Git In the Trenches

Peter Savage

February 2011

Inhaltsverzeichnis

Kapitel

Woche 1

Tag 1 - “Es muss sich etwas ändern”

Meeting mit dem Team

Falls Du bereits ein erfahrener Nutzer eines Versionsverwaltungssystems bist, kannst Du dieses Kapitel ruhigen Gewissens überspringen. Es ist lediglich eine Art Erklärung, warum man überhaupt VCS verwendet. Dieses Kapitel beleuchtet die Anforderungen von Tamagoyaki Inc. und warum sie das VCS auswählten, das ihnen am geeignetesten erschien. Tamagoyaki Inc. erstellt Software, die einen handelsüblichen PC in ein Medien-Center verwandelt. Ihr Produkt wird an Endkunden verkauft und sie sind stark darauf angewiesen, sich auf Handelsmessen gut präsentieren zu können, um gute Verkaufszahlen zu erreichen. Das folgende Gespräch beschreibt die Vorgänge, die schlussendlich zur "Wir brauchen ein VCS!Diskussion führten.

In the trenches...

John saß an seinem Schreibtisch und schaute aus dem Fenster. Der Regen tröpfelte die Scheibe

herunter, aber das störte ihn nicht. Es war ein ruhiger Montag Morgen, der Release war gut verlaufen und John dachte lediglich daran, wie er die neue Datenbankschnittstelle implementieren könnte, worum er gebeten worden war. Wegen der Musik, die in seinem Kopfhörer lief, bemerkte er kaum, wie sich sein Chef, der Chefarchitekt und der Geschäftsführer sich seinem Tisch näherten.

"John," rief Markus, sein Vorgesetzter, "schaff dein Team in das Vorstandszimmer. Sofort!"

Das hörte sich nicht gut an.

* * *

"Also John, wir wollen wissen, wie der Bug, der vor zwei Wochen hätte..." der Geschäftsführer ruderte zurück, "der vor zwei Wochen als gelöst präsentiert wurde, es in die finale Version der Software geschafft hat?"

"Es tut mir leid," begann John, bevor er abgewürgt wurde.

"Eine Entschuldigung hilft uns nicht John," sagte der Vorstandsvorsitzende Wayne Tobi. "Das war beinahe eine riesen Peinlichkeit für Tamagoyaki Inc. Wir müssen sicherstellen, dass so etwas nicht wieder passiert. Die Demonstration auf der Messe war beinahe ein völliger Fehlschlag. Glücklicherweise hatte jemand daran gedacht, eine Ersatz-Maschine mitzunehmen." Er wandte sich an Markus: "Ich will heute Abend noch einen Bericht auf meinem Schreibtisch haben, in dem steht, was das Problem war, wie es uns durch die

Finger gehen konnte und wie wir uns zukünftig gegen derlei Fehler absichern können."

"Natürlich Sir," antwortete Markus. Er war vor Peinlichkeit hellrot angelaufen.

Im Zimmer wurde es still und ein paar Minuten der Stille vergingen bevor das Meeting beendet wurde und John und sein Team gehen konnten.

* * *

"Also, Du willst mir erzählen, dass Simon eine ältere Kopie der Library vom Netzlaufwerk geholt hat und seine neuesten Änderungen in diese Kopie einpflegte?" Markus hielt seinen Ärger zurück.

"Es scheint fast so," sagte John mürrisch.

"Verdammt nochmal! Wie konnte das geschehen? Warum hat er nicht die neueste Version genommen? Und warum ist das der Qualitätssicherung nicht aufgefallen?" Markus schaute quer durch den Meeting-Raum zu John. "John, du musst sicherstellen, dass sowas nie wieder passiert. Finde eine Lösung!"

Das Problem mit der Speicherung

Es ist nicht so, dass diese Situation völlig ungewöhnlich wäre. Die meisten Leute haben es wohl schon mal geschafft älteren Code zu kopieren und diesen versehentlich anstatt der neuesten, aktuellen Version zu verwenden. Wenn man Code auf Netzlaufwerken oder lokalen Festplatten speichert ist es einfach den Überblick zu verlieren, welche Version welche ist, egal wie gut die Namenskonvention ist. Das ist, als würde man versuchen eines

von diesen Puzzles mit den gebackenen Bohnen machen, wovon man drei Schachteln hat und alle drei in eine Schachtel kippt, weil es einfacher ist. Nun aber nicht mehr ganz so einfach, oder?

Menschen neigen dazu, ihre Ordner so zu benennen, dass die Namen ihnen etwas sagen. Trotzdem bedeutet das nicht notwendigerweise, dass dieser Name auch anderen Entwicklern etwas sagt. "Version 2.3 - fixed bug a" ist auch nur dann aussagekräftig, wenn man weiss welcher Bug das ist, und so etwas wie "Version 2.3 - fixed bug a(2)" ist sogar noch schlimmer. Den Leuten zu erlauben selbständig eigene Dateinamen zu vergeben führt unglücklicherweise immer zu derartigen Problemen. Wenn diese Dateien auf einem Netzlaufwerk gespeichert werden, verschlimmert sich das Problem noch um das Zehnfache, weil es dort oftmals keinen festen Bezugspunkt gibt.

Was wäre also die Lösung? Nun, in sehr vielen Fällen kann eine Versionsverwaltung nicht nur sicherstellen, dass es einen festen Speicherort mit definierter Struktur für die Daten gibt, sondern auch, dass es eine vollständige Historie des Codes gibt. Verantwortlichkeit ist sehr wichtig im Geschäft der Software-Entwicklung, besonders dann, wenn die Software an Kunden verkauft wird. In manchen Situationen wird ein Kunde eventuell sogar anordnen, dass der Code, der für ihn entwickelt wird, in einem Versionsverwaltungssystem gespeichert wird. Auf diesem Wege kann der Kunde nachvollziehen wann ein bestimmter Teil des Quellcodes verändert wurde oder wann ein neuer Teil das erste Mal hinzugefügt wurde.

Tag 3 - 'Eine mögliche Lösung'

Feinheiten der Versionsverwaltung

Es gibt viele Programme für Versionsverwaltung, zum Beispiel Git, Mercurial, Subversion, CVS und Bazaar, um nur einige

der Open-Source Lösungen zu nennen. Wahrscheinlich ist die relevantere Frage, welches VCS man benutzt. Jedes von ihnen hat seine Vorteile und Nachteile, aber manche sind eher für bestimmte Aufgaben geeignet als andere. Fall man mit anderer Software interagiert, oder etwas mit anderen Entwicklern zusammen programmiert, sollte man aßerdem daran denken, zu erfragen welche Software diese verwenden. Für gewöhnlich ist Zusammenarbeit, Forking und Patching wesentlich einfacher, wenn man das selbe VCS wie der Upstream oder die Mitwirkenden.

In the trenches...

> "Also, es scheint so, als wäre die einzig mögliche Lösung für dieses Problem, abgesehen von Klaus Vorschlag die Arbeit auf lediglich einen Entwickler zu reduzieren - danke Klaus - ," Klaus nickte John zustimmend zu, "ein Versionsverwaltungssystem zu implementieren."
>
> Markus kaute auf seinen Lippen. "Ich verstehe Deine Anischt John, aber sind Versionsverwaltungssysteme nicht ziemlich teuer?"
>
> "Es gibt viele Open-Source-Tools dafür, die könnten wir uns zuerst ansehen," meldete sich eine neue Stimme in der Diskussion zu Wort, "manche von ihnen sollen sehr gut sein."
>
> "Beenden wir doch das Meeting, evaluieren die verschiedenen Vor- und Nachteile und treffen uns Morgen wieder, um die Ergebnisse zu diskutieren," sagte John. "Klingt das gut?"

Nun müssen wir uns also ein paar Features der verschiedenen VCS ansehen und herausfinden, wo jeweils die Stärken und

Schwächen liegen. Wir werden uns hier hauptsächlich auf Git konzentrieren, nachdem das restliche Buch davon handeln wird. Da Du dieses Buch liest, nehmen wir an, dass Du höchstwahrscheinlich bereits eine Entscheidung getroffen hast, welches Versionsverwaltungssystem Du benutzen wirst. Lass uns nun also über die verschiedenen Feautures reden, die in den meisten VCS vorhanden sind.

Dezentrale Versionsverwaltung

Versionsverwaltungssysteme kann man für gewöhnlich in zwei Kategorien einteilen; zentral oder dezentral. Git ist ein dezentrales VCS. Es wurde so entwickelt, dass beinahe alles lokal abgewickelt werden kann. Dies wird ersichtlicher, wenn wir später auf andere Feautures von Git eingehen, aber fürs erste reicht es zu verstehen, dass Git nicht auf ein zentrales Repository angewiesen ist. Das ist sehr mächtig. Wirklich!

Branching

Die meisten VCS beherrschen das Branching. Branching erlaubt es Entwicklern im Wesentlichen eine Kopie ihres Repositories zu erstellen und damit zu experimentieren, mit der Gewissheit, dass sie jederzeit zum Original zurückkehren können, falls das notwendig werden sollte. Dies gibt den Entwicklern die Freiheit mit allerlei Dingen zu herumzuspielen, ohne Angst haben zu müssen, dass der originale/saubere Code davon betroffen ist.

Git implementiert Branching auf besondere Art und Weise. Die meisten der älteren VCS setzen Branching derart um, dass eine seperate Kopie des Rpositories erzeugt wird. Das ist aber langsam und mühselig. Gits Art des Branchings gibt den Entwicklern die Möglichkeit mehrere lokale Branches zu erstellen, um in diesen testen zu können. Wegen seiner dezentralen Struktur können Entwickler auswählen welchen Branch sie

pushen wollen, wenn der Code in eine zentralere Stelle integriert werden soll, von der die anderen den Code beziehen können. Dadurch kann der Code privat getestet werden.

Die Implementation des Branching in Git ist schnell. Dadurch, dass Repositories lokal gespeichert werden, ist die Geschwindigkeit beim Erstellen eines Branches nur durch die Geschwindigkeit der Festplatten auf dem lokalen Computer limitiert.

Staging

Git geht mit COmmits anders um als die meisten anderen VCS, indem sie eine Staging Area einführen. Die Staging Area erlaubt es Entwicklern, ihre Commits vorzubereiten, bevor sie in das Repository geschrieben werden. Warum ist das nützlich oder unterschiedlich zu anderen Versionsverwaltungssystemen? In Git kann man eine Datei verändern, sie zur Staging Area hinzufügen und dann weiterhin Änderungen an der Datei vornehmen, sogar wenn man noch nicht mal etwas commitet hat. Es bleibt aber auch zu erwähnen, dass man die Staging Area nicht nutzen muss, aber es gibt sie, für Entwickler, die sie nutzen möchten. It should be noted that it's not absolutely necessary to use the staging area, but it is there for developers wishing to utilise it.

Workflow

Durch die Art und Weise wie Git entwickelt wurde ist es möglich, es in praktisch jedem möglichen Workflow zu nutzen. Drei der wichtigsten Workflows sind unten erklärt, wobei Git mit jedem von ihnen genutzt werden kann, was es zu einem der vielseitigsten Systeme macht.

Centralised Workflow

Ein zentralisierter Workflow zeichnet sich dadurch aus, dass ein einziges Repository benutzt wird. Mehrere Entwickler laden Dateien von dort in lokale Kopien davon, arbeiten an der lokalen Version und laden die Änderungen dann wieder hoch in das zentrale Repository.

Damit kann Git genauso umgehen wie beinahe jedes andere VCS auch. Ein Entwickler kann seine Änderungen nicht Upstream schicken, solange er nicht den aktuellssten Stand des zentralen Repositories lokal gespeichert hat und eventuelle Konflikte beseitigt hat.

Wenn man mit dem zentralisierten Workflow arbeitet, haben alle Entwickler die gleichen Zugriffsrechte auf das Repository und jeder Entwickler ist genauso **wichtig** wie jeder andere. Dies mag zwar in kleineren Teams funktionieren, aber sobald die Anzahl der Entwickler größer wird, könnte das zentralisierte System kompliziert werden. Wenn immer mehr Leute anfangen auf die gleichen Dateien zuzugreifen, treten Konflikte und andere Hindernisse immer häufiger auf.

Integration Manager Workflow

Der Integration Manager Workflow ist dem zentralisierten sehr ähnlich, weil es auch hier ein **Blessed Repository** gibt, das jeder Entwickler als Referenz benutzt. Der Unterschied liegt darin, dass es nur eine Person gibt, welche die Änderungen in das **Blessed Repository** einpflegt. Diese Person wird Integration Manager genannt.

Dieser Workflow funktioniert mit Git sehr gut. Entwickler arbeiten an ihrem lokalen Repository und sobald sie mit ihren Änderungen zufrieden sind, laden sie ihre Changes an einen Ort, wo sie der Integration Manger sehen kann. Dann begutachtet der Integration Manager die Veränderungen, die die Entwickler

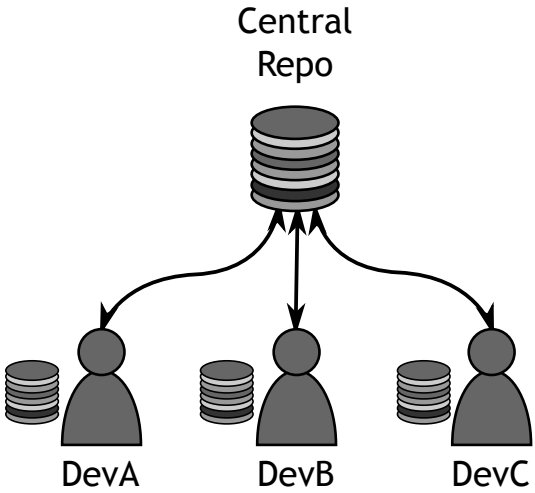

Abbildung .1: Centralised Workflow

gemacht haben, und kopiert sie in ein eigenes lokales Repository. Wenn dann alles wie geplant funktioniert, lädt der Integration Manager die Changes in das Blessed Repository, so dass alle anderen Entwickler zugriff auf den neuen Code haben.

Dictator and Lieutenant Workflow

Der Diktator und Leutnant Workflow ist sozusagen eine Erweiterung des Integration Manager Workflows. Er passt eher zu größeren Teams, wo Elemente oder Teile des Codes einem **Leutnant** zugewiesen werden können, der dafür verantwortlich ist, jede Änderung seines Bereichs abzusegnen.

Sobald die Leutnanten mit ihrem Code zufrieden sind, machen sie den Code dem Diktator zugänglich. Dieser nimmt dann eine ähnliche Rolle wie der Integration Manager des vorigen Workflows ein. Schlussendlich werden alle Changes in das Blessed Repositpory gepusht, von dem die Entwickler mit weniger Rechten den Code beziehen können.

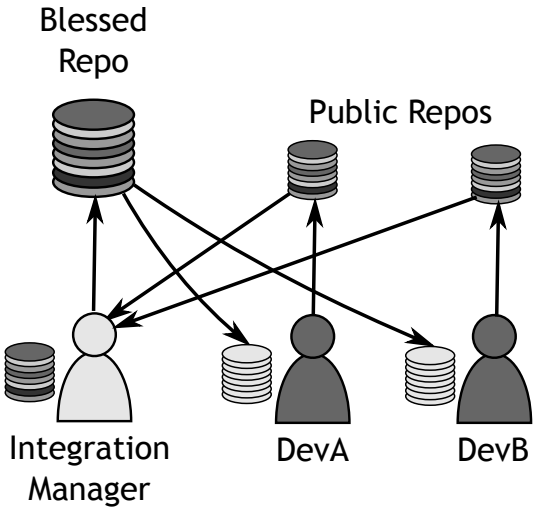

Abbildung .2: Integration Manager Workflow

Die Hauptsache, an die man sich erinnern sollte, ist, dass Git für jeden dieser Workflows geeignet ist. Dadurch ist es ein sehr flexibles System, das es Dir erlaubt, auf jedem dieser Wege zu arbeiten, wie auch immer Du Dich entscheidest.

Offline Committing

Eines der nützlichsten und am meisten unterschätzten Features eines dezentralen VCS ist wahrscheinlich das Offline Committing. Es ist vielleicht deswegen unterschätzt, weil nicht alle Versionsverwaltungssysteme über dieses Feature verfügen. Offline Committing bedeutet, dass man Dateien zum Repository hinzufügen kann, ohne mit einem zentralen Repository verbunden zu sein.

Wenn man reist, oder schlichtweg nicht im Büro ist, können Entwickler und Integratoren dennoch damit fortfahren Code zu reviewen, die letzten Änderungen begutachten, Diffs anschauen

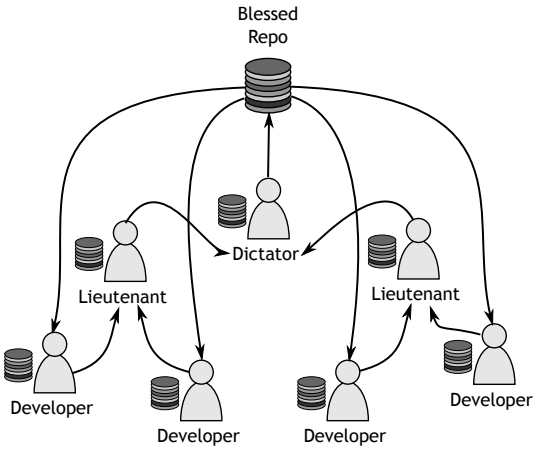

Abbildung .3: Dictator and Lieutenant Workflow

Terminology

Blessed

Ein Blessed Repository (oder Canonical) ist Repository, welches die Anerkennung des Projektmanagers hat. Es ist sozusagen der Standard, von dem alle anderen Kopien gemacht werden. Wenn es einen Ort gibt, an dem der Code fehlerfrei sein sollte, dann im Blessed Repository. Falls Du Dein Projekt öffentlich zugänglich machst, wird das Blessed Repository für gewöhnlich das sein, auf das alle zugreifen können um es als Ausgangspunkt für ihre Entwicklungen zu nehmen.

und Code-Änderungen in das Repository committen. All dies ist der Tatsache geschuldet, dass Git 99% seiner Operationen lokal ausführt. Wenn ein Repository kopiert wird, setzt Git lokal tatsächlich eine Kopie des kompletten Repositories auf, was den Entwicklern die Flexibilität gibt von überall aus zu arbeiten, ohne dass Zugang zum Firmennetzwerk vorrausgesetzt wird.

Sobald die Entwickler dann in ihr Büro zurückgekehrt sind laden sie ihre Änderungen einfach in das "öffentliche" Verzeichnis (sei es nun lokal oder ein Blessed Repository) und alle Commits, die sie in ihrer Abwesenheit fertiggestellt haben, werden dem Rest des Teams inklusive der gesamten Historie und Zwischenständen zugänglich gemacht

Developer Interaction

Eine Faktor, den man bei der Auswahl des zu verwendenden Versionsverwaltungssystems beachten sollte, ist der der Entwickler-Interaktion. Damit ist die Art und Weise gemeint, auf die die Entwickler das VCS benutzen und es bedienen. Es gibt vier Arten der Interaktion

Graphical User Interface Client (GUI)

Eine GUI erlaubt es dem Entwickler, oder Benutzer, das Repository in einer grafischen Oberfläche mit Maussteuerung zu verändern. Ein GUI-Client besteht normalerweise aus einer seperaten Anwendung, die gestartet wird, wenn ein User am Repository Änderungen vornehmen will, zum Beispiel Dateien hinzufügen oder Modifikationen committen will.

Manche Entwickler ziehen einen extra Client für die Interaktion vor, wohingegen andere es lieber mögen, wenn die Tools ineinander integriert sind.

Shell Extension Integration

Die Integration in die Shell macht es Entwicklern möglich, mit dem Repository so zu interagieren, wie sie es normalerweise beim modifizieren von Dateien und Ordnern auch tun würden. Eine der am meisten benutzten Shell-Erweiterungen für Git ist TortoiseGit, welches sich selbst in den Windows Explorer

integriert. Dadurch wird dem User ermöglicht, durch einen Rechtsklick auf ein File in einem Git-Repository kontextsensitive Aktionen durchzuführen.

Command Line Interface (CLI)

Das Kommandozeilen-Interface wird von vielen Entwicklern bevorzugt, da sie es skripten können und genau sehen, was geschieht - oftmals sehr viel dataillierter, als in einer GUI. Das Command Line Interface gibt einem vollständige Kontrolle über das Produkt, es ist ausserdem erwähnenswert, dass beinahe alle Versionskontrollsysteme ihr Leben als kommandozeilengesteuerte Programme begannen. Aber warum ist das so? Es braucht viel Zeit und Arbeit, alle Optionen und Kleinigkeiten in eine GUI zu integrieren! Das CLI wird fast immer das mächtigste aller Tools sein, besonders dort, wo VCS betroffen sind.

Day 4 - "Eine Entscheidung wurde gefällt"

Die Vorraussetzungen prüfen

Der wichtigste Aspekt bei der Wahl eines Versionskontrollsystems ist die Definierung der Vorraussetzungen. Diese können wenige sein, oder sie sind sehr speziell; aber lasst uns sehen, was John und sein Team für die wichtigsten Anforderungen an ein VCS halten und wofür sie sich schlussendlich entschieden haben.

> **In the trenches...**
>
> "Es scheint, als wäre offline-committing eine sehr nützliche Sache. " sagte Mike nickend. "Besonders für Leute wie John, die ständig reisen."

> "Dem stimme ich zu, es wäre toll in einem Flugzeug sitzen zu können und trotzdem den gesamten Code zusammenfügen zu können, die History jedes Abschnitts zu kennen, " antwortete John. "Git scheint in Bezug auf das Branching auch sehr mächtig zu sein."
>
> "In der Tat," schaltete sich Klaus ein, "Ich nutze Branching zuvor in Subversion und es war ein Lebensretter. Git wird nachgesagt, dass es das auch sehr schnell bewerkstelligt."
>
> "Dank der Tatsache, dass Git scheinbar mehrere Arbeitsweisen unterstützt können wir sie ausprobieren, um zu prüfen, wie sie für uns funktionieren." Markus schaute sein Team an. "Also legen wir uns auf Git fest?"
>
> Das Team nickte und jeder ausser John verliess den Meeting-Raum. Das versprach interessant zu werden, sehr interessant.

Da dieses Buch von Git handelt, werden wir nicht allzu tief auf die Funktionsweisen oder Features anderer VCS eingehen. Dieses Kapitel hat Dir hoffentlich genug Informationen gegeben um andere Versionskontrollsysteme auszuprobieren, falls nötig. Die Hauptsache, die man im Hinterkopf behalten sollte ist, dass Git ein dezentrales VCS ist. Deshalb ist es wichtig, daran zu denken, dass es mit den gleichen Workflows benutzt werden kann wie zentrale Systeme es tun.

Die Anforderungen von John und seinem Team sind nicht sehr speziell. Sie sind ein eher kleines Team, das die Vorteile eines gut organisierten Codes haben möchte. Außerdem möchten sie ihre Team-Funktionen und die Dynamik einbringen, um sie an das Versionskontrollsystem anzupassen und es somit zum Kern ihrer Entwicklungsarbeit zu machen.

Versionskontrolle ist kein Ersatz für einen guten Workflow. Es ist nicht erfunden worden, um alles besser zu machen. Wenn Du Leute in deinem Team hast, die nur ihr eigenes Ding durchziehen und sich nicht um ihre Arbeitsweise sorgen, wird ein VCS nicht auf einmal alles verbessern. Ein Tool ist nur ein Tool, und Versionskontrolle ist nichts anderes: Ein Tool. Du kannst dem unordentlichsten Baumeister eine schöne neue Werkzeugkiste kaufen, aber solange er nicht den Willen hat sich zu ändern wirst Du feststellen, dass alle seine Werkzeuge im größten Fach am Boden landen.

Day 5 - "Arbeiten wie ein Team"

Team-Organisation

Nun, da wir die Grundlagen behandelt haben, sehen wir uns an, wie John sein Team zusammengestellt hat, um herauszufinden, ob sie Versionskontrolle überhaupt einsetzen können. Es ist sehr wichtig, dass das team versteht wie das Modell funktioniert, was von ihnen erwartet wird und inwiefern sie Zugang zum Repository haben. Meistens werden die Leute frustriert, wenn sie nicht wissen, was sie tun oder eben nicht tun, oder weil sie keinen Zugriff auf bestimmte Teile des Codes haben.

> **In the trenches...**
>
> Es war 16:36 Uhr am Freitag und der Tisch im Meeting-Raum war voll mit leeren Cola-Dosen, Pizza-Schachteln und einer japanischen Obentobox, die einem besonders starrköpfigen Mitglied des Teams gehörte, der geschworen hatte, nie wieder Pizza zu essen. Es war Markus Idee gewesen, das Essen als Verstärkung zu bestellen, um die Diskussion wieder gestärkt fortführen zu

können. Das Team versuchte zu entscheiden, wie seine Organisation aussehen sollte.

"Es gibt also keine Möglichkeit, eine Kombination von Modellen einzuführen?" fragte Mike.

"Ich schätze nicht," sagte John. "An was hast Du denn gedacht?" Seine Brille rutschte seine Stirn herunter und er wurde nun richtig müde.

"Nun, ich nehme an, dass wir die Software in zwei Teile aufgeteilt haben. Wir haben die Library, an der Klaus, Jack und ich arbeiten. Dann haben wir noch die grafischen Elemente, um die sich Simon, Martha und Rob kümmern. Und ich kenne die Tools, die Eugene schreibt." Jeder hörte nun Mike zu, als dieser fortfuhr: "John, Du willst Dich nicht um die Libraries kümmern müssen, weil das eher Klaus Part ist. Warum setzen wir also nicht zwei Integratoren ein? Klaus und Du selbst habt die Rechte um in das Blessed-Repository pushen zu können. John kann von von seinen Jungs pullen, wohingegen Klaus den Code von seinen Team-Mitgliedern bezieht. Schlussendlich haben wir ein gutes modell der Versionskontrolle."

John hob seine Augenbrauen an, "Nicht schlecht Mike," sagte er, sichtlich beeindruckt. Nachdem sie einige Stunden damit verbracht hatten, die verschiedenen Workflows durchzugehen und Zuständigkeiten abzuklären, fühlte es sich nun gut an, endlich eine Entscheidung getroffen zu haben.

"Also fangen wir am Montag an?" fragte Markus, der vom anderen Ende des Tisches zugehört hatte.

"Allerdings!" Verkündete Klaus, "Am Montag werden wir alle Gits!"

Kapitel

Woche 2

Tag 1 - "Wir sind Programmierer, wir benutzen Git!"

Die Entwicklungsumgebung aufsetzen

Jetzt sind wir also bereit in die Tiefen von Git abzutauchen und es tatsächlich zu benutzen, oder? Naja, nicht wirklich. Zuerst müssen wir uns entscheiden, wie der Workflow, für den wir uns entschieden haben, in unserem Versionsverwaltungssystem implementiert werden soll. Die Tatsache, dass Git so vielseitig ist, ist Segen und Fluch zugleich. Es ist eine gute Idee von Anfang an festzulegen, wie sich Entwickler, Reviewer und Integratoren verhalten sollen, bevor man tatsächlich anfängt Code zu committen. Aber manchmal ist das nicht möglich. Es ist gut möglich, dass Du noch nie ein Versionskontrollsystem wie Git benutzt hast und Du daher anfängst, Dich durch das Thema durchzukämpfen. Das ist zwar normal, aber falls Du vor hast diese Art von System in eine produktive Umgebung einzupflegen, solltest Du erst herausfinden, wie das überhaupt funktioniert.

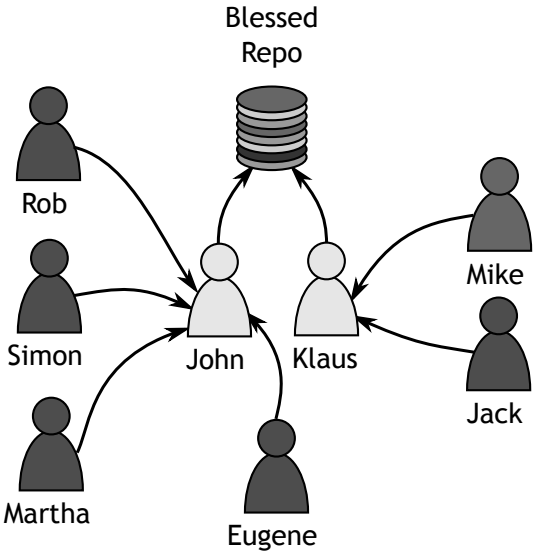

Abbildung .1: Tamagoyaki Inc's Physical Structure

Das Modell, welches vorhin vorgestellt wurde, ist leicht vorzustellen. Wir haben zwei Integratoren, die beide Zugriff auf das Blessed Repository haben, und es gibt mehrere Entwickler, deren Changes gereviewed und integriert werden; eben von jenen Integratoren. Die physische Darstellung dieses Workflow-Modells ist im Diagramm unten aufgeführt.

Die physische Struktur is einfach und gut, aber sie legt nicht genau fest, wie die Daten bewegt werden, sie legt lediglich fest, wer dafür verantwortlich ist auf der jeweilgigen Stufe des Ablaufs. Was nun benötigt wird, ist eine detaillierte Analyse wo die Daten herkommen und wo sie sich hinbewegen. Ein Datenflussdiagramm ist hierfür nützlich, aber nicht essentiell. Nichtsdestotrotz werden wir eine leicht veränderte Form des Diagrammes, um zeigen zu können, wie die Daten von einer Person zur anderen übertragen werden. Bevor wir fortfahren und einen Blick auf das Diagramm werfen, lass uns kurz in die

Schützengräben zurückkehren und nachschauen, wie die Leute mit ihrem Repository-Design zurechtkommen.

In the trenches...

"John, warum muessen wir uns an einem Montag Morgen um 09:45 Uhr hier treffen." beklagte sich Klaus. "Ich habe noch nicht einmal genug Kaffee intus, um meine E-Mails zu checken, ganz zu schweigen von einem Meeting!"

John grinste, "Ich glaube nicht, dass Dir Kaffee da helfen würde, Klaus. Es ist eine Gewinner-Persönlichkeit, die Dir da helfen wird." Der Rest des Teams lachte, bis John begann wild auf dem Whiteboard zu zeichnen. "Also, hier haben wir unser phsyisches Modell. Wir wissen, welche Leute die Verantwortung tragen werden, aber wir wissen nicht, wie wir unsere Repositories anordnen sollen."

"Guter Einwand," stimme Mike ihm zu.

"Also, wir werden offenbar ein Blessed Repository haben," sagte John, während er einen Kreis auf die Tafel malte. Er trat zurück, eine Hand am Kinn. "Weiterhin stelle ich mir vor, dass Klaus und Ich Kopien dieses Repositories auf unseren lokalen Festplatten haben. Wir werden diese modifizieren und dann unsere Changes zurück in das zentrale Repository schieben."

"Aber ich dachte Git hat kein zentrales Repository?" fragte Martha. Die anderen stöhnten.

"Nun," sagte John, "so weit ich das verstanden habe, hat es das tatsächlich nicht. Was ich meine,

ist, dass Klaus und Ich auch lokale Kopien des Repositories haben. Wir arbeiten an diesen lokalen Kopien und pushen unsere Veränderungen zurück auf den Server. Es ist eine Synchronisation, und auch eine Kopie. Ich glaube, das nennt man einen Klon." Er nickte, "Und da Klaus und ich kaum am selben Code arbeiten, müssen wir wohl kaum mergen oder mit Konflikten rechnen."

"Aber was ist mit uns Codeaffen?" fragte Martha, "Woher koennen wir unsere Kopien des Repositories beziehen?"

"Vom zentralen Server natürlich," Rob begann zu lächeln.

"Ja," sagte John, "aber Ich denke, was Martha sagen will ist, wie Ihr die Updates bekommen werdet?" Er begann im Raum herumzulaufen und der eine oder andere der Entwickler folgte ihm, als er das Fenster erreichte und stehen blieb. "Ich denke Ihr würdet Euren Branch mit dem des Blessed Repositories mergen."

Im Zimmer wurde es still und das einzige Geräusch das zu hören war, war das rattern der Klimaanlahe in der Zimmerdecke.

Simon fing an zu rede, "Naja, ich habe am Wochenende etwas über dieses Rebasing gelesen und in manchen Fällen soll ein Rebase besser als Merging sein."

"Was ist Rebasing und worin unterscheidet es sich zum Merging?" fragte Mike.

Nun, Rebasing ist ziemlich clever. Stell Dir das folgendermassen vor: Du hast einen Upstream

> Branch, in unserem Fall das Blessed Repository. Du machst Changes. Wenn sich der Upstream Branch ändert könntest Du die Changes vom Blessed Repository reinmergen. Wenn Du das machst erstellst Du einen eigenen Commit hierfür, der die Veränderungen merged. Das funktioniert zwar, aber..." er driftete ein wenig ab, "es kann unter Umständen Probleme verursachen. Ein besserer Weg das zu handhaben ist Rebasing. Rebasing kann alle Veränderungen, die Du gemacht hast nehmen, sie auf die Seite räumen und alle Upstream-Changes einfügen. Danach werden die vorher beseitigten Changes auf die gerade eingefügten Changes draufgesetzt."
>
> John atmete aus, "Das klingt ziemlich cool Simon, aber eine Sache ist glasklar, wir müssen mehr über die Grundlagen von Git lernen, bevor wir anfangen uns mit Merging und Rebasing zu beschäftigen. Lasst uns den Rest des Tages damit verbringen, mit einigen Test-Repositories zu experimentieren und morgen treffen wir uns dann wieder."

Falls Du noch nie zuvor mit einem VCS zu tun hattest, ist es eine gute Idee, zuvor einige Zeit lang damit herumzuspielen. Schon sehr bald wirst Du die Grundlagen gelernt haben und in der Lage sein, Deine neu erworbenen Fähigkeiten in die Praxis umzusetzen. Aber obwohl es gut ist, in Test-Repositories zu experimentieren, ist es ganz normal, dass man das System tatsächlich benutzen muss um Probleme zu entdecken. Der Rest dieses Kapitels ist eine sehr kurze Einführung in Git. Es ist aufgebaut wie eine Einführung, weil wir erwarten, dass Du Dir etwas Zeit nimmst um Git kennenzulernen und zu lernen wie es funktioniert.

Ein Repository initialisieren

Das erste, das wir machen müssen, ist zwei sehr wichtige Sachen verstehen:

1. Wie man ein Git Repository erstellt
2. Was ein Git Repository tatsächlich ist

Das erste davon ist relativ einfach umzusetzen.

```
john@satsuki:~$ mkdir coderepo
john@satsuki:~$ cd coderepo/
john@satsuki:~/coderepo$ git init
Initialized empty Git repository in /home/john/coderepo/.git/
john@satsuki:~/coderepo$
```

What we've done here is create a new directory called coderepo, moved into it, and then run the git init command. The result of this command is a new directory in the coderepo directory called .git. This directory will hold a local copy of our entire repository. This will allow us to create branches, merge changes, rebase things and ultimately push our changes to somewhere else.

Something that is crucial to the running of a repository, whether you are an administrator of Git, or a developer who is using it, is an understanding of how Git works. It is fine to jump in and play with the repository and test the water, but before committing to using Git in a production environment, you should understand what Git actually does in the background in some detail.

During the writing of this book several people have told me that Git is one of the only version control systems where a good understanding of how the underlying system works is not just highly recommended, but bordering on essential.

Let us take a few minutes to talk about how Git works internally and how the data is actually stored. Git doesn't store

changes to files, but actual snapshots of files at specific points in time. It refers to these by running an SHA-1 hash against the file. By doing this, it is easy to Git to detect if a file has changed. If the SHA-1 hash of a file changes, then the file must have been modified.

When a commit is made to the repository, Git stores a few things. A commit object is created. This contains information about who made the commit, the parent of the commit and points to a tree object. The tree object describes what the repository looked like at the time of the commit. In other words the tree object, tells Git what files were in there. Lastly, Git stores the files that were in the repository under their SHA-1 names in the objects directory. Of course Git is super clever here because if you have exactly the same file in multiple commits, the SHA-1 hash of that file doesn't change and therefore Git only stores one copy of the file to save space.

The commit object is also referred to by an SHA-1 hash. This is different to many other version control systems which use either a number that refers to the repository or a per file version number. Getting used to seeing 40 character SHA-1 hashes can take a little time. Saying Ï need the commit referred to as bf81617d6417d9380e06785f8ed23b247bea8f6d,ïs certainly not as easy as saying you need revision 6. However, Git handles these hashes well, and you can reference a commit using a few of the characters from the beginning, as long as those characters uniquely refer to that commit, i.e., as long as your choice isn't in any way ambiguous.

Day 2 - “Making commitments”

Let’s Make a Repository

The most simple way of committing a file into the repository is to create it, or bring it into your working copy and use the commands below.

```
john@satsuki:~/coderepo$ touch my_first_committed_file
john@satsuki:~/coderepo$ git add my_first_committed_file
john@satsuki:~/coderepo$ git commit -m 'My First Ever Commit'
[master (root-commit) cfe23cb] My First Ever Commit
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 my_first_committed_file
john@satsuki:~/coderepo$
```

What we have done here, is to create a new blank file and add it into the repository using Git’s add command. Then we have committed it into the repository. Let’s make a few changes to our working copy and see what the result is. First we are going to add another two new files, then we are going to make changes to our original file and finally we are going to run git status to see what Git has to say about our changes.

```
john@satsuki:~/coderepo$ echo "Change1" > my_first_committed_file
john@satsuki:~/coderepo$ touch my_second_committed_file
john@satsuki:~/coderepo$ touch my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
 working directory)
#
#        modified:   my_first_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
 committed)
#
#        my_second_committed_file
#        my_third_committed_file
no changes added to commit (use "git add" and/or "git
 commit -a")
john@satsuki:~/coderepo$
```

So we can see that Git is reporting that there are changes to our first committed file, and that our second and third files are **untracked**. Untracked files are ones which Git detects as being present in the working directory, but which haven't yet been added and there for upon running a commit, these files will not be added to the repository. Notice that if we tried to run a commit now, nothing would actually be committed to the repository. Even though there are changes to to my_first_committed_file, we have not asked Git to include these. So, let's go ahead and do that, and at the same time we'll make a few changes to my_second_committed_file, and add those too.

```
john@satsuki:~/coderepo$ git add my_first_committed_file
john@satsuki:~/coderepo$ echo "Change1" > my_second_committed_file
john@satsuki:~/coderepo$ git add my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   my_first_committed_file
#       new file:   my_second_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
 committed)
#
#       my_third_committed_file
john@satsuki:~/coderepo$
```

Now we can see that one of the sections has changed to "Changes to be committed". So this means that Git has recognised and remembered that we are expecting these files to be committed when we next run a git commit.

Committing the Uncommitted

In the trenches...

"John, what is going on here?" shouted Klaus

from across the hallway. The entire office had heard Klaus banging his hands down on the desk for the last fifteen minutes. “John!” the shout turned into a scream.

“Calm down Klaus, I’m just coming.” John walked over to Klaus and pulled up one of the folding plastic chairs. After a few minutes of fumbling he finally managed to take up his position next to an infuriated Klaus.

“John, Git is driving me crazy. I have added files to the repository and I keep running a commit, but the changes aren’t getting put into the blasted repo.” Klaus was clearly distressed and John resisted the urge make jokes.

John pointed at the screen. “Run a git status Klaus and I’ll show you what the problem is.”

To understand what Klaus was getting in a spin about, let’s make a change to my_second_committed_file now and see how this affects things. Remember we have already added the file, but we haven’t yet made a commit.

```
john@satsuki:~/coderepo$ echo "Change2" >>
 my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#        modified:   my_first_committed_file
#        new file:   my_second_committed_file
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be
 committed)
#   (use "git checkout -- <file>..." to discard changes in
 working directory)
#
#        modified:   my_second_committed_file
```

```
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
 committed)
#
#        my_third_committed_file
john@satsuki:~/coderepo$
```

How interesting! We now have three sections and one of our files appears twice under both Changes to be committed and Changed but not updated. What does this mean? If you remember back, we spoke about a staging area. This is one area in which Git differs to many version control systems. When you **add** a file into the repository, Git will actually make a copy of that file and move it into the staging area. If you then go ahead and change that file, you would need to run another git add in order for Git to copy your changed file into the staging area. The most important thing to remember is that Git will only ever commit what is in the staging area.

So, if we go ahead and run our commit now, we will only have the changes marked in Changes to be committed appearing in our repository.

```
john@satsuki:~/coderepo$ git commit -m 'Made a few changes to
first and second files'
[master 163f061] Made a few changes to first and second files
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 my_second_committed_file
john@satsuki:~/coderepo$
```

In our examples, we have used the syntax `git commit -m 'Message'`. This is a slightly special way of committing, it allows us to specify our commit log message on the command line. If we wanted to, we could run the command git commit and this would open a text editor that we could use to input our commands.

Let us finish off our round of committing by using the `git commit -a` option. This commits all of the changes to files

which are already tracked. Consequently we do not have to specify the files with `git add`, like we have had to previously. Any file which has been modified and has previously been added to the repository, will have it's changes committed upon running that command.

```
john@satsuki:~/coderepo$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
 working directory)
#
#        modified:   my_second_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
 committed)
#
#        my_third_committed_file
no changes added to commit (use "git add" and/or "git commit
 -a")
john@satsuki:~/coderepo$
```

```
john@satsuki:~/coderepo$ git commit -a -m 'Finished adding
 initial files'
[master 9938a0c] Finished adding initial files
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

```
john@satsuki:~/coderepo$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#        my_third_committed_file
nothing added to commit but untracked files present (use
"git add" to track)
john@satsuki:~/coderepo$
```

Day 4 - "Let's do this right, not fast"

Uh-Oh I Think I Made A Mistake

So now we are fairly well acquainted with adding files into the repository and performing commits. In a short while we will learn about how to view the changes we have made and perform diffs against various objects. Before we close out the week, we need to go back to the trenches one last time.

In the trenches...

"Rob, ya got a second?" asked Mike.

"Sure, what's up?" replied Rob from across the office. "Gimme two secs to make this commit." The office went silent again whilst Rob's fingers darted across the keyboard. "Ahh. Damn it!" shouted Rob.

Mike rose from his chair and walked over to Rob. "What's up?"

"I just added a file into the staging area, but I don't want it there." He shook his head, "Well not yet anyway."

Mike chuckled, "Sorry for interrupting dude."

"Nah, it's OK, I just need to know how to pull this file out of the index."

"Git reset," shouted a voice. The stillness of the office was interrupted by a chair free wheeling across the floor. The occupant of the chair was Klaus. He seemed proud that he was finally getting to grips with things. "You can use git reset to reset a file that's in the index." He grabbed at the keyboard, "Here, lemme show you."

The git reset is great at removing things from the index that you don't want to be there. Of course, it can do a great many other things, but for now, let us concern ourselves with the scenario presented above. We are working away, and have added a number of files into the index ready for committing, when we discover that we are actually not ready to commit them. In the following example, we are going to add the file my_third_committed_file and then remove it from the index.

```
john@satsuki:~/coderepo$ git add my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#        new file:   my_third_committed_file
#
john@satsuki:~/coderepo$
```

Notice how my_third_committed_file is now ready to be committed to repository. The problem is we need to add something more to it before we do. Remember that when we run the git add command, we are copying the file from our working copy to the index. If we decide we no longer want that file in the repository, we can run the following.

```
john@satsuki:~/coderepo$ git reset my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
 committed)
#
#        my_third_committed_file
nothing added to commit but untracked files present (use
 "git add" to track)
john@satsuki:~/coderepo$
```

We have discarded the file which was residing in the index. This is very important to note. We are not moving the file from the index back into our working directory, we are literally just

deleting the file from the index. Our working copy remains unaffected. We could run the git reset command without appending a file. If we did this, all the files in the index would have been discarded.

In the trenches...

"So, I think we are all agreed, I'll keep a version of the repository under Git version control. Until everyone else feels comfortable with some of the more advanced features." John looked around the room for any disagreements but there were none.

"Agreed John," said Markus, "I'm pleased with how you guys are progressing, very pleased, but like John said, it's far better for us to take our time and to implement this correctly, than to rush it and to end up with something that we can't administrate and that we don't know how it works."

"So next week, I want you all to start playing with diffing and logs and don't forget we have an important release due too." John pushed his glasses further up his nose. "The week after that we'll start looking at branching and by then we may be at the stage where we can implement our model."

Everyone nodded in agreement.

Now we know how to add files into the repository. The question is, what do we do if we need to remove a file, or even rename it. Well, git has some commands to help with that. `git rm` and `git mv` delete and move files respectively. Usually when you want to remove files from the repository, or move them, this

is how you will handle it, but what if you have already deleted a tracked file manually? Well, you have two options. You could run `git commit -a`, but remember this will commit all changes to tracked files. You could also run a `git rm <filename>` with the name of the file you have just deleted. Git will then push that change into the staging area ready for commit. The same applies to moving a file

However, it is worth noting something in the way that Git handles renames. Git does not track renames explicitly. This means that by running the `git mv <source> <dest>` command, you are essentially running a Linux `mv` command, followed by the `git rm` on the source file and `git add` on the destination file. Running the `git mv` command is a shorthand way of doing just that. It is worth playing with this to ensure that you understand what is happening. As an exercise, inspect the repository after each command so that you understand at what point Git recognises your actions as a rename.

We have run through a few basic commands in Git. If you are familiar with version control systems, then possibly the only real difference you will have noticed is that of the staging area. It really is powerful, and allows you to organise and prepare your commits, so that they are both meaningful and coherent.

For Tamagoyaki Inc, their plan to implement version control was far too aggressive. Most of the members of the team had never even used a version control system. When deciding to implement version control, it is essential to ensure that you are doing it for the right reasons. Version control is a tool to help you to keep things in order, but remember tools are nothing without process. It is process that is key to the order.

Knowledge

How do we change the commit message editor?

We spoke earlier about the configuration file and how it stores information about our Git instance. Git can use any text editor you require, even a graphical one, though the need rarely arises. As mentioned earlier, Git has a preference lever when talking about configuration. First and foremost it will look in the repositories own 'config' file in the .git folder. Then, it will look in the users `~/.gitconfig` file. Finally, Git will look in your distributions own global folder.
If we wanted to change the editor that Git would use to modify commit messages, we can either modify the files directly, or run a command similar to the following;

```
git config core.editor "nano"
```

If we want the changes to apply globally, meaning it would affect all repositories we administrate as this user, unless overridden by a repository setting, we would run the following;

```
git config --global core.editor "nano"
```

It is worth noting that you can also use the $EDITOR environment variable to accomplish the same thing. Many people use this in preference to the modifying the Git configuration simply because many other programs honour this setting.

Summary - John's Notes

Commands

- `git add` - Add files into the index or staging area
- `git commit` - Commit files into the repository, using text editor for commit message
- `git commit -m '<Message>'` - Commit files into the repository, using the command line to supply commit message
- `git commit -a` - Commit all tracked files into the repository that have changed, using text editor for commit message
- `git reset <path>` - Remove file from index or staging area
- `git status` - Show the status of tracked, changed, untracked files

Terminology

- **Branch** - A way of working on the same set of code in parallel without modifications overlapping
- **Commit** - A group of objects and a tree in a Git repository

Kapitel

Danksagungen

“Vielen herzlichen Dank!”

Korrektur und Ideen

Diese Menschen haben Zeit und Arbeit geopfert, um das Projekt zu forken, zu testen und grossartige Vorschläge zum Buch einzubringen:

Og Maciel
Alistair Buxton
Miia Ranta (Myrtti)
Hassan Williamson (HazRPG)
(Synth_sam)
Jonas Bushart

LaTeX Support

Diese Menschen gaben unschätzbaren LaTeX-Support:

Matthew Johnson
Ben Clifford

Git Support

#git auf freenode.net

GitHub.com für das hosten des GITT git-Repositorys

Wikipedia.org füe das Bereitstellen von Informationen über Versionskontrollsysteme