Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Scientific Programming

Vergleich von Git und SVN

Seminararbeit

vorgelegt von

Volker Mauel
Matrikelnummer 855252

Jülich im Dezember 2013

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

_____

_____

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: _____

Aachen, den _____

Unterschrift der Studentin / des Studenten

Zusammenfassung

Aus der professionellen Softwareentwicklung sind Versionsverwaltungen heutzutage nicht mehr wegzudenken, da sie die Zusammenarbeit in Teams erheblich erleichtern. Zudem dienen sie als Anschlussstelle für weitere Dienste, wie beispielsweise Hudson/Jenkins für Continuous Integration[1]. Jedoch ist ein System nur dann hilfreich, wenn es dem Benutzer die eigene Arbeit erleichtert und ihn dabei nicht behindert. Hinzu kommt, dass das eingesetzte System effizient mit den vorhanden Ressourcen umgehen muss, um wirtschaftlich zu sein.

Diese Seminararbeit vergleicht zwei populäre Versionsverwaltungssysteme. Zum einen Subversion, welches jahrelang als Quasi-Standard galt und stetig weiterentwickelt wurde. Zum anderen das verhältnismäßig junge Git, welches unter anderem von Linux-Entwickler Linus Torvalds entwickelt wurde, der dabei gezielt versuchte, die Probleme der anderen Systeme zu vermeiden. Dabei werden die wichtigsten Interna beider Systeme erläutert und typische Workflows innerhalb der beiden Systeme dargestellt und verglichen. Abschließend wird ein Fazit gezogen welches der beiden Systeme für welchen Anwendungsfall empfehlenswert ist.

[1] Continuous-Integration-Systeme sind für das automatisierte Kompilieren und Testen von Software zuständig. Dabei wird die Versionsverwaltung in regelmäßigen Abständen auf eine neue Version überprüft und diese, gemäß den Vorgaben, erstellt und/oder gestestet.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CAS Content-Addressed Storage
SVN Subversion
CVS Concurrent Versions System

Anmerkung

Die Testumgebung bildet ein aktuelles Debian 7.2, welches die Pakete aus den offiziellen Stable-Debian-Repositories (de.debian.org) bezieht. Aus diesem Grund wurden nicht die neusten Versionen von Git und Subversion eingesetzt. Features, die erst in neueren Versionen eingeführt wurden, werden bewusst nicht als Teil dieser Seminararbeit behandelt.

```
root@taurus   # svn –version
svn, version 1.6.17 (r1128011) compiled Oct 3 2013, 02:29:26
```

```
root@taurus   # git –version
git version 1.7.10.4
```

Kapitel 1

Einleitung

In dieser Seminararbeit geht es darum einen genaueren Blick auf die beiden Versionsverwaltungen Git und Subversion (SVN) zu werfen. Zunächst wird in Kapitel 2 erklärt, worum es sich bei einem Versionsverwaltungssystem handelt. Außerdem wird kurz die Geschichte der beiden Systeme vorgestellt.

Im dritten Kapitel wird Subversion näher erläutert. Dabei geht es zunächst um die Architektur, welche sich auf ein zentrales Repository stützt, die interne Datenhaltung im Repository und der Arbeitskopie, sowie einige grundlegende Eigenschaften, wie beispielsweise die Versionsnummerierung, die Verwendung von Tags und Branches und die Atomarität von Aktionen. Zusätzlich werden in diesem Kapitel einige Nachteile an der Versionsverwaltung mit Subversion genannt.

Ähnlich dazu wird im vierten Kapitel die Versionsverwaltung mit Git erklärt, jedoch zusätzlich, die durch die Architektur von Git bedingten neuen möglichen Arbeitsabläufe.

In Kapitel 5 findet ein direkter Vergleich zwischen Git und Subversion statt, in welchem man zum einen den Speicherverbrauch beider Systeme am Beispiel eines leeren Repositories und des Mozilla-Repositories näher betrachtet. Zum anderen erhält man einen kurzen Überblick über die wichtigsten Aktionen, wie Branching, Tagging, Merging und die Verwendung des Logs in beiden Systemen. In Kapitel 5.4 werden außerdem ausgewählte Statistiken zur Popularität beider Systeme gezeigt, um die Verteilung der Nutzungsverhältnisse zu erkennen.

Im letzten Kapitel wird ein Fazit gezogen und die persönliche Meinung des Autors erläutert.

Kapitel 2

Versionsverwaltungen

Im Folgenden wird erklärt, worum es sich bei einer Versionsverwaltung handelt, sowie deren generelle Funktionalität erläutert.

2.1 Definition Versionsverwaltung

Eine Versionsverwaltung ist ein System mit dem Änderungen an Dateien auf Befehl der Benutzer aufgezeichnet werden und bei Bedarf jede der vorherigen Dateiversionen wiederhergestellt werden kann. Der zentrale Ort an dem die Änderungen der Dateien aufgezeichnet werden nennt sich Repository. Zusätzlich zur Benutzung eines Repositories von lediglich einem Benutzer, bieten die meisten Versionsverwaltungen die Möglichkeit, dass mehrere Benutzer gleichzeitig an Dateien arbeiten können, ohne dass dabei eine Änderung eines Nutzers verloren geht. Die Kombination dieser Eigenschaften bietet die Grundlage der Softwareentwicklung im Team. Dies bedeutet jedoch keineswegs, dass Versionsverwaltungen lediglich für die Softwareentwicklung interessant sind. Ebenso kann man die Systeme dazu nutzen, um Änderungen an Bildern, Audiodateien oder sogar Binärdateien nachzuverfolgen und alte Versionen wiederherzustellen.

Gerade große Projekte in der Bild- und Tonbearbeitung brauchen viel Platz auf der Festplatte und im Arbeitsspeicher, da sie selbst die Option bieten alle Änderungen nicht-destruktiv[1] durchzuführen und somit eine eigene Versionsverwaltung enthalten. Die Anforderungen an die Rechner könnten jedoch immens gesenkt werden, wenn man lediglich die aktuell nötigen Daten bereithält, anstatt des gesamten Werdegangs einer Datei. Dies kann man erreichen, indem man die programmeigene Versionsverwaltung deaktiviert und stattdessen auf ein System wie Git oder Subversion setzt.

2.2 Geschichte der Versionsverwaltungssysteme

Da eine detaillierte Erklärung zu den einzelnen Systemen den Umfang dieser Arbeit übersteigen würde, beschäftigt sich diese Seminararbeit im Folgenden lediglich mit den aktuell populärsten Systemen zur Versionsverwaltung Subversion und Git, sowie deren direkten Vorgängern.

[1] Nicht-destruktiv bedeutet, dass die Änderungen reversibel sind, da das Ausgangsmaterial nicht zerstört wird.

2.2.1 Geschichte von Subversion

Die Geschichte von SVN beginnt im Jahr 2000 bei der Firma CollabNet. Dort wurde es als ablösendes System für Concurrent Versions System (CVS) entwickelt.[Wika] Im Gegensatz zu CVS bot SVN einige entscheidende Vorteile, allen voran die direkte Nutzung auf dem Dateisystem (FSFS genannt) ohne zusätzliche Datenbank, sowie der Organisation von Dateiversionen als Projektversion. In CVS war es zwar so, dass einzelne Änderungen an Dateien aufgezeichnet wurden, jedoch vollkommen unabhängig vom Projektkontext. SVN hingegen fasst alle Dateien zu einer Version, in diesem Zusammenhang Revision genannt, zusammen. Somit kann man beispielsweise via `svn co -r 5 REPOSITORY` auf die Version des Projektes zum Zeitpunkt der Revision 5 zugreifen und erhält eine Arbeitskopie[2] von diesem Zustand.

2.2.2 Geschichte von Git

Die Entwicklung Gits begann Linus Torvalds im Jahr 2005, nachdem durch eine Lizenzänderung das bis dahin genutzte System (BitKeeper) den Linux-Entwicklern nicht mehr kostenlos zur Verfügung stand.[Wikb] Dabei wurde besonderer Wert auf die Unterstützung verteilter Versionierung (siehe Kapitel 4), die Sicherheit gegen Verfälschung und eine hohe Effizenz gelegt. Schon wenige Tage nach Beginn der Arbeit an Git stellte Linus Torvalds eine erste Version vor, welche die von ihm und anderen Kernelentwicklern gewünschten Anforderungen erfüllte.

[2] Bei SVN erhält man immer eine Kopie der Dateien des Repositories. In dieser Kopie erfolgte Änderungen an den Dateien werden von SVN aufgezeichnet.

Kapitel 3

Versionsverwaltung mit Subversion

Nachfolgend wird die Versionsverwaltung mit Subversion näher erläutert. Dabei geht es im Wesentlichen darum, die Verwendung Subversions darzustellen und einige Interna, wie die Datenhaltung im Repository, zu erläutern.

3.1 Architektur von Subversion

SVN basiert im wesentlichen auf einer Client-Library, welche mittels Kommandozeilen- oder GUI-Applikationen verwendet werden kann. Diese Client-Library bildet einerseits über das Client-Interface eine feste Schnittstelle für den Zugriff auf ein Subversion-Repository, ist jedoch andererseits eine Schnittstelle zur Library, welche für die Verwaltung der lokalen Arbeitskopie zuständig ist. Das Repository ist mit einem definierten Interface über verschiedene Wege erreichbar, wie beispielsweise das lokale Dateisystem, eine Netzwerkfreigabe, das SVN-eigene Protokoll oder das DAV-Protokoll in Verbindung mit einem Webserver, welcher dieses unterstützt. Das Subversion-Repository bietet zwei verschiedene Möglichkeiten die Änderungen zu protokollieren. Dies erfolgt entweder, wie zuvor schon bei CVS, über eine Berkeley-DB oder direkt im Dateisystem. Alle Änderungen an den Dateien der lokalen Arbeitskopie werden bei einem Commit[1] zusammen mit einer Notiz in das Repository übertragen und dort als neue Revision hinterlegt. Die nachfolgende Grafik veranschaulicht die Architektur.

[1] Ein Commit (engl. to commit sth. to so.: Jemandem etwas anvertrauen) ist die Übertragung der Änderungen in das Repository, benannt nach dem Befehl `svn commit`.

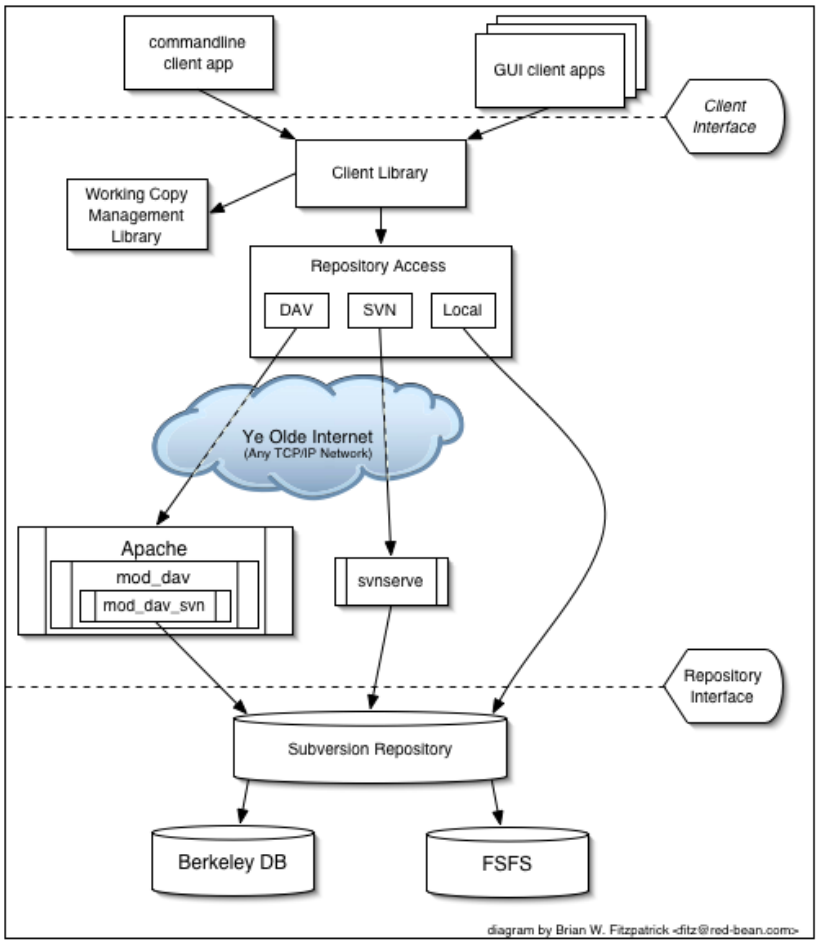


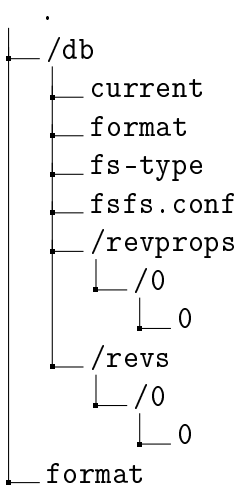
Abbildung 3.1: Die Architektur von SVN im Überblick [CSFP11, S. XV].

3.1.1 Zentrales Repository

In SVN gibt es ein zentrales Repository (erstellt mittels `svnadmin create NAME`), in welchem alle Änderungen protokolliert werden. Von dort aus kann man eine lokale Arbeitskopie erstellen, um selbst Änderungen an den Dateien vorzunehmen und diese abschließend mittels `svn commit` zurück ins Repository zu übertragen. Dieser Aufbau birgt jedoch den Nachteil, dass es gleich mehrere Punkte gibt, an denen ein Commit fehlschlagen kann. Beispielsweise wenn das Netzwerk nicht verfügbar, oder Server offline ist. In diesem Fall ist man gezwungen mit weiteren, möglicherweise erst zu testenden, Änderungen so lange zu warten, bis das Repository wieder erreichbar ist.

3.1.2 Datenhaltung im Repository

Beim Erstellen eines Repositories wird von Subversion die Struktur zur internen Haltung der Daten angelegt. Die wichtigsten Inhalte des Ordners finden sich an folgenden Orten:



Für die Datenhaltung ist dabei lediglich der Unterordner /db sowie die Datei format wichtig. In der Datei format wird dabei das Format der im Ordner /db hinterlegten Dateien beschrieben. Sie enthält eine Ganzzahl von 1 bis 6, wobei jeder Zahl eine Versionsempfehlung zugeordnet ist. Dies ermöglicht es das Repository von neueren Systemen aus zu verwenden, ohne es bei einem Commit unbrauchbar zu machen. Die folgende Tabelle zeigt die empfohlenen Versionen für die Formate:

Format 1	understood by Subversion 1.1+
Format 2	understood by Subversion 1.4+
Format 3	understood by Subversion 1.5+
Format 4	understood by Subversion 1.6+
Format 5	understood by Subversion 1.7+

Tabelle 3.1: Versionszuordnung des Formates zur SVN-Version.[Svn]

Auf dem Testsystem, welches im Rahmen dieser Seminararbeit verwendet wurde, erzeugt Subversion ein Repository mit Format 5. Dies besagt im Wesentlichen, dass dieses Repository ein Sharded-Layout[2] verwendet. Dies bedeutet, dass unterhalb des /db/revs-Ordners alle X(standardmäßig 1000) Revisionen ein neuer Ordner (Shard[2]) erzeugt wird, in welchem die nächsten X Änderungen abgelegt werden. Subversion verwendet im FSFS-Modus sogenannte Forward-Diffs. Dabei werden Basis-Dateien erstellt. Für die jeweiligen Änderungen dieser Dateien wird lediglich aufgezeichnet, in welcher Zeile welche Änderung gemacht wurde. Dies geschieht um Festplattenplatz einzusparen, da sonst jedes mal der vollständige Inhalt gespeichert werden müsste. Gleichzeitig werden auch beim

[2]Shard nach dem engl. Scherbe/Teilstück.

Erstellen eines neuen Shards (zum Beispiel bei Revision 1000) für diese Revision neue Basisdateien mit den bisherigen Änderungen erstellt.

Dies hat den Vorteil, dass Subversion bei einem Checkout nicht erst alle Änderungen auf die Basisdateien anwenden muss. Falls sich das Repository beispielsweise bei Revision 50000 befindet und man Revision 40000 auschecken möchte, müssten nicht erst alle 40000 Änderungen erneut auf die Basisdateien angewendet werden. Im schlechtesten Fall, wenn beispielsweise die aktuelle Revision bei 50000 liegt und Revision 49999 ausgecheckt werden soll, würden somit bei einem sharded-Repository gerade einmal 999 Änderungen angewendet werden, was verglichen mit 49999 Änderungen beim Auschecken eines nicht-sharded Repository wenig ist.

In der Datei /db/current findet sich ein Verweis auf die aktuelle Revision des Repositories, um ein schnelles Auschecken der neusten Revision zu ermöglichen.

Der Inhalt der Dateien unterhalb von /db/revs ist dabei wie folgt strukturiert: Die erste Zeile beginnt entweder mit "PLAIN\n" oder mit "DELTA\n" gefolgt von den Änderungen im SVN-Diff-Format. Im Feld id: erhält jede Datei eine einzigartige ID, wie beispielsweise 0-1.0.r1/293. Hierbei ist die Nummer vor dem Slash die Revisionsnummer der Datei und die Nummer danach die Transaktions-ID (siehe Abschnitt Atomarität von Aktionen). Falls eine Datei, die eingecheckt wurde, bereits vorher in der Versionsverwaltung geführt wurde, enthält die /db/revs-Datei zusätzlich das pred:-Feld mit einem Verweis auf die Revision im gleichen Format wie id:. Das type:-Feld sagt aus, ob es sich um eine Datei (file) oder einen Ordner (dir) handelt. Das count:-Feld enthält eine fortlaufende Nummer für die Anzahl der Änderungen an einer Datei. Das text:-Feld beinhaltet Angaben zur Revision, Offset, Länge, Größe und Hash einer Datei und der Änderung.

3.1.3 Datenhaltung in der Arbeitskopie

Wie bereits erwähnt bietet SVN die Möglichkeit alle Änderungen im Dateisystem selbst zu speichern. Dies geschieht mittels eines Ordners namens .svn im Hauptordner der lokalen Arbeitskopie. Beim Hinzufügen einer neuen Datei via `svn add DATEINAME` (um SVN mitzuteilen, Änderungen der Datei ins Repository zu übertragen) und der Übertragung ins Repository (mittels `svn commit`) wird im Ordner ./.svn/text-base/ eine Kopie der neuen Datei unter dem Namen DATEINAME-base hinterlegt. Mit Hilfe dieser Referenzkopie können dann bei der Änderung einer Datei diese nachverfolgt werden und statt der vollständigen Datei übertragen und gespeichert werden.

Eine Übersicht der lokal getätigten Änderungen kann man via `svn diff` ausgeben lassen. Um sich die gesamte Versionshistorie anzusehen, kann man diese mittels `svn log` beim Server abfragen und ausgeben lassen. Optional besteht die Möglichkeit, die Historie auf lediglich eine einzelne Datei zu beschränken, indem man zusätzlich noch den Dateinamen angibt (`svn log DATEINAME`).

3.1.4 Versionsnummerierung

Revisionen in Subversion werden durchlaufend nummeriert. Hierdurch ist es einfach möglich eine Revision eindeutig zu identifizieren und diese mittels `svn co PFAD -r NUMMER` aus dem Repository auszuchecken oder via `svn update -r NUMMER` eine bestehende Arbeitskopie auf den Stand des Repositories zu einer bestimmten Version umzustellen.

3.1.5 Tags & Branches

Tags (also benannte Revisionen) und Branches (alternative Entwicklungszweige) sind in Subversion nur begrenzt möglich, da diese im Grunde genommen nur Kopien einer bestimmten Revision sind. Gängige Praxis ist es deshalb in einem leeren Repository zunächst die drei Unterordner trunk, tags und branches anzulegen, wobei der Ordner trunk dem Hauptzweig der Entwicklung entspricht. Um einen Tag zu erstellen verwendet man `svn copy PFAD/trunk PFAD/tags/TAGNAME -m "Tagmessage"`. Dass es sich bei diesem Tag lediglich um eine Kopie handelt, erkennt man an dem copy-Kommando. Ebenso verhält es sich, unter Änderung des Ordners, mit dem Erstellen von Branches.

3.1.6 Atomarität von Aktionen

Alle Aktionen in Subversion sind atomar, was bedeutet, dass sie entweder vollständig oder vollständig nicht funktionieren. Als Beispiel wäre an einen Commit zu einen Remote-Server zu denken, wobei während dem Commit plötzlich die Verbindung zu dem Server unterbrochen wird. Wären die Aktionen nicht atomar, könnte es zu inkonsistenten Zuständen kommen, sodass beispielsweise nur einige der im Commit zu verändernden Dateien wirklich übertragen wurden und die neue Revisionsnummer tragen, andere Dateien jedoch noch auf dem alten Stand sind. Dies hätte weitreichende Folgen für die Entwicklung im Team, weshalb Subversion dafür Sorge trägt, dass die Änderungen atomar sind.

Ein weiteres Beispiel wäre das gleichzeitige Committen zweier Personen. Hierbei regelt Subversion den Zugriff so, dass die Änderungen sequenziell abgearbeitet werden, also zunächst einer der Commits bearbeitet wird und danach der andere. Falls hierbei die gleiche Datei von beiden Nutzern bearbeitet wurde erhält der Nutzer, dessen Commit hinten angestellt wurde, einen Fehler und die Möglichkeit den Konflikt der Dateien manuell zu beheben.

3.2 Nachteile

Nachfolgend werden einige Nachteile genannt, die dem Nutzer spätestens bei längerer Verwendung von Subversion auffallen.

3.2.1 Tags & Branches

Da Tags und Branches lediglich Kopien von vorhandenen Dateien sind und diese nur in einem besonderen Ordner abgelegt werden, ist es schnell möglich, dass man im falschen Ordner Dateien verändert. Durch separate Befehle fürs Tagging und Branching würde die Gefahr der versehentlichen Änderung der falschen Dateien minimiert.

3.2.2 Hoher Speicherverbrauch

Der Speicherverbrauch von Subversion ist verhältnismäßig hoch, da die via Forward-Diffs archivierten Änderungen in Klartextdateien gespeichert werden. Eine Möglichkeit zur Verbesserung wäre diese Dateien zusätzlich zu komprimieren.

3.2.3 Netzwerkanbindung erforderlich

Da Subversion nahezu alle Operationen auf dem (Remote-)Repository durchführt, ist im typischen Anwendungsfall Subversions, also der Entwicklung einer Software in einem zentralen Repository mit mehreren Entwicklern, ständig eine Verbindung zu diesem Repository erforderlich.

Kapitel 4

Versionsverwaltung mit Git

Da Git im Gegensatz zu Subversion eine verteilte Versionsverwaltung ist, gibt es einige grundlegende Unterschiede. Der größte Unterschied ist, dass anstatt einer Arbeitskopie wie bei Subversion, ein lokaler Klon eines Remote-Repositories, oder nur ein lokales Repository existiert. Nachfolgend werden zur weiteren Erklärung Parallelen zu Subversion gezogen, aber auch die Unterschiede und neuen Möglichkeiten aufgezeigt.

4.1 Die Architektur von Git

Zur einfacheren Erklärung gehe man zunächst von einem Remote-Repository aus. Damit ein Entwickler Änderungen durchführen kann, benötigt er zunächst eine Kopie. Wie bereits erwähnt, handelt es sich dabei nicht um eine Arbeitskopie (also eine bestimmte Revision des Remote-Repositories), sondern um eine vollständige Kopie des Repositories. Dieses Kopieren oder Klonen geschieht mit dem Befehl `git clone REPOSITORY`. Nach erfolgreichem Abschluss der Aktion existiert ein Ordner mit dem Namen des Repositories im aktuellen Verzeichnis. In diesem kann, wie bei Subversion, gearbeitet werden. Anders als in Subversion hat man in Git jedoch die Möglichkeit alle Änderungen zunächst lokal durchzuführen und im eigenen Repository aufzuzeichnen und erst abschließend alle Änderungen gemeinsam in das Remote-Repository zu übertragen. Um diese Trennung deutlich zu vollziehen, verfügt Git über zusätzliche Kommandos. `git commit -a -m "MSG"` ist dafür zuständig die Änderungen an den Dateien in das lokale Repository zu übertragen. Diese werden den anderen Entwicklern zugänglich gemacht, indem man `git push` verwendet, womit die Änderungen ähnlich einem `svn commit` in das Remote-Repository übertragen werden. Möchte ein anderer Entwickler besagte Änderungen in sein Repository übertragen, muss er ein `git pull` (analog zu `svn update`) ausführen.

4.1.1 Workflows in Git

Da der Bedarf eines einzelnen Remote-Repositories entfällt, weil jedes lokale Repository auch gleichzeitig als Quelle/Ziel für `clone`-, `pull` und `push`-Kommandos dienen kann, sind in Git völlig neue Workflows (Arbeitsabläufe) möglich, wie sie vermehrt in großen Softwareprojekten zu finden sind. Das bekannteste Beispiel hierfür ist der Linux-Kernel, wo der sogenannte „Dictator and Lieutenants Workflow“ eingesetzt wird. [Cha09, S. 99]

Der Dictator and Lieutenants Workflow

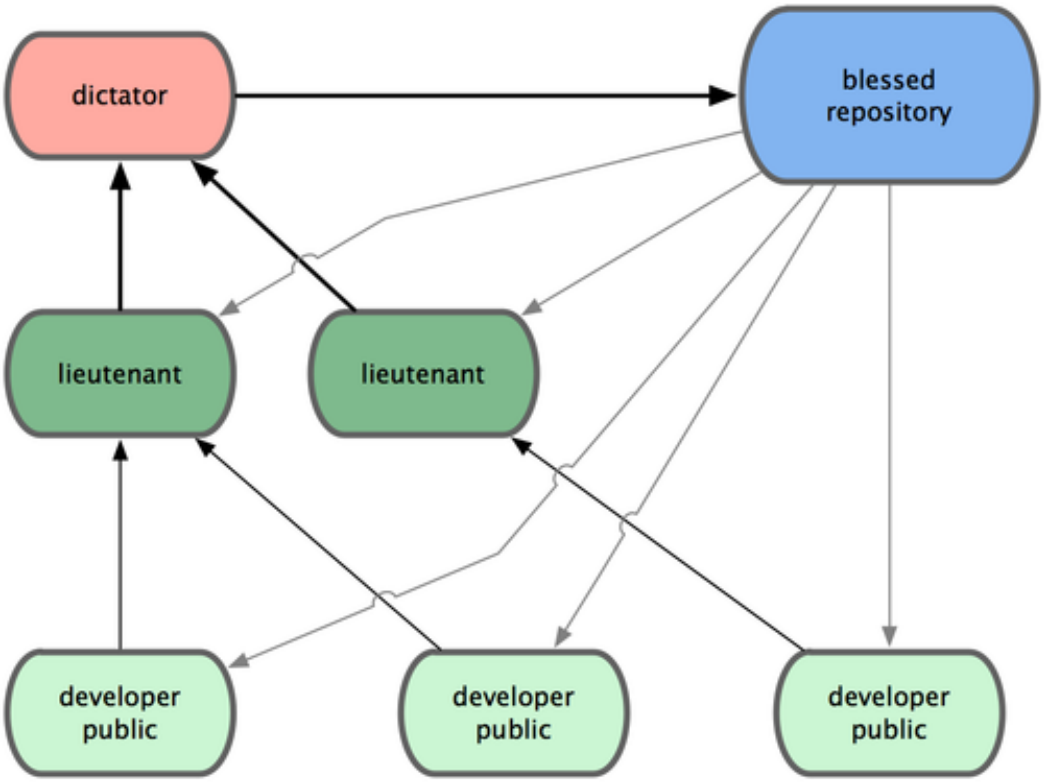


Abbildung 4.1: Dictator and Lieutenants Workflow [Cha09, S. 99].

Abbildung 4.1 zeigt das Prinzip des Workflows. Hierzu ist es wichtig zu wissen, dass die dicken schwarzen Pfeile die Push-Richtungen darstellen und die dünnen grauen Pfeile die Pull-Richtungen. In diesem Workflow gibt es mehrere Entwickler, die den Lieutenants unterstellt sind, welche wiederrum dem Dictator unterstellt sind.

Zunächst pullt jeder Entwickler die aktuellste Version aus dem blessed repository, in welches lediglich der Dictator pushen darf. Die Änderungen der Entwickler landen im Repository des jeweiligen Lieutenants. Dies hat zur Folge, dass bereits dort, also so früh wie möglich, die unterschiedlichen Änderungen der verschiedenen Entwickler zusammengeführt und Konflikte gelöst werden. Im nächsten Schritt sind die Lieutenants dafür verantwortlich die gepushten Änderungen der Entwickler in das Repository des Dictators zu überführen und die dabei auftretenden Konflikte zu lösen. Der Dictator ist dann für die Endkontrolle zuständig, bevor er die Änderungen freigibt und ins blessed repository überträgt.

Man erkennt deutlich, dass dieser Workflow für große Projekte gedacht ist, da hier darauf geachtet wird Änderungen sobald wie möglich zusammenzuführen. Im Kontext des Linux-Kernels wäre sonst beispielsweise Linus Torvalds als Dictator dafür zuständig die Merges für Rechtschreibfehlerkorrekturen eines einfachen Modulentwicklers durch-

zuführen, was unnötig viel Zeit in Anspruch nähme. Deshalb wird diese Aufgabe an die Programmierer, welche die Änderungen durchgeführt haben, delegiert.

Zentralisierter Workflow

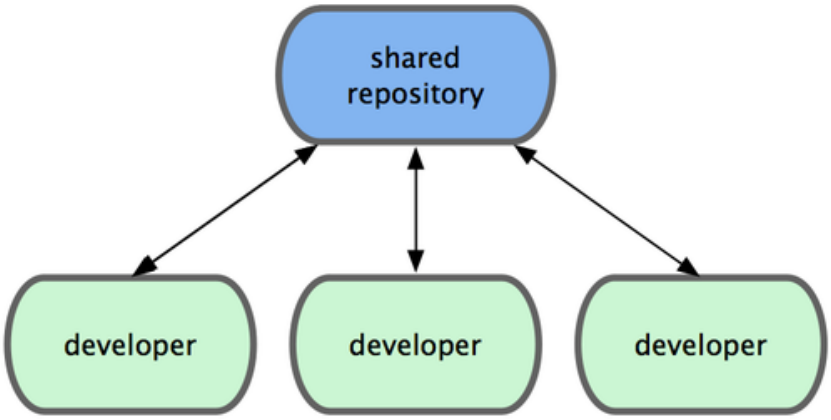


Abbildung 4.2: Zentralisierter Workflow [Cha09, S. 98]

Die Möglichkeit zur Nutzung alternativer Workflows bedeutet jedoch keineswegs, dass man dazu gezwungen wird die Möglichkeit wahrzunehmen. Der aus Subversion bekannte Workflow wird „zentralisiert“ genannt. Hierbei gibt es lediglich ein einziges Remote-Repository, in welches alle Entwickler eines Projekts ihre Änderungen übertragen und aus welchem sie diese beziehen.

Forking Workflow

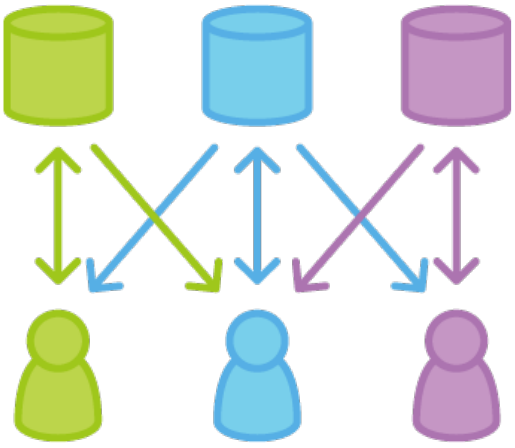


Abbildung 4.3: Forking Workflow.[Atl]

Bei dem Forking Workflow handelt es sich um den bekanntesten Workflow im Zusammenhang mit Git. Bekannt geworden ist es vor allem durch Plattformen wie GitHub und BitBucket. Dort können Nutzer den Sourcecode ihrer Projekte öffentlich hosten lassen. Interessierte Nutzer und Entwickler, die dem Projekt etwas beisteuern wollen, können dann auf dem Server einen Fork (im Grunde eine Kopie des Repositories) erstellen, in welchen sie aber ihre Änderungen pushen dürfen. Wenn ein Nutzer in seinem Fork nun einen Teilaspekt fertiggestellt hat, kann er dem Besitzer des Original-Repositories einen sog. Pull-Request zusenden, in welchem er ihm mitteilt, dass er etwas geändert hat, und es gerne im offiziellen Repository sehen würde. Der Besitzer des Originals kann dann einen Pull auf das Repository des Nutzers ausführen, um dessen Änderungen zu erhalten. Nachdem er die Konflikte in seinem lokalen Repository aufgelöst hat, kann er abschließend die neue Version in das Original-Repository bei einem der Hoster stellen und es somit jedem Nutzer zugänglich machen. Für kleine bis mittelgroße Open-Source-Projekte bietet sich diese Art des Workflows an, da somit prinzipiell jeder als Entwickler fungieren kann, ohne die Sicherheit des Projekts zu gefährden.

4.1.2 Versionsnummerierung

Anders als in Subversion sind die Revisionen in Git nicht mit fortlaufenden Nummern gekennzeichnet, sondern ergeben sich aus den getätigten Änderungen. Aus den Änderungen und einer Hash-Funktion (Hash-Funktionen sind Einweg-Funktionen) wird ein Hash-Wert erstellt, der die Revision eindeutig kennzeichnet. Hash-Werte (kurz Hashs) können zur einfacheren Verwendung abgekürzt werden, solange diese weiterhin eindeutig sind. Falls also beispielsweise lediglich ein Commit im gesamten Repository mit 3e beginnt, bezeichnet dies eine eindeutige Revision.

4.1.3 Tags & Branches

Tags und Branches sind ein integraler Bestandteil Gits, weswegen die Aktionen über eigenständige Kommandos zu erreichen sind.

Branches

Um einen neuen Branch zu erstellen genügt es `git checkout -b NAME` zu verwenden. Dies erstellt einen neuen Branch mit dem Namen „NAME“ und wechselt direkt zu diesem. Anders als in Subversion sind die Branches so fest in Git integriert, dass man hierfür keine eigene Ordnerstruktur (mit trunk, tags, branches; siehe Abschnitt 3.1.5) benötigt, sondern Git diese Daten intern selbstständig ablegt. Möchte man nun von einem Branch zurück zum Master-Branch wechseln, verwendet man `git checkout master`.

Tags

Ebenso wie Branches sind auch Tags ein Bestandteil von Git. Mittels `git tag NAME` erstellt man aus dem aktuellen Branch einen Tag. Möchte man diesen erneut anzeigen, so

wechselt man, ähnlich wie bei Branches, via `git checkout tags/NAME` zu diesem Tag. Standardmäßig werden Tags bei einem Push in ein Remote Repository nicht übertragen. Dies muss manuell über das Kommando `git push origin NAME` geschehen.

4.1.4 Interne Datenhaltung des Repositories

Git fällt in die Kategorie der Content-Addressed Storage (CAS) Systeme. Das bedeutet, dass einem bestimmten Key (im Fall von Git dem Hash eines Commits) immer genau ein Objekt zugeordnet wird. Beim Aufruf des `git add DATEI`-Befehls wird die Datei, wie beim Aufruf von `git hash-object -w DATEI`, zunächst gehasht und dann intern im Ordner .git/objects/3c/6ee4... abgelegt, wenn das Ergebnis des `hash-object`-Befehls „3c6ee4...“ ergibt. Den Inhalt der Datei kann man sich mittels `git cat-file -p 3c6ee` ausgeben lassen. Git speichert jedoch noch weitere Informationen, wie zum Beispiel den Typ eines Objekts. Diesen kann man mittels `git cat-file -t 3c6ee` in Erfahrung bringen. Bei einfachen Dateien, wie zum Beispiel auch Sourcecode, verwendet Git den Typ „blob“[1]. Zusätzlich zum bloßen Hashen und Ablegen der Datei in einem bestimmten Ordner komprimiert Git den Inhalt mit zlib, wodurch sich eine Speicherplatzersparnis ergibt. Bei einer Beispieldatei, welche 6 Textzeilen enthielt, ergibt sich so eine Platzersparnis von knapp 35% des in Git gespeicherten Objektes gegenüber der selben Datei im Klartext.

Erfolgt nun ein Commit (`git commit -a -m "TEXT"`), so erzeugt Git zwei weitere Objekte. In diesem Beispiel findet sich unter dem Hash 6f903b ein Objekt vom Typ Commit und unter dem Hash 85c90f ein Objekt vom Typ Tree. Wirft man einen Blick in die Commit-Datei (mittels `git cat-file -p HASH`), so findet man dort die Informationen des Commits, wie den Autor, den TEXT des Commits und den zugehörigen Tree. Der Inhalt der Tree-Datei besteht dabei aus einer einfachen Auflistung der zugehörigen Dateien im Format „<MODUS> <TYP> <HASH>\t<DATEINAME>“, wobei der Modus für Dateien im Normalfall 100644 ist und der Typ einem der Typen eines Git-Objektes entsprechen muss (also Tree, Blob oder Commit). Über den Hash innerhalb der Tree-Datei (in unserem Fall 3c6ee4) lässt sich eindeutig die Datei identifizieren, welche in diesem Commit enthalten sein soll.

4.1.5 Datenhaltung bei Änderungen

Wenn man nun, wie in der Softwareentwicklung üblich, eine Datei modifiziert (also beispielsweise Zeilen löscht/neue hinzufügt), erkennt Git die Änderungen ebenso wie Subversion. Markiert man die Änderung mittels `git add DATEI` zur Übertragung ins Repository, so erzeugt Git ein völlig neues Objekt (im Beispiel unter dem Hash 164f9f). Bei einem Commit zum Übertragen der Änderungen in das lokale Repository werden erneut zwei Objekte erzeugt (Tree und Commit), wobei das Commit-Objekt zusätzlich zu den bisherigen Angaben auch einen „parent“-Wert enthält, welcher eindeutig den vorherigen Commit (in diesem Fall 6f903be) identifiziert.

[1] Blob steht für Binary Large Object.

4.1.6 Garbage-Collection & Pack-Dateien

Da so im Laufe der Zeit sehr viele Objekte und damit einzelne Dateien entstehen würden, besitzt Git die Möglichkeit der Garbage-Collection. Hierbei werden Objekte, welche durch Änderungen auf andere Objekte entstanden sind, auf die jeweiligen Änderungen reduziert.[Cha09, S. 217f] Im Gegensatz zu Subversion arbeitet Git mit Backwards-Diffs, es speichert also die aktuelle Version als Basis-Datei und merkt sich die Änderungen als Diff-Datei. Dadurch ist es möglich sehr schnell auf die aktuellste Version einer Datei zuzugreifen (beispielsweise beim Clonen eines Repositories), zum Zugriff auf ältere Dateien müssen jedoch diese Diffs auf die Basis-Datei angewendet werden. Um den Speicherplatzverbrauch weiter zu reduzieren packt Git die längere Zeit nicht benötigten Objekte zusammen in eine Datei, da diese so effizienter komprimiert werden können. Dieses Komprimieren kann auch manuell mit `git gc` ausgelöst werden.

4.1.7 Vorteile dieses Systems

Zunächst erscheint es umständlich, dass man, um das gleiche Ergebnis wie bei SVN zu erreichen, zwei Befehle absetzen muss (`git commit` und `git push`), jedoch bietet dieses System einen entscheidenden Vorteil im Hinblick auf die unter Abschnitt 3.2 genannten Nachteile. Man kann unabhängig von etwaigen Netzwerk- oder Serverproblemen seine Änderungen der Dateien aufzeichnen und in einem weiteren Schritt, also dann, wenn das Problem behoben wurde, die Änderungen in das Remote-Repository übertragen und damit anderen Entwicklern zur Verfügung stellen.

Durch den Vermerk des Ausgangscommits im „parent“-Feld eines neuen Commits ist gesichert, dass auch bei verteilter Versionierung immer feststellbar ist, wo ein Commit seinen Ursprung gefunden hat. Zusätzlich ist es dadurch nicht möglich, dass jemand rückwirkend Commits verändern kann, da die Änderung zu einem anderen Hash des Commits führen würde. Außerdem ist ein schneller Zugriff auf die neuste Revision möglich, da nicht, wie etwa bei Subversion, vor Erhalt der eigentlichen Daten erneut Diffs auf Basisdateien angewendet werden müssen. Weiterhin ist es durch die vollständige Kopie des Repositories möglich, dass jedes Repository als Backup dienen kann. Zusätzlich ermöglicht die einfache Struktur der Git-Objekte diese auch ohne Git, lediglich mit Shell-Kommandos, zu verwenden (siehe [Cha09, S. 205-213]).

4.2 Mögliche Probleme

Nutzern zentralisierter Versionsverwaltungen wie Subversion und CVS fällt der Einstieg in Git zwar meist leicht, jedoch schöpfen sie das volle Potential Gits nicht aus. Gerade wenn es um die Erstellung oder Nutzung von Branches geht, benötigen die Entwickler einige Zeit um den erweiterten Funktionsumfang von Git vollständig zu erlernen und zu verwenden.

Kapitel 5

Direkter Vergleich von Git und SVN

Im folgenden Kapitel findet ein direkter Vergleich zwischen Git und Subversion statt. Dabei geht es vor allem darum, die Vorteile des jeweiligen Systems aufzuzeigen und Anwendern des einen Systems einen kurzen Einblick in das andere zu geben.

5.1 Speicherverbrauch

Dieser Abschnitt bezieht sich auf den Speicherverbrauch beider Systeme. Dabei wird zunächst ein leeres Repository für Git und Subversion erstellt, und anschließend mit Linux-Boardmitteln (im speziellen `du -h`) die Größe beider Repositories ermittelt und verglichen. Danach gibt es einen Vergleich des Speicherverbrauchs von aktiv verwendeten Repositories am Beispiel der Mozilla-Repositories.

5.1.1 Leeres Repository

Da Git mit dem Ziel entwickelt wurde, möglichst ressourcenschonend zu arbeiten, liegt es nahe zu prüfen, ob es dieser Anforderung gerecht wird. Hierzu erstellt man mit `svnadmin create foo` ein neues Subversion-Repository und ermittelt dessen Größe via `du -h foo`. Analog ist das Vorgehen bei Git anzuwenden. Dieser Vergleich wurde auf dem beschriebenen Testsystem ausgeführt, wobei folgende Werte ermittelt wurden: die Größe eines leeren Git-Repositories liegt bei 92 KByte, die Größe eines leeren Subversion-Repositories hingegen bei 144 KByte. Natürlich sind diese Werte nicht sonderlich relevant, jedoch sind sie bereits ein möglicher Indikator für die Größenverhältnisse eines befüllten Repositories.

5.1.2 Beispiel: Mozilla-Repositories

Die Mozilla-Repositories sind mit ihrer 10-jährigen Entwicklungshistorie ein repräsentatives Beispiel für die Effizenz von Git. Berichten zufolge soll das SVN-Repository fast 12 GB groß sein und aus 240.000 Dateien bestehen. Im Gegensatz dazu ist die gleiche Entwicklung in einem Git-Repository gerade einmal 420 MB groß und in lediglich 2 Dateien gespeichert. Damit benötigt Subversion fast das 30-fache des von Git belegten Speichers. [Gitb, Smaller Space Requirements]

5.2 Geschwindigkeit

Dadurch, dass bei Git nahezu alle Befehle (außer push & pull) auf dem lokalen Repository ausgeführt werden, und für diese nicht erst eine Netzwerkverbindung hergestellt werden muss, ist es Subversion in jedem Fall überlegen. Hinzu kommt, dass beim Auschecken der neusten Revision nicht erst Diffs auf die Basisdateien angewendet werden müssen, sondern die Dateien lediglich zlib-komprimiert und mit einem Header versehen sind.

5.3 Git und Subversion in der Anwendung

Nachfolgend findet eine kurze Gegenüberstellung einiger Kommandos von Git und Subversion statt.

5.3.1 Beispiel: Branching

Das Branching in Subversion und Git verläuft vom Prinzip bereits sehr unterschiedlich (siehe Abschnitt 3.1.5 und 4.1.3).

Branch in SVN

Das Branching in Subversion ist im Wesentlichen eine Kopie in den branches-Ordner des Repositories. `svn copy PFAD/trunk PFAD/branches/BRANCHNAME -m "message"` erzeugt dabei den Branch im zuvor genannten Ordner. Für den Wechsel zum Branch ist ein entsprechender Ordnerwechsel nötig.

Branch in Git

Im Gegensatz zum Branching in Subversion ist das Branching in Git mittels eines einfachen `git checkout -b BRANCHNAME` erledigt. Hierzu ist es nicht nötig den gesamten Pfad anzugeben oder einen eigenen Ordner für Branches anzulegen. Da Branches in Git vollständig von Git selbst übernommen werden, ist zum Wechseln des Branches ein `git checkout BRANCHNAME` nötig. Der aktuelle Branch ist in der Auflistung (`git branch`) mit einem * gekennzeichnet.

5.3.2 Beispiel: Tag

Das Tagging in Subversion ist, wie bereits in Abschnitt 3.1.5 erwähnt, ebenfalls eine Kopie der Dateien mittels `svn copy`. Im Gegensatz dazu besitzt Git ein eigenständiges Kommando zum Taggen.

Tagging in SVN

Mittels `svn copy PFAD/trunk PFAD/tags/TAGNAME -m "msg"` erzeugt man in SVN einen Tag. Dieses findet sich in der Arbeitskopie im vorher dafür angelegten Ordner tags.

Tagging in Git

Bei Git erfolgt das Tagging via `git tag TAGNAME`. Dabei werden die Tags nicht in einem speziellen Unterordner abgelegt, sondern deren Verarbeitung erfolgt intern, genau so wie bei Branches, direkt in Git und bildet damit einen Marker für eine bestimmte Revision.

5.3.3 Beispiel: Merge

Zum Mergen in Subversion ist es nötig die URL zu kennen, unter welcher der Hauptentwicklungszweig (Trunk) zu finden ist.

Merge in SVN

Der Aufruf von `svn merge URL` lädt die neuste Version von URL und versucht automatisch die Dateien zusammenzuführen.

Merge in Git

In Git benötigt man zum Mergen zweier Entwicklungszweige lediglich deren Namen. Mittels eines Aufrufs von `git merge BRANCHNAME` versucht Git, ebenso wie Subversion, die Änderungen automatisch zusammenzuführen.

5.3.4 Beispiel: Log

Das Log stellt das Werkzeug zum Auffinden der Revision einer Änderung dar. Deshalb ist es wichtig, dass dieses übersichtlich ist.

Log in SVN

In Subversion besteht die Möglichkeit mit `svn log [datei]` das gesamte Log des Repositories, oder nur das Log einer bestimmten Datei anzuzeigen. Die Ausgabe ist dabei relativ simpel gehalten.

```
------------------------------------------------------------------------
<Revision> | <Username> | <Date> | <XXX> lines

<Message>
```

Falls das Log mehrere Einträge enthält, werden diese durch einen horizontalen Strich voneinander getrennt.

Log in Git

Das Log in Git ist ähnlich aufgebaut, bietet jedoch einige weitere Optionen, wie beispielsweise das Färben der Ausgabe. Die grundsätzliche Ausgabe sieht dabei wie folgt

aus:

```
commit <Hash>
Author: <Name und E-Mail>
Date: <Datum>

<Message>
```

Zusätzlich dazu gibt es die Möglichkeit Branches, sowie Merges im Log anzeigen lassen kann. Dies ist mit der Option `--graph` möglich.

```
*   commit 857018b535ff5dbac4c7d5dc5145f50c1e8656f1
|\  Merge: 4031a22 c976ca9
| | Author: Volker Mauel <v.mauel@fz-juelich.de>
| | Date:   Mon Nov 11 09:41:45 2013 +0100
| |
| |     Merge branch 'master' of http://...
| |
| * commit c976ca9add87860308a4851d62f54696d6c61eba
| | Author: Volker Mauel <v.mauel@fz-juelich.de>
| | Date:   Sun Nov 10 11:05:03 2013 +0100
| |
| |     interna von svn erweitert
| |
| * commit b760e09b5ee9032d08cc8e3e36cbe2cda56d8697
| | Author: Volker Mauel <v.mauel@fz-juelich.de>
| | Date:   Sat Nov 9 18:52:07 2013 +0100
| |
| |     Subversion weitergeschrieben
| |
* | commit 4031a227ce1f9bbee23e80a050de945baedd44a8
|/  Author: Volker Mauel <v.mauel@fz-juelich.de>
|   Date:   Mon Nov 11 09:41:40 2013 +0100
|
|       svn.tex
|
```

5.4 Statistiken

Um eine grobe Einschätung hinsichtlich der Präferierung beider Versionsverwaltungen zu erhalten, ist es naheliegend Google Trends zu Rate zu ziehen.

5.4.1 Google Trends von Januar 2004 bis November 2013

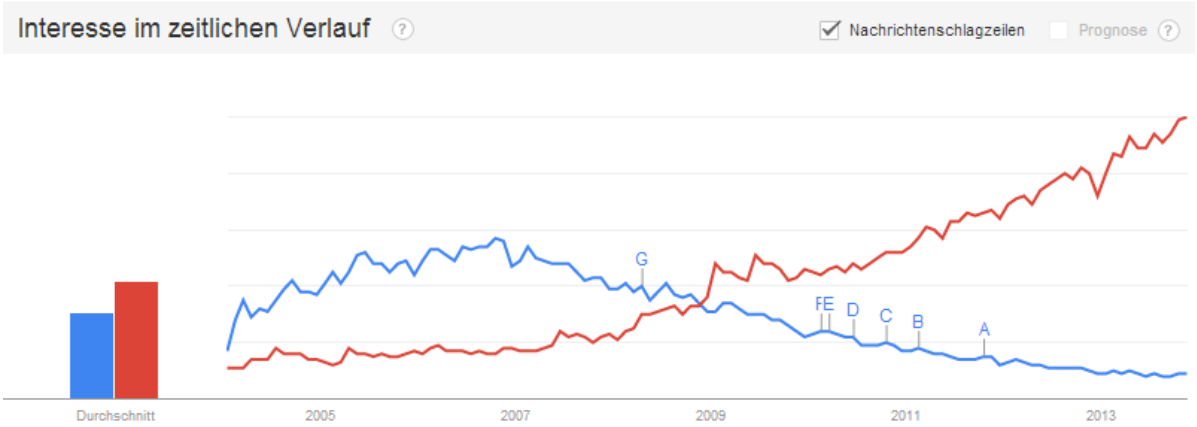


Abbildung 5.1: Google Trends zu Git(rot) und SVN(blau) von 2004 bis 2013.[Goo]

Hierbei ist deutlich zu erkennen, dass das Interesse an Git (rot) im Laufe der Zeit immer weiter gestiegen ist. Im Jahr 2009 überstiegen die Suchanfragen für Git erstmalig die für Subversion. Etwa seit dem Jahr 2007 scheint das Interesse an Subversion stetig abzunehmen.

5.4.2 Git in großen Projekten

Das vermutlich größte Projekt, welches in einem Git-Repository entwickelt wird, ist der Linux-Kernel. Dabei ist es erstaunlich, dass das Clonen des Repositories nur knapp 7 Minuten dauert, obwohl das Repository etwa 3,3 Millionen Objekte verwaltet (Stand 03.12.2013). Hiervon fallen lediglich 37 Sekunden für den tatsächlichen Transfer der Dateien an.

Weitere Projekte wie Android, Ruby on Rails, QT, Gnome, KDE und Eclipse verwenden ebenfalls Git zur Versionskontrolle. Meistens befinden sich die Quelldateien der Projekte auf öffentlichen Plattformen wie GitHub oder BitBucket, damit sich, gemäß dem Forking-Workflow (siehe Abschnitt 4.1.1), andere Entwickler an der Entwicklung beteiligen können.[Gita]

5.4.3 Git in Unternehmen

Jedoch wird Git nicht exklusiv von großen Open-Source-Projekte verwendet, welche das Potential von Git erkannt haben, sondern auch von Unternehmen. Etablierte Unter-

nehmen des IT-Sektors wie Google, Facebook und Microsoft haben ebenfalls öffentliche Repositories bei GitHub, oder bieten eine Git-Integration in ihren eigenen Produkten[1].[Gita]

[1]Microsoft bietet mit Visual Studio 2013 die Möglichkeit Git als Versionsverwaltung zu nutzen. Zusätzlich ist es möglich auf der Serverseite des Team-Foundation-Server Git-Repositories zu hosten.

Kapitel 6

Fazit

Sowohl Subversion als auch Git sind sehr gute Systeme zur Versionsverwaltung. Beide erfüllen die Vorgabe Änderungen an einem Projekt mitzuverfolgen und bieten die Möglichkeit ältere Versionen wiederherzustellen.

Dennoch ist Git zu präferieren. Dies liegt nicht alleine daran, dass es schneller als Subversion ist, sondern auch, dass es den Entwicklern mehr Freiheiten bietet. Jeder Entwickler kann lokal mit Branches arbeiten und ist nicht dazu gezwungen diese auch auf den Server zu übertragen. Hinzu kommt, dass Branches sehr wenig Festplattenspeicherplatz beanspruchen: Zum Erzeugen eines Branches wird nämlich lediglich eine Datei mit einem Verweis auf den ursprünglichen Commit angelegt, aus dem der Branch entstanden ist. Weiterhin benötigt Git für die meisten Kommandos keine Verbindung zum Server, wodurch mobiles Arbeiten an Notebooks erleichtert wird, da man wie gewohnt vorgehen kann, Änderungen abschließt und committed, um diese anschließend, sobald man im Büro oder Zuhause ist, in ein Remote-Repository übertragen kann.

Hinzu kommt, dass es für Git sehr einfach möglich ist, mittels HTTP / WebDAV Repositories auszuliefern und Änderungen darüber in Empfang zu nehmen.

Aufgrund eigener Erfahrungswerte in der Nutzung von beiden Versionsverwaltungssystemen, entschied ich mich zur Verwendung von Git für meine Projekte. Dies ermöglicht mir eine internet- bzw. netzwerkunabhängige Entwicklung ohne auf die Verwendung eines Repositories verzichten zu müssen. Der geringe Speicherplatzverbrauch der Repositories ist in Anbetracht der hohen Preise für SSDs zusätzlich von Vorteil.

Alles in allem würde ich für zukünftige Projekte ebenfalls Git empfehlen, da es die bisherige Arbeitsweise mit Subversion (den zentralisierten Workflow) weiterhin unterstützt und sogar noch zusätzliche Möglichkeiten bietet. Der Anschluss an andere Systeme, wie beispielsweise Jenkins für Continuous Integration, also automatisierte Builds und Tests, ist ebenso gegeben, wie die Integration in die meistgenutzten IDEs (Visual Studio, XCode, Eclipse, Netbeans). Der Umstieg fällt Entwicklern durch die ähnlichen Kommandos relativ leicht, da sie im Prinzip nur wenig Neues lernen müssen. Git bietet zusätzlich die Option ein Subversion-Repository als Remote-Respository zu nutzen, wodurch der Umstieg erleichtert wird. (vgl. [Cha09, Kapitel 8]) Damit ist es möglich in einer Umgebung, in der Subversion als Versionskontrollsystem eingesetzt wird, trotzdem von den Vorzügen Gits zu profitieren.

Literaturverzeichnis

[Atl] Git Workflows. Zugriff 3.12.2013. URL: https://www.atlassian.com/git/workflows#!workflow-forking.

[Cha09] Scott Chacon. Pro Git. 2009. URL: https://github.s3.amazonaws.com/media/progit.en.pdf.

[CSFP11] Ben Collins-Sussman, Brian W. Fitzpatrick und Michael C. Pilato. Versionskontrolle mit Subversion. [Revision 4543]. 2011. URL: http://svnbook.redbean.com/de/1.6/svn-book.pdf.

[Gita] Git Homepage. Zugriff 3.12.2013. URL: http://git-scm.com/.

[Gitb] GitSvnComparison. Zugriff 3.12.2013. URL: https://git.wiki.kernel.org/index.php/GitSvnComparison.

[Goo] Google Trends Git und Subversion. Zugriff 3.12.2013. URL: http://tinyurl.com/gitsvngoogletrends.

[Svn] [Revision 1128011]. URL: http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_fs_fs/structure.

[Wika] Wikipedia: Apache Subversion. Zugriff 25.09.2013. URL: http://de.wikipedia.org/wiki/Apache_Subversion.

[Wikb] Wikipedia: Git. Zugriff 18.11.2013. URL: http://de.wikipedia.org/wiki/Git.