vorher im ***Repository***(***Lock***). Wenn Sally

[2]Aufgrund der Versionenspeicherung verschwinden Harrys Änderungen natürlich nicht wirklich, aber sie sind in der aktuellen Version nicht mehr zu ersehen. Jemand, der erst jetzt ein ***Update*** oder einen ***Check-Out*** macht, sieht nun gar nichts von Harrys Arbeit, wenn er nicht explizit danach fragt.

jetzt auch an A arbeiten möchte, müsste sie es ebenfalls sperren. Dies kann sie jedoch nicht, da A bereits gesperrt ist. Jedoch kann sich Sally die Version von A im ***Repository*** ansehen, auch wenn sie es nicht bearbeiten kann. In der Zwischenzeit bearbeitet Harry A(***Modify***) und ***commit***tet seine Version am Ende. Jetzt entsperrt er A natürlich wieder(***Unlock***), wodurch Sally endlich Dokument A in der neuen Version sperren, ***update***n und bearbeiten kann. Abbildung 3 veranschaulicht diesen Vorgang.

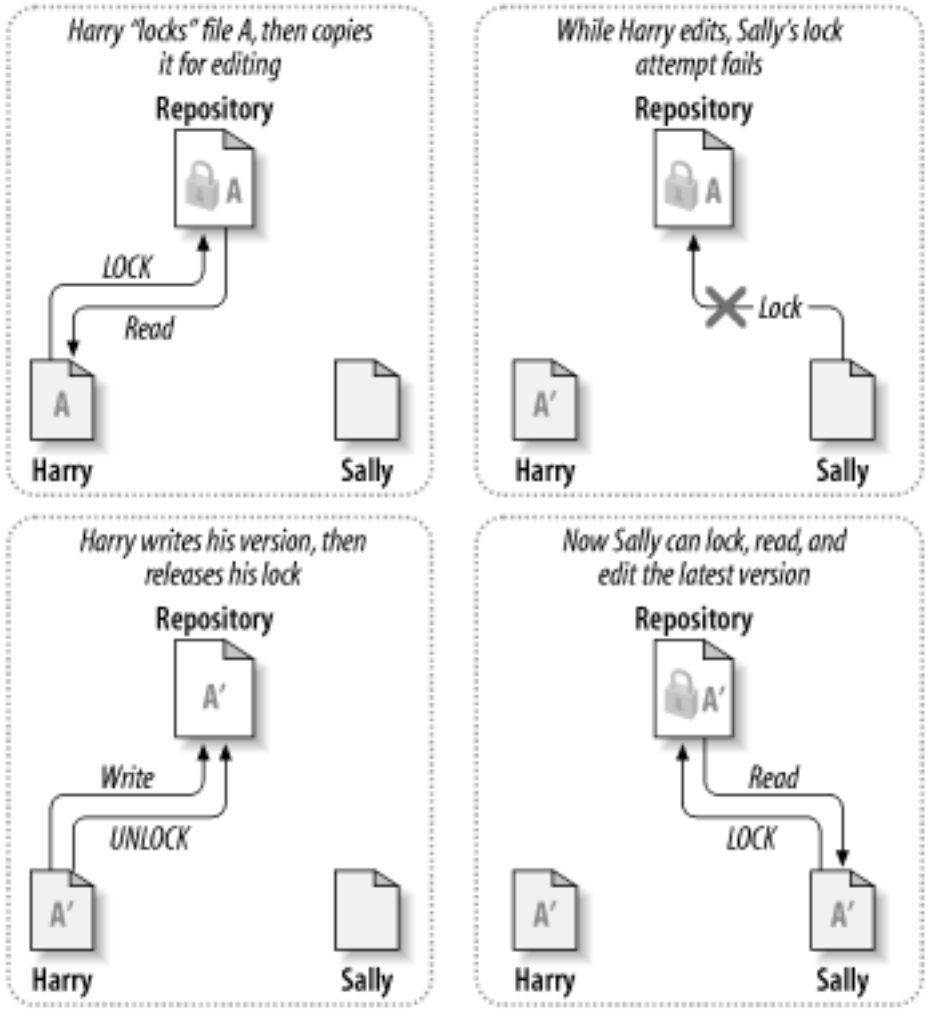

Abbildung 3: "***Lock-Modify-Unlock***"-Ablauf

Es versteht sich, dass sich hier aber ähnliche Probleme ergeben wie bei dem Modell ohne Kopien. Wenn Harry jetzt das Entsperren vergisst und/oder in Urlaub fährt oder krank wird, bleibt A für alle anderen gesperrt. Allerdings gibt es die Möglichkeit, dass einer der Administratoren die Sperrung aufheben kann. Jedoch verzögert sich dadurch die Arbeit wieder, da Sally zum Beispiel nicht unbedingt direkt weiß, ob Harry noch an A arbeitet.

Eine weitere Möglichkeit ist, dass Harry und Sally an ganz unterschiedlichen Stellen von A arbeiten möchten. Hier würden die Änderungen sich nicht überschneiden und beide könnten problemlos gleichzeitig A bearbeiten. Dabei muss dann aber sicher gestellt werden, dass die

beiden ihre Änderungen in irgendeiner Weise am Ende zusammenmischen. Stattdessen verstreicht jedoch unnötig Zeit, bis beide ihre Änderungen vorgenommen haben.

2.3 Copy-Modify-Merge

Bei dieser Methode passiert anfänglich nichts anderes, als in 2.1 bereits beschrieben. Harry und Sally kopieren sich Dokument A(***Copy***) und beide bearbeiten es(***Modify***). Nun macht Sally zuerst ihren ***Commit***. Zum Zeitpunkt von Harrys ***Commit*** geht das Szenario vom Ende des Abschnitts 2.1 weiter. Wenn Harry seine Version ins ***Repository*** stellen möchte vergleicht die Software, ob er die aktuelle Version im ***Repository*** bereits kannte, also ob Harry seine Änderungen an dieser getätigt hat. Da Sally aber bereits ***commit***tet hat meldet die Software Harry, dass die Ursprungsversion seiner ***Working-Copy Out-of-Date*** ist und lehnt den ***Commit*** ab. Abbildung 4 zeigt den Ablauf bis zu diesem Punkt Nun muss Harry

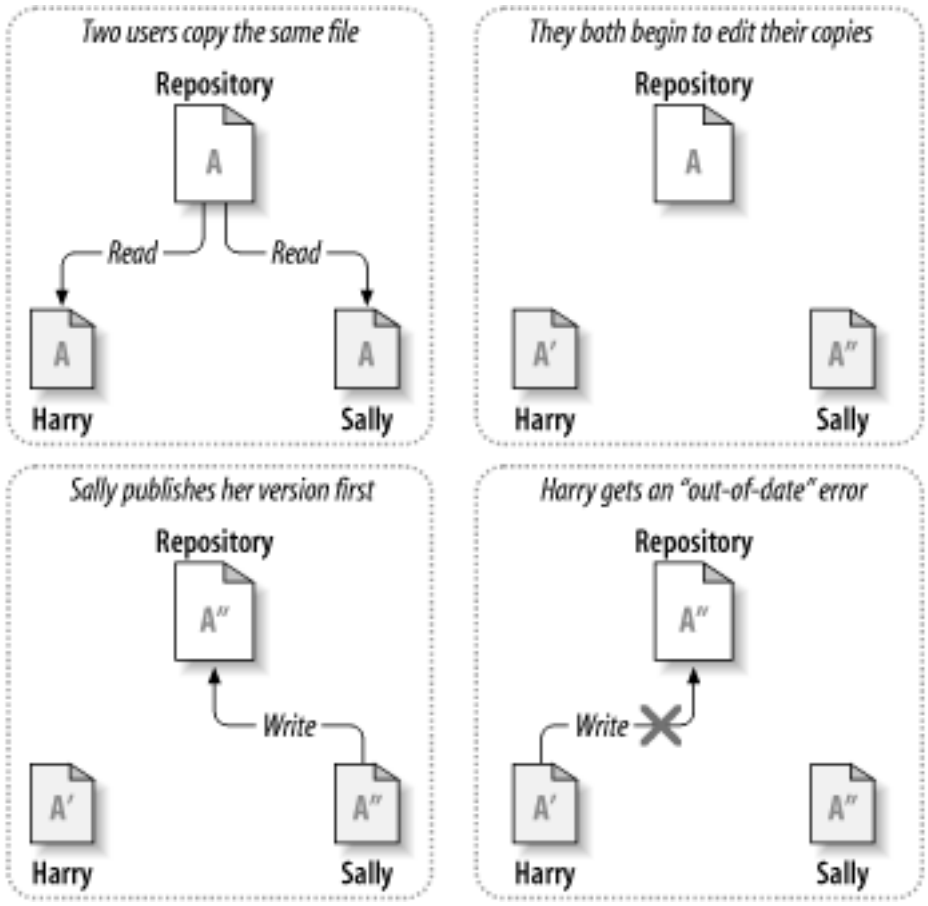

Abbildung 4: "***Copy-Modify-Merge***"-Ablauf bis zur ***Out-of-Date***-Fehlermeldung

vorerst ein ***Update*** machen. Er hat jetzt seine Version und die aktuellste ***Repository***-Version bei sich. Diese beiden Versionen müssen nun bei Harry gemischt werden(***Merge***), bevor er sie wieder ***commit***ten kann. Am Ende sind nun die Änderungen von Harry **und** Sally Teil der aktuellsten Version, was in Abbildung 5 gezeigt wird.

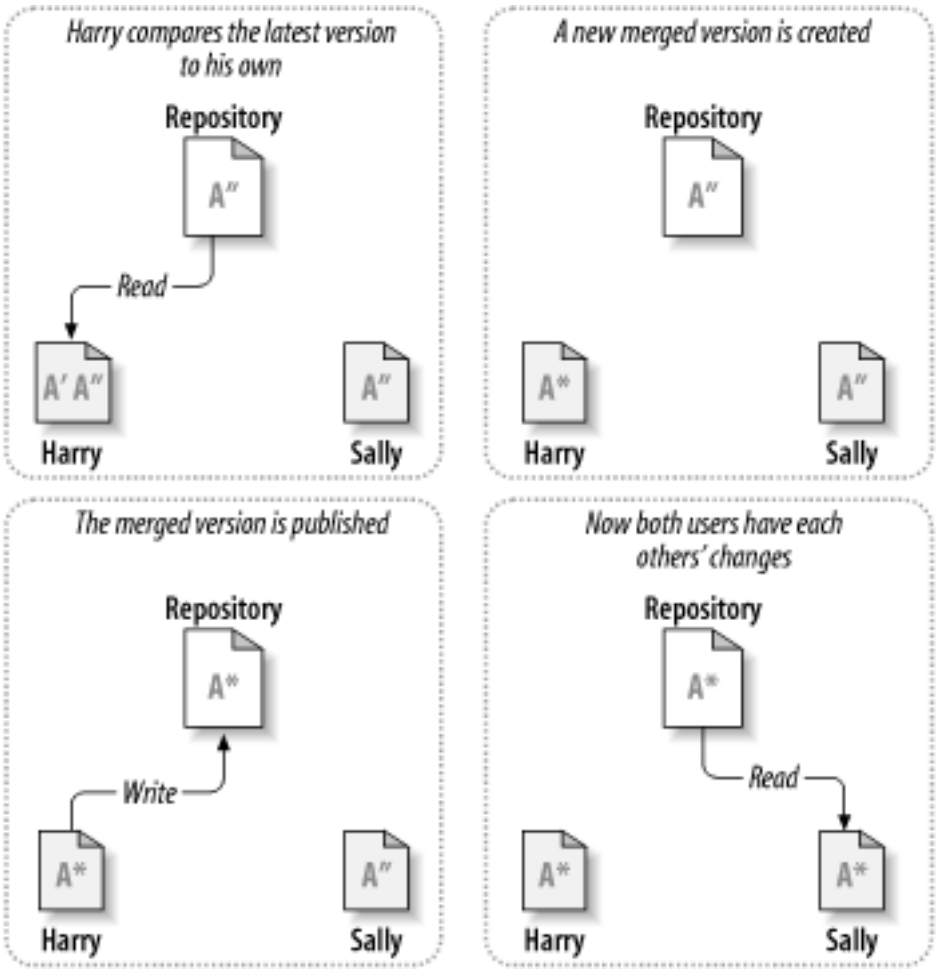

Abbildung 5: Weiterer "***Copy-Modify-Merge***"-Ablauf

2.4 Merge

In nahezu allen Systemen wird das ***Copy-Modify-Merge***-Modell zur Anwendung gebracht. Das ***Lock***en von Dokumenten wird in manchen Systemen lediglich als Feature mit angeboten, um Dokumente gezielt vor gleichzeitigen Bearbeitungen zu schützen. Somit ist das ***Merge***n, also das Mischen von verschiedenen Dokumentversionen, eine essentielle Technik in Versionsverwaltungssystemen, um Parallelarbeit zu ermöglichen. Gleichzeitig ist die Software nicht immer dazu in der Lage das ***Merge***n komplett allein durchzuführen. Daher wird dieser Technik ein eigener Abschnitt gewidmet.

Sollten die zu ***merge***nden Versionen an komplett unterschiedlichen Stellen verändert worden sein, kümmert sich die Software um den gesamten ***Merge***-Vorgang. Die von beiden Versionen unbearbeiteten Stellen werden einfach beibehalten, während die bearbeiteten Stellen aus der jeweiligen Version genommen werden. So entsteht ein komplettes Dokument mit allen Änderungen von beiden Bearbeitern, das nun ***commit***tet werden kann. Hierbei muss gesagt werden, dass Versionsverwaltungssysteme den Vergleich auf Basis der Zeilen vornehmen. Damit ist das automatische ***Merge***n nur bei Textdokumenten möglich. Binärdokumente müssen

in jedem Fall von Hand gemischt werden, da hier kein Text im direkten Sinn vorhanden ist, oder sie müssen gesperrt werden.

Doch auch bei Textdokumenten kann ein manuelles ***Merge***n von Nöten sein, nämlich wenn die beiden ***Merge***-Kandidaten an gleichen Stellen geändert wurden. Software ist schlicht nicht der Lage zu evaluieren, welche Version die bessere ist bzw. in welcher Weise man diese Änderungen vielleicht auch verbinden kann. Versionsverwaltungssoftware erkennt nur Zeichen, aber nicht die Bedeutung dahinter. Hier können nur die Bearbeiter kontrollieren, was für die Anforderungen wirklich das Beste ist. Also muss derjenige, der das ***Merge***n vornimmt, die Änderungen in beiden Versionen vergleichen und manuell die Änderung übernehmen, die besser ist, oder ein Konglomerat aus beiden bilden. Hierfür ist es natürlich notwendig, mit den Urhebern der anderen Änderungen zu kommunizieren, denn die möchten ja sicher nicht, dass ihre Arbeit einfach verschwindet. Der aufmerksame Leser wird jetzt sagen, dass hier ja wieder Zeit aufgewendet wird. An dieser Stelle muss man die Differenz zwischen Zeitersparnis durch die parallele Arbeit und dem Zeitaufwand sehen. Wenn man nun keine zu großen Änderungen zwischen zwei ***Commit***s tätigt, dürfte diese Differenz eindeutig zugunsten der Zeitersparnis ausfallen.

3 Branching

Ein noch größeres Hilfsmittel für Parallelarbeit als das ***Merge***n bietet uns das Abzweigen einer Untermenge von Dokumenten, das ***Branch***en. Stellen wir uns vor Harry soll in Dokument A eine große Änderungen vornehmen. In der Software-Entwicklung könnte dies zum Beispiel die Implementierung einer großen Methode oder gar eines ganzen Programm-Features sein. Das dauert natürlich länger und benötigt sicher einige Zwischen-***Commit***s. Doch eigentlich möchte man möglichst immer konsistente und korrekte Dokumente im ***Repository*** haben. Das wäre nicht der Fall, wenn erst ein Teil einer großen Gesamtänderung im Dokument steht. Einerseits würde dann ein Teil des Dokuments unter Umständen für andere sinnlos erscheinen und andererseits würde das Dokument in bestimmten Fällen sogar unbrauchbar. Nehmen wir zum Beispiel wieder an, das Dokument gehört zu einem Software-Projekt und in diesem Dokument stünde Quellcode. Wenn dann eine unfertige Methode dort drin steht oder diese auf Klassen zugreift, die noch gar nicht existieren, ist sie nicht mehr kompilierbar und kann so weder ausgeführt geschweige denn getestet werden. Zwar könnte man diesen Code auskommentieren, aber dies würde das Dokument für andere nur wieder unübersichtlicher machen.

Ein Vorschlag wäre, dass Harry seine Arbeit bis zum Ende nicht ***commit***tet. Dies hat aber nun einige große Nachteile. Zum einen verliert Harry die in Kapitel 1 bereits angesprochene einfache Möglichkeit Änderungen 1:1 zurück zu nehmen, da seine alten Versionen nicht automatisch gespeichert bleiben. Zum anderen möchte oder muss Harry vielleicht an verschiedenen Rechnern arbeiten und müsste A dafür erst manuell transferieren. Bei einem Dokument mag dies noch in Ordnung sein, aber sobald man größere Mengen bearbeiten muss, um etwas Neues einzubauen, wird das auch wieder sehr aufwendig. Gleichzeitig muss er die Dokumente auch selbst an andere Personen verteilen, wenn er bereits während der Arbeit ein Feedback für diese erhalten möchte.

Selbstverständlich gibt es noch weitere Anwendungsfälle. Beispielsweise könnte gewünscht sein, einen Teil des Projekts in eine etwas andere Richtung zu entwickeln oder verschiedene Richtungen erst zu testen. In einem Software-Projekt könnte dies der Fall sein, wenn man

sich noch nicht im Voraus entscheiden kann, mit welcher Technik ein bestimmtes Feature realisiert werden soll. Diese alle im einzig vorhandenen Entwicklungszweig zu implementieren ist zudem nur mit extremer Mehrarbeit auch für damit nicht beauftragte Gruppen verbunden. Schließlich müsste dann jede Technik eigene Methodennamen und/oder Schnittstellen haben. Nun müsste jede Gruppe, die das zu implementierende Feature lediglich nutzt, in ihrem Teil alle Techniken ansprechen, um diese testen zu können. Das wird schnell unübersichtlich und kompliziert. Zudem müssen die Techniken, die man am Ende ablehnt, wieder aufwendig herausgesucht und gelöscht werden. Hier würde es auch nicht helfen, dass Versionsverwaltungssysteme das Zurücksetzen von Änderungen vereinfachen. Gerade weil alle Gruppen, die mit den verschiedenen Techniken beauftragt sind, ihre Implementierungen in das gleiche Dokument ***commit***ten, müsste man erst aufwendig alle ***Commit***s heraus filtern, in denen die abgelehnten Techniken bearbeitet wurden.

Viel einfacher wäre es doch, weiterhin die Funktionen des Versionsverwaltungssystems zu nutzen, aber dennoch den eigentlichen Entwicklungszweig nicht anzurühren, bis wir sicher sind, dass unsere Änderungen wirklich „einsatzfähig“ sind. Eine weitere Möglichkeit wäre, ein eigenes ***Repository*** für den Nebenzweig zu erstellen. Aber auch hierbei wird die Arbeit durch das notwendig werdende Kopieren zwischen den ***Repositories*** nur unnötig komplizierter.

Also erstellen wir diesen Nebenzweig ganz einfach im gleichen ***Repository***. Dafür kopiert man einfach die Dokumente, auf die der eigene Zweig aufbauen soll, in einen separaten Ordner. Nun kann man seine eigenen Änderungen auf diesem Entwicklungszweig tätigen und stört andere Personen überhaupt nicht. Der Hauptentwicklungszweig enthält noch immer einen konsistente Version, im Software-Szenario also zum Beispiel ein funktionierendes Programm. Natürlich sind noch weitere ***Branch***es und sogar ***Branch***es von ***Branch***es möglich. Dies ist in Abbildung 6 verdeutlicht. In seinem Zweig kann man sich nun sprichwörtlich gese-

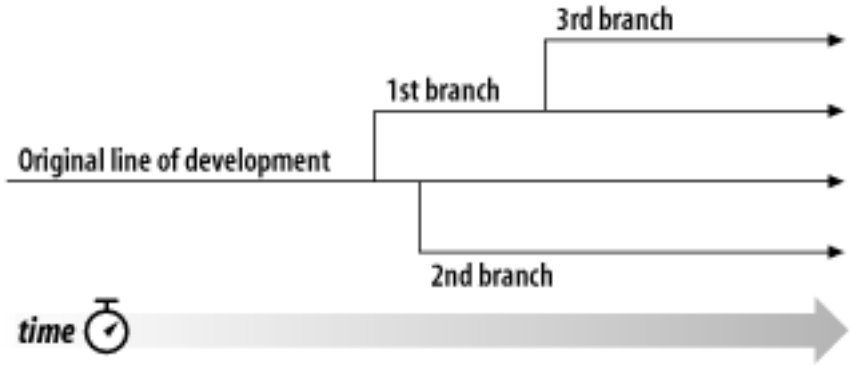

Abbildung 6: ***Branch***en vom Hauptzweig und anderen ***Branch***es

hen austoben. Wenn die Gesamtbearbeitung dann am Ende doch von entscheidungsbefugten Personen abgelehnt wird löscht man den Zweig und muss nach der dann vielleicht ganz umsonst gewesenen Arbeit keine Zeit mehr für das Zurücksetzen der Änderungen aufwenden und kann sofort die Arbeit an einem anderen Teilprojekt aufnehmen.

Wenn die Arbeit jedoch nicht umsonst war, sondern in den Hauptzweig übernommen werden soll, bleibt noch ein letzter Schritt. Dies ist natürlich das ***Merge***n des ***Branch***es in den Hauptzweig. Doch damit muss der ***Branch*** noch nicht beendet sein. Durch das ***Merge***n wird der ***Branch*** ja nicht automatisch gelöscht. Vielleicht wollte der Auftraggeber oder Projekt-

leiter einfach nur einen Teil der großen Änderung bereits im Hauptdokument haben, welches schon fertig gestellt war. Abbildung 7 zeigt diesen Vorgang. Beim ***Merge***n des ***Branch***es mit

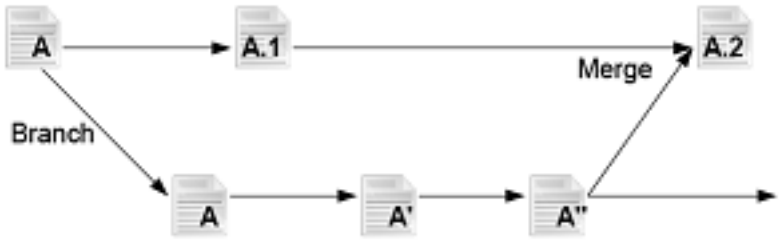

Abbildung 7: Allgemeiner ***Branch***lebenszyklus

dem Hauptzweig ist es wahrscheinlich, dass ein großer Teil manuell eingefügt werden muss. Im Endeffekt ist dies aber weniger Mehrarbeit als bei mehreren Zwischen***merge***s, da im ***Branch*** ja auch Teile wieder gelöscht oder verändert worden sein können. Bei einem großen ***Branchmerge*** hat man dann jedoch nur die „richtigen“ Änderungen zu evaluieren.

4 Verwaltungsmodelle

Bei den heutigen Versionsverwaltungssystemen herrschen zwei Modelle vor. Beide basieren im Grunde auf den bereits beschriebenen Konzepten und Methoden. Jedoch liegen den Modellen jeweils andere Konzepte zu Grunde. Wobei diese Modelle trotzdem durch das jeweils andere realisiert werden können, wenn auch mit wenigen Abstrichen. Zum Schluss dieses Kapitels werden die beiden Modelle noch einmal verglichen.

4.1 Zentrale Versionsverwaltung

Die zentrale Versionsverwaltung ist im Grunde die direkte Umsetzung der bisher beschriebenen Grundlagen. Es gibt ein zentrales ***Repository***. Dieses kann entweder auf einem eigenen Server liegen oder auf einem einfachen Rechner, auf den alle Projektangehörigen über ein Netzwerk zugreifen können. Dieses kann ein lokales Netzwerk sein, ein Intranet oder natürlich auch das Internet selbst. Nun kann der Benutzer sich eine lokale ***Working-Copy*** auf sein System aus***check***en, diese bearbeiten und seine Änderungen in das ***Repository commit***ten. Wie bereits erwähnt wird in nahezu jedem System das ***Copy-Modify-Merge***-Modell verwendet, wobei einige auch das Sperren von Dokumenten anbieten. Natürlich ist das ***Branch***ing ebenfalls in solchen zentralen Systemen möglich, indem ein Teil des ***Repositories*** in diesem kopiert wird. Dieser ***Branch*** ist dann für jeden anderen, der auf das ***Repository*** zugreifen darf, zugänglich.

4.2 Verteilte Versionsverwaltung

Die verteilte Versionsverwaltung ist ein relativ neues Modell, dass jedoch immer mehr Verbreitung findet. Im ersten Moment wird man an das erste Beispiel aus Kapitel 1 denken, wo jedes Dokument bei seinem Urheber abgelegt wird. Dies hat jedoch mit dem in diesem Abschnitt beschriebenen Modell nur wenig zu tun. Vielmehr bedeutet „verteilt“ in diesem Kontext, dass viele ***Repositories*** vernetzt werden, anstatt ein einziges, zentrales zu haben.

Im Gegensatz zum zentralen Modell, wo das ***Branch***en zwar ebenfalls eine wichtige Methode ist, aber dennoch nur ein Feature, hat das verteilte Modell die Idee der Abzweigungen als Grundprinzip. Allerdings wird die in Kapitel 3 erläuterte Anwendung noch etwas modifiziert. Ganz zu Anfang gibt es natürlich nur ein ***Repository***, in dem die ersten Dokumente liegen. Diese ***Repository*** gehört aber nur zu einem Teilnehmer oder zumindest nur einer Teilnehmergruppe. Alle weiteren Teilnehmer holen sich dann anfangs eine Kopie von diesem ***Repository***. Das klingt genauso wie bei der zentralen Versionsverwaltung, jedoch holen sich die Teilnehmer keine wirkliche ***Working-Copy***. Vielmehr erschaffen sie einen neuen ***Branch***. Doch dieser wird nicht auf dem Ursprungs***repository*** erzeugt. Die Teilnehmer haben ihr eigenes ***Repository*** auf dass sie den Inhalt des anderen ***Repositories*** kopieren. Nun können sie mit ihrer Kopie arbeiten, ohne die Arbeit der anderen Teilnehmer anzutasten. Sie haben also ihren eigenen ***Branch*** des Ursprungsprojekts.

Mit ihrem eigenen ***Repository*** können die Teilnehmer nun arbeiten, als würden sie ein zentralisiertes System benutzen. Sie ***commit***ten ihre Änderungen also auch nicht in das Ursprungs***repository***, im Allgemeinen dürfen sie es nicht mal, sondern lediglich auf ihr eigenes. Bisher ist das System nichts anderes, als wenn jeder noch mal bei sich ein zentralisiertes System installiert und in dieses seine ***Working-Copy*** hinzufügt. Es gibt allerdings zwei wichtige Unterschiede. Erstens gibt es grundsätzlich kein übergeordnetes zentrales ***Repository***. Das erste dient hier nur als Ausgangspunkt, ist danach aber gleichwertig zu den anderen ***Repositories***. Als zweites sind die einzelnen ***Repositories*** in diesem Modell alle miteinander vernetzt. Zwar darf man wie bereits erwähnt normalerweise nicht in ein anderes ***Repository*** ***commit***ten, man kann jedoch trotzdem in die anderen ***Repositories*** schauen und sich auch deren Inhalte holen. Man ***pull***t sie, also zieht sie von den ***Repositories*** anderer Personen.

4.3 Vergleich Zentral vs. Verteilt

Wie im einleitenden Text zu diesem Kapitel gesagt, kann man mit den beiden Modellen das jeweils andere teilweise simulieren. Mit einem zentralisiertem System braucht man dazu eben nur ein eigenes ***Repository*** bei sich zu installieren. Da hier aber verschiedene ***Repositories*** nicht miteinander vernetzt werden können, können die anderen Teilnehmer den Inhalt eines anderen lokalen ***Repositories*** nicht einsehen. Dazu muss man den Inhalt seines lokalen ***Repositories*** erst in seine ***Working-Copy*** des zentralen ***Repositories*** kopieren und dieses dann ***commit***ten.

Andersherum kann man in einem verteilten System einem ***Repository*** einfach die Bedeutung eines zentralen ***Repositories*** per eigener Definition zukommen lassen. In diesem Fall stellt sich jedoch die Frage, warum man dann nicht direkt ein zentralisiertes System benutzt. Außerdem muss man sich dadurch wieder auf das Netzwerk verlassen.

Dies ist nämlich einer der großen Vorteile des verteilten Modells. Wenn das Netzwerk beim zentralen Modell nämlich einmal ausfällt hat man keine Möglichkeit seine Änderungen zu ***commit***ten oder ein ***Update*** zu machen. Im verteilten Modell ***commit***tet man einfach in sein lokales ***Repository***. So kann man trotz Netzwerkausfalls von der Speicherung aller Dokumentversionen profitieren.

Gleichzeitig kann man auch ganz unabhängig vom Ursprungsprojekt arbeiten. Das kann zum Beispiel bei Open-Source-Projekten zur Anwendung kommen. Hier kann sich jemand die bisher entwickelte Software ganz nach seinen eigenen Vorstellungen modifizieren ohne das eigentliche Projekt zu korrumpieren. Gleichzeitig können die Ursprungsentwickler sich andere Teile, an die sie vielleicht sogar nie gedacht haben, mit Hilfe der beschrieben Technik des

Pullens in ihr Projekt integrieren.

Auf der anderen Seite kann es aber auch gewünscht sein, dass ein Projekt nicht von anderen Person für eigene Zwecke ohne weiteres weiterentwickelt werden kann. Genauso muss man bedenken, dass die Entwicklung durch jemand anderen in einem verteilten System komplett außerhalb der Kontrolle der Verantwortlichen des ursprünglichen Projektes liegt.

Auch sind verteilte Systeme bei kleinen und mittleren Projekten möglicherweise zu übertrieben. In unserem Projekt wäre so ein System komplett fehl am Platz. Wir haben ein bestimmtes Projektziel und keinerlei Notwendigkeit, geschweige denn Ressourcen, um nebenbei noch in eine komplett andere Richtung zu entwickeln, so dass ein extra ***Repository*** notwendig wäre. ***Branch***es können wir ja auch einfach mit einem zentralisierten System benutzen. Zudem verkompliziert ein verteiltes System den Entwicklungsvorgang zusätzlich, da man im Allgemeinen alle anderen ***Repositories*** nach den aktuellsten Versionen durchsuchen muss oder man müsste noch extra Zeit dafür einplanen, in der jemand aus allen ***Repositories*** die aktuellsten Versionen zusammensucht und zentral zur Verfügung stellt.

5 History

Es wurde bisher viel darüber gesprochen, wie Versionsverwaltungen arbeiten und welche Hilfsmittel sie bieten. Der Grund für die Wahl des Oberbegriffs „Versionsverwaltung“ wurde bisher jedoch eher stiefmütterlich behandelt. So wurde lediglich des Öfteren der Umstand angeführt, dass die Systeme alle alten Versionen eines Dokuments abspeichern. In den gespeicherten Versionen könne man nachsehen, wie eine Stelle früher exakt ausgesehen hat und diese dadurch 1:1 auf diesen alten Stand zurückbringen. Wir reden hier jedoch von Software und damit liegt es nahe, dass dieser Ablauf ebenfalls automatisiert werden kann. Doch durch das Anlegen dieser Dokumentengeschichte, der History, sind noch viele andere Anwendungen möglich, die Versionsverwaltungen so nützlich machen.

5.1 Hilfsmittel History

Bei jedem ***Commit*** wird eine Versionsnummer vergeben. Je nach System hat dabei jedes Dokument seine eigene Nummer oder aber das gesamte ***Repository*** hat eine einheitliche Nummer, die mit jedem ***Commit*** inkrementiert wird. Diese Versionsnummern ermöglichen es, zwei Versionen gezielt miteinander zu vergleichen.

Genauso ermöglichen die Versionsnummern das ***Merge***n von Teilen eines ***Branch***es. Anstatt den ganzen ***Branch*** in einen anderen Zweig zu mischen kann nur der Unterschied zwischen zwei bestimmten Versionen in diesen eingefügt werden. Dies kann zum Beispiel der Fall sein, wenn man bei einem Software-Projekt im ***Branch*** zwischendurch nur einen Bugfix ***commit***tet. Dieser ist für die Hauptversion der Software vielleicht ebenfalls interessant. Jedoch ist das Hauptziel des ***Branch***es noch nicht erreicht. Also sagen wir dem System, dass es nur die Änderung des ***Commit***s in den Hauptzweig ***merge***n soll, der den Bugfix enthält(siehe Abbildung 8).

Doch auch das einfache ***Merge***n am Ende eines ***Branch***es wird durch die History vereinfacht. Hätten wir immer nur eine aktuelle Version eines Dokuments im System könnten wir durch das Erstellen einer Kopie trotzdem einen Nebenzweig eröffnen. Wenn jetzt im Hauptzweig jedoch Stellen entfernt werden, weiß das der Nebenzweig nicht und man müsste die zu ***merge***nden Dokumenten auch an diesen Stellen vergleichen. Durch die Versionierung können wir jedoch dem System sagen, dass es nur die Änderungen zwischen der aktuellen

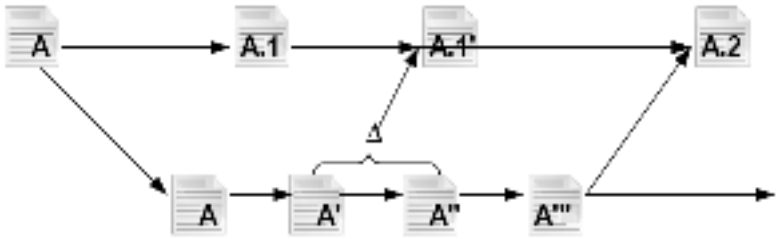

Abbildung 8: ***Merge***n von Teiländerungen eines ***Branch***es

Versionen und der Version, die für den ***Branch*** kopiert wurde, in den Hauptzweig ***merge***n soll. Dies kann man in Abbildung 9 sehen.

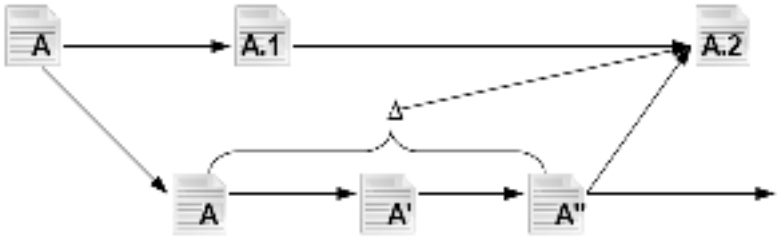

Abbildung 9: ***Merge***n eines ganzen ***Branch***es

Darüber hinaus wird in der History auch immer gespeichert, wer welchen ***Commit*** getätigt hat. So wird es überhaupt erst möglich zum Beispiel im Falle einer ***Out-of-Date***-Meldung direkt heraus zu kriegen mit wem man bezüglich des bevorstehenden ***Merge***s kommunizieren muss. Gleichzeitig könnte eine Person zum Beispiel ein gesamtes Dokument überprüfen um die Struktur neu zu ordnen. Wenn diese Person nun auf eine Stelle trifft, die für sie erstmal keinen Sinn oder Nutzen ergibt, kann man nachsehen, mit welchem ***Commit*** diese Stelle auf den vorhandenen Stand gebracht wurde und damit auch, wer dafür verantwortlich ist. Dieser Verantwortliche kann dann zu Rate gezogen werden.

Damit man nicht unbedingt immer in den Dokumenten selbst nachschauen muss, was geändert wurde, kann man die ***Commit***s auch mit Kommentaren versehen. Bei Verbesserungen von Rechtschreibfehlern in Quell-Code-Kommentaren reicht es ja, wenn man bereits aus dem ***Commit***kommentar ersehen kann, dass es sich bei den Änderungen eben nur um sprachliche Verbesserungen handelt.

5.2 Datenmessi Versionsverwaltung (?)

In jedem bisherigen Kapitel wurde mehr oder weniger deutlich erwähnt, dass Versionsverwaltungssysteme sich gerade dadurch auszeichnen, dass sie alle Versionen einen Dokuments abspeichern und so jederzeit jede Änderung 1:1 rückgängig gemacht werden kann. Im Abschnitt 5.1 wurde darauf noch näher eingegangen und erläutert, wie diese History auch noch in anderer Weise nützlich ist und wie jede Änderung auch einfacher nachvollzogen werden kann. Doch ist das so denn wünschenswert? Immerhin wurde in Abschnitt 1.3.2 doch gerade das hohe Datenaufkommen als Problem der angebotenen Lösungen kritisiert. Jetzt basiert

die History aber doch gerade auf der Lösung, alte Versionen nicht auszutauschen, sondern unter der aktuellen weiterhin im ***Repository*** zu behalten. Dazu kommt noch die Methode des ***Branch***ings, die als so nützlich empfunden wurde, dass sie ein eigenes Kapitel bekam. Wenn man sich erinnert wurde jedoch gesagt, dass ***Branch***es im Grunde Kopien von Dokumenten sind, auf denen dann unabhängig von den Ursprungsdokumenten gearbeitet werden kann. Insgesamt häufen dann aber gerade diese beiden als so wunderbar dargestellten Dinge, die History und das ***Branch***ing, Unmengen an Daten an. Zwar wird Speicher immer günstiger, doch müssen wir ihn deswegen verschwenden?

Doch diese Sorgen können problemlos beruhigt werden. Diese Anhäufung von Dokumentversionen und Kopien sieht nämlich nur für den Benutzer so aus, als wäre jede einzelne Instanz davon ein echtes Dokument. In Wirklichkeit speichern Versionsverwaltungssysteme lediglich die Änderungen zwischen zwei Versionen. Das bedeutet, dass nur das initiale Dokument wirklich als Ganzes im ***Repository*** liegt. Danach speichert die Software bei jedem ***Commit*** nur welche Stellen in welcher Weise modifiziert wurden. Das Gleich passiert bei Kopien. Es wird keine wirkliche Kopie angelegt. Vielmehr wird eine Referenz auf die kopierte Version erstellt und danach ebenfalls nur noch die Änderung von Version zu Version gespeichert. Diese Deltas zwischen zwei Versionen werden ***Changeset***s genannt. Es entstehen also kaum neue Daten, lediglich das ***Changeset*** wird angelegt. So kann die Software auch noch schneller arbeiten, wenn man das in 5.1 beschriebene Szenario des ***Merge***ns von Änderungen zwischen bestimmten Versionen durchführt. Würde jede Version als echtes, vollständiges Dokument gespeichert, müsste die Software erst selbst das Versionsintervall überprüfen und die darin getätigten Änderungen extrahieren. So nimmt es jedoch einfach nur alle gespeicherten ***Changeset***s dieses Intervalls. Bei überschneidenden Teilen in ***Changeset***s wird natürlich der aktuellere Teil genommen, da hier der ***Commit***ter ja bereits evaluiert haben sollte, dass dies die beste Version ist. Ansonsten wäre die Änderung ja nicht in der aktuelleren Version vorhanden. Wenn man nun die aktuelle oder dank der History eine bestimmte ältere Version betrachten möchte, wird diese von der Software durch Anwendung aller ***Changeset***s bis zur angeforderten Version auf die Anfangsversion erstellt, so dass dem Benutzer ein komplettes Dokument angeboten werden kann.

Zum Schluss sei jedoch noch einmal darauf hingewiesen, dass Versionsverwaltungssysteme grundsätzlich auf den Zeilen in Textdokumenten arbeiten. Natürlich können auch Binärdokumente verwaltet werden, da diese aber keine Texte im eigentlichen Sinne sind, können die Systeme nicht einfach das ***Changeset*** speichern. Bei Binärdokumenten wird also wirklich bei jedem ***Commit*** und bei jeder Kopie eine komplette Instanz des Dokuments gespeichert. Hier sollte man also doch auf das Datenaufkommen achten.

6 Zusammenfassung

Versionsverwaltungssysteme automatisieren viele Dinge, die man bei manueller Verwaltung von Dokumenten nur mit sehr viel Zeitaufwand realisieren könnte. Die Software hingegen übernimmt die meisten Vorgänge und führt sie automatisch nach einem simplen Befehl des Benutzers aus. Alle Dokumente werden in einem ***Repository*** gehalten, aus dem sich die Mitarbeiter die für sie relevanten Dokumente auf ihren lokalen Rechner holen können. Nach Bearbeitung werden die Änderungen, also das ***Changeset***, ins ***Repository committet***, um sie allen zur Verfügung zu stellen. Hier muss der Benutzer nun nicht mehr selbst überprüfen, ob die letzte Version im ***Repository*** auch wirklich die Basisversion seiner Bearbeitung ist.

Die Software meldet ihm diesen Fall automatisch. Genauso automatisch geschieht das ***Merge***n seiner Version und der aktuellen des ***Repositories***, wenn die ***Changeset***s sich nicht überschneiden. Erst bei Überlappungen muss der Benutzer manuell eingreifen. Manche Systeme bieten auch das Sperren von Dokumenten an, um Überlappungen zu vermeiden. Doch aufgrund des potentiellen Zeitverlustes ist dies nur eine Option für bestimmte Anwendungsfälle. Zur weiteren Unterstützung der parallelen Arbeit kann man sich auch ***Branch***es erstellen, um viele Änderungen erst ins eigentliche Dokument einzuspeisen, wenn die Änderungen komplett und korrekt sind. Neben dem Modell mit einem einzigen zentralen ***Repository*** gibt es auch noch das verteilte Verwaltungsmodell, in dem viele ***Repositories*** vernetzt sind. Jeder kann ein Projekt nach seinem Willen und seinen Vorstellungen weiter entwickeln, wobei jeder auch ohne Probleme die Entwicklungen anderer vernetzter Personen in das eigene Projekt übernehmen kann. Das Kernelement aller Versionsverwaltungen ist jedoch die History der Dokumente. Jede Version eines Dokuments bleibt erhalten und bei jedem ***Commit*** eines ***Changeset***s werden der ***Commit***ter und ggf. ein Kommentar dazu abgespeichert, so dass jede Änderung nachvollziehbar und zuzuordnen ist. Da jedoch nur die Änderungen zwischen zwei Versionen gespeichert werden, entstehen keine Datenberge, wenn es sich nicht gerade um Binärdokumente handelt.

7 Fazit

Versionsverwaltungen sind bei der Arbeit mit sich weiter entwickelnden Dokumenten ein sehr komfortables Hilfsmittel. Die Zeit, die man ohne solche Systeme in Zwischenschritte, wie das Beschaffen der aktuellsten Version oder das Reproduzieren von getätigten Bearbeitungen, investieren muss, wird durch die Übernahme nahezu aller solchen Aufgaben durch die Software drastisch reduziert. Lediglich in Bereichen, wo eine Entscheidung über das weitere Vorgehen zwingend notwendig ist, muss der Mensch noch selber eingreifen. Dies ist jedoch ein Zeitaufwand, den jeder von uns gerne auf sich nimmt. Schließlich haben es Maschinen(vielleicht zum Glück) in ihrem Urteilsvermögen noch nicht so weit gebracht, dass man blind darauf vertrauen könnte, dass sie auch die richtige Entscheidung treffen. Abgesehen von diesem Eingriff muss der Benutzer aber trotzdem im gesamten Umgang mit Versionsverwaltungen einige Dinge beachten, um die Möglichkeiten auch effektiv nutzen zu können. So ist es ungünstig, nach jeder Zeile einen ***Commit*** zu machen, da andere Personen dann nur noch damit beschäftigt sind, ihre ***Working-Copy up***zu***daten***. Gleichzeitig sollten die ***Commit***s aber auch nicht zu große ***Changeset***s beinhalten, sonst wird es enorm schwer, diese wieder in den Entwicklungszweig einzufügen. Auch sollte der Benutzer die Möglichkeit der Kommentare zu ***Commit***s nutzen. So können unter Umständen lange Kommunikationswege bei der Evaluierung von Änderungen entfallen, da im Kommentar bereits die nötige Information steht. Trotz dieser zusätzlichen Anforderungen an den Benutzer lösen Versionsverwaltungen aber dennoch genug Sorgen, wie zum Beispiel die des Speicherbedarfs. Wer also meint, er bräuchte keine Versionsverwaltungssoftware mag damit möglicherweise Recht haben, er schafft sich aber unter Umständen mehr Arbeit und Kosten als ihm lieb ist.

Abbildungsverzeichnis

Die Abbildungen 1 bis 6 wurden [CSFP04] entnommen.

Literatur

[CS05] Ben Collins-Sussman. The risks of distributed version control. http://blog.red-bean.com/sussman/?p=20, November 2005.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[Gui07] *A Visual Guide to Version Control*, September 2007. http://betterexplained.com/articles/a-visual-guide-to-version-control/.

[Tor07] Linus Torvalds. Re: clarification on git, central repositories and commit access lists. http://lwn.net/Articles/246381/, August 2007. Dies ist eine Antwort von Linus Torvalds auf eine Mail von Adam Treat, Mitglied der KDE-Community.

[Whe04] David A. Wheeler. Comments on open source software / free software (oss/fs) software configuration management (scm) systems. http://www.dwheeler.com/essays/scm.html, April 2004.

[Wik08a] Wikipedia. Distributed revision control — wikipedia, the free encyclopedia, 2008. [Online; accessed 27-May-2008].

[Wik08b] Wikipedia. Revision control — wikipedia, the free encyclopedia, 2008. [Online; accessed 27-May-2008].