CM Paper

Git oder Subversion?

Git oder Subversion?

Subversion hat seinen Vorgänger CVS praktisch vollständig verdrängt, denn es ist schlicht das bessere Werkzeug für zentralisierte Versionskontrolle. Doch dann kam Git: genial, offen und verteilt. Mit seiner explosiv wachsenden Präsenz sogar in großen Softwarehäusern ist Git inzwischen weit mehr als nur das New Kid On The Block. Aber ist Git *besser* als Subversion? So mancher grundlegende Unterschied zwischen beiden Tools spricht dagegen, dass Subversion das Schicksal seines Vorgängers teilen wird.

Subversion entstand mit dem ausdrücklichen Ziel, ein besseres CVS zu werden. Noch heute, inzwischen *Apache™ Subversion®*, ist diese historische Abstammung deutlich erkennbar, und wird ganz besonders durch Gits völlige Andersartigkeit kontrastiert. Denn Git entstand teils mit dem ausdrücklichen Ziel, anders als CVS und Subversion zu sein – es ist nicht einfach „das bessere Subversion", sondern funktioniert auf nahezu jeder Ebene essenziell anders als Subversion, von genereller Arbeitsweise bis hin zu Merging Details. Welches der Tools „besser" ist lässt sich ebenso wenig sagen wie man, sprichwörtlich, Äpfel mit Birnen vergleichen kann. Git eröffnet einerseits ganz neue Möglichkeiten für die Arbeit mit Versionskontrolle und kann vor allem beim Merge glänzen, andererseits kann Git ganz naturbedingt nicht alle Features abdecken, die für Subversion selbstverständlich sind.

Was Git nicht bieten kann

File Locking ist, wenn eine Datei für einen Benutzer reserviert wird, sodass kein anderer gleichzeitig Änderungen an derselben Datei vornehmen kann. Angenommen, in einem Repository ist eine Pixelgrafik eines Cover-Fotos abgelegt. Nun hat Fred beispielsweise den Auftrag, alle Hautpartien zu glätten, während Frieda einen Farbabgleich vornehmen soll. Frieda ist schneller fertig und speichert ihre Änderungen im Repository. Fred stellt etwa drei Stunden später fest, dass der Ausgangspunkt seiner Arbeit sich inzwischen verändert hat. Entweder er überschreibt Friedas Version der Pixelgrafik und verliert so den Farbabgleich, oder er schießt seine eigene Arbeit in den Wind. So oder so muss

eine der Arbeiten *nocheinmal* gemacht werden, da die gemachten Änderungen nicht automatisch verschmolzen werden können. Um solche Situationen zu vermeiden, bietet Subversion die Möglichkeit, ausgewählte Pfade im Repository mit einem *svn:needs-lock* Property zu sperren [goo.gl/APFgi]. Im Beispiel wäre die Pixelgrafik damit schreibgeschützt, bis Frieda ein exklusives Bearbeitungsrecht in Form eines *File Locks*, welches im zentralen Subversion Repository abgelegt wird, erlangt hat. Fred wird daraufhin warten müssen, bis Frieda fertig ist und das Lock wieder freigibt, bevor wiederum er das exklusive Bearbeitungsrecht erlangen kann. So wird Doppelarbeit vermieden.

Das ausschlaggebende Wort im obigen Beispiel ist: *zentral*. Git arbeitet nicht zentralisiert (siehe Abb. 1), sondern verteilt (siehe Abb. 3). Neben vielen Vorteilen, die diese Arbeitsweise bietet, gibt es aber auch den Nachteil, dass es nicht *das eine*, definitive, zentrale Repository gibt (bestenfalls gibt es ein konventionell vereinbartes Repository als zentralen Sammelpunkt, siehe Abb. 2). Fred hat sein Git Repository, Frieda hat ihr Git Repository; beide können in ihren Git Repositories beliebig viele Commits vornehmen, bevor sie dann beim nächsten Pull oder Push feststellen werden, dass eine der Arbeiten erneut gemacht werden muss. Es gibt schlicht keinen zentralen Mechanismus, der Fred unmissverständlich mitteilt, dass Frieda noch beschäftigt ist – also wird bei der Arbeit mit Git ein separater Mechanismus benötigt, mit dem Mitarbeiter sich exklusives Bearbeitungsrecht an schwer verschmelzbaren Dateien mitteilen. Neben Pixelgrafiken gehören dazu alle Dateiformate, für die es keinen verlässlichen automatischen Merge gibt – also auch Office-Dokumente, Video, Audio und viele mehr.

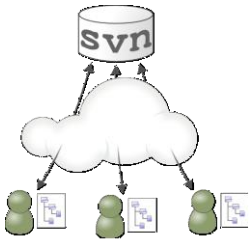

Abb. 1 Subversion hat ein zentrales Repository

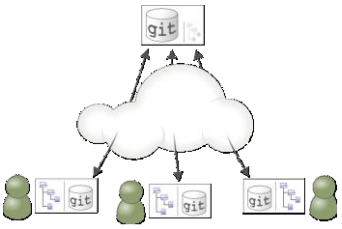

Abb. 2: Git Arbeitskopien sind vollständge Repositories; oft wird sich auf ein zentrales „upstream“ Repository geeinigt

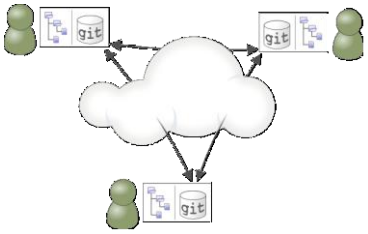

Abb. 3: Git kann problemlos ohne bestimmte Zentrale, also peer-to-peer, verwendet werden

Ein weiteres Feature, das nur Subversion bietet, hat dagegen nichts mit seiner zentralisierten Struktur zu tun. Jeder beliebige Unterpfad kann mit Subversion ganz unabhängig vom Rest des Pfadbaums heruntergeladen und bearbeitet werden (Abb. 4). Im gleichen Zuge wird **pfadbasierte Zugangsberechtigung** möglich: in einem Subversion Repository können verschiedenen Teams unterschiedliche Lese- und Schreibrechte in verschiedenen Bereichen zugeteilt werden [goo.gl/pwhrd]. Zum Beispiel kann einem externen Dienstleister ein sehr eingeschränkter Lesezugang zu nur kleinen Teilen ihrer Subversion Repositories gewährt werden, sodass eventuelle Betriebsgeheimnisse geschützt bleiben. Zudem können die Zugangsberechtigungen mit Subversion in beliebiger Komplexität zusammengestellt und jederzeit leicht verändert werden. Mit Git dagegen ist jede Arbeitskopie eine vollständige Kopie nicht nur des gesamten Pfadbaums, sondern auch der gesamten Entwicklungshistorie (Abb. 5). Lese- und Schreibrecht wird in Git meist für das gesamte Repository erteilt und kann bestenfalls auf einzelne Branches aufgelöst werden. Eine Unterteilung des Pfadbaums in Unterbereiche können Sie bei Git mit sogenannten Submodules [goo.gl/LZl8s] erreichen – dies sind quasi ineinander geschachtelte Git Repositories. Zwei große Nachteile von Submodules sind aber, dass die verschiedenen Zugangsbereiche schon von Anfang an festgelegt sein müssen und im Nachhinein nur umständlich verändert werden können [goo.gl/70nzx und goo.gl/ypb5I], und dass überschneidende Zugangsbereiche nur bedingt umsetzbar sind: Gibt es im Repository zum Beispiel ein Dutzend Produkte, deren technische Bereiche nur die dem jeweiligen Produkt zugeteilten Entwickler sehen dürfen, deren Dokumentation jedoch für redaktionelle und übersetzende Teams sichtbar sein soll, müsste in Git erstens jeder Produktbereich und zweitens die Dokumentation jedes Produkts von vorn herein in einem jeweils eigens abgespaltenen Submodule, also einem separaten Git Repository, abgelegt sein. Diese rigide Trennung ist in Subversion nicht nötig, sodass dort die Historie bereichsübergreifend ohne Brüche aufgezeichnet wird.

Kurz erwähnt sei, dass Subversions Historie vollständig unveränderlich ist. In Git ist es durch eine ungünstige Verkettung von Fehlanwendung und Fehlkonfiguration [goo.gl/oMQoH] möglich, dass die Branch-Historie unsinnig umstrukturiert wird, sodass ursprüngliche Commit Objekte nun keinem Branch mehr angehören. Bleibt das unbemerkt, so könnte einige Zeit später (per Voreinstellung zwei Wochen) Gits Garbage Collector die ursprünglichen Commit Objekte einsammeln und beseitigen. Tatsächlich ist das Risiko hier kaum größer, als wenn in Subversion erlaubt wird, ein vergangenes Commit Log zu korrigieren, welches dann durch Fehlanwendung verloren gehen könnte. Trotzdem lässt sich eindeutig sagen: Subversion hat keinen Garbage Collector. Einigen wenigen Anwendern könnte das als Feature gelten.

Darüber hinaus bietet Subversion noch andere weniger spektakuläre Features, die nichtsdestotrotz Git überlegen sind:

- Strukturen leerer Verzeichnisse werden in Subversion vollständig aufgezeichnet. Git dagegen zeichnet hauptsächlich Inhalte auf („Git tracks content, not files"), wodurch leere Verzeichnisse dann verloren gehen.
- Subversion kann für die Arbeit mit großen Binärdateien besser geeignet sein, da eine Arbeitskopie jeweils nur die neueste Version beinhalten muss statt der gesamten Historie.
- Subversions Kommandozeile ist für Viele einfacher zu erlernen als Gits [goo.gl/9FqkC]; einige Leser mögen aufschreien: natürlich ist dieser Punkt rein subjektiv.
- Subversion bietet, neben seiner Kommandozeile mit Skripting-freundlicher --xml Option, seit der ersten Stunde eine wohldefinierte API, d. h. eine vollständige Programmierschnittstelle in C/C++, und via Bindings auch für Python, Perl, Ruby und Java. Git beschränkt sich leider auf die sogenannten Plumbing Commands der Git Kommandozeile – jedoch ist mit *libgit2* [goo.gl/xlyi1] inzwischen eine „saubere" Neuimplementierung von Git mit vollwertiger C API in Entwicklung.

Was Git wirklich besser kann

Allein Gits verteilte Natur bietet unbestreitbar unzählige Vorteile, darunter:

- Auch ohne dauernde Netzwerkanbindung sind alle Git Features verfügbar, während Subversion für jeden Commit, jeden Merge und jedes Log eine Verbindung zum Server braucht. Mit Git können Sie also an beliebigen Orten, auch ganz ohne Netzanbindung, in gänzlich neuer Qualität arbeiten als es mit Subversion möglich war.

- Jede erstellte Arbeitskopie bildet eine redundante Kopie der gesamten Historie, also jeweils ein weiteres robustes Backup. Das befreit zwar nicht von der Notwendigkeit einer mehrfach gesicherten Backup-Strategie (wie das unlängst gerade noch abgewendete „KDE Git disaster" [goo.gl/HnQ9n] anschaulich verdeutlicht), jedoch besonders für verteilte Teams eigenverantwortlicher Individuen ist die inhärente Redundanz und Unabhängigkeit attraktiv. Subversions Arbeitskopien dagegen sichern nur jeweils den allerneusten Stand.
- Fällt das „Haupt-Repository" aus (oder entspricht nicht mehr den Vorstellungen), können Mitarbeiter nahtlos eine andere Arbeitskopie zum neuen „Haupt-Repository" machen. Um jedoch ein Subversion Repository gleichwertig zu ersetzen muss Zugang zu, oder ein vollständiges Backup (bspw. ein *svnsync* Mirror [goo.gl/2Ym8c]) von dem *einen* zentralen Subversion Repository verfügbar sein.

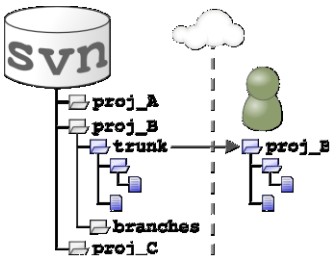

Abb. 4: Ein Subversion Repository enthält oft viele verschiedene Projekte. Ein Benutzer lädt nur einen kleinen Teil zum Bearbeiten herunter.

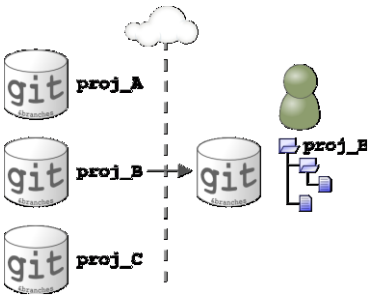

Abb. 5: Ein Git Benutzer lädt immer das gesamte Repository herunter und hat dadurch die gesamte Historie sowie alle Branches lokal verfügbar.

Gits verteilte Natur ist auch einer der zwei Gründe dafür, dass ein Git Merge um Längen schneller läuft als ein Subversion Merge: In jeder Git Arbeitskopie liegt die gesamte Historie lokal bereit und kann rasend schnell vollständig analysiert werden, während ein Subversion Merge mit langwieriger Netzwerklatenz erst alle Information über die Merge-Quelle vom Subversion Server abfragen muss. Natürlich muss ein Git Nutzer beim ersten *clone* Vorgang mitunter weitaus länger auf seine Arbeitskopie warten als ein Subversion Nutzer auf ein schlichtes *checkout*; danach jedoch wird die Wartezeit mindestens wettgemacht, indem Git nur noch vergleichsweise selten Kontakt zu einem Server benötigt [goo.gl/BtwQE].

Der zweite Grund, warum Subversion im Merge langsamer ist als Git, lautet: Subversions Modell ist übermäßig komplex. Git betrachtet nicht umsonst den gesamten Pfadbaum als Einheit; dies entspricht schlicht der alltäglichen Praxis und erlaubt gleichzeitig eine dramatische Vereinfachung des Merge Modells (abb. 6). Ironischerweise empfiehlt Subversion aus Effizienzgründen seinen Nutzern, in ähnlichen Strukturen zu arbeiten wie Git sie voraussetzt [goo.gl/QixKM]: ein Unterpfad-Merge (Abb. 7) ist zwar möglich, macht aus dem Unterpfad aber gewissermaßen einen Spezialfall. Eine Häufung solcher Spezialfälle verlangsamt den Merge spürbar. Nahezu immer wird auch ein Subversion *trunk* als unzertrennbare Einheit verwaltet. Allerdings „weiß" Subversion nichts davon: dem Benutzer ist es jederzeit erlaubt, aus der einheitlichen Struktur auszubrechen und Unterpfade abzuspalten. Einerseits ist es damit möglich und empfehlenswert, viele verschiedene Projekte in ein und demselben Subversion Repository in beliebigen Strukturen zu verwalten (bei der Apache Software Foundation ASF werden alle gegenwärtig 149 Projektverzeichnisse im selben Subversion Repository verwaltet [goo.gl/AnJb7]). Andererseits muss Subversion aber in jeder neuen Revision auf jeder Pfadebene mit einer Vielzahl von möglichen Komplexitäten rechnen und quasi jede Datei und jeden Ordner einzeln auf Spezialfälle überprüfen. Im Kontrast zu Subversions komplexem, jedoch starrem Merge Algorithmus gelangt Git mit weitaus weniger expliziten Metadaten schlicht einfacher und damit schneller ans Ziel.

Git ist im Merging nicht nur schneller als Subversion. Das „Evil Twin" Problem illustriert, dass Git auch inhaltlich bessere Merges liefern kann: Wenn zum Beispiel mit Subversions internen Operationen eine gegebene Datei gelöscht und dann wieder hinzugefügt wird (*svn delete* gefolgt von *svn add*), dann ist somit für Subversion eine neue Datei-Identität entstanden, auch wenn die „neue" Datei denselben Namen und identischen Inhalt beibehält. Naiven Subversion-Nutzern passiert das mitunter, wenn sie beispielsweise mit TortoiseSVN [goo.gl/pfpkE] eine Datei aus Versehen gelöscht haben – sie sollten dann eigentlich mit einer Revert-Operation das Delete rückgängig machen, gehen aber den Weg über ein erneutes Hinzufügen der gelöschten Datei via Add-Operation und haben damit einen unerwünschten Bruch in der Datei-Identität erzeugt. Künftig weigert sich Subversion in allen anderen Arbeitskopien und Branches, dortige Änderungen in selbige Datei zu übernehmen und unterbricht Updates und Merges mit einem Tree Conflict [goo.gl/mHOl5]. Die neu hinzugefügte Datei hat zwar denselben Pfad und denselben Inhalt, jedoch eine völlig andere interne Datei-Identität: sie ist ein „Evil Twin" (engl.: bösartiger Zwilling). Subversion befolgt also ganz explizit die Angaben der Benutzer mithilfe seiner internen Move-, Copy-, Delete- und Add-Operationen, auch wenn es zu deren Ungunsten sein mag. Git dagegen belässt die Aufzeichnung expliziter Abstammung auf der Ebene der Branches und macht sich nicht die Arbeit, die Herkunft jeder einzelnen Datei penibel zu notieren. Die Historie ist für Git nichts weiter als eine Sammlung von Schnappschüssen zu verschiedenen Zeitpunkten. Eine Umbenennung erkennt Git schlicht und einfach daran, dass eine neu entstandene Datei zu einem hohen Anteil identische Inhalte zu einer just verschwundenen Datei hat. Für Git ist es gänzlich egal, *wie* der gegenwärtige Zustand erstellt worden ist – das „Evil Twin" Problem *gibt* es demnach mit Git überhaupt nicht.

Subversion	Git
Mit Subversion gibt es immer *ein* zentrales Repository und viele Arbeitskopien des jeweils neusten Stands (siehe Abb. 1).	Jede Git Arbeitskopie ist gleichzeitig ein vollständiges Repository, das mit beliebigen anderen kommuniziert (Abb. 2 und 3).
Jeder Unterpfad kann unabhängig vom Rest des Pfadbaums im Repository bearbeitet werden.	Der gesamte Pfadbaum einer Arbeitskopie ist eine unzertrennbare Einheit.
Pfad-Identitäten werden *explizit* verfolgt.	Jede Revision gilt als Schnappschuss eines arbiträren Pfadbaums. Pfad-Identitäten werden, wo nötig, mit pfiffigen Heuristiken *implizit* abgeleitet.

Tabelle 1: Drei essenzielle Unterschiede zwischen Git und Subversion, die deren Arbeitsweise definieren.

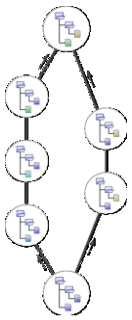

Abb. 6: Gits Merge Modell

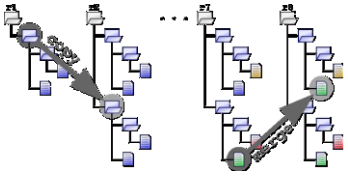

Abb. 7: Mit Subversion entsteht ein Branch durch eine einfache Kopie. Im Merge können zwar beliebige Unterpfade ausgewählt werden – besser wäre jedoch ein Merge an der Branch-Wurzel.

Perfektes Merge?

Ein Merge stellt einen sehr komplexen Vorgang dar – vor allem hinter den Kulissen. Gegenüber Subversion bietet Git so manche willkommene Vereinfachung, jedoch hat auch Gits Merge Modell kleine Mankos: beispielsweise im Cherry Picking ist Subversions Fähigkeit von Vorteil, jeden Pfad in jeder Revision gesondert aufzeichnen zu können [goo.gl/WSuzj]. Mit Cherry Picking (engl.: Kirschen Pflücken) wird ein Merge bezeichnet, der nur einen kleinen Teil meist „aus der Mitte" eines Branches in einen anderen überträgt. Beispielsweise wird beim Entwickeln eines neuen Features ein kritischer Bugfix gefunden, der so schnell wie möglich in die Hauptentwicklungslinie übertragen werden soll. Da der Rest des Features noch nicht reif ist, wird nur diejenige Revision übertragen, in der der Bugfix enthalten ist – quasi wird nur die eine reife Kirsche gepflückt. Subversion und Git unterstützen beide Cherry Picking. Jedoch, wenn bei einem solchen Merge Konflikte von Hand gelöst werden mussten, werden mit Git dann an derselben Stelle erneut Konflikte auftauchen, sobald der Feature-Branch in die Hauptentwicklungslinie integriert wird. Denn Git versucht erneut, alle Änderungen zu verschmelzen, während Subversion sich explizit jedes Cherry Picking merkt und bei der späteren Integration die bereits „abgehakten" Datei-Revisionen einfach auslassen wird. So werden einmal gelöste Konflikte nicht wiederkehren. Allerdings ist dieser Fall – nämlich ein konfliktbehaftetes Cherry Picking mit nachfolgendem Branch-Merge – schon mit der einzige, bei dem Subversions explizite Metadaten einen Vorteil bringen. Gits Merge Modell ist und bleibt überzeugender.

Die Wahl

Für viele, sogar sehr viele offene Projekte ist Gits verteilte Natur, die den sozialen Strukturen der Entwicklungsteams ganz natürlich entspricht, das überragende Argument schlechthin. Kleine Vorteile von Subversion werden dadurch ohnehin tief in den Schatten gestellt. Das Erstellen von Subversion Mirrors mit *svnsync* ist im Vergleich zu Git geradezu rudimentär. Wenn Sie Git einmal verstanden haben, gibt es zudem wahrscheinlich weniger Merge-Konflikte als mit Subversion. Doch nicht für alle ist Git das Nonplusultra – Subversion bleibt die bessere Wahl für Projektgruppen

- die unbedingt pfadbasierte Zugangsberechtigung benötigen;
- die unbedingt File Locking benötigen;
- die nicht jede Arbeitskopie mit vielen großen binären Dateien in ihrer Entwicklungshistorie belasten möchten und/oder
- die alle Arbeit garantiert an einem zentralen Punkt bündeln möchten.

Gibt es kein bestimmtes notwendiges Feature, das direkt die Wahl für Git bzw. Subversion ergibt, bleibt es beim Abwägen der Präferenzen. Beide Tools bieten ausgereifte und erstklassige Versionskontrolle; beide haben exzellente öffentliche Communities sowie Angebote zu professionellem Support; beide sind exzellent mit anderen Werkzeugen integrierbar oder längst integriert; für beide gibt es vielerlei einfach nutzbare GUI Tools, die per Mausklick funktionieren [http://goo.gl/JvGu0 und http://goo.gl/xelCl]; für beide finden sich freie und professionelle Repository-Hoster im Netz; beide skalieren hervorragend. Gleichzeitig haben sie aber sehr tiefgreifende sowie auch viele oberflächliche Unterschiede. Generell sollten Sie Ihre Arbeitsprozesse vor der endgültigen Entscheidung sorgfältig planen und auch testen, um sich Teufels Merge Küche zu ersparen. Aber schlussendlich kann durchaus die beste Wahl sein, sich danach zu richten, welches der Tools in Ihren Entwicklungsteams beliebter ist und somit ganz naturgemäß effizienter genutzt werden wird – und sei es eines der vielen anderen frei verfügbaren Versionskontrollwerkzeuge [goo.gl/3uW2s].

Neels Hofmeyr, Dipl. Ing. in Technischer Informatik, ist seit 2005 bei der *elego Software Solutions GmbH* (elego.de) als SCM Consultant, Softwareentwickler und gesponsorter Subversion Committer tätig. Er hat dank *elego* jahrelange Erfahrung in der Migration zwischen und im Vergleichen von Git, Subversion und anderen VCS, sowie in maßgeschneiderter Integration mit Tools wie Atlassian® JIRA®, CollabNet Teamforge® und OpenERP™.