Paper

CM Paper

Git vs. Subversion – Welches Tool passt?

Zentral vs. Verteilt

Merging-Strategie

Und mehr

Einleitung

Aus dem wilden Wald der heute verfügbaren Versionskontroll-Werkzeuge [1] ragen zwei Rivalen merklich hervor: Subversion und Git. Beide sind frei verfügbar, beide erlauben großen Teams von Software-Entwicklern gleichzeitig und auf mehreren Entwicklungslinien zu arbeiten; sie beherrschen Branching und Merging, können Konflikte lösen, und sie zeichnen die historische Entwicklung des Quellcodes Version für Version genau auf. Sie scheinen dasselbe zu bieten, und so ist die Wahl zwischen Git und Subversion oft nicht einfach.

Dabei sind Git und Subversion alles andere als ähnlich! Ein Blick unter die Haube zeigt zwei völlig verschiedenartige Strukturen der Versionskontrolle, deren Eigenschaften bis ins Design der Entwicklungsprozesse durchschlagen.

User Interface

Beide Tools bieten in erster Linie ein Kommandozeilen-Interface. Während der Einstieg in die Subversion Kommandozeile relativ leicht fällt, erfordern komplexere Merges ein gutes Verständnis der internen Strukturen. Bei Git ist es eher umgekehrt: schon im Einstieg ist Wissen über die internen Vorgänge nötig, dagegen vermeidet Git durch seinen Strukturaufbau übermässige Komplexität (wiederum insbesondere beim Merge) und erscheint in anspruchsvollen Fällen einfacher.

Subversion bietet mit dem frei verfügbaren *Subversion Book* [2] und der Kommandozeilen-Hilfe (*svn help*) zweifellos die bessere Dokumentation. Beim Einstieg in Git sind eher die zahlreich verfügbaren Tutorials hilfreich [3].

Doch so unterschiedlich die Kommandozeilen sind, so ähnlich sind sich die für beide Tools verfügbaren Frontends (GUIs, Web Anwendungen oder Plugins, von Drittanbietern [4] [5]). Das User Interface sollte für die Wahl des Tools *nicht* ausschlaggebend sein.

Die wahren Unterschiede zwischen Git und Subversion, die sich nicht durch einfache *Wrapper* oder genau definierte Prozesse kaschieren lassen, liegen in den grundlegend definierenden Philosophien der beiden Tools. Diese könnten kaum verschiedener sein.

Zentral vs. Verteilt

Subversion folgt einem zentralisierten Modell. Alle Benutzer schicken ihre Änderungen an dasselbe zentrale Repository (Bild 1).

- Benutzer können wahlweise nur kleine Teile vom Repository herunterladen und bearbeiten (*sparse checkout*).
- Im selben Zuge kann die Zugangsberechtigung zum Repository pfadgenau konfiguriert werden.
- Das zentrale Repository erlaubt es, bestimmte Pfade vorübergehend für einen Benutzer zu reservieren (globales *locking*, sinnvoll für binäre Dateien, z.B. Bilder).
- Alle Branches und Tags befinden sich im zentralen Repository.
- Lokale Entwicklungen werden erst dann versioniert, wenn sie ins zentrale Repository übermittelt werden.

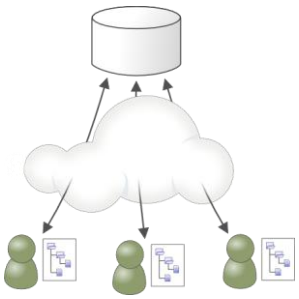

Bild 1: Subversions zentralisiertes Modell

Git dagegen implementiert ein verteiltes Modell (Bild 2), in dem jede Working Copy ein kompletter Klon des Repository ist. Man kann sich natürlich auf ein bestimmtes Repository als zentralen Sammelpunkt einigen (Bild 3). Trotzdem bleibt Git grundlegend verteilter Natur:

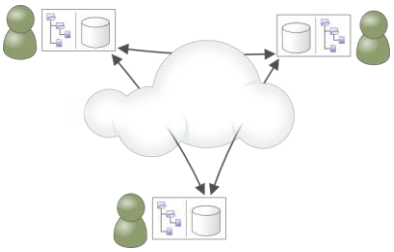

Bild 2: Gits verteiltes Modell

- Das Erstellen vollständiger Repository-Klone ist ein alltäglicher und zwingender Vorgang.
- Eine Git Working Copy ist unabhängig von der Netzanbindung voll funktionstüchtig, und Benutzer, die schlechte Anbindung zum Sammelpunkt haben, können sich problemlos untereinander abgleichen.
- Fällt ein Repository aus, kann trotzdem nahtlos weitergearbeitet werden.
- Vollständig versioniertes Arbeiten ist auch lokal möglich. Branches können sowohl nur im lokalen Repository/Working Copy existieren als auch mit verwandten Repositories abgeglichen werden.
- Branches werden explizit gehandhabt und sind leicht überschaubar. Sie können aber auch in einer privaten Working Copy "versteckt" bleiben, wenn sie nicht abgeglichen werden.
- Globales *locking*, pfadgenaue Authentifizierung und *sparse checkouts* sind im verteilten Modell inhärent gar nicht möglich, selbst wenn sich alle Benutzer auf ein bestimmtes Repository als zentralen Sammelpunkt einigen (Bild 3). Dagegen sind Git Benutzer nicht von einem funktionstüchtigen und immer erreichbaren zentralen Server abhängig.

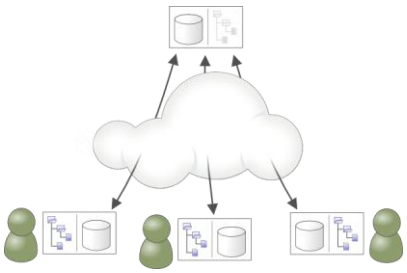

Bild 3: Git mit vereinbartem zentralen Sammelpunkt

Mit *svnsync* kann auch ein Subversion Repository geklont werden und somit verteilte Redundanz bieten, jedoch sind diese Klone lediglich *read-only* Kopien. Jeder *svn commit* benötigt zwingend Kontakt zum zentralen Haupt-Repository (daher das Konzept eines *write-through proxy* [6]).

Die erste Entscheidung ist die Wahl zwischen differenzierter Zugangsberechtigung und blockierbarer Pfade (*locking*) mit Subversion einerseits, oder Unabhängigkeit von der Netzanbindung und hoher Redundanz (im Sinne von Datensicherung) mit Git andererseits.

Merging

Ein weiterer grundlegender Unterschied zwischen Git und Subversion ist, kurz gesagt, deren Merge Verhalten. Die Historie wird in beiden Tools sehr verschieden aufgezeichnet, mit weitreichenden Auswirkungen. Subversion ist in der Lage, Merge-Operationen im Repository genauer zu registrieren, braucht aber im Durchschnitt viel länger für ein Merge als Git.

Merge-Auflösung

Git betrachtet den vom Benutzer bearbeiteten Dateibaum in einer gegebenen Revision als unzerteilbare Einheit. Die Verknüpfungen zwischen Branches und Revisionen gelten jeweils für *den gesamten Dateibaum* (Bild 4).

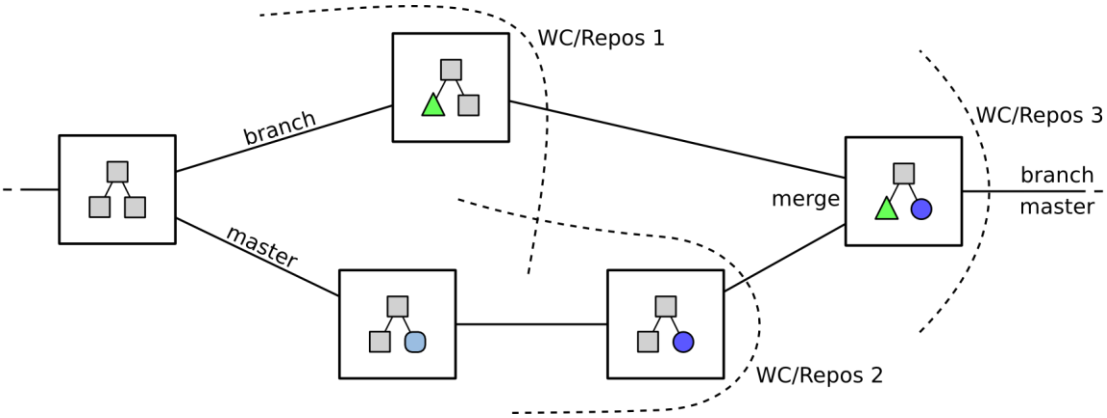

Bild 4: Branching & Merging in Git (vereinfacht) mit Beispielen möglicher Working Copies

Subversion folgt dem grundlegenden Paradigma eines intelligenten Dateisystems, in dem *einzelne Pfade* mit anderen verknüpft sein können. Eine *Cheap Copy* (Kopie durch interne Referenz) bildet die Grundlage für Subversions Branching und Merging (Bild 5). Ein Branch ist

also eine "billige" Kopie *eines Teils* des Dateibaums. Die von Benutzern bearbeiteten Working Copies und somit alle vorgenommenen Merge Operationen basieren ebenfalls auf *Teilen* des gesamten Subversion-Dateisystems. Subversion bietet so einerseits eine feinere Auflösung, muss andererseits aber auch eine höhere Komplexität erlauben und verwalten.

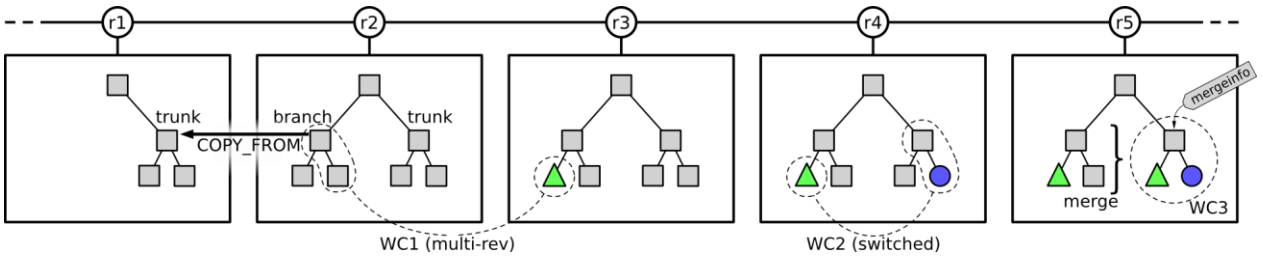

Bild 5: Branching & Merging in Subversion (vereinfacht) mit Beispielen möglicher Working Copies

Cherry Picking

Beim *Cherry Picking* möchte man nur vereinzelte Change Sets von einem Branch in einen anderen übernehmen. Diese sollten bei späteren Merges dann möglichst nicht mehr auftauchen — sie sind "schon erledigt".

Solang ein *Cherry Picking* konfliktlos verläuft, sind Git und Subversion ebenbürtig. Spätere Merges überspringen schon erledigte Änderungen automatisch. Sobald aber Konflikte gelöst werden mussten, hat Gits Modell einen Nachteil: Git merkt sich nicht explizit, welche Revisionen oder Pfade eines anderen Branches bereits eingebracht wurden. Ist nach dem Lösen eines Konflikts das Ergebnis des *Cherry Picking* nicht mehr identisch mit dem des ursprünglichen Change Sets, wird derselbe Konflikt beim nächsten Merge der betreffenden Branches unweigerlich erneut auftauchen.

Subversion dagegen merkt sich diese fragmenthaften Merges revisionsgenau (und pfadgenau) in betreffenden *svn:mergeinfo* Properties und wird nicht mehr versuchen, die schon erledigten Revisionen (und Pfade) später erneut zu mergen.

Subversions Auflösung unterscheidet aber nicht mehr zwischen Change Sets *innerhalb* einer Revision einer gegebenen Datei. Ein *Cherry Picking* z.B. nur der Hälfte der Änderungen an einer Datei-Revision verhält sich dann ähnlich wie Git auf der Branch-Ebene. Entweder, die restlichen Änderungen dieser Datei werden verworfen (mit mergeinfo), oder ein vermeintlicher Text-Konflikt wird später erneut auftauchen (ohne mergeinfo). Diese Art von *Cherry Picking* ist bei den meisten Benutzern jedoch äusserst selten.

Merge-Geschwindigkeit

Git besticht mit herausragend geringen Laufzeiten beim Merge. Benutzer beklagen sich dagegen oft über Subversions langsamen Merge, insbesondere bei sehr großen Working Copies. Auch wenn nur eine kleine Änderung einer einzigen Datei ansteht, muss Subversion semantisch jeden Pfad einzeln untersuchen.

Git ist allerdings langsamer darin, einzelne Pfade zurückzuverfolgen — Pfad-Verwandtschaften zwischen Revisionen müssen jedes Mal aufs Neue heuristisch ermittelt werden und sind u.U. nicht eindeutig (z.B. wegen großer Ähnlichkeit zweier Dateien). Im Entwicklungsalltag stört Gits Langsamkeit im Umgang mit einzelnen Pfaden jedoch weniger als Subversions Langsamkeit im Umgang mit großen Dateibäumen.

Merge-Praxis

Ist nun Subversions hohe Auflösung sinnvoll? Ironischerweise empfehlen *Best Practises* in der Benutzung von Subversion, jeden Branch und jede Revision als Einheit zu behandeln. Ein Subversion Merge sollte ideal immer an der Branch-Wurzel (wie z.B. */trunk* oder */branches/release-1.x*) erfolgen und Revisionen immer vollständig beinhalten. Fragmenthafte Merges erzeugen nämlich semantische Unterbrechungen im Pfadbaum [7], sodass einzelne Unterpfade in zukünftigen Merges eine Spezialbehandlung benötigen. Dies skaliert ausgesprochen schlecht. Man möchte also solche Unterbrechungen möglichst nicht innerhalb eines Branches erzeugen, sondern nur an der Branch-Wurzel selbst. Spezialfälle profitieren zweifellos von der Möglichkeit, Revisionen auf nachvollziehbare Weise in einzelne Pfade zu unterteilen. Für die meisten Benutzer von Subversion macht das allerdings keinen Sinn, und auch sie sollten möglichst nur in der gröberen, zu Git analogen, Auflösung arbeiten. Jede Revision sollte also möglichst nur streng zusammenhängende Änderungen beinhalten, damit sie bei späteren *Cherry Picking* Merges nicht weiter unterteilt werden muss.

Am Merge wird deutlich sichtbar, dass Git aus den Fehlern von Subversion lernen konnte, und mit einer vernünftigen Vereinfachung sein Merge-Verhalten deutlich verbessern und beschleunigen konnte. Jedoch fehlt bei Git die genaue Aufzeichnung von *Cherry Picking* Merges.

Subversion ist nun am Zuge, seinerseits von Git zu lernen, und über sein ursprüngliches Ziel, ein besseres CVS zu sein, hinauszuwachsen. Gits Merge-Vereinfachungen lassen sich durchaus als konfigurierbare Begrenzungen innerhalb des gegenwärtigen Subversion Modells umsetzen. Schon heute ist das per Hook-Skript durchsetzbar und oft empfehlenswert [8]. Allerdings wäre eine interne Lösung wünschenswert, die ebenfalls hilfreiche Merge Heuristiken und Optimierungen ermöglichen und grafische Darstellung vereinfachen könnte. Subversions Weiterentwicklung ist hier ohne großen Bruch möglich.

Die Wahl

Wessen Entwicklungsprozesse mit Git abbildbar sind, der findet in Git ein ausgereiftes Tool zur Versionskontrolle, das zwar von Benutzern hinreichendes Verständnis der internen Strukturen fordert, aber robust, relativ einfach und durch seine "implizite" Arbeitsweise oft vielfach schneller arbeitet als Subversion.

Umgekehrt bietet Subversion einige Schlüssel-Features, denen Git aufgrund seiner grundlegenden Struktur nicht gerecht werden kann. Viele Anwender schätzen gerade die zentralisierte, "explizite" und pfadgenau aufgelöste Struktur von Subversion.

Kombinationen beider Tools, wie z.B. *git-svn* [9], sind mit Vorsicht zu genießen, da man mit ihnen gewissermaßen die Schwächen beider Tools kombiniert. Ein mit git-svn vorgenommenes Merge kann kein *mergeinfo* im Subversion Repository verzeichnen, und Subversion ist nicht in der Lage, Gits direkte Branch-Verknüpfungen wiederzugeben. In jede Richtung gehen so Metadaten verloren, die dann manuell kompensiert werden müssen.

Entwerfen Sie also, unabhängig voneinander, Ihre Entwicklungsprozesse für beide Tools, und nutzen Sie die jeweiligen Stärken aus. Scheuen Sie nicht davor zurück, vereinfachende Sichtweisen auf Ihre Entwicklungsprozesse auszuprobieren. Jeder Vorgang wird in der Entwicklung täglich wiederholt werden, ausgiebige Planung lohnt also [10]. Versichern Sie sich mit detaillierten Probeläufen, die richtige Wahl getroffen zu haben.

Darüberhinaus ist es natürlich wichtig, sich ernsthaft mit den oft stark polarisierten Meinungen der Entwicklerteams auseinanderzusetzen und diese, falls nötig, gut verständlich und objektiv widerlegen zu können.

Referenzen

[1] http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

[2] http://svnbook.red-bean.com

[3] z.B. http://www.gitready.com

[4] https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools

[5] http://svn.apache.org/viewvc/subversion/trunk/www/links.html?view=co&pathrev=900404

[6] http://subversion.apache.org/docs/release-notes/1.5.html#webdav-proxy

[7] http://www.collab.net/community/subversion/articles/merge-info.html

[8] Ein *pre-commit* Hook kann erzwingen, dass *svn:mergeinfo* nur an bestimmten Pfaden gesetzt wird.

[9] http://www.kernel.org/pub/software/scm/git/docs/git-svn.html

[10] http://www.elegosoft.com/files/Downloads/Publications/Das-ABC-des-Branching-und-Merging.pdf

Über den Autor

Neels Hofmeyr studiert Technische Informatik an der Technischen Universität Berlin. Einen Teil seines Studiums absolvierte er an der University of Stellenbosch, Südafrika. Er ist als Mitarbeiter des Deutschen Subversion-Sponsors *elego Software Solutions GmbH* seit Mitte 2008 aktiv an der Entwicklung von Subversion beteiligt. Die *elego Software Solutions GmbH* unterstützt aktuell mit vier Committern die Weiterentwicklung von Subversion. *elego* bietet in Kooperation mit *CollabNet*, dem Hauptsponsor der Subversion-Entwicklung, Subversion-Support für den europäischen Raum. D.h. Unternehmen können professionelle Unterstützung bei Installation, Migration und Integration, sowie der Administration und der Anwendung von Subversion in Anspruch nehmen.