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

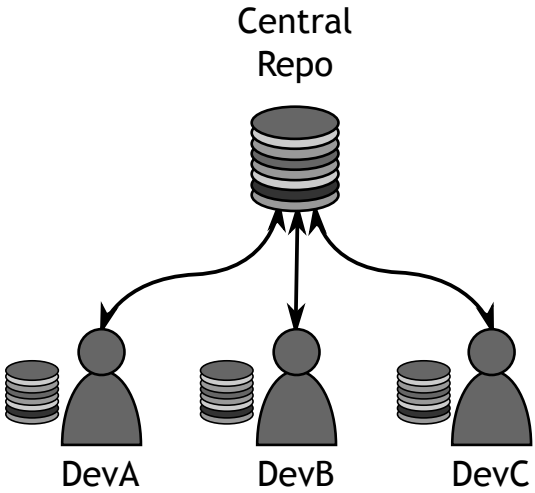

Abbildung .1: Centralised Workflow

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.

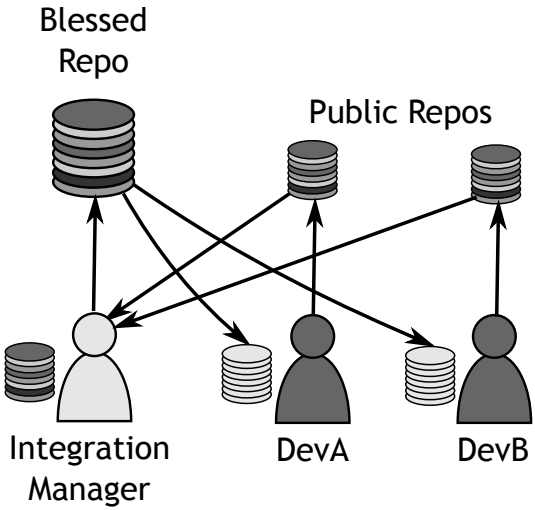

Abbildung .2: Integration Manager Workflow

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

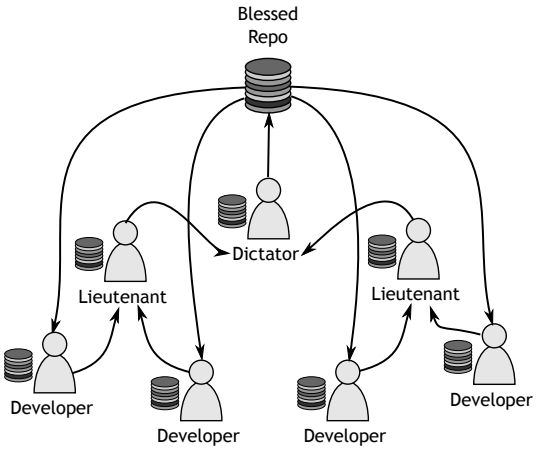

Abbildung .3: Dictator and Lieutenant Workflow

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.

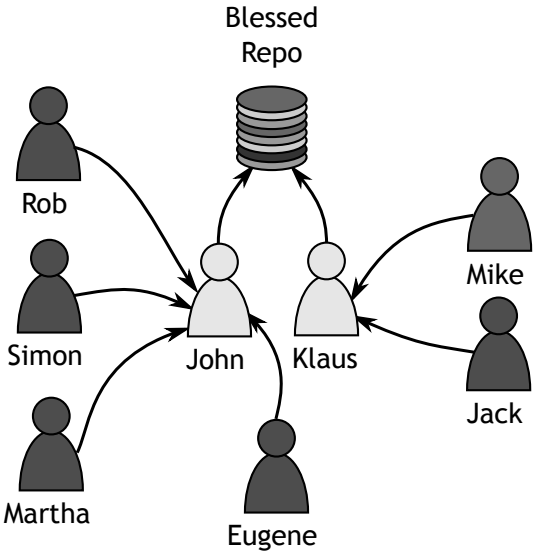

Abbildung .1: Tamagoyaki Inc's Physical Structure

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$
```

Was wir hier gemacht haben, ist folgendes: Wir haben ein neues Verzeichnis namens coderepo erstellt, in dieses Verzeichnis gewechselt und das Kommando git init aufgerufen. Das Resultat dieses Aufrufs ist ein neues Verzeichnis im coderepo-Ordner, das .git heißt. Dieses Verzeichnis beinhaltet eine lokale Kopie unseres kompletten Repositories. Es ermöglicht uns die Erstellung von Branches, das Mergen von Changes, Rebasing und letzten Endes auch das Pushen von unseren Changes auf einen Server.

Etwas, das sehr wichtig beim Betreiben eines Repositories ist, sowohl für Git-Administratoren als auch Entwickler die es benutzen, das Verständnis wie Git funktioniert. Es ist schön ins kalte Wasser zu springen und durch Experimente das Wasser zu testen, aber bevor man sich verpflichtet Git in einer produktiven Umgebung zu benutzen, sollte man verstehen, was Git im Hintergrund macht.

Während Ich dieses Buch geschrieben habe, haben mir verschiedene Leute erzählt, dass Git eines der wenigen VCS ist, wo ein gutes Verständnis des zugrunde liegenden Systems nicht nur hilfreich, sondern sogar beinahe nötig ist.

Nehmen wir uns ein paar Minuten Zeit um darüber zu reden, wie Git intern funktioniert und wie die Daten tatsächlich gespeichert werden. Git speichert Changes nicht in Dateien, sondern Snapshots zu bestimmten Zeitpunkten. Es bezieht sich auf diese, indem es einen SHA-1 Hash gegen sie vergleicht. Dadurch ist es für Git einfach zu erkennen, ob eine Datei verändert wurde. Wenn sich ein SHA-1 Hash einer Datei ändert hat sich der Inhalt der Datei offensichtlich verändert.

Wenn ein Commit im Repository gemacht wird, speichert Git einige Dinge: Ein Commit Object wird erstellt, welches Informationen darüber enthält, wer den Commit gemacht hat, den Ursprung des Commits und ein Tree Object. Das Tree Object beschreibt, wie das Repository zum Zeitpunkt des Commits ausgesehen hat. In anderen Worten: Der Tree Object teilt Git mit, welche Dateien im Repository enthalten waren. Zu guter Letzt speichert Git die Files die im Repository waren mit dem Namen ihres SHA-1 Hashes im objects-Ordner. Natürlich ist Git an dieser Stelle sehr clever, denn, wenn man die gleiche Datei in mehreren Commits ändert, verändert sich dennoch nicht der SHA-1 Hash und deswegen speichert Git nur eine Kopie der Datei; um Platz zu sparen.

Das Commit Object ist ebenfalls durch einen SHA-1 Hash gekennzeichnet. Hierbei unterscheidet sich Git zu vielen anderen VCS, die entweder eine Numer benutzen, oder eine Versionsnummer in der Datei selbst. Sich an 40-stellige SHA-1 Hashes zu gewöhnen kann etwas dauern. Zu sagen Ïch brauche den Commit mit dem Hash bf81617d6417d93380e06785f8ed23b247bea8f6dïst wahrscheinlich nicht so einfach wie schlichtweg zu sagen, dass man Version 6 benötigt. Trotzdem kann Git sehr gut mit den hashes umgehen und man kann sich auf einen bestimmten Commit beziehen, indem man ein paar der Zeichen vom Anfang des Hashes nennt, solange sich diese Zeichen eindeutig auf einen Commit beziehen.

Day 2 - “Erzeugen von Commits”

Lasst uns ein Repository erstellen!

Der einfachste Weg eine Datei in ein Repository zu committen ist, sie zu erstellen, oder sie in das Arbeitsverzeichnis zu kopieren und folgende Kommandos zu benutzen:

```
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$
```

Damit haben eine leere Datei erstellt und sie mit dem git-add Kommando dem Repository hinzugefügt. Danach haben wir es committet. Machen wir nun ein paar Changes in unserem Arbeitsverzeichnis und betrachten dann, was die Resultate sind. Zuerst fügen wir zwei weitere (neue) Dateien hinzu, dann werden wir unser ursprüngliches File verändern um schlussendlich git-status aufzurufen. Dort werden wir sehen, was Git zu unseren Changes zu sagen hat.

```
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$
```

Hier berichtet Git also, dass unser zuerst committetes File verändert worden ist und dass unsere zweite und dritte Datei **untracked** sind. Untracked Files sind Dateien, die Git im Arbeitsverzeichnis erkennt, die aber bisher nicht hinzugefügt worden sind. Daher würden sie bei einem Commit auch nicht dem Repository hinzugefügt werden. Beachte, dass bei einem Commit zum jetzigen Zeitpunkt nichts zum Repository committet werden würde. Obwohl my_first_committed_file verändert worden ist, haben wir Git noch nicht angewiesen diese Veränderungen hinzuzufügen. Machen wir nun also weiter und das nachholen, zur selben Zeit werden wir Veränderungen an my_second_committed_file vornehmen und diese auch hinzufügen.

```
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$
```

Nun können wir sehen, dass sich einer dieser Abschnitte geändert hat, er heißt nun "Changes to be committed". Das heißt, dass Git erkannt hat, dass wir diese Dateien beim nächsten git-commit committen wollen.

Committing the Uncommitted

In the trenches...

"John, was geht hier vor?" Rief Klaus quer durch den Korridor. Das ganze Büro hatte Klaus in den letzten 15 Minuten die Hände auf den Tisch schlagen hören. "John!" Sein Ruf wandelte sich in einen Schrei.

"Beruhige Dich Klaus, ich komme gerade erst." John ging hinüber zu Klaus und zog sich einen der Plastikstühle heran. Nach ein paar Minuten des herumtastens schaffte er es, eine Position neben dem wütenden Klaus einzunehmen.

"John, Git macht mich verrückt. Ich habe Dateien zum Repository hinzugefügt und bearbeite weiterhin den Commit, aber die Änderungen werden nicht ins verdammte Repo übertragen." Klaus war sichtlich verzweifelt und John widerstand der Versuchung Witze zu reißen.

John zeigte auf den Bildschirm. "Rufe git-status auf und ich zeige Dir wo das Problem liegt."

Um zu verstehen, was Klaus so verrückt macht, modifizieren wir nun my_second_committed_file und sehen uns an wie sich die Dinge nun verhalten. Zur Erinnerung: Wir haben die Datei zwar schon geadded, aber noch noch keinen Commit gemacht.

```
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$
```

Interessant! Wir haben jetzt drei Abschnitte und eine der Dateien taucht nun zwei Mal auf, ein Mal unter "Changes to be committedünd ein Mal unter "Changed but not updated". Was bedeutet das? Wenn Du Dich zurückerinnerst, haben wir über die Staging Area gesprochen. Das ist ein Bereich, in dem sich Git von vielen anderen Versionsverwaltungssystemen unterscheidet. Wenn Du eine Datei zum Repository **hinzufügst**, macht Git in Wirklichkeit eine Kopie der Datei und verschiebt sie in die Staging Area. Wenn Du sie danach veränderst, musst Du nochmal "git addäufrufen, um die abermals veränderte Datei in die Staging Area zu kopieren. Die wichtigste Tatsache, die man im Hinterkopf behalten sollte ist, dass Git nur das committet, was sich in der Staging Area befindet.

Wenn wir nun fortfahren und git-commit ausführen, werden nur die Datein unter "Changes to be commitedïn unserem Repository auftauchen.

```
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 unserem Beispiel haben wir die Syntax `git commit -m 'Message'` verwendet. Das ist ein etwas spezieller Weg des

Committens, der es uns erlaubt, unsere Commit-Message auf der Kommandozeile anzugeben. Wenn wir wollten, könnten `git commit` aufrufen, woraufhin sich ein Text-Editor öffnen würde, in dem wir unsere Nachricht eingeben könnten.

Beenden wir nun unseren Rundumschlag des Committens, indem wir `git commit -a` benutzen. Das bewirkt, dass alle Veränderungen an einer Datei committet werden, die bereits erfasst wurden. Schlüssigerweise müssen wir die Dateien nicht mehr extra mit `git add` angeben, wie wir es zuvor hätten tun müssen. Jede Datei, die geändert wurde und vorher schon zum Repository hinzugefügt wurde, wird mit diesem Kommando committet.

```
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 - "Lasst es uns richtig machen, nicht schnell"

Oh-oh, ich glaube ich habe einen Fehler gemacht

Nun sind wir also ziemlich gut mit dem hinzufügen von Files und dem Durchführen von Committs vertraut. In Kürze werden wir lernen, wie man die Änderungen seiner Changes betrachten kann und einen Diff gegen verschiedene Objekte erstellt. Doch bevor wir nach dieser Woche ins Wochenende gehen, müssen wir uns noch ein letztes Mal in die Schützengräben begeben...

In the trenches...

"Rob, hast Du mal eine Sekunde Zeit für mich?" fragte Mike.

"Klar, was ist los?" antwortete Rob vom anderen Ende des Büros. "Gib mir noch zwei Sekunden um diesen Commit fertigzustellen." Im Büro wurde es still, während Robs Finger über die Tastatur glitten. "Ah, verdammt!" rief Rob.

Mike stand auf und ging zu ihm hinüber. "Was ist los?"

"Ich habe gerade eine Datei zur Staging Area hinzugefügt, aber ich will sie dort gar nicht haben." Er schüttelte seinen Kopf, "Nunja, zumindest noch nicht."

> Mike kicherte, “Entschuldige die Unterbrechnung, Kumpel.”
>
> “Ach, ist schon ok. Ich muss nur wissen, wie ich diese Datei wieder aus dem Index lösche.”
>
> “Git reset,” rief eine Stimme. Die Stille des Büros wurde vom Rollen eines Stuhls unterbrochen. Der Besitzer des Stuhls war Klaus. Er schien stolz zu sein, dass er endlich mit Git klarkam. “Du kannst git reset benutzen um eine Datei im Index wiederherzustellen.” Er griff nach der Tastatur, “Hier lass es mich Dir zeigen.”

Git-reset ist großartig wenn es darum geht, Sachen aus dem Index zu entfernen. Natürlich kann es auch viele andere Dinge, aber fürs Erste beschäftigen wir uns mit dem oben beschriebenen Szenario. Wir arbeiten vor uns hin und haben eine Vielzahl an Dateien zum Index hinzugefügt, bis wir entdecken, dass wir noch nicht so weit sind um sie committen zu können. Im folgenden Beispiel fügen wir erst my_third_committed_file dem Index hinzu, um es danach wieder zu entfernen.

```
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$
```

Beachte, dass my_third_committed_file jetzt so weit ist, um committet zu werden. Das Problem ist, dass wir noch etwas mehr hinzufügen müssen, bevor wir das tun können. Erinnere Dich, wenn man git-add aufruft wird die Datei vom Arbeitsverzeichnis in den Index kopiert. Falls wir uns entscheiden, dass wir das

File nicht mehr im Repository brauchen, können wir folgendes aufrufen:

```
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$
```

Hiermit haben wir die im Index befindliche Datei gelöscht. Hierbei gibt es etwas zu beachten: Wir verschieben nicht die Datei vom Index zurück in unser Arbeitsverzeichnis, sondern wir löschen lediglich die Datei aus dem Index. Unser Arbeitsverzeichnis bleibt davon unberührt. Wir könnten auch das git-reset Kommando ohne eine Angabe der zu löschenden Dateien aufrufen, dann würden alle Files, die sich im Index befinden, aus diesem gelöscht werden.

In the trenches...

"Ich denke wir stimmen jetzt alle überein, Ich werde eine Version des Repositories mit Git verwalten, bis sich jeder mit den etwas fortgeschritteneren Funktionen von Git auskennt." John schaute sich im Raum um, ob es Einwände gab, aber es gab keine.

"Ok John, " sagte Markus, "Ich bin sehr zufrieden mit Euren Fortschritten, aber wie John bereits gesgt hat. es ist viel besser für uns, wenn wir uns noch etwas Zeit nehmen und alles richtig implementieren, als es sofort produktiv zu benutzen

> und dann aber nicht administrieren können und auch nicht wissen wie es funktioniert."
>
> "Ich möchte, dass Ihr alle Euch nächste Woche mit den Diffs und Logs auseinandersetzt. Und vergesst nicht, dass wir auch ein wichtiges Release haben." John schob seine Brille weiter hoch auf seine Nase. "Die Woche danach werden wir einen Blick auf Branches werden, danach werden wir wohl unser Modell implementieren können."
>
> Jeder nickte zustimmend.

Wir wissen nun, wie man ein File zu einem Repository hinzufügt. Die Frage ist aber, was zu tun ist, wenn man eine Datei löschen oder umbenennen will. Auch dafür gibt es bei Git Kommandos: `git rm` und `git mv` löschen beziehungsweise bennenen Dateien um. Das wäre also der Weg wie man normalerweise derartige Operationen durchführen würde, aber was ist, wenn man die betreffende Datei bereits manuell gelöscht hat? Dann hat man zwei Optionen: Man könnte `git commit -a` ausführen, aber das würde ja die Changes an allen bereits erfassten Dateien erneut committen. Eine weitere Möglichkeit wäre `git rm <filename>` mit dem Dateinamen angehängt aufzurufen, woraufhin Git diese Veränderung auch in der Staging Area durchführen wird. Das Selbe gilt auch für das umbennen von Dateien.

Dennoch ist es einen Blick wert, wie Git Umbenennungen handhabt: Git erfasst Umbenennungen nicht extra. Das bedeutet, dass der Aufruf von `git mv <source> <dest>` zuerst das normale Linux `mv`-Programm ausführt, gefolgt von `git rm` auf das ursprüngliche File und danch ein `git add` auf die neu benannte Datei. `git mv` ist nur ein kürzerer Weg das zu tun. Es ist eine gute Übung das auszuprobieren, um zu verstehen was wirklich passiert. Als Übung kannst Du das Repository nach

jedem Kommando inspizieren, um festzustellen, ab wann Git Deine Aktionen als Umbenennung registriert.

Wir haben nun ein paar essentielle Kommandos von Git kennengelernt. Wenn Du dich bereits mit Versionsverwaltungssystemen beschäftigt hast, ist der einzige Unterschied, den Du fesgestellt hast, vielleicht die Staging Area. Sie ist sehr mächtig und erlaubt es Dir, Deine Commits zu organisieren und vorzubereiten, so dass sie aussagekräftig und zusammenhängend sind.

Für Tamagoyaki Inc war der Plan der Implementation eines VCS zu aggressiv. Die meisten Teammitglieder haben noch nie ein Versionsverwaltungssystem benutzt. Aber wenn man sich entscheidet, ein VCS einzusetzen, ist es sehr wichtig sicherzustellen, dass man dies aus den richtigen Gründen tut. Versionskontrolle ist ein Programm um die Dinge im Blick zu behalten, aber Programme helfen nichts ohne einen definierten Prozess. Der Prozess ist der Schlüssel zur Ordnung.

Knowledge

Wie man den Editor für die Commit-Messages ändert

Wir haben vorhin schon kurz über die Konfigurations-Datei gesprochen und darüber, wie sie die Informationen über unsere Git-Instanz speichert. Git kann jeden Texteditor benutzen, den man sich wünschen kann, sogar einen grafischen, obwohl dafür kaum Bedarf besteht. Wir bereits erwähnt, hat Git mehrere Ebenen was die Konfigurations-Dateien betrifft. Zuerst wird die eigene Konfigurations-Datei im .git-Ordnerdes Repositories betrachtet, danach in der `~/.gitconfig`-Datei im Home-Verzeichnis des Users und zu guter letzt im globalen Ordner der verwendeten Distribution.

Wenn man den Editor zum Verfassen von Commit-Messages wechseln möchte kann man entweder diese Datein direkt verändern, oder ein Kommando wie folgendes benutzen:

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

Wenn wir Dinge global verändern wollen (also für alle Repositories des Users), würden wir folgendes Kommando ausführen:

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

Man kann auch die $EDITOR -Variable des Environments verändern. Allerdings würden dann auch andere Programme als nur Git den eingestellten Editor verwenden, was unter Umständen nicht erwünscht ist.

Zusammenfassung - John's Notizen

Kommandos

- `git add` - Fügt Dateien zum Index oder zur Staging Area hinzu
- `git commit` - Committet Dateien ins Repository, benutzt einen Text-Editor für die Commit-Message
- `git commit -m '<Message>'` - Committet ebenfalls ein File, die Commit-Message wird aber auf der Shell angegeben
- `git commit -a` - Committet alle erfassten Dateien ins Repository, benutzt einen Text-Editor für die Commit-Message
- `git reset <path>` - Entfernt Dateien vom Index oder von der Staging Area
- `git status` - Zeigt den Status von erfassten, veränderten und nicht erfassten Dateien an

Terminology

- **Branch** - Eine Möglichkeit gleichzeitig am selben Code zu arbeiten, ohne dass sich die Veränderungen überlappen
- **Commit** - Eine Gruppe von Objekten in einem 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