
Daily GIT.

Table of Contents

1. Über den Autor	2
2. Vorwort	3
3. Eine kleine Geschichte zur Einleitung	4
3.1. Eine Idee	4
3.2. Snapshots	5
3.3. Tags	6
3.4. Branches	8
3.5. Kollaboration	11
3.6. Merges	13
3.7. Rebase	14
3.8. No space left on device	15
3.9. Schlussendlich	16
4. Tag 1	17
4.1. Der Hintergrund	17
4.2. Git Basics	18
4.3. Daily Work	22
4.4. Internals	26
4.5. Branches	31
4.6. Remotes	54
4.7. Push	64
4.8. All is Lost - Reflog	72
4.9. Best Practices	73
4.10. Hooks	87
4.11. Daily Alias	92
4.12. Tag 1 endet	94
5. Tag 2	97
5.1. Git Branching Modelle	97
5.2. Git Flow	101
5.3. Tag 2 endet	137
6. Tag 3 - Die Migration nach Git	138
6.1. Eine erste Migration	142
6.2. Der bessere Weg	145
6.3. SVN / Git Ignores	152

6.4. Tag 3 endet	153
6.5. Links	155
7. Tag 4 - Verteile Repositories	155
7.1. Repository Splitting	157
7.2. Distributed Repositories	163
7.3. Tag 4 endet	182
8. Fazit	182
9. Appendix A	183
9.1. Branch Per Feature (BPF)	183

Das werden Sie täglich brauchen, nicht mehr, nicht weniger.

1. Über den Autor



Martin Dilger ist freiberuflicher Trainer und Consultant. Er betreut Firmen beim Einsatz und der Umstellung auf Git. Er bietet Schulungen für Entwickler, Operations und Entscheider mit dem Ziel, jeden Tag ein kleines bisschen produktiver zu werden.

Martin Dilger ist regelmäßiger Autor im Java Magazin und dem Java Spektrum und schreibt regelmäßig zu aktuellen Themen im [Effective Trainings Blog¹](#). Er ist außerdem Sprecher auf Konferenzen und besucht regelmäßig Firmen und hält Vorträge zu aktuellen Themen rund um Agilität, Webentwicklung, Enterprise Java und Softwarequalität.

In seinem "anderen" Leben lebt Martin Dilger mit seiner Familie in München, ist liebender Ehemann und Vater und passionierter Jogger.

¹ <http://www.effectivetrainings.de/blog>

2. Vorwort

Git ist die wahrscheinlich wichtigste Neuerung im Bereich Softwareentwicklung der letzten 10 Jahre (Stand 2013). Es gibt bereits unzählige Bücher, Tutorials, Videos und Schulungen zu diesem Thema. Warum also nochmal ein Buch?

Dieses Buch behandelt Schritt für Schritt alle Themen rund um Git, die Sie als Entwickler brauchen werden, um produktiv zu arbeiten. Es gibt drei Wege, wie sie anfangen können mit Git zu arbeiten.

Der effektivste aber leider wahrscheinlich auch der mit der steilsten Lernkurve - Sie fangen einfach an mit Git zu arbeiten. Dabei spielt es keine Rolle, ob in Ihrem Projekt derzeit CVS, Subversion oder vielleicht sogar schon Git im Einsatz ist. Git bietet für alle gängigen Systeme sogenannte Bridges, die die Zusammenarbeit zwischen Git und anderen Systemen ermöglichen.

Die zweite Möglichkeit ist, sie suchen sich jemanden, der Ihnen erklären kann, wie man richtig mit Git arbeitet. Haben Sie diese Möglichkeit sollten Sie sie unbedingt nutzen. Sie werden mit Sicherheit auf Fragen (und Fragezeichen) stossen, die Ihnen zu Beginn seltsam erscheinen werden. Glauben Sie mir, jeder Entwickler macht diese Erfahrung. Git ist nicht einfach, aber immer logisch. Es dauert allerdings eine ganze Weile, bis ein Git-Neuling diese Erkenntnis akzeptieren kann.

Die dritte und letzte Möglichkeit, Sie kaufen sich ein Buch oder lesen Online-Artikel. Hier haben Sie zunächst die Qual der Wahl, denn es gibt eine Fülle an sehr guten und leider auch an weniger guten Artikeln, Büchern und Tutorials.

Dieses Buch vereint alle drei Varianten. Das Buch beschreibt einen mehrtägigen Dialog zwischen einem sehr erfahrenen Git-Veteranen und einem neuen Entwickler im Team, der bisher noch gar nicht mit Git gearbeitet hat. Die beiden Entwickler besprechen alle wichtigen Themenbereiche rund um die Arbeit mit Git und dezentraler Versionskontrolle direkt am Rechner. Angefangen beim ersten Klonen eines Projektes, über die Arbeit mit Branches und Workflows bis hin zur Migration von bestehenden Subversion-Projekten. Wo immer sinnvoll sind im Buch Übungen hinterlegt, die das Verständnis noch vertiefen sollen.

Die Art und Weise wie das Buch geschrieben ist liegt mit Sicherheit nicht jedem, ich persönlich hätte mir genau dieses Buch zum Start gewünscht.

Ich freue mich, wenn Sie das Buch bewerten würden.

Martin Dilger, München im Dezember 2014

3. Eine kleine Geschichte zur Einleitung

Um mit Git zu arbeiten ist es wahrscheinlich wichtiger als bei jedem anderen System, dass man ein Verständnis dafür entwickelt, wie das System arbeitet und funktioniert. Aus meiner Erfahrung in unzähligen Trainings zum Thema Git weiß ich, dass sich viele Entwickler schwer tun, die Konzepte hinter Git zu verstehen. Anfangs scheint es einen eher zu erschlagen als zu helfen.

Das Schöne ist, Git ist vom Prinzip und von den Konzepten sehr einfach und kinderleicht zu verstehen. Glauben Sie nicht? Ich erzähle öfter eine Geschichte, die das Verständnis für die Konzepte hinter Git ungemein erleichtert.

Die Idee zu dieser Geschichte stammt (leider) nicht von mir, sondern basiert auf dem wundervollen Blog-Eintrag [The Git Parabel²](#) von Tom Preston Werner (dem GitHub Gründer). Ich habe die Geschichte nur ein wenig adaptiert, damit ich sie in meinen Trainings verwenden kann. Meiner Ansicht nach ist dies der beste und unterhaltsamste Weg, sich dem Thema Git zu nähern.

Ich kann Ihnen nur empfehlen, bevor sie mit dem eigentlichen Buch beginnen, lesen Sie entweder "The Git Parabel" oder die Geschichte von Lars hier in diesem Buch. Es wird Ihnen sehr helfen im weiteren Verlauf des Buches.

3.1. Eine Idee

Lars ist motivierter junger Mann, mitte 30, lebt in Nürnberg und war bisher angestellt als Software-Entwickler bei einem mittelständischen Softwareunternehmen. Lars hat nur ein großes Problem - *er ist extrem ehrgeizig*. Die Arbeit als *einfacher* Entwickler füllt ihn nicht mehr aus, er möchte mehr und entwickelt seit einigen Wochen insgeheim an seiner großen Geschäftsidee. Diese Idee hat Potential und Lars sieht sich schon reich, berühmt und als Revolutionär in einer kleinen eCommerce-Nische, die er für sich entdeckt hat. Heute ist ein neuer Tag, es ist Montag früh, Lars sitzt mit einem Kaffee an seinem Küchentisch und ist bereit mit der Arbeit an seinem neuen Produkt zu beginnen.

Die Umsetzung der Features geht flott voran und nach einigen Stunden hat Lars bereits das grundsätzliche Setup des Projektes fertiggestellt. Er beginnt mit der Umsetzung der ersten Features. Während die Codezeilen fast wie von selbst auf dem Bildschirm landen fällt Lars auf, dass er eine wichtige Sache noch nicht

² <http://tom.preston-werner.com/2009/05/19/the-git-parable.html>

bedacht hat. Er arbeitet bisher ohne Versionskontrollsysteem. Kein professioneller Softwareentwickler würde jemals ohne Versionskontrolle arbeiten. Aber die Systeme, mit denen Lars bisher gearbeitet hat reizen ihn nicht. Aus seiner täglichen Arbeit kennt er sich bestens mit *Subversion* und *CVS* aus, aber beide Systeme scheinen ihm für sein Projekt ungeeignet und zu aufwendig im Setup. Sowohl Subversion als auch CVS sind sogenannte zentralisierte Versionskontrollsysteeme. Für diese Systeme wird üblicherweise ein Server benötigt, der aufgesetzt, konfiguriert und gewartet werden muss. Da Lars momentan allein arbeitet scheint ihm dieser Ansatz zu kompliziert. Hinzu kommt leider, dass Lars keine Zeit hat sich in mögliche andere Systeme einzuarbeiten. Er braucht etwas einfaches, schnelles und vor allem, ein System das er versteht.

3.2. Snapshots

Was Lars eigentlich haben möchte sind einfache Snapshots des Projektes zu bestimmten Zeitpunkten. Beispielsweise immer dann, wenn er ein neues Feature abgeschlossen oder er einen wichtigen Meilenstein erreicht hat möchte er einen Snapshots erstellen und diesen Stand somit einfrieren. Lars überlegt, was der einfachste Weg ist, diese Snapshots in seinem Projekt zu erstellen. Für den Anfang reicht es Lars, wenn er einfach seinen kompletten Workspace mit allen Dateien die sich aktuell darin befinden in ein eigenes Verzeichnis ausserhalb des Workspaces sichert. Er erzeugt sich also einen Ordner *Snapshots*, und in diesem Ordner einen ersten Ordner *Snapshot-1*, in den er den kompletten Inhalt seines Workspaces kopiert, nachdem das initiale Setup abgeschlossen ist. Das einfachste System der Welt aber es erfüllt seinen Zweck.

Zusätzlich legt Lars in jedem Snapshot-Verzeichnis eine kleine Datei *message.txt* ab. In dieser Datei hinterlegt er das Datum an dem der Snapshot erstellt wurde sowie eine kleine Nachricht, was sich mit diesem Snapshot geändert hat. Auf diese Art und Weise kann Lars sehr einfach ein Changelog seines Projektes erstellen, indem er über die Snapshots iteriert und die *message.txt* ausliest. Zusätzlich kann er sich einen bestimmten Snapshot zurück in den Workspace holen, indem er ihn einfach direkt wieder dorthin zurück kopiert.

16.10.2014

Initial Project Setup

Lars kommt gut voran, denn er arbeitet wie besessen, schläft kaum noch, merkt keinen Unterschied zwischen Tag und Nacht und vergisst jeglichen Tagesrhythmus. Und so ist er nach wenigen Wochen bereits bei Snapshot-100 angelangt. Der Snapshot-100 ist

die erste Version, die Lars als *Feature-Complete* betrachten würde. *Feature-Complete* zumindest für eine erste Version - es könnte bereits Kunden geben, die durchaus bereit sein könnten hierfür Geld auszugeben.

Er überlegt sich, eine Version 0.9 zu releases. Damit wäre den Kunden klar, dass das System noch nicht perfekt ist. Er telefoniert kurzerhand mit einigen seiner potentiellen Kunden und tatsächlich sind einige dabei, die Interesse an der Lösung haben und bereit sind, hierfür Geld auszugeben.

3.3. Tags

Lars möchte diesen besonderen Snapshot jetzt irgendwie markieren, denn er möchte sich nicht im Kopf merken, welche Snapshots jetzt als Releases an Kunden ausgespielt wurden. Für einen, evtl. sogar zwei Snapshots würde das noch funktionieren, definitiv aber nicht für mehr. Wieder denkt Lars nach und überlegt sich, was die einfachste Möglichkeit wäre, einen Snapshot als Release zu markieren. Die Systeme mit denen Lars bisher gearbeitet hat kennen das Konzept der **Tags**. Tags sind unveränderliche Marken in der Historie, die wichtige Meilensteine markieren. Genau dieses Konzept möchte Lars auch in seinem eigenen System etablieren. Er erzeugt hierfür einfach einen neuen Ordner Tags, ebenfalls wieder ausserhalb des Workspaces. In diesem Verzeichnis *Tags* erstellt er eine neue Datei *Release-0.9.txt*. In dieser Datei steht nichts anderes als die Referenz auf den Snapshot, der das entsprechende Release markiert.

Snapshot-100

Um jetzt zu überprüfen, welche Snapshots bereits an Kunden ausgeliefert wurden muss Lars lediglich im *Tags*-Ordner die Dateien *Release-** öffnen und sieht sofort die richtige Referenz. Um zu erfahren, welche Version die neueste ist und damit welche Version aktuell beim Kunden liegt öffnet er einfach die *Release-**-Datei mit dem höchsten Rang und prüft, welche Referenz dort abgelegt ist. Lars legt außerdem fest, dass Tags, die einmal erstellt wurden genauso wie Snapshots nicht veränderbar sind. Der *Release_0.9*-Tag ist also für alle Zeit mit dem Snapshot-100 verbunden.

Lars ist zufrieden und arbeitet schnell weiter. Lars hat viele neue Ideen für Features und schon am Abend ist er bei **Snapshot 103** angelangt.

Am darauffolgenden Tag klingelt plötzlich sein Telefon. Diesen Klingelton kennt Lars nicht. Als er abhebt hat er einen aufgeregten Kunden am Apparat, der sich über ein Problem in der Plattform beschwert. Lars hat seinen ersten Bug-Report. Das Problem ist trivial und Lars sieht sofort, wie es zu beheben ist. Er möchte schon direkt mit dem

Bugfix beginnen und stellt plötzlich fest, dass sein Versionskontrollsyste eine kleine Lücke aufweist.

Er ist aktuell bei *Snapshot 103*, und hat bereits mit der Implementierung eines großen neuen Features begonnen. Dieses Feature ist noch nicht abgeschlossen und kann keinesfalls bereits an seinen Kunden ausgeliefert werden. Es wäre also sinnvoll, den Bugfix auf Basis des letzten Releases, also *Snapshot-100* zu machen.

Wie aber lautet die Versionsnummer dieses Bugfix-Releases? Eine einfache Variante wäre **Snapshot-104**. Bisher war es einfach zu erkennen, was der aktuellste verfügbare Snapshot war. Lars musste hiefür einfach den mit der höchsten Version wählen. Das ist jetzt leider nicht mehr so einfach möglich, denn der neue *Snapshot 104* hat zwar eine höhere Version, ist aber von den implementierten Features schon weit hinter dem *Snapshot 103* zurück. Die Versionen sind jetzt zwar immer noch eindeutig, aber sagen nichts mehr darüber aus, was der Snapshot tatsächlich beinhaltet.

Da der Bugfix schnell an seine Kunden ausgeliefert werden muss bleibt es zunächst bei *Snapshot 104* und Lars erstellt ein neues Bugfix-Release. Natürlich erstellt er im gleichen Moment auch ein weiteres *Tag*. Er legt also im Verzeichnis *Tags* eine weitere Datei *Release-0.9.1.txt* an.

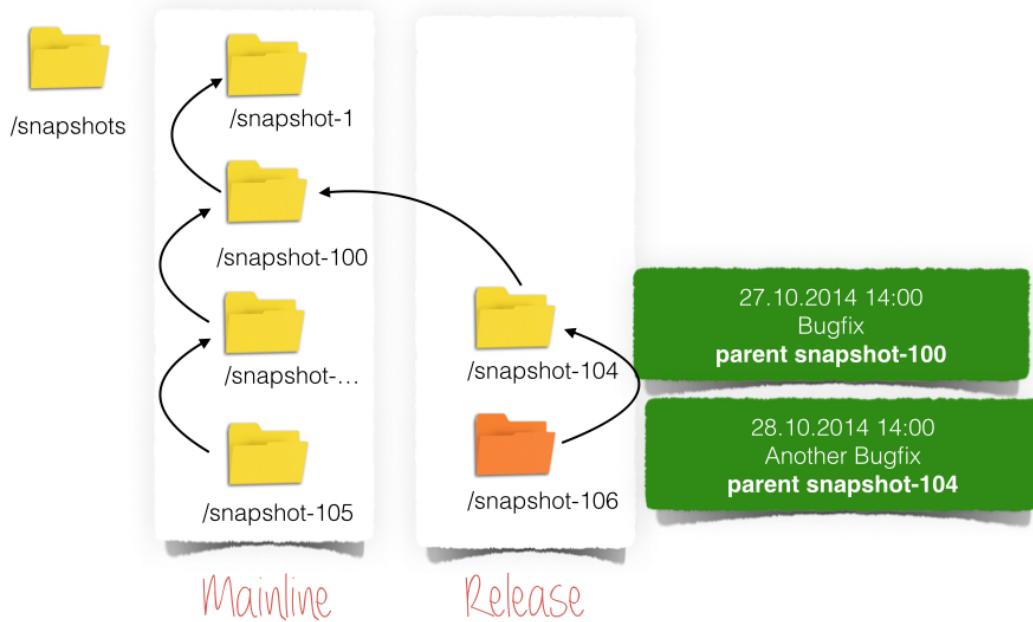
[Snapshot-104](#)

Nachdem das Release ausgespielt ist atmet Lars zunächst einmal tief durch. Das erste Problem ist gelöst. Wie aber soll er jetzt das Problem mit den Snapshots lösen? Er sitzt an seinem Küchentisch und sinniert über die Problematik und es will ihm keine richtig gute Lösung einfallen. Sein Blick schweift aus dem Fenster. Draussen geht ein leichter Herbstwind und er sieht den Apfelbaum in seinem Garten leicht im Wind hin und herwiegen. Seine Gedanken schweifen ab und er überlegt sich unbewusst, dass ein Baum tatsächlich eine sehr interessante Konstruktion der Natur ist. Egal wie weit man einem Ast folgt, egal wieviele Verästelungen man passiert - eins ist immer ganz einfach - den Weg zurück zum Stamm zu finden. Und plötzlich fällt der Groschen und er schlägt sich mit der flachen Hand auf die Stirn. Das grundsätzliche Problem seines Systems ist, dass er Versionen linear verwaltet. Was er aber stattdessen braucht ist eine Baumstruktur.

Der *Snapshot 104* war der erste Snapshot, der sich vom Stamm entfernt hat, wie ein erster kleiner Ast, der plötzlich gewachsen ist. Was sein System aber im Gegensatz zum Baum im Garten nicht bietet ist von diesem Ast aus einen einfachen Weg zurück zum Stamm zu finden.

3.4. Branches

Jetzt da Lars das eigentliche Problem identifiziert hat ist eine Lösung schnell gefunden. Für jeden Snapshot merkt sich Lars in der *message.txt* zusätzlich, welcher Snapshot der unmittelbare Vorgänger war. Für den *Snapshot 104* ist dies der *Snapshot 100*, für den *Snapshot 103* ist dies der *Snapshot 102*.



Ohne es zu wissen hat Lars eine extrem einfache und effiziente Art und Weise gefunden, parallele Entwicklung auf Zweigen zu realisieren. Der *Snapshot 104* ist genau betrachtet nichts anderes als der oberste Commit in seinem Release-Branch, auch wenn es diesen Branch offiziell bisher gar nicht gibt. Der *Snapshot 103* ist der oberste Commit in seiner Mainline. Lars kann jetzt ohne Probleme auf Basis des *Snapshot 104* einen weiteren Bugfix-Release erstellen, indem er den den *Snapshot 104* einfach in seinen Workspace kopiert, die notwendigen Änderungen vornimmt und einen weiteren Snapshot erstellt, der als Parent den bereits existierenden *Snapshot 104* referenziert. Seine *Mainline*, also der *Snapshot 103* ist hiervon überhaupt nicht betroffen. Lars kann auf Basis des *Snapshot 103* problemlos weitere neue Features implementieren, ohne dass sein Release-Branch davon betroffen ist. Änderungen eines Snapshots können maximal diejenigen Snapshots betreffen, die über eine

Parent-Referenz erreichbar sind. Der *Snapshot 105* ist von *Snapshot 104* über keinen Weg erreichbar, genauso umgekehrt.

Das hantieren mit den Versionsnummern ist aber unbequem, denn Lars muss sich merken, dass seine Mainline derzeit auf *Snapshot 105* steht. Diese Information existiert nur in Lars Kopf. Lars findet aber grundsätzlich die Idee auf Branches zu arbeiten sehr attraktiv. Sollten es aber mehr Branches werden kann sich Lars unmöglich für alle Branches merken, auf welchen Snapshots diese aktuell stehen. Er behilft sich mit einem sehr einfachen Workaround. Er erstellt wieder ausserhalb seines Workspaces ein neues Verzeichnis **/branches** und in diesem Verzeichnis die beiden Dateien **master.txt** und **release.txt**. In diesen beiden Dateien befindet sich nichts weiter als eine Referenz auf den jeweiligen Snapshot auf dem der Branch aktuell steht.

In der Datei **master.txt** steht also nichts weiter als:

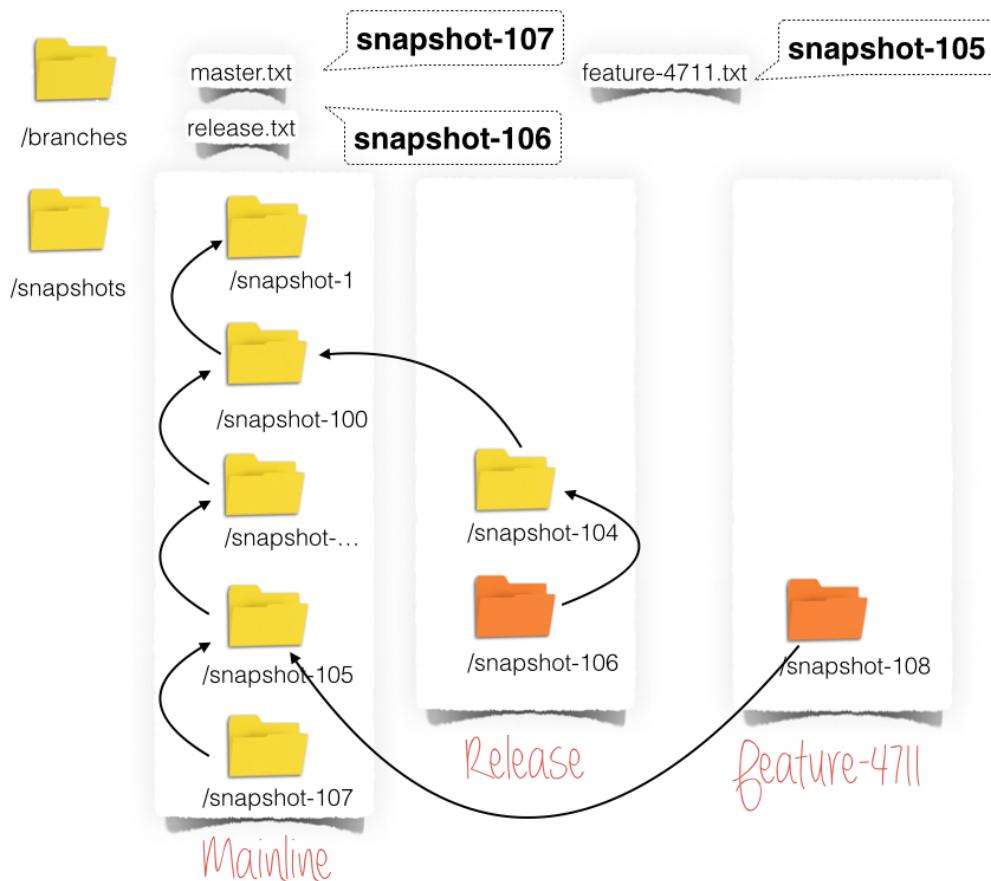
Snapshot-105

Analog steht in der Datei **release.txt**:

Snapshot-104

Um auf Basis des letzten Releases also einfach einen neuen Bugfix-Release zu erstellen muss Lars lediglich in der Datei **release.txt** die Version des letzten Snapshot auslesen, diesen in den Workspace kopieren und den Bug fixen. Lars darf nur keinesfalls vergessen nach jedem Erstellen eines neuen Snapshot die jeweilige Referenz in der **master** bzw. der **release.txt** Datei zu aktualisieren und zwar auf den gerade eben neu angelegten Snapshot.

Lars ist wirklich erstaunt, wie einfach und gut dieses Prinzip der Branches funktioniert. Er wundert sich, dass ausser ihm scheinbar bisher noch niemand auf diese Idee gekommen ist.



Um beispielsweise einen neuen Branch auf Basis des *Snapshot 105* zu erstellen erzeugt Lars lediglich eine neue Datei *feature-4711.txt* im Ordner */branches*. In diese Datei schreibt er *Snapshot 105*. Damit ist klar, dass ein Branch mit dem Namen *feature-4711* existiert, der aktuell auf dem *Snapshot 105* steht. Wechselt er jetzt auf diesen Branch, indem er den Snapshot aus *feature-4711.txt* in seinem Workspace kopiert kann er auf dieser Basis den neuen *Snapshot 108* erstellen und die Änderung betrifft zunächst nur den Branch und hat keinerlei Einfluss auf die Mainlinen oder den Release-Branch.

Lars hat die ganze Nacht hindurch gearbeitet und quält sich morgens völlig übermüdet aus dem Bett. Er setzt sich sofort wieder an seinen Rechner und möchte weiterarbeiten. Er kann sich aber nicht mehr erinnern, was er gestern Nacht überhaupt implementieren wollte. Er muss zunächst einmal wieder scharf nachdenken um überhaupt zu wissen, welcher Snapshot und welcher Branch da aktuell in seinem Workspace sind. Das ärgert Lars so sehr, dass er sich hierfür eine Lösung überlegt. Er müsste sich einfach irgendwo merken, was denn aktuell in seinem Workspace für eine Version liegt. Er erzeugt sich ausserhalb des Workspaces eine neue Datei *HEAD.txt*. Wann immer Lars einen Branch und somit einen bestimmten Snapshot in seinen Workspace holt, dann schreibt der den Namen des ausgecheckten Branches in diese Datei. So ist immer

klar, welche Version aktuell im Workspace liegt. Holt sich Lars also beispielsweise den *Feature-4711*-Branch in seinen Workspace, schreibt er gleichzeitig *feature-4711* in die Datei HEAD. So kann er auch im völlig übernächtigten Zustand schnell herausfinden, woran er eigentlich gerade arbeitet. Wechselt er jetzt vom *Feature-4711*-Branch auf den *Release*-Branch muss er nur sicherstellen, dass in der Datei *HEAD* anschließend *release* und nicht mehr *feature-4711* steht.

Manchmal ist es auch notwendig, dass sich Lars einen Snapshot holt, der nicht auf einem aktuellen Branch liegt. Nehmen wir an, der *master*-Branch steht aktuell auf dem *Snapshot 109*. Lars wundert sich über eine bestimmte Stelle im Code und er möchte sich gerne anschauen, wie der Stand im Source-Code vor dem *Snapshot 109* war. Er kopiert sich also den *Snapshot 108* in seinen Workspace. Der Workspace steht also aktuell nicht auf einem Branch, sondern lediglich auf einer bestimmten Snapshot-Version. Das passt noch nicht so ganz ins Konzept, denn bisher geht er davon aus, dass immer ein Branch ausgecheckt ist. In diesem Fall wählt er den einfachsten Weg und schreibt in die Datei HEAD nicht den Namen des Branches sondern die tatsächliche Snapshot-Version. Lars nennt das einen *Detached HEAD*, denn der ausgecheckte Snapshot ist nicht mit einem Branch verbunden.

3.5. Kollaboration

Einige Tage später stellt Lars fest, dass er den Aufwand zur Entwicklung der Plattform unterschätzt hat, denn er kommt nicht so schnell voran wie seine Kunden das erwarten. Lars entschliesst sich, Hilfe zu holen. Er kennt Markus, einen sehr guten Entwickler in München, mit dem er bereits in mehreren Projekten zusammengearbeitet hat. Nach einem kurzen Telefonat ist Markus von der Idee überzeugt und ist bereit mit einzusteigen. Lars erklärt Markus kurz am Telefon das Konzept seiner selbstgestrickten Versionsverwaltung. Markus ist zunächst etwas skeptisch, dennoch willigt er ein, das System zu benutzen. Lars schickt Markus also seinen Workspace, alle Snapshots und Tags in einem Zip-Archiv per Mail.

Die beiden Entwickler einigen sich kurz auf Arbeitspakete, die es zu erledigen gilt und arbeiten dann aufgrund der geografischen Trennung unabhängig voneinander. Sie vereinbaren, jeden Tag abends kurz zu telefonieren und die jeweils neuesten Stände abzulegen. Sowohl Lars als auch Markus haben einen sehr produktiven Tag und jeder erzählt dem anderen abends stolz, was erreicht wurde. Es stellt sich heraus, dass sowohl Lars als auch Markus jeweils drei neue Features umgesetzt haben. Lars ist mittlerweil bei **Snapshot 109** angelangt. **Markus auch.**

Jetzt erst fällt Lars auf, dass die linearen und aufsteigenden Versionen, die er aus *Subversion* und *CVS* kennt zwar für klassische Systeme mit einem Server in der Mitte funktionieren, nicht jedoch für das dezentrale System, das sich Lars ausgedacht hat. Sowohl Lars als auch Markus arbeiten lokal und ohne Verbindung zum jeweils anderen. Eine Synchronisation der Stände findet nur abends statt, nachdem die beiden telefoniert haben.

Auf diese Art und Weise aber sind ständige Versionskonflikte vorprogrammiert. Lars denkt nach und versucht eine Lösung für dieses Problem zu finden. Auf der Suche geht er nocheinmal die Liste an bisher erstellten Snapshots durch und versucht herauszufinden, was einen Snapshot eindeutig identifizieren und somit als Version dienen könnte. Lars wird schnell fündig, denn was einen Snapshot eindeutig identifiziert ist die *message.txt*, die für jeden Snapshot angelegt wird.

27.10.2014 17:13

Markus (markus@IT-Rockstar.de)
Customer Service IF
parent Snapshot-105

Seit Markus mit an Bord ist haben sich die beiden Entwickler zusätzlich darauf geeinigt, dass sowohl der Name und die E-Mailadresse zusätzlich in die *message.txt* mit aufgenommen wird. So ist sofort ersichtlich, wer einen Snapshot wann, auf welcher Basis und mit welchen Änderungen erstellt hat. Wie aber kann Lars auf Basis dieser *message.txt* jetzt eine Version erstellen? Die erste Idee die Lars einfällt ist mit einer Hash-Funktion zu arbeiten. Eine gute Hash-Funktion garantiert, dass die generierten Werte eindeutig sind und keine Konflikte auftreten können. Die Wahrscheinlichkeit, dass ein Snapshot mit einer identischen *message.txt* vom gleichen Autor, zur gleichen Zeit aber mit unterschiedlichen Änderungen gespeichert wird kann getrost als unwahrscheinlich betrachtet werden.

Lars verwendet den SHA-1 Hash-Algorithmus und berechnet aus einer Datei *message.txt* beispielsweise den Hashwert 1a1a98e5b306ee1804c2106f919aa09e84d99356. Dieser Hashwert ist zwar schwerer zu lesen als ein Snapshot-102, ist aber genauso eindeutig und deshalb als Version ganz brauchbar. Lars startet also einen Konvertierungsprozess und konvertiert alle Versionen in die entsprechenden Hashwerte. Hierbei konvertiert er alle Snapshots im /*snapshots*-Verzeichnis, alle Referenzen in den Dateien im Ordner /*branches* und auch alle Referenzen im Ordner /*tags*.

Eine typische *message.txt* sieht also nach der Konvertierung so aus.

27.10.2014 17:13

Markus (markus@IT-Rockstar.de)
Customer Service IF
parent 1d22431e22ed6234ecd079f94d5277005f22b219

Die Parent-Referenz ist jetzt ebenfalls einfach nur ein Hashwert des jeweiligen Parent-Snapshots. Weder Lars noch Markus stört es aber besonders, dass die Versionen jetzt schwerer zu lesen sind als zuvor, denn üblicherweise arbeiten sie nur noch mit den Branch-Namen wie **master** und nicht mit dem entsprechenden Hashwert.

3.6. Merges

Damit ist das Problem der Versionskonflikte ein für alle mal gelöst. Sowohl Lars als auch Markus können fortan völlig problemlos und unabhängig voneinander arbeiten. Wenn sie sich abends synchronisieren schickt Markus alle neuen Snapshots an Lars, der sich die Änderungen zunächst in einen Workspace kopiert und ein Code-Review vornimmt und anschließend die Änderungen von Markus in den **master** übernimmt. Wie aber kann Lars die Änderungen von Markus übernehmen? Nach kurzen Nachdenken scheint die Lösung auf der Hand zu liegen. Es gibt bereits fertige diff-Algorithmen, die mit fast jedem Betriebssystem ausgeliefert werden. Lars untersucht einfach, welche Dateien sich bei Markus im Vergleich zu seinem letzten Snapshot geändert haben. Anschließend erstellt Lars einen neuen Snapshot auf Basis seines letzten Snapshots und übernimmt in diesen zusätzlich alle Änderungen von Markus. Für die meisten Dateien ist das kein Problem, da sie entweder von Lars oder aber Markus bearbeitet wurden, nicht jedoch von beiden. Beide haben jedoch Änderungen an der **pom.xml** des Projektes vorgenommen, da für beide neuen Features neue Dependencies notwendig waren. Für die **pom.xml** übernimmt Lars sowohl seine als auch die Änderungen von Markus und stellt anschließend sicher, dass das Projekt immer noch baut. Lars fällt bei dieser Gelegenheit auf, dass der neue Snapshot nicht nur einen, sondern sogar zwei Parents hat, da sowohl Lars letzter Commit als auch Karls letzter Commit als Basis dieses Snapshots dienen.

28.10.2014 17:13

Lars (lars@IT-Rockstar.de)
Merged Karls changes
parent 6f66e0ce3f8c0b98f0b17ec5a5b1d7b509fe57c9
parent 1d22431e22ed6234ecd079f94d5277005f22b219

3.7. Rebase

Die beiden kommen sehr gut voran und das Produkt entwickelt sich rasend schnell weiter. Es wurden schon weitere Releases an die Kunden gegeben und alle sind sehr zufrieden mit dem Portal. Markus arbeitet derzeit an einem sehr großen neuen Feature, das bisher kein Konkurrent anbietet. Markus hat hierfür auf einem eigenen Branch bereits drei Snapshots erstellt. Nach wie vor telefonieren Lars und Markus jeden abend um sich zu synchronisieren. Beim heutigen Telefonat stellt sich heraus, dass Lars ein wichtiges Refactoring vorgenommen hat, das auch Markus gerne auf seinem Branch haben würde. Markus denkt sich sogar, es wäre wünschenswert gewesen, wenn er seinen Branch direkt erst heute auf dem letzten Snapshot von Lars erstellt hätte und nicht schon gestern. Der große Vorteil wäre, dass die Änderungen von Lars schon ab dem ersten Snapshot für Markus verfügbar gewesen wären. Leider geht das nicht - oder etwa doch? Markus spricht mit Lars darüber und Lars fängt sofort wieder an nachzudenken.

Was wäre, wenn Markus einfach den letzten Snapshot von Lars als Grundlage für seine beiden Snapshots verwendet. Markus findet die Idee gut. Lars schickt Markus seine letzten Snapshots und Markus überlegt, wie er seine bereits erzeugten Snapshots so umschreiben kann, dass sie auf den letzten Snapshots von Lars basieren. Markus hat zwei Snapshots erstellt, die aufeinander basieren. Er nimmt sich also den ersten Snapshot und macht zunächst ein Diff, um festzustellen, wie stark sich sein Snapshot von dem von Lars unterscheidet. Markus kopiert den Snapshot von Lars in seinen Workspace und übernimmt zusätzlich alle Änderungen aus seinem Snapshot ebenfalls in den Workspace. Markus gefällt das sehr, denn jetzt sieht es so aus, als hätte er die Änderungen direkt auf der Basis des Refactorings von Lars gemacht. Markus erstellt aus dem neuen Workspace jetzt einen komplett neuen Snapshot. Dieser neue Snapshot hat natürlich einen komplett anderen Hashwert als sein voriger Snapshot, denn der Parent-Zeiger in der *message.txt* hat sich geändert, und schon die kleinste Änderung an der Datei hat zur Folge dass ein komplett neuer Hashwert entsteht. Anschließend übernimmt Karl die Änderungen aus seinem zweiten Snapshot ebenfalls in den Workspace und erstellt auf dieser Basis einen zweiten, neuen Snapshot, der ebenfalls wieder einen neuen Hashwert erhält.

Seine beiden alten Snapshots kann Markus jetzt getrost löschen, denn sie werden nicht mehr gebraucht, da alle Änderungen in den beiden neuen Snapshots bereits vorhanden sind. Markus schickt die beiden neuen Snapshots an Lars, der die Änderungen anschließend in den **master**-Branch übernimmt.

3.8. No space left on device

Einige Wochen später stellt Markus fest, dass durch die ganzen erzeugten Snapshots und die enormen Datenmengen seine Festplatte langsam an ihre Grenzen stösst. Auch wenn sich pro Feature und Snapshot nur eine einzige Datei ändert wird aktuell für jeden Snapshot jede Datei dupliziert und in einem neuen Snapshot gespeichert. Die gespeicherten Daten sind unglaublich redundant. Als Markus abends mit Lars telefoniert spricht er dieses Problem an und die beiden überlegen, wie sie die Speichernutzung effizienter gestalten können. Sie könnten natürlich im ersten Schritt anfangen, und alle Snapshots komprimieren. Das würde die Symptome lindern, aber nicht das Problem lösen. Früher oder später werden die beiden wieder an die Grenzen der Platte stossen, da jeder Snapshot nach wie vor eine große Menge Speicherplatz verbraucht.

Lars verspricht Markus darüber nachzudenken. Er sitzt noch lange an seinem Küchentisch und grübelt. Draussen ist es längst dunkel und der Mond steht schon hoch am Himmel. Lars hämmert seinen Kopf gegen die Tischplatte, denn er kommt einfach nicht auf eine saubere Lösung für dieses Problem.

Er geht noch einmal die Fakten durch. Pro Snapshot ändern sich durchschnittlich nur ganz wenige Dateien, die meisten Dateien bleiben identisch. Was wäre, wenn pro Snapshot nicht die Dateien selbst, sondern Referenzen auf Dateien gespeichert werden würden? Lars findet die Idee gut und macht ein kleines Experiment.

Zunächst erzeugt er einen neuen Ordner */objects*.

Anschließend berechnet er für jede Datei in *Snapshot-1* den SHA-1-Hashwert. Er komprimiert diese Datei mit einem passenden Algorithmus und kopiert sie in das neue *objects*-Verzeichnis. Im letzten Schritt benennt er die Datei noch um in den Hashwert, den er zuvor berechnet hat. Im nächsten Schritt erstellt Lars für jedes Verzeichnis im Projekt eine temporäre Textdatei, in die er für jede Datei in diesem Verzeichnis einen Eintrag der folgenden Art vornimmt.

```
blob 041cb757d223fdc2a9108af9b44034d33b063d79 pom.xml  
tree 28d2f9e028c26ddc195e3c38281b1627f0509cd5 /src
```

Diese Datei ist nichts anderes als eine Auflistung aller Inhalte eines Verzeichnisses mit den jeweiligen berechneten Hashwerten. Für das obige Beispiel also speichert Lars, dass im Root-Verzeichnis eine Datei pom.xml mit dem Hashwert 041cb757d223fdc2a9108af9b44034d33b063d79 liegt, als auch

ein weiteres Unterverzeichnis, was er als *tree* markiert mit dem Hashwert 28d2f9e028c26ddc195e3c38281b1627f0509cd5 und dem Namen *src*. Anschließend berechnet Lars noch den Hashwert über das komplette Verzeichnis und speichert diese Datei unter diesem Hashwert ebenfalls im *objects*-Verzeichnis.

Eine ähnliche Datei erstellt er für das hier referenzierte Unterverzeichnis *src* und speichert auch diese Datei unter ihrem Hashwert im *objects*-Verzeichnis. Der ursprünglichen Snapshot-Verzeichnis legt Lars nicht mehr die rohen Daten ab, sondern nur noch die Referenz auf das Root-Verzeichnislisting. Von diesem Listing aus kann Lars alle Dateien für diesen Snapshot anhand der hinterlegten Hashwerte aus dem *objects*-Verzeichnis laden.

Nachdem der erste Snapshot konvertiert ist betrachtet Lars sein Werk. Er ist sich noch nicht ganz sicher, ob er viel gewonnen hat.

Er betrachtet den zweiten Snapshot und er führt nochmals diesselbe Prozedur durch. Er berechnet die Hashwerte der vorhandenen Dateien und stellt für die meisten Dateien fest, dass die gezippten Dateien mit den Hashwerten bereits im *objects*-Verzeichnis vorhanden waren. Nur die Dateien, die sich tatsächlich mit diesem Snapshot geändert hatten haben auch neue Hashwerte erhalten. Alle anderen Dateien muss Lars nicht doppelt speichern sondern kann sie einfach auch im zweiten Snapshot referenzieren.

Diese Prozedur führt er für alle Snapshots durch mit dem Ergebnis, dass nur noch ein minimaler Bruchteil des ursprünglichen Speicherplatzes verbraucht wird.

3.9. Schlussendlich

Durch die Änderung ist das System sowohl performanter als auch einfacher zu handhaben. Lars und Markus sind mittlerweile sehr zufrieden mit dem Funktionsumfang. Das Projekt wird ein voller Erfolg. Eines Tages surft Lars durch Zufall im Internet und stösst auf Git, ein dezentrales Versionskontrollsystem, dessen Konzepte ihm sehr bekannt vorkommen. Zufälligerweise ist Lars durch logisches Kombinieren genau auf die gleichen Konzepte gekommen, die auch im dezentralen Versionskontrollsystem Git zur Anwendung kommen.

Wir spulen jetzt einige Jahre nach vorne. Lars hat seine Firma mittlerweile verkauft und arbeitet wieder als Angestellter bei einer kleinen Softwarefirma in Nürnberg.

4. Tag 1

4.1. Der Hintergrund

Karl ist Freiberufler, Consultant und Softwareentwickler aus Leidenschaft. Er arbeitet seit vielen Jahren sehr produktiv mit verschiedenen Frameworks und für verschiedene Kunden.

Karl lebt mit seiner Frau und seinem Sohn in einer kleinen Wohnung in München. Der Beruf bringt es leider mit sich, dass Karl gelegentlich Projekte auch außerhalb Münchens annehmen muss.

Heute startet ein neues Projekt für Karl und zwar in Nürnberg. Karl überlegt meistens sehr lange, ob er Projekte annehmen soll die mit Reisezeit, Pendeln und Übernachtungen verbunden sind. Das Projekt in Nürnberg jedoch ist technologisch so interessant, dass Karl die Unannehmlichkeiten hierfür in Kauf nimmt.

Aufgrund seines Sohnes möchte Karl aber vermeiden, die ganze Woche in Nürnberg im Hotel zu übernachten. Aus diesem Grund pendelt er morgens nach Nürnberg und abends wieder zurück. Das bedeutet, Karl verbringt jeden Tag 2,5 Stunden im Zug. Zeit die produktiv genutzt werden will.

Wie es der Zufall will arbeitet das Team von Karl mit dem dezentralen Versionskontrollsyste **Git³**.

Einer der Vorteile die ein dezentrales System mit sich bringt ist *Offline-Fähigkeit*.



Ein bekannter Scherz unter GIT-Nutzern ist, dass die Offline-Fähigkeit von Git nur in der Theorie gegeben ist, da man Git ohne Google nur schwer auf der Kommandozeile verwenden kann.

Offline-Fähigkeit wird oft gleichgesetzt mit der Möglichkeit ohne Netzwerk zu arbeiten. Im Fall von Karl ist dies korrekt und auch wichtig, da Karl so auch im ICE nach Nürnberg produktiv in seinem Repository arbeiten kann.

Ein viel wichtigerer Aspekt ist jedoch die Möglichkeit, *lokal* mit Git zu arbeiten. Die meisten Operationen in Git sind lokale Operationen und nur ein ganz kleiner Teil ist wirklich mit einer Remote-Operation verbunden. In zentralisierten Systemen wie *Subversion* benötigen im Gegensatz alle Operationen eine Verbindung zum zentralen Repository-Server.

³ <http://www.git-scm.org>

4.2. Git Basics

Karl kommt in Nürnberg an und freut sich, seine neuen Kollegen zu treffen. Die erste und wichtigste Aufgabe eines Entwicklers in einem neuen Projekt, neben dem obligatorischen Kaffeetratsch in der Küche, ist das Auschecken des Projektrepositories und das Einrichten des Arbeitsplatzes.

Karl hat bisher keinerlei Erfahrung mit Git und freut sich, dass sich sofort alle Kollegen bereit erklären, ihn bei der Einarbeitung zu unterstützen.

Erster Ansprechpartner für Karl ist Lars, der für Neuzugänge im Team eine Art Mentorenrolle übernimmt.

Karl und Lars setzen sich zusammen an Karls Tisch und richten den Arbeitsplatz ein.

» *Karl, zunächst solltest Du dir Git installieren. Git wird in den nächsten Monaten dein wichtigstes Tool werden.*

» Das habe ich bereits zu Hause gemacht, wir hatten ja vereinbart, dass ich mit meinem eigenen PC hier im Büro arbeite, richtig? Soweit ich das bei unserem letzten Gespräch verstanden habe arbeiten die meisten Entwickler bei euch ja mit Windows.

» *Ja, das stimmt, es gibt einige Entwickler, die mit Linux arbeiten, aber die meisten arbeiten auf Windows Systemen.*

» Ich würde gerne mit meinem Mac-Book arbeiten. Wie ist das denn eurer Erfahrung nach? Sind die verschiedenen Git-Implementierungen kompatibel?

» *Soweit ich weiß hatten wir bisher nie Probleme mit Git und verschiedenen Systemen. Unsere Windows-Entwickler arbeiten mit Msys-Git⁴. Die haben zwar nur Versionen mit Preview im Namen, laufen aber stabil. Die Linux-Entwickler arbeiten mit der normalen Git-Version oder teilweise auch mit EGit unter Eclipse. Alle arbeiten bisher problemlos zusammen.*

» Das klingt gut, ich habe natürlich versucht, mich ein wenig schlau zu machen. Ich habe gelesen, dass es des öfteren mal zu Problemen zwischen Windows und Linux/BSD Systemen kommt, weil Windows das Zeilenende anders behandelt?

» *Ja das stimmt, die verschiedenen Systeme behandeln Zeilenumbrüche jeweils unterschiedlich. Man könnte jetzt natürlich denken, dass dies ein Problem ist, wenn*

⁴ <http://code.google.com/p/msysgit/>

ein Entwickler auf einem Linux-Rechner eine Klasse öffnet, die auch einem Windows-Rechner erstellt wurde, gerade wenn es um einheitliche Formatierung geht. Git kümmert sich aber darum.⁵ In der Standardeinstellung, mit der wir hier auch arbeiten wird Git für jede Datei, die du eincheckst jedesmal alle Zeilenenden in das Linux-Format konvertieren. Beim Auschecken werden die Zeilenenden wieder in das Format für dein aktuelles System konvertiert.

» Ziemlich intelligent gemacht. Ich würde sagen, wir fangen einfach direkt an zu arbeiten und klären weitere Fragen dann, wenn sie auftauchen?«

Klonen

Karl möchte natürlich schnellstmöglich mit der Arbeit starten. Bevor er aber anfangen kann muss er sich zunächst den Source-Code des Projektes besorgen, an dem er arbeiten soll.

Würde das Team mit Subversion arbeiten würde sich Karl eine Revision aus dem zentralen Repository auschecken. Eine Revision entspricht einem bestimmten Entwicklungsstand zu einem bestimmten Zeitpunkt. Jedesmal wenn ein Entwickler in das zentrale Repository eincheckt wird der globale Revisionzähler hochgezählt. Das funktioniert deshalb, weil sich alle Entwickler über das zentrale Repository synchronisieren. Es gibt keine Möglichkeit, Commits zu machen ohne mit dem zentralen Repository zu sprechen.

Das Team arbeitet aber **Git**-sei-Dank nicht mit Subversion sondern mit Git. Um sich den Sourcecode des Projektes zu holen muss Karl das Projektrepository klonen. Die *Clone*-Operation unterscheidet sich fundamental vom Auschecken einer Revision in Subversion, da nicht nur ein bestimmter Entwicklungsstand geladen wird sondern das komplette Repository inklusive jeglicher Historie, aller Commits, aller Tags und Branches und allem was jemals in diesem Repository geschehen ist.

Karl klonnt sich also das Team-Test-Repository unter Anleitung von Lars. Das Test-Repository ist eine Spielwiese auf der neue Entwickler sich zunächst mit Git vertraut machen können.

```
git clone ssh://karl@repository-server.intern.com/repos/test-projekt.git  
mein-test-projekt.git  
Cloning into 'git-ws-repos-local'...  
done.
```

⁵ <http://git-scm.com/book/ch7-1.html#Formatting-and-Whitespace>

Checking connectivity... **done**

Im Team ist der Zugriff auf das Repository über SSH gesteuert. Über SSH lassen sich problemlos Zugriffsrechte auf Maschinenebene und notfalls auch auf Repository-Ebene über Unix-File-Permissions steuern.



Für kleine Teams mit einfachen Zugriffsrechten ist dies eine passende Lösung. Für komplexere Teams mit vielen unterschiedlichen Rollen und Rechten sollte hierfür ein System wie Stash oder Gitosis verwendet werden.

.git/config

Mit der Klon-Operation hat sich Karl das komplette Repository zu sich lokal auf den Rechner geholt.

» *Karl, da du mit Git noch nicht so vertraut bist empfehle ich dir, mal einen Blick in das .git-Verzeichnis zu werfen.*

» Es gibt ein .git-Vereichnis?

» *Ja genau, in diesem Verzeichnis ist das eigentliche Repository zu finden. Schau dir das mal an.*

HEAD ①
branches ②
config ③
hooks/④
index⑤
objects/⑥
ref⑦

① Referenz auf den obersten Commit, auf dem das Repository aktuell steht.

② Deprecated, wurde früher zum Speichern von Branches verwendet

③ Konfiguration für dieses Git-Repository

④ Hooks

⑤ Git-Index Binary

⑥ Objekt-Datenbank

⑦ Referenzen auf Branches

» *Es würde jetzt nichts bringen, wenn ich dir das alles erkläre. Mit den meisten dieser Dateien und Verzeichnissen wirst du in den nächsten Tagen sowieso in Berührung*

kommen. Die wichtigste Datei, die du dir vielleicht gleich mal anschauen solltest ist die .git/config Datei. In dieser Datei befindet sich die Konfiguration für dein Repository.

```
[core] ①
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = false

[remote "origin"] ②
    url = ssh://karl@repository-server.intern.com/repos/test-
projekt.git
    fetch = +refs/heads/*:refs/remotes/origin/*

[branch "master"] ③
    remote = origin
    merge = refs/heads/master
```

- ① Core-Konfiguration, für Entwickler meist uninteressant
- ② Remotes - hier ist konfiguriert, dass unter dem Namen "origin" ein Repository mit der hinterlegten url zu finden ist.
- ③ Branch Tracking Information - hier ist hinterlegt, dass sich der Branch "master" automatisch mit dem Remote-Repository "origin" verbinden soll.

» Zugegeben, Karl, für Deine tägliche Arbeit wirst du die Informationen hier selten brauchen. Aber glaub mir, es ist gerade auch für Entwickler enorm wichtig zu verstehen wie Git tatsächlich funktioniert. Und das beispielsweise "origin", was du sehr oft sehen wirst nichts anderes als ein Name für ein Repository hinter einer URL ist. Man könnte das Repository auch "Karl_Remote" nennen.

» Das Einzige was du zwingend konfigurieren musst ist dein Name und deine E-Mailadresse, damit Git weiß, wen es als Autor in den Commit schreiben soll.

```
git config user.name "Karl"
git config user.email "Karl@effectivetrainings.de"
```

» Diese Konfiguration landet übrigens auch in der .git/config. Schau dir das nochmal an!.

```
[user]
name = Karl
email = karl@effectivetrainings.de
```

4.3. Daily Work

Karl möchte natürlich schnellstmöglich seinen ersten Commit machen. Da wir uns im Test-Repository befinden ist das auch kein Problem. Zunächst verschafft sich Karl einen Überblick über das Repository.

```
git status
# On branch master
nothing to commit, working directory clean
```

Wir sehen hier bereits wichtige Informationen. Initial befinden wir uns auf dem Branch *master*, der automatisch angelegt wurde. Der *master*-Branch ist vergleichbar mit dem Subversion-Trunk.



master ist nur ein Name und der master-Branch ein Branch wie jeder andere. Der Name *master* ist nur Konvention und hat ansonsten keine Bedeutung.

Weiterhin sehen wir, dass wir derzeit keine lokalen Änderungen in unserem Repository haben. Der Stand des lokalen Repositories entspricht also dem des entfernten Repository.

Log

Eine Übersicht über die bisher im Repository gemachten Commits bekommt Karl mit Hilfe von *git log*.

```
git log
Commit: cea024d4f4af1080b2a4d52f8477c6dc6647cdef ①
Author: dilgerm <martin@effectivetrainings.de> ②
Date:   (54 minutes ago) 2014-01-15 09:34:09 +0100 ③
Subject: initial commit ④
```

- ① Der Hash-Wert des Commits
- ② Der Autor
- ③ Datum und Uhrzeit des Commits
- ④ Die Commit-Message

Je nach Bedarf ist dies aber bereits zu viel Information. In 90% der Fälle möchte Karl nicht alle Information sehen, sondern beispielsweise nur wann der letzte Commit im Repository gemacht wurde. Das *log*-Kommando lässt sich bis zur [Unkenntlichkeit](#)

parametrisieren⁶. Es gibt jedoch einige Parameter die von den meisten Entwicklern im Team fast täglich verwendet werden.

One Liner

```
git log --oneline ①  
cea024d initial commit
```

- ① Zeigt einen abgekürzten Hash-Wert und nur die Commit-Message.

File-History

```
git log --oneline README ①  
cea024d initial commit
```

- ① Zeigt nur die Commits, die die Datei README betreffen.

```
git log --oneline -- README①  
cea024d initial commit
```

- ① Zeigt nur die Commits, die die Datei README betreffen, funktioniert auch wenn die Datei README nicht mehr vorhanden ist.



Der -- Operator dient als Trennung bei vielen Git-Kommandos und trennt die Kommandoparameter von den betroffenen Dateinamen.

Author-Commits

```
git log --author=dilgerm -- README①  
cea024d initial commit
```

- ① Zeigt nur die Commits, die vom Autor *dilgerm* sind.

Daily Standup

```
git log --oneline --since '1 day ago' --no-merges --author $(git config --  
get user.name) ①
```

⁶ <https://www.kernel.org/pub/software/scm/git/docs/git-log.html>

- ① Zeigt alle Commits des Autors und des letzten Tages ohne Merges. (Beispiel von <https://coderwall.com/p/vyl8zg>)

Vom Change zum Commit - Developer Workflow

Höchste Zeit, dass sich Karl ein wenig intensiver mit der Arbeitsweise mit Git vertraut macht.

```
echo 'Karl was here' >> Karl.txt ①
git status ②
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# Karl.txt ③
```

- ① Erzeuge eine neue Datei mit Inhalt und Namen des Entwicklers.
- ② Überprüfe den Status des Repositories
- ③ Status zeigt an, dass eine neue ("untracked") Datei vorhanden ist.

Die Datei *Karl.txt* ist Git bisher nicht bekannt. Das ändern wir, indem wir sie zum Index hinzufügen.

```
git add Karl.txt ①
git status
On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   Karl.txt ②
#
```

- ① Karl macht Git mit der Datei *Karl.txt* bekannt.
 - ② Karl sieht nun nicht mehr untracked-files sondern *Changes to be Committed*. Die Datei *Karl.txt* ist also für den nächsten Commit vorgemerkt.
- Karl kann jetzt endlich seinen ersten Commit machen.

```
git commit -m "Karls first Commit" ①
[master 85f37a2] Karls first Commit ②
1 file changed, 1 insertion(+) ③
```

```
create mode 100644 Karl.txt
```

- ① Schreibt alle vorgemerkt Änderungen in das Git Repository (Commit)
 - ② Zusammenfassung des Commits
 - ③ Statistik des Commits
-

```
git log --oneline  
85f37a2 Karls first Commit  
cea024d initial commit
```

Zuletzt sorgt ein *git status* nochmal für Sicherheit.

```
git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 1 commit.  
#   (use "git push" to publish your local commits)  
#  
nothing to commit, working directory clean
```

Karl sieht, dass nach dem Commit das Working-Directory wieder sauber ist. Git erkennt sogar, dass unser lokaler Branch einen Commit weiter ist als der zugeordnete Remote-Branch. Dies funktioniert nur, wenn das **Tracking**⁷ der Branches richtig initialisiert ist. Ein Branch kann jederzeit mit einem beliebigen Remote-Branch verbunden werden.

```
git branch -u origin/master ①
```

- ① Verbindet den aktuell ausgecheckten Branch mit dem origin/master Branch. Funktioniert leider erst ab Git 1.8.x

Für ältere Git-Versionen (1.7.x) war noch dies notwendig.

```
git branch --set-upstream master origin/master
```

Karl kann sich auch den Vergleich mit dem Remote-Tracking-Branch explizit anzeigen lassen.

```
git branch -v  
* master 85f37a2 [ahead 1] Karls first Commit
```

⁷ <http://git-scm.com/book/ch3-5.html#Tracking-Branches>

Hiermit sieht Karl, auf welchem Commit der aktuelle Branch steht, um wieviele Commits der lokale Branch vom Tracking-Branch abweicht und was die Commit-MESSAGE war. Zugegeben, diese Information hätte man besser in *git log* untergebracht, aber Git ist leider nicht bekannt für seine Konsistenz.

Der typische Entwickler-Flow sieht also folgendermaßen aus.



Diesen Flow durchläuft jeder Entwickler im Team jeden Tag dutzende Male. Je länger ein Entwickler mit Git arbeitet, desto kleiner und feingranularer werden üblicherweise die Commits im Repository.

Übung

Erzeugen Sie genauso wie Karl in einem Repository Ihrer Wahl eine neue Datei <IhrName>.txt

Schreiben Sie beliebigen Inhalt in diese Datei.

Überprüfen Sie, wie sich das Repository verändert mit Hilfe von *git status* und *git log*.

Committen Sie Ihre Änderung.

Überprüfen Sie Ihren Commit erneut mit Hilfe von *git log*.

Überprüfen Sie mit Hilfe von *git status* dass keine weiteren Änderungen mehr in Ihrem Repository vorhanden sind.

4.4. Internals

Karl scheint zufrieden.

» Das ist ja ganz einfach!

» *Ja das ist es. Aber es ist trotzdem auch wichtig, dass Du verstehst, was genau jetzt passiert ist. Git ist nicht immer einfach, aber immer logisch. Bevor wir weitermachen, würde ich gerne mit dir über einige Dinge sprechen, die jetzt im Repository passiert sind. Je genauer du verstehst, wie Git arbeitet, desto leichter wirst Du dir später tun, wenn die ersten Probleme auftreten.*

» Liebend gern, Lars. Was muss ich wissen?

» *Am besten du wirfst einen Blick in dein .git-Verzeichnis. Und hier speziell in das / objects-Verzeichnis.*

» Ah ja, objects klingt gut, ist Git denn objektorientiert programmiert?

» *Nein, das hat nichts mit den Objekten einer Programmiersprache zu tun. Vielleicht ist dieses Verzeichnis auch einfach nur unglücklich benannt. Alle Objekte, die du in Git speicherst, also primär Dateien und Verzeichnisse, werden als Objekte bezeichnet. Git kennt hauptsächlich vier Arten von Objekten - Blobs, Trees, Commits und Tags. Wenn Du in dein objects-Verzeichnis schaust solltest Du etwas in der Art sehen.*

```
|- 05
  └── 8f0f82590adfebbd4d4fc2c55ede64771390d3
|- 30
  └── 4360cabab487e6f7b707b5aa96774f85bf17b77
|- 64
  └── c4b2cbdcbe14b6b14e04f1e787c21bfc8fc802
|- 67
  └── e764f05cec7cfb6219cd57d69c421ba3eeaae4
|- 92
  └── dfda6b9bfa599a0524d1b5c341802170d3d510
|- ce
  └── a024d4f4af1080b2a4d52f8477c6dc6647cdef
|- d1
  └── 63091526fd797675973f966af5313c8fbb2ea1
|- ff
  └── 3e2ea55c4cda9ebdb9f87d5b7e1dfa26b6393e
|- info
|- pack
```

» Wow, das sieht aber kompliziert aus.

» *Ja stimmt, wenn man nicht weiß, was das ist könnte man denken, Git ist total kompliziert oder? Dabei wirst du sehen, dass Git grundsätzlich kinderleicht zu*

verstehen ist, weil das Prinzip dahinter so einfach ist. Du siehst also viele seltsame Verzeichnisse, die nur aus jeweils zwei Zeichen bestehen, beispielsweise 05, richtig?

» Ja genau!

» *Wir hatten schon darüber gesprochen, dass Git sehr unterschiedlich ist zu beispielsweise dem zentralen System Subversion. In Subversion synchronisieren sich alle Entwickler über das zentrale Repository. Deswegen ist es einfach eine Revision zu verwalten. Die Revision wird einfach immer weiter nach oben gezählt, jedesmal wenn ein Entwickler einen Commit macht. Was meinst du, würde das in Git auch funktionieren?*

» Hm, gute Frage, da alle Commits zunächst lokal sind müsste die Revision lokal hochgezählt werden. Wenn ich jetzt aber an einem neuen Feature arbeite und Du gleichzeitig an einem anderen, dann würde bei uns beiden die Revision lokal hochgezählt werden. Probleme hätten wir erst, wenn wir versuchen würden unsere Arbeit zusammenzubringen, richtig? Welche Revision würde dann gewinnen?

» *Sehr gut aufgepasst, Karl. Wir arbeiten dezentral, wir können also unmöglich eine globale Revision verwalten. Git muss sogar sicherstellen, dass die Revisions weltweit funktionieren, egal wieviele Entwickler an einem Projekt arbeiten.*

» Das hat was mit diesen Hash-Werten zu tun, die wir hier sehen, oder?

» *Perfekt! Genau, Git arbeitet anders. Git berechnet den SHA-1-Hash-Wert über die Inhalte aller am Commit beteiligter Dateien jedesmal neu wenn wir committen.*

» Ja, das könnte funktionieren. Wenn wir beide aus Versehen die gleiche Änderung machen, ich bei mir und Du bei dir am PC, dann hätten diese beiden Commits also den gleichen Hash-Wert und somit die gleiche "Revision" in Git?

» *Genau Karl, das kann man so sehen. Die Revision in Git ist der Hash-Wert des Commits. Das ist aber noch nicht alles. Ich zeig dir mal was.*

```
git hash-object Karl.txt
058f0f82590adfebbd4d4fc2c55ede64771390d3
```

» *Mit Hilfe von git hash-object lässt sich der SHA-1 Hash eines Objektes berechnen. Vergleiche doch diesen Wert mal mit den Verzeichnissen in deinem objects-Verzeichnis.*

» Warte mal, ich sehe ein Verzeichnis "05" und eine Datei "8f0f82590adfebbd4d4fc2c55ede64771390d3". Das kann kein Zufall sein. Kann es

sein, dass Git den Hash-Wert meiner Datei berechnet hat, die ersten zwei Zeichen des Hash-Wertes als Verzeichnis nimmt und den restlichen Hash-Wert als Dateinamen?

» *Genau Karl, so ist es. Git verwendet die ersten beiden Zeichen als Verzeichnisnamen um Betriebssystem-Beschränkungen zu umgehen. Es können nunmal nicht unendlich viele Dateien in einem Verzeichnis gespeichert werden. Den Rest des Hash-Wertes verwendet Git als Dateinamen. Kannst Du dir vorstellen, was Git genau speichert?*

» Ich nehme an, einfach meine Textdatei?

» *Nicht ganz, versuch doch mal die Datei zu öffnen*

```
cat .git/objects/05/8f0f82590adfebbd4d4fc2c55ede64771390d3
xK??OR04a?N, ?Q(O,V?H-J?06?
```

» Hm, sieht binär aus?

» *Genau, Git speichert die Dateien nicht im Rohformat sondern packt alles nochmal sehr effizient mit Hilfe von ZLib zusammen. Die Dateien liegen also binär vor und Git entpackt die Dateien nur wenn notwendig.*

» Ok, verstanden. Aber eine Frage hätte ich dann doch noch?

» *Und die wäre?*

» Wenn ich mir den Commit mit *git log* anschause.

```
git log
Commit: ff3e2ea55c4cda9ebdb9f87d5b7e1dfa26b6393e
Author: Karl <karl@effectivetrainings.de>
Date:   (33 minutes ago) 2014-01-15 13:55:37 +0100
Subject: Karls first Commit
```

» Der Hash-Wert des Commits ff3e2ea55c4cda9ebdb9f87d5b7e1dfa26b6393e stimmt doch nicht überein mit dem Hashwert 058f0f82590adfebbd4d4fc2c55ede64771390d3 meiner Datei?

» *Sehr gut beobachtet! Wir haben uns bisher nur den Hash-Wert deiner Textdatei angesehen. Ich habe aber vorhin schon erwähnt, dass Git 4 Arten von Objekten kennt. Erinnerst Du dich noch?*

» Ja, Commits, Trees, Blobs und Tags?

» Korrekt, wir haben uns jetzt deinen ersten Blob angeschaut. Das siehst du auch, wenn du dir den Typen der Datei anschaugst.

```
git cat-file -t 058f0f82590adfebbd4d4fc2c55ede64771390d3  
blob
```

» Jede Datei die mit Git gespeichert wird landet als **Blob** in der Objektdatenbank. Was ist jetzt ein **Tree**? Du kannst dir das einfach als Repräsentation eines Verzeichnisses vorstellen. Du siehst den **Tree** sogar, indem du den Hash-Wert des Commits mit `cat-file -p` betrachtest. Das **p** steht für "pretty". Das Kommando `git cat-file` ist ein sogenanntes **Plumbing-Kommando**.



Kommmands in Git sind nach dem **Composite-Pattern** aufgebaut. Sie sind unterteilt in sogenannte **Plumbing** oder **Low-Level**- und **Porcellain-Kommandos**. In den meisten Fällen arbeiten Entwickler nur mit dem "guten Porzellan". Es macht aber Sinn, sich durchaus auch mit den Low-Level Operationen wie `cat-file` zu beschäftigen.

```
git cat-file -p ff3e2ea55c4cda9ebdb9f87d5b7e1dfa26b6393e  
  
tree 64c4b2cbdcbc14b6b14e04f1e787c21bfc8fc802  
parent cea024d4f4af1080b2a4d52f8477c6dc6647cdef  
author Karl <karl@effectivetrainings.de> 1389790537 +0100  
committer Karl <karl@effectivetrainings.de> 1389790766 +0100
```

Karls first Commit

» Hier siehst du den **Tree**. Den können wir uns jetzt nochmal genauer betrachten.

```
git cat-file -p 64c4b2cbdcbc14b6b14e04f1e787c21bfc8fc802  
100644 blob 058f0f82590adfebbd4d4fc2c55ede64771390d3 Karl.txt  
100644 blob 304360cabaa487e6f7b707b5aa96774f85bf17b77 README
```

» Siehst du? Der **Tree** referenziert also die beiden **Blobs**, genauso wie das Verzeichnis die beiden Dateien referenziert. So einfach ist das. Ein Commit referenziert immer einen **Tree**, ein **Tree** referenziert immer einen oder mehrere **Blobs** oder auch weitere **Trees** als Unterverzeichnisse. Eine letzte Sache noch, wir haben uns vorher mit Hilfe von `cat-file -p` den Commit selbst angeschaut. Eine Sache hierbei war interessant und ist dir vielleicht entgangen.

```
git cat-file -p ff3e2ea55c4cda9ebdb9f87d5b7e1dfa26b6393e
```

```
tree 64c4b2cbdcbe14b6b14e04f1e787c21bfc8fc802
parent cea024d4f4af1080b2a4d52f8477c6dc6647cdef
author Karl <karl@effectivetrainings.de> 1389790537 +0100
committer Karl <karl@effectivetrainings.de> 1389790766 +0100
```

Karls first Commit

» *Der Commit referenziert seinen Parent-Commit. Commits in Git schweben nicht irgendwie im luftleeren Raum sondern sind miteinander über eine Parent-Child Hierarchie verbunden. Jeder Commit hat entweder keinen, genau einen oder beliebig viele Parent-Commits. Kein Commit ist klar, dies kann nur für den allerersten Commit im Repository der Fall sein. Die meisten Commits haben genau einen Parent-Commit, nämlich der direkt vorangegangene Commit. Werden Branches zusammengeführt entstehen sogenannte Merge-Commits. Commits haben die Eigenschaft so viele Parents zu haben wie Branches zusammengeführt wurden. In den meisten Fällen also zwei, das erkläre ich dir aber, wenn wir dazu kommen. In Ordnung?*

» Ja, in Ordnung. Mir ist zwar noch nicht ganz klar, wozu ich diese ganzen Informationen brauche, aber ich denke, das wird mir später klar.

4.5. Branches

Karl hat von Lars bereits einen sehr guten Überblick über die grundsätzliche Arbeitsweise mit Git erhalten. Jetzt wird es höchste Zeit, dass Karl seine Arbeit als Entwickler aufnimmt. Der erste Schritt besteht nun darin, sich das Projekt-Repository auszuchecken.

Übung

Klonen Sie sich das Repository unter <https://github.com/dilgerma/effective-git-workshop> in ein Verzeichnis *Projekt.git*

```
git clone https://github.com/dilgerma/effective-git-workshop
Project.git
```

Gib mir ein Ticket - ich starte

Karl möchte am liebsten sofort loslegen. Lars muss ihn ein wenig bremsen.

» *Karl, wir arbeiten nicht direkt auf dem master branch. Der master ist der aktuelle Entwicklungsstand, es sollten aber nur fertige Features zurückgeführt werden. Der master sollte zumindest stabil sein.*

» Ok, verstanden. Das bedeutet, dass wir auf eigenen Feature-Banches arbeiten? Ist das nicht ganz schön kompliziert?

» *Nur solange du mit Subversion arbeitest. Nein, Scherz beiseite. Das Arbeiten mit Branches ist quasi DAS Feature von Git. Branches sind so schnell und leichtgewichtig, dass es wirklich Spaß macht damit zu arbeiten.*

» Mit Branches zu arbeiten macht Spaß? Wow, das hör ich tatsächlich wirklich zum ersten Mal.

» *Ich zeige dir mal, wie Branches funktionieren. Es wird nämlich schnell klar, wieso Branches in Git so einfach sind, wenn man weiß wie sie funktionieren. Am besten wäre es, du wirst einen Blick in das .git/refs Verzeichnis.*

```
#alle Verzeichnisse
ls .git/refs
heads❶
remotes❷
tags

#alle Branch-Dateien
ls .git/refs/heads/
master ❸

#Enthalten jeweils Hash-Wert eines Commits
cat .git/refs/heads/master
ad261f23894095de696ffd43a0d01af1e7249a02 ❹

#Zeige obersten Commit im aktuellen Branch
git log --oneline
ad261f2 Initial commit ❺
```

❶ Hier sind Branches konfiguriert

❷ Hier sind Remote-Repositories konfiguriert

❸ Für jeden Branch befindet sich hier eine eigene Datei

❹ In der Datei steht jeweils nur ein Hashwert

❺ Der Hash-Wert des obersten Commits des aktuell ausgecheckten Branches entspricht dem Hash-Wert in refs/heads/<branchname>

» Das versteh ich nicht, warum brauch ich einen Hash-Wert in einer Datei?

» *Es ist ganz einfach. Git braucht irgendeine Art Mapping, um Branch und Commit zusammenzubringen. Wir referenzieren den master-Branch als master und nicht als ad261f2. Git arbeitet fast komplett File-basiert. Das Mapping besteht also darin, dass wir eine Datei master haben, in der der Hash-Wert ad261f2 steht. Dadurch weiß git, dass der Branch master existiert (weil die Datei vorhanden ist) und der oberste Commit im Branch master den Hash-Wert ad261f2 hat (weil dieser Hash-Wert in der Datei steht).*

Erzeugen wir uns doch einfach einen Feature-Branch. Ich habe dir gestern bereits ein sehr einfaches Ticket herausgesucht, an dem du heute arbeiten kannst. Branches haben bei uns immer eine feste Bezeichnung. Normalerweise arbeiten wir mit GitFlow⁸, aber für den Anfang ist es glaube ich besser, wenn wir das erst ein paar Mal manuell machen, als Fingerübung quasi.

```
git branch feature-4711 ①
git branch ②
feature-4711
* master ③
```

① git branch <branch-name> erzeugt einen neuen Branch

② git branch ohne Parameter zeigt die lokal verfügbaren Branches

③ Der * zeigt den aktuell ausgecheckten Branch an.

» Lars, kannst du mir sagen, wie ich jetzt auf meinen neuen Branch wechseln kann?

» *Klar, das geht mit checkout. Du checkst dir quasi einen Branch aus.*

```
#wechsel auf branch feature-4711
git checkout feature-4711

#Wechsel und neu erzeugen eines Branches
git checkout -b feature-4711
```



Checkout hat je nach Kontext und Parametern ganz unterschiedliche Bedeutungen. Weitere Bedeutungen werden später noch erläutert.

⁸ <http://www.effectivetrainings.de/blog/2012/04/22/git-flow-einfaches-arbeiten-mit-dem-perfekten-git-workflow/>

Übung

Was hat sich durch das Erzeugen des Branches im .git/refs/heads-Verzeichnis verändert?

Erzeugen Sie den Branch **feature-4711** vom master. Editieren Sie die README-Datei auf dem Branch feature-4711. Committen Sie Ihre Änderung.

Wie hat sich das .git/refs/heads-Verzeichnis verändert?

```
#Eine neue Datei ist entstanden
ls .git/refs/heads/
feature-4711
master

#Da wir den Branch vom master branch gezogen haben stehen sowohl
#master als auch feature-4711 aktuell auf dem gleichen Commit.
cat .git/refs/heads/master
ad261f23894095de696ffd43a0d01af1e7249a02
cat .git/refs/heads/feature-4711
ad261f23894095de696ffd43a0d01af1e7249a02

#Branchwechsel mit checkout
git checkout feature-4711

#editiere readme und commit.
git commit -m "Adjusted Readme"
[feature-4711 bebc4db] Adjusted Readme
 1 file changed, 2 insertions(+)

#Die Dateien unterscheiden sich, weil der Feature-Branch einen Commit
#weiter
cat .git/refs/heads/master
ad261f23894095de696ffd43a0d01af1e7249a02
cat .git/refs/heads/feature-4711
bebc4dbc18cb05d7fd2df59db7cf249bc793dbf0
```

» Das ist interessant. Wie weiß denn Git, welchen Branch ich aktuell ausgecheckt habe?

» *Gute Frage, dazu wirst Du am besten einen Blick in dein .git-Verzeichnis.*

```
git log HEAD ①
Commit: bebc4dbc18cb05d7fd2df59db7cf249bc793dbf0
Author: dilgerm <martin@effectivetrainings.de>
Date:   (37 minutes ago) 2014-01-15 16:57:42 +0100
Subject: Adjusted Readme
```

```
#Welcher Branch aktuell ausgecheckt ist steht in der Datei HEAD
cat .git/HEAD
ref: refs/heads/feature-4711
```

- ① HEAD ist nur eine andere Bezeichnung für den aktuell obersten Commit in der Historie

» Um zu wissen, auf welchem Commit mein Repository aktuell steht schreibt Git jedesmal, wenn ich den Branch wechsel den Pfad der Branch-Datei aus refs/heads in die Datei HEAD in meinem .git-Verzeichnis. Um zu wissen auf welchem Branch wir uns aktuell befinden macht Git intern etwa folgendes.

```
Pfad = Lade Dateipfad aus .git/HEAD
Commit-Hash = Lese Datei aus Pfad
Branch-Name = Lese Dateinamen aus Pfad
Setze obersten Commit im Repository auf Commit-Hash
```

Branch nachträglich erstellen

» Karl, du wirst sehen, du wirst dich sehr schnell an das Arbeiten mit Branches gewöhnen. Spätestens in 3 Wochen wirst du dich fragen, wie du jemals ohne Arbeiten konntest.

» Ich werde dich in ein paar Wochen nochmal darauf ansprechen.

» Im Eifer des Gefechts passiert es übrigens jedem mal, dass er die Umsetzung einer Story aus Versehen auf dem **master** startet. Da das relativ oft passiert, auch mir zum Beispiel noch, zeige ich dir noch schnell, wie du das ganz einfach lösen kannst. Nehmen wir für diesen Fall an, du willst von deinem aktuellen Branch auf einen weiteren Branch wechseln. Stell dir einfach vor, du arbeitest mit zwei anderen Kollegen an diesem Feature und möchtest etwas ausprobieren. Dazu möchtest Du gerne einen eigenen Branch erzeugen, auf dem du lokal bei dir arbeiten kannst. Leider hast Du bereits zwei Commits auf dem aktuellen Feature-Branch gemacht, von denen du nicht sicher bist, ob du sie behalten möchtest. Das ist sehr einfach zu lösen indem du vom aktuellen Branch einen weiteren Branch "feature-4711-experiment" ziehst und dann den Branch "feature-4711" um die beiden fraglichen Commits zurücksetzt.

» Du sagst immer, dass das alles ganz einfach ist. Für mich klingt das ganz schön kompliziert.

» *Keine Sorge, das sind nur die ganzen Begriffe, du wirst dich sehr schnell daran gewöhnen. Erzeuge doch bitte mal die zwei Commits deines Experimentes auf dem aktuellen Branch.*

```
git branch
* feature-4711
[...]

#erster commit
echo "Karls erster Commit" >> Karls-experiment.txt
git add Karls-experiment.txt
git commit -m "erster experiment commit"
[feature-4711 5f5a42d] erster experiment commit
 1 file changed, 1 insertion(+)
 create mode 100644 Karls-experiment.txt

#zweiter commit
echo "Karls zweiter Commit" >> Karls-experiment.txt
git add Karls-experiment.txt
git commit -m "zweiter experiment commit"
[feature-4711 a1dbcc2] zweiter experiment commit
 1 file changed, 1 insertion(+)

#log
git log
Commit: a1dbcc20f620573097866445302991d877e76232
Author: dilgerm <martin@effectivetrainings.de>
Date:   (63 seconds ago) 2014-01-21 18:02:57 +0100
Subject: zweiter experiment commit

Commit: 5f5a42da684db9c5fa4f50c390bd2c78946c8238
Author: dilgerm <martin@effectivetrainings.de>
Date:   (2 minutes ago) 2014-01-21 18:01:44 +0100
Subject: erster experiment commit
[...]
```

» *Sehr gut Karl. Jetzt möchtest Du diese beiden Commits aber nicht auf deinem aktuellen Branch, sondern auf dem Branch "feature-4711-experiment" haben, weil du dir nicht sicher bist, ob deine Idee für die Umsetzung in die richtige Richtung geht.*

```
git checkout -b "feature-4711-experiment"  
Switched to a new branch 'feature-4711-experiment'
```

```
git branch  
  feature-4711  
* feature-4711-experiment  
[...]
```

```
#log  
git log  
Commit: a1dbcc20f620573097866445302991d877e76232  
Author: dilgerm <martin@effectivetrainings.de>  
Date:   (4 minutes ago) 2014-01-21 18:02:57 +0100  
Subject: zweiter experiment commit
```

```
Commit: 5f5a42da684db9c5fa4f50c390bd2c78946c8238  
Author: dilgerm <martin@effectivetrainings.de>  
Date:   (5 minutes ago) 2014-01-21 18:01:44 +0100  
Subject: erster experiment commit  
[...]
```

» Die Branches sind identisch und beide haben die zwei Commits des Experimentes. Jetzt wechselst du einfach zurück auf deinen "feature-4711"-Branch und setzt den Branch mit Hilfe von **reset** um die zwei Commits zurück, die du dort nicht haben möchtest. Am besten du machst einfach was ich dir sage, ich werde dir **reset** später noch genauer erklären.

```
git checkout feature-4711  
  
#branch um zwei commits zurücksetzen  
git reset --hard HEAD~2  
HEAD is now at bebc4db Adjusted Readme
```

» Damit hast du genau die Situation, die du wolltest. Die beiden Commits des Experimentes sind auf dem richtigen Branch und der Feature-4711-Branch sieht so aus als wäre nie etwas passiert. Ich weiß, das ging alles ein wenig schnell, stell dir für jetzt einfach vor, **git reset** ist eine Möglichkeit, jeden beliebigen Branch auf einen Commit deiner Wahl zu setzen. Beispielsweise den Commit, der "vor zwei Commits" gemacht wurde.

Merge

» Interessant, so langsam glaube ich bekomme ich ein ungefähres Bild und eine Idee, wie das mit Git tatsächlich gedacht ist.

» *Ja, es ist wunderschön, nicht wahr?*

» Soweit bin ich noch nicht. (**grins**)

» *Ok, machen wir weiter und gehen zurück auf den "feature-4711"-Branch. Auf dem haben wir jetzt eine Änderung gemacht. Irgendwann ist der Zeitpunkt gekommen, diese Änderung wieder auf den master zurückzubringen. Wann dieser Zeitpunkt gekommen ist, darüber lässt sich streiten. Es gibt Teams, die eine frühestmögliche Integration neuer Features zurück in den master bevorzugen. Wir gehören auch dazu, allerdings möchten wir nur Features zurückführen, die tatsächlich abgeschlossen sind. Andere Teams arbeiten direkt auf dem master. Der Vorteil ist, dass jeder Entwickler spätestens beim Update sofort sieht, ob zwei Features miteinander in Konflikt stehen. Der Nachteil ist klar, der master-Branch ist nicht mehr stabil, da auch halbfertige Features zurückgeführt werden. Wir bevorzugen allerdings einen stabilen master und nehmen dafür auch in Kauf, dass wir Merge-Konflikte erst etwas später bemerken und auflösen.*

» Ich denke, das befürworte ich. Frühe Integration ist immer gut, aber stabile Branches sind wichtiger. Vor allem könnt ihr dann auch Features zurückhalten, wenn diese beispielsweise bereits implementiert sind, aber erst später den Kunden zur Verfügung gestellt werden sollen. Ich habe hierfür früher immer Feature-Flags einbauen müssen.

» *Ja, Feature-Flags sind eine Alternative und haben auch ihre Daseinsberechtigung. Ich persönlich habe aber schon lange keine mehr verwendet.*

Genug geredet, was müssen wir tun um den Branch zurückzuführen?

» Lass mich raten, die Operation nennt sich *Merge*?

» *Das ist korrekt. Zunächst wechseln wir zurück auf den master. Um das Ganze ein wenig interessanter zu machen provozieren wir einen Merge-Konflikt.*

```
git checkout master
#edit README.md
git merge feature-4711
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
```

Automatic merge failed; fix conflicts and **then** commit the result.

» *Wir haben den ersten Konflikt in unserem Repository.*

» Interessant, welches Merge-Tool verwenden wir im Team?

» *Da gibt es keine Vorgabe, ich denke die meisten verwenden KDiff, einige aber auch die Entwicklungsumgebung direkt.*

» Ok, ich überlege mir das noch.

» *Am besten du schaust dir mal den aktuellen Stand deines Repositories an.*

```
git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# You have unmerged paths. ❶
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
# both modified:      README.md ❷
#
```

❶ Git zeigt einen Konflikt an

❷ Git benennt die Dateien, die Konflikte enthalten.

» Kann ich mir anschauen, was den Konflikt verursacht hat?

» *Natürlich, dazu gibt es wie fast immer in Git mehrere Möglichkeiten. Eine einfache Möglichkeit ist diff. Diff zeigt dir ohne weitere Parameter die Änderungen zwischen deinem Workspace und dem Repository an. Tritt ein Merge-Konflikt auf schreibt Git beide Änderungen in die betroffenen Dateien getrennt mit "=====".*

```
git diff
diff --cc README.md
index da95ff7,c3eb4a8..0000000
--- a/README.md
+++ b/README.md
@@@ -1,3 -1,4 +1,9 @@@
```

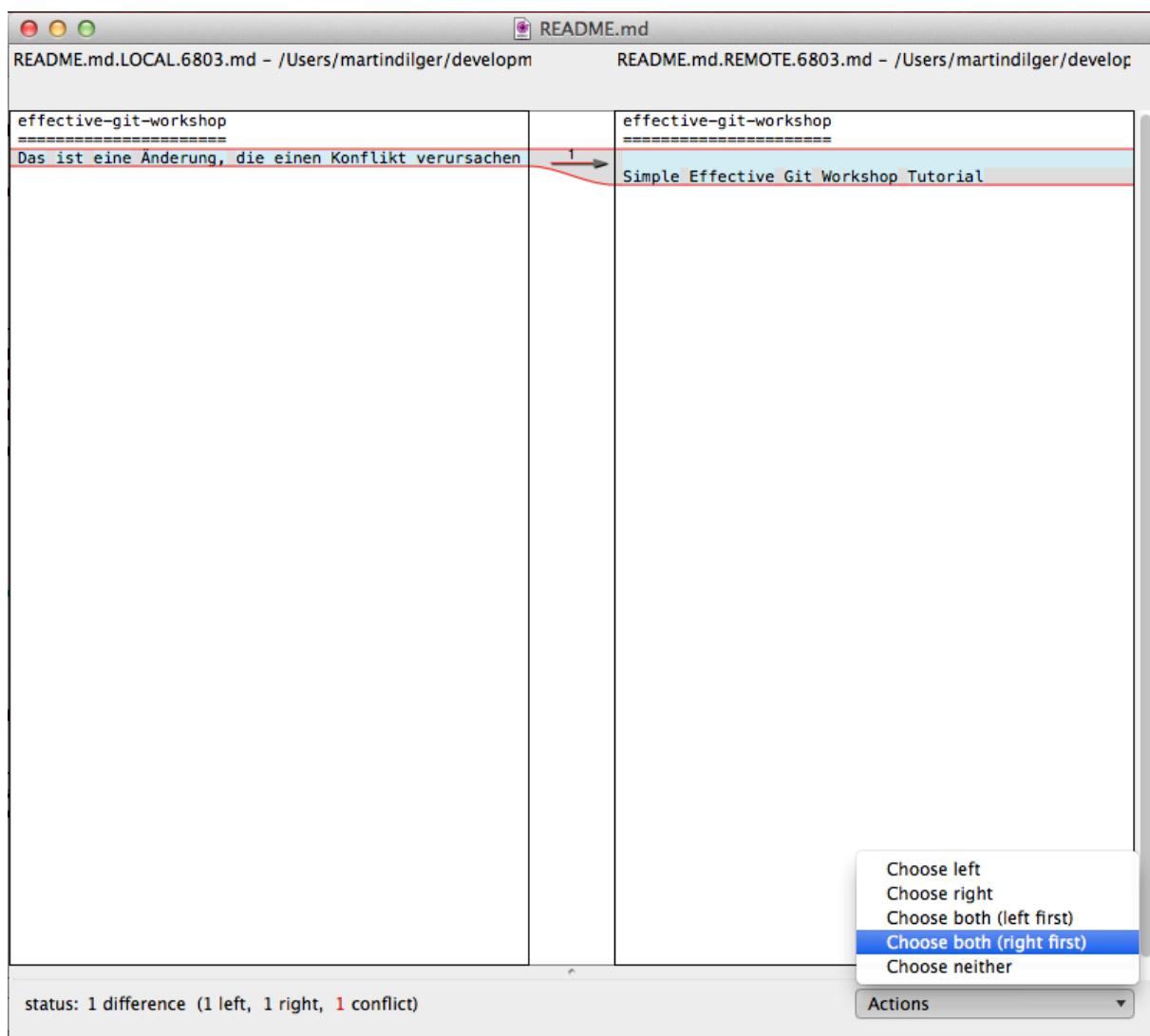
```
effective-git-workshop
=====
++<<<<< ours ①
+Das ist eine Änderung, die einen Konflikt verursachen sollte.
++||||||| base
++===== ②
+
+ Simple Effective Git Workshop Tutorial
++>>>>>> theirs ③
```

- ① Änderung auf dem aktuellen Branch
 - ② Trenner zwischen beiden Konflikten
 - ③ Änderung auf dem Branch der gemerged werden soll
- » Die Anzeige mit Diff ist aber wirklich schwer zu lesen. Eine weitere Möglichkeit ist die Anzeige im Mergetool. Du kannst das Mergen starten indem du git merge-tool aufrufst.
-

```
git mergetool
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse ecmerge
p4merge araxis bc3 codecompare emerge vimdiff ①
Merging:
README.md ②

Normal merge conflict for 'README.md': ③
{local}: modified file
{remote}: modified file
```

- ① Mögliche Tools, Git prüft nicht, ob diese Tools tatsächlich alle installiert sind.
- ② Die Datei die den Merge-Konflikt verursacht hat
- ③ Art des Merge-Konfliktes. In diesem Fall wurde die Datei auf beiden Branches editiert. Andere Möglichkeiten wären beispielsweise das Löschen der Datei.



» Ok, wir haben den Konflikt gelöst, wir müssen das aber unbedingt noch Committen. Das wird gerne vergessen. Am besten wir prüfen vorher nochmal schnell den Status des Repositories.

```
git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# All conflicts fixed but you are still merging. ❶
#   (use "git commit" to conclude merge)
#
# Changes to be committed:
#
# modified:   README.md
```

- ❶ Git erkennt, dass wir den Merge-Konflikt aufgelöst haben und weiß, dass wir bisher vergessen haben, diese Änderung zu committen.

Übung

Wechseln Sie auf den master-Branch.

Editieren Sie die Datei README.md, so dass ein Merge Konflikt mit der Änderung auf feature-4711 entsteht.

Führen Sie den Branch zurück und lösen Sie den Merge-Konflikt.

Welche Besonderheit erkennen Sie in der Historie?

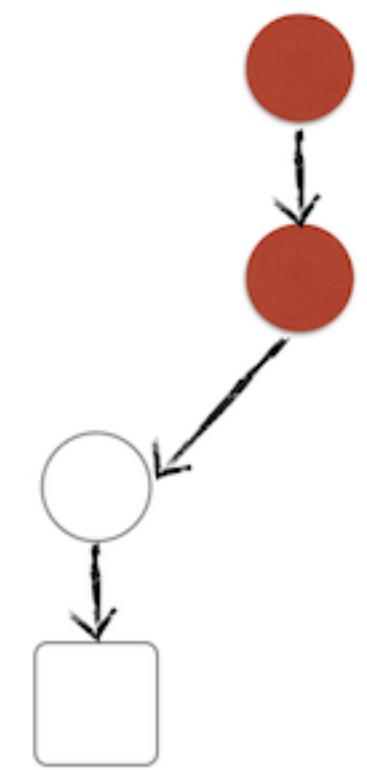
```
git checkout master
echo "Das ist eine Änderung, die einen Konflikt verursachen sollte."
>> README.md
git merge feature-4711
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

» *Karl, wenn Du dir jetzt mal die Historie auf dem master betrachtest siehst du eine kleine Besonderheit. Sehr einfach siehst Du das zum Beispiel mit folgendem Kommando.*

```
git log --graph
* Commit: bb2d3b275870e891a76b73d2597efc0a10fa373d ❶
| \ Author: dilgerm <martin@effectivetrainings.de>
| | Date: (67 seconds ago) 2014-01-16 16:51:54 +0100
| | Subject: merged fb-4711
| |
| |
| *
| * Commit: bebc4dbc18cb05d7fd2df59db7cf249bc793dbf0 ❷
| | Author: dilgerm <martin@effectivetrainings.de>
| | Date: (24 hours ago) 2014-01-15 16:57:42 +0100
| | Subject: Adjusted Readme
| |
| |
* | Commit: 817e46017f4094a4f33fc1f5dd423257a20a7c99 ❸
```

```
| / Author: dilgerm <martin@effectivetrainings.de>
| Date: (23 hours ago) 2014-01-15 18:06:18 +0100
| Subject: Konflikt
|
* Commit: ad261f23894095de696ffd43a0d01af1e7249a02
  Author: Martin Dilger <martin.dilger@googlemail.com>
  Date: (25 hours ago) 2014-01-15 06:56:25 -0800
  Subject: Initial commit
```

- ❶ Der entstandene Merge-Commit, der zwei Parents hat
 - ❷ Der Commit, den Karl auf dem Feature-Branch gemacht hat
 - ❸ Der Commit auf dem master, der den Konflikt verursacht hat.
- » *Ein Merge-Commit entsteht immer dann, wenn wir zwei oder mehr Branches zusammenführen und kein sogenannter Fast-Forward-Merge möglich ist.*
- » Was bitte ist ein Fast-Forward-Merge?
- » *Ein Fast-Forward-Merge, beispielsweise vom Feature-Branch auf den master, ist dann möglich, wenn auf dem master nichts passiert ist seit wir den Branch gezogen haben. Der komplette master-Branch ist also bereits in unserem Feature-Branch enthalten. In diesem Fall ändert Git beim Merge auf den master einfach den Inhalt der Datei **master** in **/refs/heads** auf den Hash-Wert des obersten Commits unseres Feature-Branches. Es kann kein Konflikt auftreten und es muss auch kein **Merge-Commit** erzeugt werden. Damit ist der Merge bereits abgeschlossen und der Branch vollständig zurückgeführt. Das ist der einfachste Fall eines Merges.*
- » Hm, Lars, ich bin mir nicht sicher, ob ich das richtig verstehre. Können wir das kurz am Whiteboard durchsprechen?
- » *Gute Idee, wir verwenden das sowieso viel zu selten.
Also pass auf. Nehmen wir einfach mal an, wir hätten den Merge noch nicht gemacht.
Das sah ja ungefähr so aus, richtig?*



master fb-4711

» Jetzt nehmen wir weiterhin an, wir hätten keinen Commit auf dem master gemacht und hätten folglich auch keinen Merge-Konflikt beim Merge gehabt. Alles was Git jetzt machen muss ist den Branch-Zeiger auf den neuesten Commit in meinem Branch zu setzen. Damit sind alle Commits aus dem Branch auch auf dem master verfügbar.



Rebase

» Danke, Lars, ich glaube, Merges hab ich verstanden. Es ist wirklich einfacher als gedacht.

» *Ja, nicht wahr?*

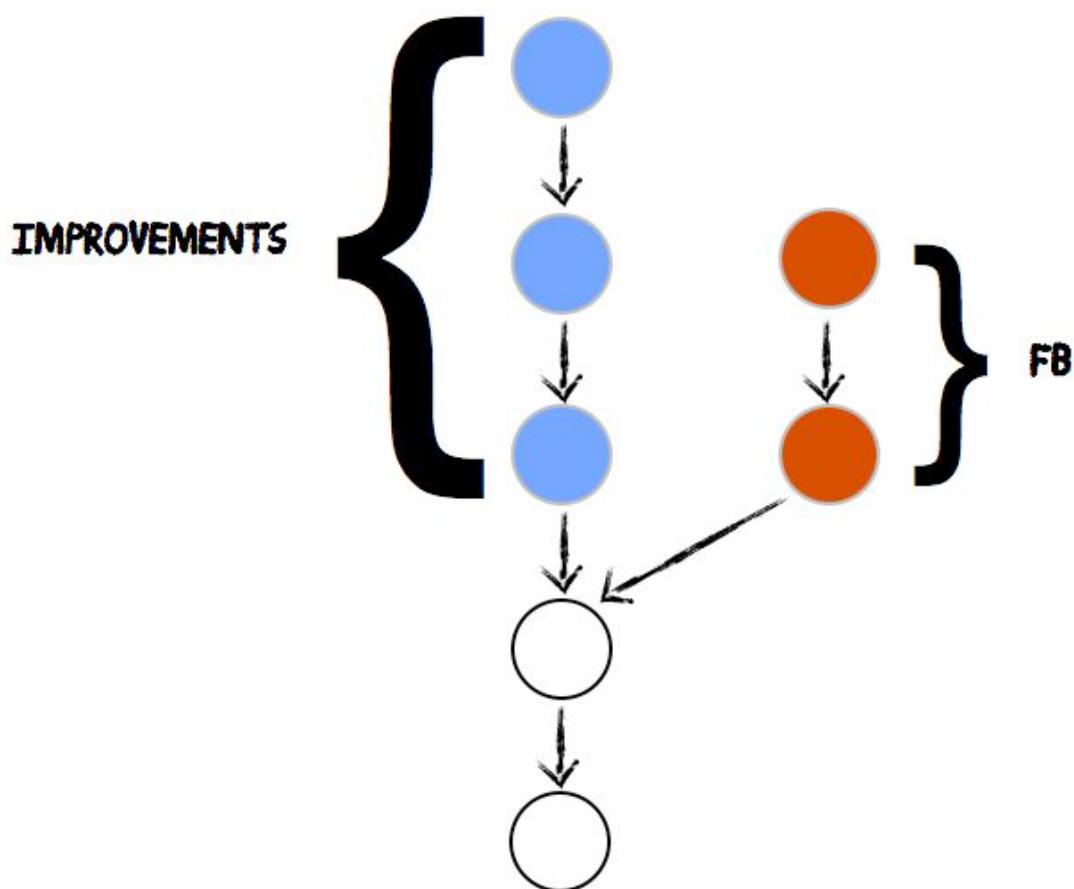
» Ich habe aber auch gelesen, dass es in Git mehrere Möglichkeiten gibt, Branches zu zusammenzubringen. Was war das? Rebase?

» *Ah, davon hast Du also auch bereits gehört. Es scheint schwierig, die Konzepte zu verstehen, vor allem da auf den ersten Blick Merge und Rebase ein recht ähnliches Ziel verfolgen – das Zusammenbringen von mindestens zwei Branches. Die scheinbare Komplexität liegt aber definitiv auch an den kryptischen Beschreibungen, die für Rebase im Web zu finden sind.*

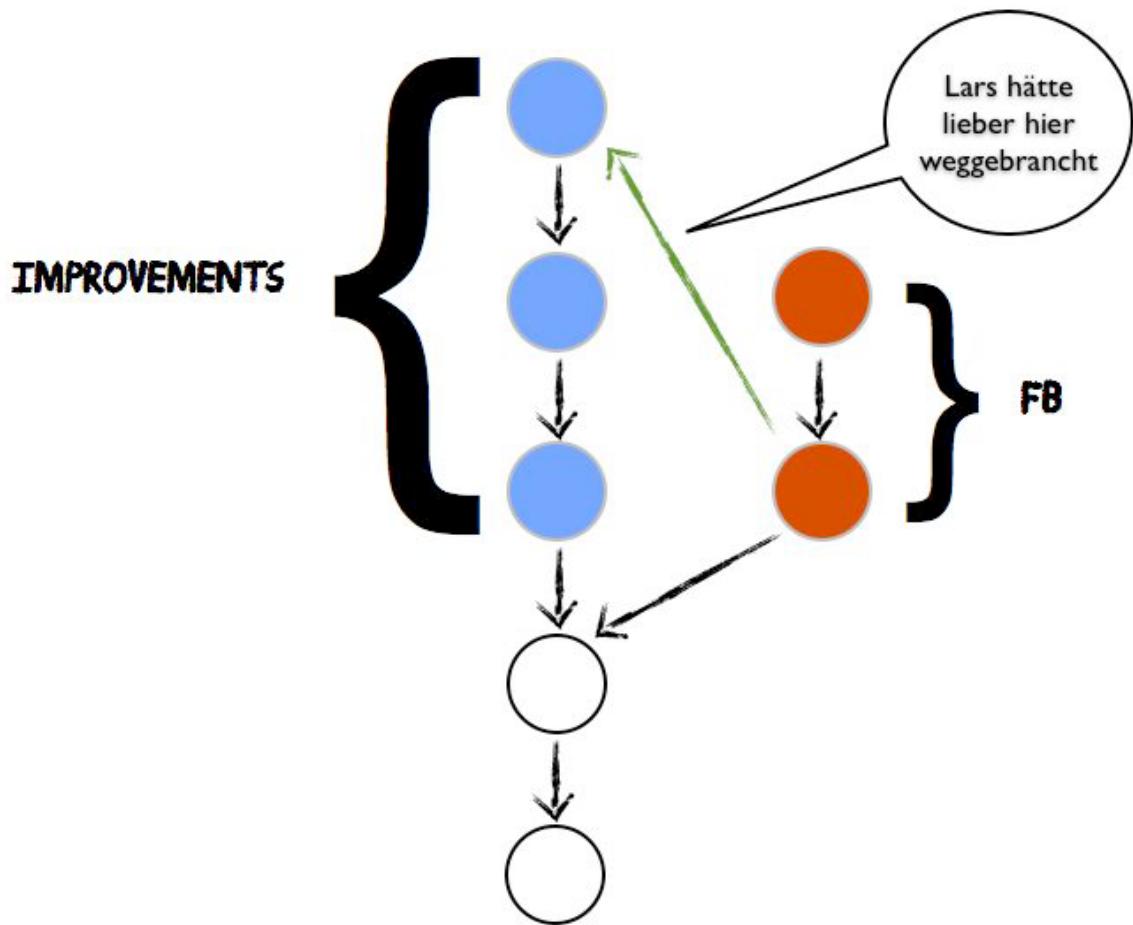
Forward-port local commits to the updated upstream head aus Git Rebase MAN Page

Rebase is recreating your work of one branch onto another. von
www.fiveminutes.eu

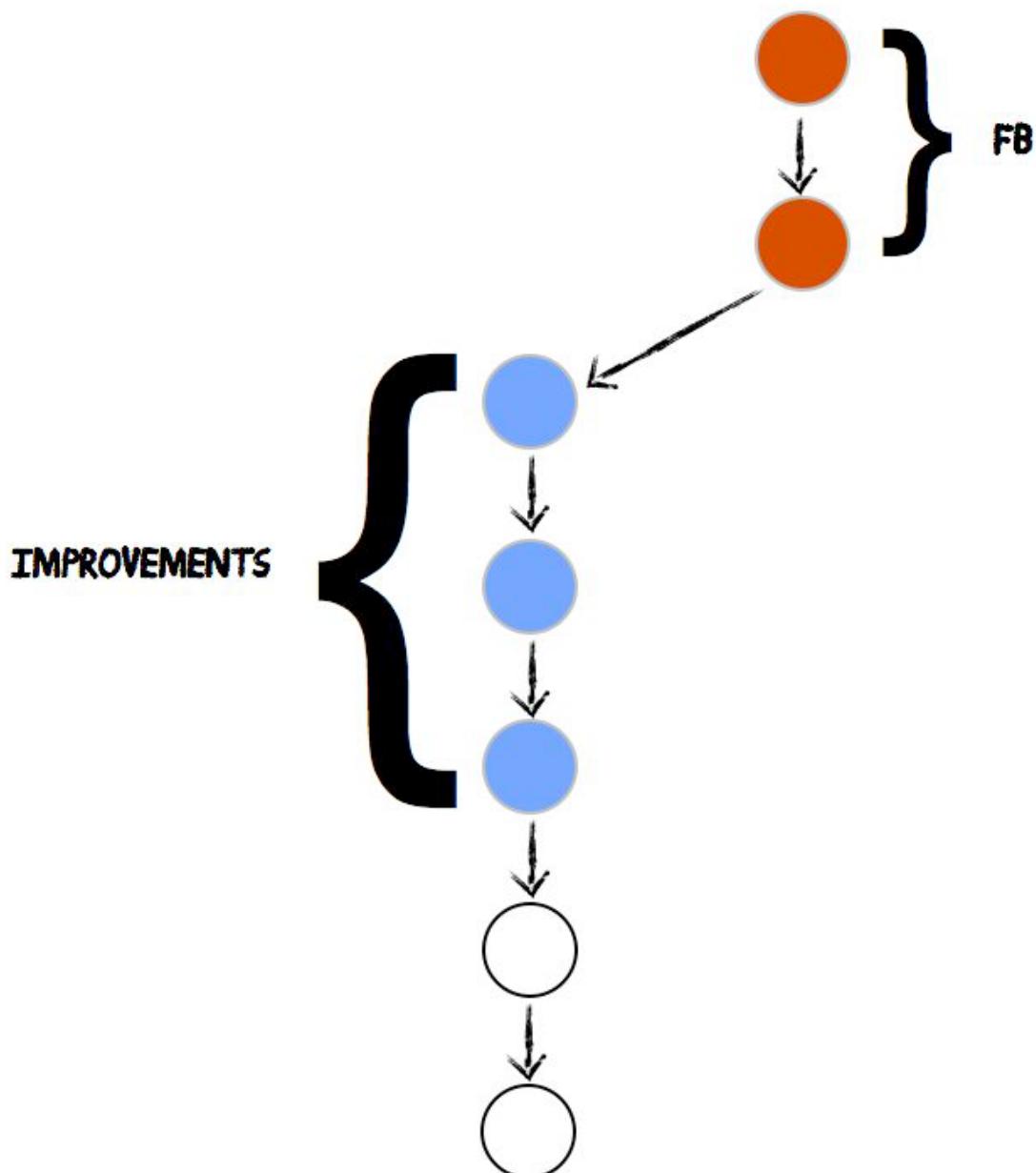
» Meine einfache Erklärung für Rebase ist diese: "Hätte ich meinen Branch doch nicht gestern sondern heute gezogen". Karl, nehmen wir an, Ich erzeuge einen neuen Feature-Branch vom master weg und arbeite auf diesem. Währenddessen arbeitest Du auf dem master weiter und machst einige wichtige Änderungen, die die Performance der Anwendung ernorm verbessern.



» Am nächsten Morgen denke ich mir, dass die Änderungen von dir wirklich praktisch wären. Eine Möglichkeit wäre zu mergen. Eigentlich denke ich mir aber – "Hätte ich den Feature Branch doch nicht schon gestern gezogen sondern erst heute". In diesem Fällen ist ein Rebase das Tool der Wahl.



» Wenn ich ein **Rebase** auf meinen Feature-Branch gegen den master mache, dann nimmt Git meine beiden orangen Commits zur Seite und holt die Commits vom master auf meinen Feature-Branch. Die beiden Branches sehen also kurzzeitig gleich aus. Anschließend nimmt Git die Commits, die es zuvor zur Seite gelegt hat und packt sie wieder oben drauf. Die lokalen Commits (port-local-commits aus der Beschreibung) sind also immer die obersten auf dem Branch nach dem Rebase.



» Zugegeben, das war alles sehr theoretisch, höchste Zeit dass wir wieder zurück an deinen Schreibtisch gehen und das Ganze mal praktisch ausprobieren. Ich habe noch einen weiteren Task, den wir zusammen bearbeiten können, das Feature-4811.

Übung

- Sie sollen das Feature 4811 implementieren.
- Erzeugen Sie einen neuen Feature-Branch `feature-4811`

- Erzeugen Sie eine neue Datei feature-4811.txt auf dem Branch mit beliebigem Inhalt.
- Erzeugen Sie einen Commit mit der Message **feature-4811 - done**
- Gehen Sie zurück auf den master
- Erzeugen Sie eine Datei master.txt mit beliebigem Inhalt
- Erzeugen Sie auch hier einen Commit
- Gehen Sie zurück auf Ihren Branch und führen Sie die beiden Branches zusammen

» Ok, legen wir los. Zunächst würde ich vorschlagen, wir erzeugen einen neuen Feature-Branch.

```
git checkout -b feature-4811
Switched to a new branch 'feature-4811'
```

» Dann implementieren wir das Feature. Keine Angst, es ist ganz einfach.

```
echo 'done' >> feature-4811.txt
git add feature-4811.txt
git commit -m "feature-4811 - done"
[feature-4811 a499d61] feature-4811 - done
 1 file changed, 1 insertion(+)
 create mode 100644 feature-4811.txt
```

» Anschließend gehen wir zurück auf den master. Denk dran, was wir machen möchten ist ein Update vom Feature-Branch gegen den Master. Um das machen zu können müssen wir auf dem master einige Commits machen.



Oft brauchen wir Commits um experimentieren zu können. Ein einfaches Skript ist beispielsweise folgendes und kann mit "git makeCommits <Anzahl Commits>" verwendet werden. Legen Sie dieses Skript am besten irgendwo in Ihrem Pfad ab und nennen es "git-makeCommits". Git sucht im Pfad nach Skripten die dem Muster "git-<Skriptname>" folgen. All diese Skripte können dann automatisch "git <skriptName>" aufgerufen werden und sehen wie native Git-Kommandos aus. Für das **makeCommits**-Skript wäre der Aufruf also beispielsweise "git makeCommits 4".

```
#!/bin/bash
```

```
for ((i=1;i<=$1;i++))  
do  
    echo "commit $i" >> file$i.txt  
    git add file$i.txt  
    git commit -am "committing file $i"  
done
```

» Wir wechseln also zurück auf den Master.

```
git checkout master  
Switched to branch 'master'  
makeCommits 5  
[master 47bbeff] committing file 1  
 1 file changed, 1 insertion(+)  
  create mode 100644 file1.txt  
[master 4242232] committing file 2  
 1 file changed, 1 insertion(+)  
  create mode 100644 file2.txt  
[master 7c31a49] committing file 3  
 1 file changed, 1 insertion(+)  
  create mode 100644 file3.txt  
[master f6137c2] committing file 4  
 1 file changed, 1 insertion(+)  
  create mode 100644 file4.txt  
[master af4c0d0] committing file 5  
 1 file changed, 1 insertion(+)  
  create mode 100644 file5.txt  
  
#show log  
git log --oneline  
af4c0d0 committing file 5  
f6137c2 committing file 4  
7c31a49 committing file 3  
4242232 committing file 2  
47bbeff committing file 1  
bb2d3b2 merged fb-4711  
817e460 Konflikt  
bebc4db Adjusted Readme  
ad261f2 Initial commit
```

» Wir haben jetzt zwei Möglichkeiten, die beiden Branches zusammenzubringen. Ein Merge würde so aussehen.

```
git checkout feature-4811
```

```
Switched to branch 'feature-4811'  
git merge master  
Merge made by the 'recursive' strategy.  
  file1.txt | 1 +  
  file2.txt | 1 +  
  file3.txt | 1 +  
  file4.txt | 1 +  
  file5.txt | 1 +  
 5 files changed, 5 insertions (+)  
create mode 100644 file1.txt  
create mode 100644 file2.txt  
create mode 100644 file3.txt  
create mode 100644 file4.txt  
create mode 100644 file5.txt
```

» Das Problem ist, wieder ist ein Merge-Commit entstanden. Wenn wir jedesmal einen Merge-Commit erzeugen, wenn wir ein Update gegen den master-Branch machen ist unsere Historie bald nicht mehr zu lesen.

```
git cat-file -p HEAD  
tree 6e82111762c37110f5c8a979164624c9a17c5ea7  
parent a499d6178cd0115fe92aaa169f708578fe0e10db ①  
parent af4c0d0f1bccb91fa59eae43a5323299ad47a776 ②  
author dilgerm <martin@effectivetrainings.de> 1389898629 +0100  
committer dilgerm <martin@effectivetrainings.de> 1389898629 +0100  
  
Merge branch 'master' into feature-4811
```

① MASTER Branch

② FEATURE-Branch

» Um die Historie sauber zu halten machen wir hier im Team alle Updates auf unseren Branches von entfernten Repositories grundsätzlich über Rebase und nicht über Merges.



Als Faustregel gilt: Updates auf dem Branch auf dem ich gerade arbeite mache ich über Rebase, alles andere über Merge.

» Am besten, wir gehen nochmal einen Schritt zurück und tun so, als ob wir den Merge nicht schon gemacht hätten.



Git erlaubt es, beliebig weit in der Zeit zurückzureisen mit **reset**.

```
git log --oneline  
1e2f263 Merge branch 'master' into feature-4811 ①  
af4c0d0 committing file 5 ②  
f6137c2 committing file 4  
7c31a49 committing file 3  
4242232 committing file 2  
47bbeff committing file 1  
a499d61 feature-4811 - done ③  
bb2d3b2 merged fb-4711  
817e460 Konflikt  
bebc4db Adjusted Readme  
ad261f2 Initial commit
```

```
#reset merge commit  
git reset --hard HEAD~1 ④  
HEAD is now at a499d61 feature-4811 - done
```

```
#log  
git log --oneline  
a499d61 feature-4811 - done ⑤  
bb2d3b2 merged fb-4711  
817e460 Konflikt  
bebc4db Adjusted Readme  
ad261f2 Initial commit
```

-
- ① Der Merge-Commit vereint die beiden Branches
 - ② Der oberste Commit vom master, der gemerged wurde
 - ③ Der oberste Commit auf dem Feature Branch ist viel weiter unten.
 - ④ Wir setzen den Commit-Zeiger des Branches einen Commit (HEAD~1) vom obersten Commit zurück.
 - ⑤ Der Merge ist Rückgängig gemacht.
» *Karl, nochmal, mit Reset beschäftigen wir uns später. Wir haben das Update vom master auf dem Feature-Branch jetzt rückgängig gemacht.*
» Ja, unglaublich wie einfach das ging.
» *Das kannst du so allerdings nur machen, wenn deine Arbeit noch nicht auf ein zentrales Repository gepusht wurde.*



Grundsätzlich gilt - Sie dürfen nur Ihre eigene Geschichte verändern, nicht die von anderen. Sobald Sie einen Commit veröffentlicht haben sollten Sie nicht mehr unbedacht mit **reset** oder **rebase** arbeiten.

» Solange du lokal arbeitest kannst du aber fast alles machen. Ok, jetzt versuchen wir das Update mit rebase.

```
git rebase master
First, rewinding head to replay your work on top of it...
Applying: feature-4811 - done
```

» Karl, schau dir das an, hier siehst du genau, was ein rebase eigentlich macht. Zunächst sagt git "rewinding HEAD". Das bedeutet, Git setzt den Zeiger auf den obersten Commit des Branches, gegen den der Rebase gemacht wird. In diesem Fall der master. Soweit verstanden?

» Ja, ich denke, das ist klar.

» Die beiden Branches sind dann also für einen ganz kurzen Moment identisch. Dann sagt Git "Applying: <Commit>". Die Commits, die du also auf dem Feature-Branch gemacht hast werden auf den neuen Stand vom master wieder aufgespielt.

» Ziemlich genial.

» Ja nicht wahr? Am besten, wir schauen uns kurz an, was genau jetzt passiert ist.

```
git log --oneline
f1aa978 feature-4811 - done
af4c0d0 committing file 5
f6137c2 committing file 4
7c31a49 committing file 3
4242232 committing file 2
47bbeff committing file 1
bb2d3b2 merged fb-4711
817e460 Konflikt
bebcb4db Adjusted Readme
ad261f2 Initial commit
```

» Siehst du? Wir haben eine saubere Historie. Es gibt aber ein Problem. Fällt dir etwas auf?

» Nein, für mich sieht das ziemlich gut aus.

» Wirf einen Blick auf den Hash-Wert des Commits "feature-4811", und vergleiche den mit dem Hash-Wert den der Commit zuvor hatte.

» Die unterscheiden sich?

» Genau, vorher hatte der Commit den Hash-Wert **a499d61**, jetzt hat der den Wert **f1aa978**. Durch den Rebase verändert sich alles.

» Ist das ein Problem?

» Das kommt darauf an, Karl. Stell dir vor, der Commit wäre bereits veröffentlicht, und ein anderer Entwickler im Team hat bereits ein Update gemacht. Stell dir weiterhin vor, du machst jetzt bei dir lokal ein rebase gegen einen anderen Branch. Der Hash-Wert des Commit verändert sich hierdurch. Jetzt habt ihr zwei Commits mit identischem Inhalt aber unterschiedlichen Hash-Werten. Git würde versuchen, diese beiden Commits zu mergen. Es könnte funktionieren, muss aber nicht. Dies ist meiner Erfahrung nach die größte Fehlerquelle bei der Arbeit mit Git. Ein rebase an der falschen Stelle kann zu den kuriosesten Fehlern führen und glaub mir, ich habe schon Entwickler weinen sehen, weil Sie nicht mehr weiter wussten. Hinzukommt, dass alles in Git sehr einfach wieder rückgängig gemacht werden kann. Das hast du vorher bereits an dem Merge gesehen. Das gilt für alles bis auf Rebase. Ein Rebase ist nicht mehr ohne weiteres Rückgängig zu machen, da du die commits veränderst. Wir haben hier im Team festgelegt, dass rebase die Strategie der Wahl ist, wenn es um Updates von entfernten Repositories geht. Aber bitte immer mit Bedacht. Falls Du dir nicht sicher bist, frag lieber einen Entwickler, denn die haben genau dasselbe wie Du durchgemacht.

» Wo wir gerade beim Thema update von entfernten Repositories sind. Vielleicht zeig ich dir einfach mal kurz, was alles möglich ist.

4.6. Remotes

» Wir hatten ja bereits kurz über Remote-Repositories gesprochen. Du erinnerst dich an die config-Datei in deinem .git-Verzeichnis?

» Ja klar.

```
[remote "origin"]
  url = https://github.com/dilgerma/effective-git-workshop
  fetch = +refs/heads/*:refs/remotes/origin/*
```

» Ich versuch das mal zu erklären, dann sehen wir auch gleich, ob ich es verstanden habe.

» Ein Remote-Repository kann irgendein Repository sein, das nicht mein aktuelles ist, richtig?

- » *Ja, das ist korrekt.*
- » Das Repository kann also auch auf dem gleichen Rechner und nur in einem anderen Verzeichnis sein?
- » *Genau.*
- » Jedes Repository hat einen festgelegten Namen. In diesem Fall "origin". Origin ist ein beliebiger Name, der aber meistens für das Haupt-Repository verwendet wird, richtig?
- » *Genau, du hast es verstanden.*
- » Jedes Repository kann aber nur ein Remote-Repository haben?
- » *Nein, überhaupt nicht. Git gibt hier überhaupt keine Einschränkungen vor. Stell dir vor, du hast dein Remote namens "origin". Das ist das Entwickler-Repository. Hier findet die Entwicklung statt. Es ist jetzt durchaus möglich, ein weiteres Repository "release" zu definieren, in das nur gespusht wird, wenn wir ein Release machen. Das ist nur ein Beispiel, wir machen das hier nicht so, weil für diesen Use-Case Branches verwendet werden, aber möglich ist alles.*
- » *Um dein Repository mit einem weiteren Remote-Repository zu verbinden machst du einfach folgendes.*

```
#bsp: git remote add backup .. /backup.git  
git remote add <Name> <URL>
```

- » Ok, soweit verstanden. Kannst du mir auch erklären, was das bedeutet?

```
fetch = +refs/heads/*:refs/remotes/origin/*
```

- » *Klar, das ist die sogenannte **Refspec**. Die Refspec definiert, wie genau die Repositories miteinander kommunizieren. Diese Refspec gibt nur an, dass wir Updates für alle lokalen Branches unter /refs/heads vom Repository "origin" updaten.*
- » Und was bedeutet **fetch**?
- » *Gute Frage! Wir haben noch gar nicht über die verschiedenen Befehle gesprochen, mit denen Du mit entfernten Repositories kommunizieren kannst.*

Fetch

» Ein Fetch ist die einfachste Remote-Operation in Git. Es bedeutet eigentlich nichts anderes, als das wir uns vorsorglich schonmal alle Änderungen aus dem Remote-Repository holen, um später damit arbeiten zu können. An unserer Working-Copy und am ausgecheckten Branch ändert sich hierdurch nichts.

» Interessant, gibt es eine vergleichbare Operation in Subversion?

» Nein, gibt es nicht, denn das ist eines der Vorteile von dezentralen Systemen. Wir haben alles lokal verfügbar. Durch ein fetch holen wir uns nur, was wir nicht sowieso schon haben. Wir haben bereits über das **objects**-Verzeichnis gesprochen, du erinnerst dich? Im **objects**-Verzeichnis speichert Git alle Objekte, also Commits, Trees, Blobs und Tags als binär-Daten. Wenn Du ein fetch machst, dann holt sich Git alle Objekte, die im Remote-Repository verfügbar sind aber noch nicht bei dir lokal. Nichts weiter, die Objekte liegen dann nur in deinem **objects**-Verzeichnis und du kannst sie jederzeit verwenden. Aber erst, wenn du sie brauchst, nicht vorher.

» Ein guter Use-Case für einen **Fetch** bei mir ist immer Freitagabend. Ich arbeite öfter mal am Wochenende. Oft stecke ich aber Freitagsabends kurz vor dem Wochenende noch mitten in einer Story, die ich umsetzen möchte. Da wir für Entwickler keine Möglichkeit anbieten, sich per VPN einzuhängen mache ich grundsätzlich, bevor ich ins Wochenende gehe nochmal ein **fetch** gegen das Remote-Repository. So habe ich alle Änderungen lokal am Rechner verfügbar, die bis zu diesem Zeitpunkt eingeccheckt waren.

» Überprüfen wir doch mal, ob sich in der Zwischenzeit etwas im Remote-Repository getan hat. Da wir bereits einige Stunden hier sind, sollten die ersten Commits bereits gemacht worden sein.

Übung

Simulieren Sie einen Commit auf dem entfernten Repository.

- Klonen Sie sich das Remote-Repository erneut in ein anderes Verzeichnis
- Erstellen Sie eine neue Datei remote.txt mit dem Inhalt "remote commit" auf dem master
- Committen Sie diese Datei

- Pushen Sie diese Datei mit "git push origin master"

» *Karl, der nächste Schritt besteht nun darin, das Update zu machen.*

```
git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0) ❶
Unpacking objects: 100% (3/3), done.
From https://github.com/dilgerma/effective-git-workshop
      ad261f2..b93516d  master       -> origin/master ❷
```

- ❶ Git zählt die Objekte, die remote- aber noch nicht lokal verfügbar sind.
- ❷ Git zeigt an, welcher Branch von welchem Commit (ad261f2) auf welchen Commit (b93516d) upgedatet werden würde.

» *Karl, Frage - kannst du mir sagen, wieso Git genau 3 Objekte geladen hat?*



Können Sie erklären, wieso genau 3 Objekte geladen wurden?

» Ha, das ist einfach. Im Remote-Repository war ein neuer Commit richtig? Du hast mir heute morgen erklärt, dass ein **Commit** immer aus mindestens 3 Teilen bestehen. Der **Commit** selbst, der **Tree** für das Verzeichnis und der **Blob** für die Datei. Also 3 Objekte.

» *Perfekt, du hast es wirklich verstanden. Der Remote-Commit hat den Hash-Wert b93516d. Schau doch mal in dein objects-Verzeichnis.*

```
cd .git/objects
cd b9
ls
3516dc1b3bda32ced75dd9c0883735e4b7ea64
```

» *Du siehst, der Commit ist lokal vorhanden. Mach nochmal ein log, damit wir sehen, ob sich etwas geändert hat.*

```
git checkout master
git log --oneline -n 1
af4c0d0 committing file 5
```

» Du siehst, der oberste Commit auf dem master ist nach wie vor **af4c0d0**. Lokal hat sich also nichts geändert. Willst du die Änderungen in deinem Workspace haben musst du sie aktiv mergen.

```
git merge origin/master
Merge made by the 'recursive' strategy.
 remote.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 remote.txt

#log
git log --oneline -n 1
bea3c24 Merge remote-tracking branch 'origin/master'
```

» Ein Update besteht also immer aus einer **fetch** und einer **merge**-Operation.

» Das heißt, ich muss jedesmal fetchen und mergen, wenn ich ein Update machen möchte?

» Das wäre ziemlich kompliziert, oder? Zunächst hatten wir ja definiert, dass wir Updates nicht mit **merge** sondern mit **rebase** machen, in Ordnung? Und die Entwickler von Git wissen natürlich auch, dass zwei Operationen für ein Update gegen das entfernte Repository ziemlicher Overhead ist. Deswegen gibt es das **pull**-Kommando.

Pull

» Das **pull**-Kommando verwenden wir generell, um Updates von einem entfernten Repository zu machen. Die Syntax ist identisch mit allen Git Remote Operationen. Ein **pull** kombiniert **fetch** und **merge** und macht intern nichts anderes als erst ein **fetch** gegen das Remote-Repository auszuführen und anschließend einen **merge** in den aktuell ausgecheckten Branch zu machen.

```
git pull origin master ❶
From https://github.com/dilgerma/effective-git-workshop
 * branch           master    -> FETCH_HEAD
 Already up-to-date.
```

git pull <Remote-Repository> <Branch>

» Ist unser Branch als Tracking-Branch konfiguriert brauchen wir nur folgendes.

```
git pull
```

Already up-to-date.

» Durch die Tracking-Branch Konfiguration ist die Angabe des Remote-Repositories und des Branches obsolet. Karl, du siehst verwirrt aus. Das wird dir aber nochmals klar, wenn du dir die config anschaust.

```
cat .git/config
```

```
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = false
[remote "origin"]
url = https://github.com/dilgerma/effective-git-workshop
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"] ❶
remote = origin ❷
merge = refs/heads/master ❸
```

❶❷❸ Tracking Branch Information

❷ Branch tracked welchen Remote?

❸ Branch tracked welchen Branch?

Übung

Erinnern Sie sich noch, wie ein Branch als **Tracking-Branch** konfiguriert wird?

» Aber du hast gesagt, wir verwenden **rebase** und nicht **merge** für updates?

» Korrekt, dazu kommen wir jetzt. Was du natürlich machen könntest wäre folgendes.

```
git rebase origin/master
```

» Damit würdest du deinen lokalen Branch gegen den master-Branch im Remote-Repository **origin** rebasen. Das Problem ist nur folgendes.

```
cat .git/refs/remotes/origin/master
```

b93516dc1b3bda32ced75dd9c0883735e4b7ea64

» *Der rebase macht kein Update vorher. Wir würden also gegen unseren lokalen Stand rebasen, was nicht unbedingt der letzte Stand aus dem Remote-Repository sein muss. Was fehlt ist ein fetch zuvor. Wir möchten aber nicht jedesmal ein fetch machen müssen, wenn wir ein Update machen möchten.*

» Gibt es denn einer Alternative?

» *Ja die gibt es, wir können auch für rebase das pull-Kommando verwenden.*

git pull --rebase origin master

» *Mit der Option --rebase macht der Pull statt fetch + merge ein fetch + rebase.*

» Aha, das klingt interessant, machen das alle im Team so?

» *Wir haben diese Option als Standard konfiguriert. Am besten machen wir das bei dir genauso. Dann sparst du dir, ständig diese Option mit anzugeben.*

```
git config --global branch.autosetuprebase always ❶
#for existing branches
git config branch.master.rebase true
```

» *Damit ist ein Pull immer auch ein Rebase. Für Branches die bereits existieren müssen wir das automatische rebase manuell aktivieren. Alle Branches die du von jetzt an neu erzeugst sind bereits auf rebase umgestellt.*

» Klasse, danke Lars.

» *Kein Problem, am besten du versuchst mal einen Rebase-Update gegen das Remote-Repository.*

```
git pull
First, rewinding head to replay your work on top of it...
Applying: Adjusted Readme
Applying: Konflikt
Using index info to reconstruct a base tree...
M README.md
Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
```

```
Failed to merge in the changes.  
Patch failed at 0002 Konflikt  
The copy of the patch that failed is found in:
```

```
When you have resolved this problem, run "git rebase --continue".  
If you prefer to skip this patch, run "git rebase --skip" instead.  
To check out the original branch and stop rebasing, run "git rebase --  
abort".
```

» Ich habe einen Merge-Konflikt?

» Ja, interessant oder? Kannst du dir erklären, wo der herkommt? Das ist übrigens genau die Stelle, wo sich die meisten Entwickler erstmal hilfesuchend umschauen und nicht weiter wissen.

» Tut mir leid, das verstehe ich nicht. Es sieht so aus, als wäre dies genau der gleiche Merge-Konflikt den wir bereits vorhin beim Merge auf den Feature-Branch gelöst haben?

» Sehr gut erkannt!

» Aber den haben wir doch bereits aufgelöst?

» Richtig, aber erinnere dich, was wir gerade besprochen haben. Wir funktioniert ein **rebase intern**?

» Ok, ich versuche das mal zu erklären. Du hast gesagt, Git nimmt meine Commits, die ich auf dem Branch gemacht habe und legt sie beiseite.

» Ja genau.

» Anschließend holt sich Git alle Commits aus dem entfernten Repository und setzt den Branch-Zeiger für meinen aktuellen Branch in **.git/refs/heads** auf den neuesten Commit des Branches im Remote-Repository gegen den ich den Rebase mache. Es werden also alle Commits Schritt für Schritt wieder hinzugefügt.

» Genau, du hast es fast. Mit dem **rebase** linearisieren wir die Historie. Die Merge-Commits verschwinden faktisch. Die Lösung des Merge-Konfliktes war aber in dem entsprechenden Merge-Commit. Dieser kann aber von **rebase** nicht verwendet werden. Da Git die Commits Schritt für Schritt wieder einspielt treten alle Konflikte erneut auf, die wir bereits aufgelöst haben.

» Ist das etwa immer so?

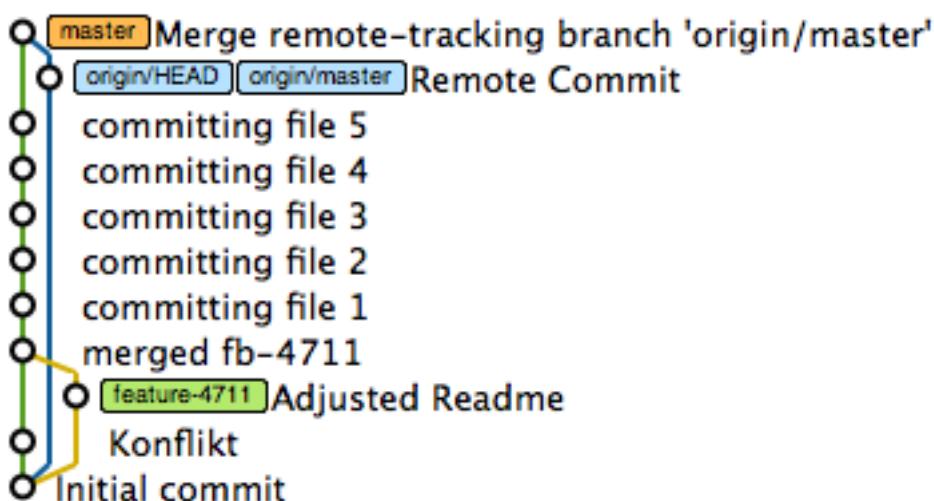
» Normalerweise ist das kein Problem. Vielleicht machen wir uns das Ganze nochmal klarer. Wir brechen den **rebase** am besten ab.

```
git rebase --abort
```



Mit **git rebase --abort** kann jeder Rebase abgebrochen werden, egal wieviel schon gemerged und verändert wurde. Nach dem **abort** ist der Status des Repository exakt wie vor dem Start des Rebase.

» Wenn Du dir jetzt die Historie anschaust.



» Du siehst den Commit "Adjusted Readme" und den zugehörigen Merge-Commit "merged-fb-4711"?

» Ja.

» Du siehst auch, dass der Merge-Commit bisher nur lokal verfügbar ist? Wenn wir jetzt einen **rebase** machen wird Git zunächst alle Commits aus **origin/master** zu uns lokal holen. Anschließend wird es alle Commits, die bisher nur lokal verfügbar sind wieder neu hinzufügen. Unser Commit "Adjusted Readme" würde also auf den Commit "Remote Commit" gesetzt werden. Der Merge-Commit "merged-fb-4711" verschwindet einfach. Da sich beim Rebase die Hash-Werte der beteiligten Commits ändern kann Git die Lösung des Merge-Konflikts nicht wiederverwenden, denn die gilt nur für die beiden Commits mit den alten Hash-Werten. Soweit klar?

» Ja, ich denke das habe ich verstanden.

» Sehr gut, es ist also zu erwarten, dass der Konflikt erneut auftritt und in diesem Fall müsstest Du den Konflikt erneut auflösen, was wir jetzt auch tun werden.



Würde Karl jetzt aber seine Änderungen pushen, könnte er den nächsten Rebase problemlos machen, da dann keine Commits ausschließlich lokal verfügbar sind und beim **rebase** keine Probleme auftreten.

Übung

- Machen Sie ein Update gegen origin
- Lösen Sie den Merge-Konflikt
- Bringen Sie den rebase zu Ende.

```
git pull
First, rewinding head to replay your work on top of it...
Applying: Adjusted Readme
Applying: Konflikt
Using index info to reconstruct a base tree...
M README.md
Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Failed to merge in the changes.
Patch failed at 0002 Konflikt
```

```
#merge
git mergetool
```

» *Ganz wichtig, Karl. Wir sind noch nicht fertig. Stell dir vor wir wandern jetzt in der Historie von unten nach oben. Wir sind jetzt genau an der Stelle, wo der Merge-Konflikt auftritt.*

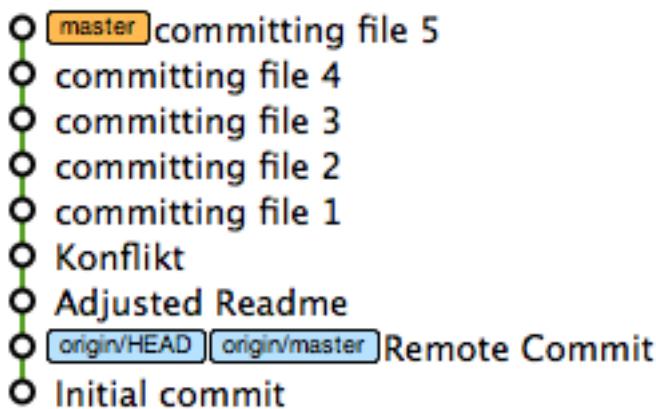
```
git log --oneline
5b90802 Adjusted Readme
b93516d Remote Commit
ad261f2 Initial commit
```

» *Wir müssen Git jetzt explizit sagen, dass wir fertig sind und weiter machen möchten.*

```
git rebase --continue
```

```
Applying: Konflikt
Applying: committing file 1
Applying: committing file 2
Applying: committing file 3
Applying: committing file 4
Applying: committing file 5
```

» Das Update ist damit vollständig und die Historie sieht so aus.



» Ich kann gut verstehen, wenn dir das alles noch ein wenig kompliziert vorkommt, aber ich verspreche dir, das wird sich bessern. Für heute ist es nur wichtig, dass du das Prinzip verstehst.

» Das habe ich verstanden, ich werde mir das dann nochmal in Ruhe anschauen. Was ich mir merke ist auf jedenfall, wenn ich ein **rebase** gegen einen Branch mache, dann kann es passieren, dass Merge-Konflikte die schon gelöst sind erneut auftreten.

» Genau, allerdings nur in seltenen Fällen. Wir haben jetzt schon zwei Features fertig implementiert. Karl, höchste Zeit, dass Du deine Sachen auch den anderen Entwicklern zur Verfügung stellst. Die Operation hierfür ist **push**.

4.7. Push

» Bevor wir lange reden, würde ich vorschlagen, du pushst Deine Commits zunächst.

```
git push origin master
Counting objects: 23, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), 1.82 KiB | 0 bytes/s, done.
Total 21 (delta 6), reused 0 (delta 0)
```

```
To https://github.com/dilgerma/effective-git-workshop  
b93516d..6e5a1d7 master -> master
```

» *Ab jetzt stehen deine Commits allen Entwicklern zur Verfügung. Es gibt aber einige Besonderheiten zu beachen beim Push. Zunächst stellt Git sicher, dass du nur pushen kannst, wenn Du zuvor ein Update gemacht hast. Ist dein Stand auf dem lokalen Branch älter als der auf dem Remote-Repository, wird Git den Push verbieten.*

» Ok, das ist bei Subversion nicht anders, ich hätte eigentlich erwartet, dass Git damit umgehen kann.

» *Nein, das kann es nicht. Stell dir das so vor. Der Branch, also beispielsweise der master im Remote-Repository zeigt auf einen bestimmten Commit, seinen HEAD. Dein lokaler Branch genauso. Ein push macht nichts anderes, als alle Objekte aus deinem objects-Verzeichnis auf den Server zu kopieren und dann den HEAD-Zeiger des Branches neu zu setzen.*

» Ok, soweit hatten wir das alles schon besprochen.

» *Genau, wenn du aber einen push ausführst obwohl im Remote-Repository bereits neuere Commits da sind würdest du den HEAD-Zeiger überschreiben und die neuen Commits verschwinden einfach, weil diese in deiner lokalen Historie nicht vorkommen.*

» Ach so, ja stimmt. Gut, das ist ja sogar gut das Git das nicht kann.

» *Oh, Git kann das sehr wohl. Das nennt sich forced push. Vielleicht versuchen wir das einfach mal?*

» Bis du sicher?

» *Ja, keine Angst, wir verwenden ein Spielzeug-Repository hierfür. Das Szenario lässt sich ganz einfach bauen. Wir definieren uns lokal auf dem Rechner ein zweites Remote-Repository das den gleichen Stand wie unser Repository hat. Dann setzen wir bei uns den Commit-Zeiger einige Commits in die Vergangenheit und versuchen zu pushen.*

Übung

- Erzeugen Sie lokal ein leeres Repository
- Deklarieren Sie dieses Repository als Remote in Ihrem Entwickler-Repository

- Pushen Sie Ihren aktuellen Stand in dieses Repository
- Setzen Sie Ihren Commit-Zeiger auf HEAD~3 (*git reset --hard*)
- Versuchen Sie erneut zu pushen
- Was geschieht?

» *Ok, Karl, versuchen wir das zusammen?*

```
mkdir testremote.git
cd testremote.git/
#init bare repository
git init --bare
Initialized empty Git repository in /Users/martindilger/development/git/
workshops/testremote.git/
#print directory
ls
HEAD
branches
config
description
hooks
info
objects
refs
```



Wichtig, Remote-Repositories sollten immer als Bare-Repository mit **--bare** deklariert werden.

» Kannst du mir den Unterschied zwischen einem Bare- und einem normalem Repository erklären?

» *Besser du nennst sie Bare- und Entwickler-Repository. Ein Bare-Repository hat keine Working-Copy. Das haben wir auch gerade gesehen. Das was du in deinem Entwickler-Repository im .git-Verzeichnis findest liegt bei einem Bare-Repository direkt im Root-Verzeichnis. Es gibt also keine Möglichkeit, Dateien zu bearbeiten oder zu Committen. Das Bare-Repository ist einzige und allein da um darauf zu pushen oder davon zu pullen.*

» Interessant, können auch Entwickler-Repositories Remotes sein oder geht das sowieso nur mit Bare-Repositories?

» *Gute Frage! Auch Entwickler-Repositories können Remotes sein. Das macht beispielsweise Sinn, wenn sich zwei Entwickler gegenseitig als Remote deklariert haben und sie direkt zusammenarbeiten. Es gibt aber eine wichtige Einschränkung. Bevor wir uns um unser echtes Remote-Repository kümmern machen wir vielleicht einen kleinen Ausflug und erzeugen uns ein weiteres Remote-Repository als Entwickler-Repository.*

```
mkdir dev-remote
cd dev-remote
git init ❶

#connect
git remote add dev-remote ../dev-remote/
git push dev-remote master
Counting objects: 28, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (28/28), 2.34 KiB | 0 bytes/s, done.
Total 28 (delta 7), reused 0 (delta 0)
remote: error: refusing to update checked out branch: refs/heads/master
! [remote rejected] master -> master (branch is currently checked out)
```

❷❸ Achtung, diesmal ohne --bare

» *Das Entwickler-Repository verweigert den Push auf den aktuell ausgecheckten Branch. Kannst du mir sagen warum?*

» Ich denke das Problem ist, dass der Entwickler wahrscheinlich gerade auf diesem Branch arbeitet, oder?

» *Ja, das geht schon in die richtige Richtung.*

» Würden wir jetzt auf den master pushen auf dem der Entwickler aktuell arbeitet, würden wir ihm seine bereits lokal gemachten Commits einfach überschreiben, weil wir den Commit-Zeiger einfach auf unseren obersten Commit setzen würden der aber bereits sehr alt sein kann, richtig?

» *Perfekt, Karl. Ich hätte es selbst nicht besser erklären können. Genau aus diesem Grund brauchen wir Bare-Repositories als Remotes, weil für Bare-Repositories diese Einschränkung nicht gilt.*

» Danke, das klingt einleuchtend, wir wollten uns aber eigentlich mit **forced-pushes** beschäftigen, richtig?

» Ja genau, lass uns das Entwickler-Repository wieder aus der Remotes-Liste löschen und stattdessen das Bare-Repository deklarieren.

```
#remove dev remote  
git remote rm dev-remote  
#add bare remote  
git remote add test-remote ../testremote.git
```

» Anschließend pushen wir unseren aktuellen Stand auf dieses Repository.

```
git push test-remote  
Counting objects: 28, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (20/20), done.  
Writing objects: 100% (28/28), 2.34 KiB | 0 bytes/s, done.  
Total 28 (delta 7), reused 0 (delta 0)  
To ../testremote.git  
 * [new branch] master -> master
```

» Soweit so gut und nichts neues, richtig?

» Ja, das kenn ich bereits.

» Was wir jetzt machen ist, wir gehen ein wenig zurück in die Vergangenheit.

```
git log --oneline  
6e5a1d7 committing file 5  
0d01e62 committing file 4  
22ea185 committing file 3  
8b46f14 committing file 2  
932e981 committing file 1  
4ef00cd Konflikt  
5b90802 Adjusted Readme  
b93516d Remote Commit  
ad261f2 Initial commit
```

```
#reset  
git reset --hard HEAD~3  
HEAD is now at 8b46f14 committing file 2
```

```
#log again  
git log --oneline  
8b46f14 committing file 2  
932e981 committing file 1
```

```
4ef00cd Konflikt  
5b90802 Adjusted Readme  
b93516d Remote Commit  
ad261f2 Initial commit
```

» Mit Hilfe von **git reset** gehen wir vom HEAD, das ist der jeweils oberste Commit in einem Branch, zum angegebenen Commit.



git reset muss mit Vorsicht verwendet werden.

» Karl, es gibt fast unzählige Varianten wie man für **reset** angeben kann, wohin man resetteten möchte. Wir haben schon gesehen, **HEAD~3** geht vom obersten Commit 3 Commits zurück. Wir hätten das genauso gut als **HEAD^^^** schreiben können, oder aber wir hätten direkt mit dem Hash-Wert des Commits arbeiten können.

» Ok, das schlage ich dann in der [Dokumentation⁹](#) nochmal nach. Kann ich jetzt pushen?

» Ja, versuch es bitte.

```
git push test-remote  
To ../testremote.git  
! [rejected]           master -> master (non-fast-forward)  
error: failed to push some refs to ' ../testremote.git '
```

» Genau, hier siehst du es sehr schön. Das ist ein sogenannter **Non-Fast-Forward-Push**. Das bedeutet nichts anderes als das auf dem Remote-Branch Commits liegen, die wir lokal noch nicht haben. Das macht auch Sinn, weil wir lokal mit **reset** einige Commits zurückgegangen sind. Wir simulieren damit praktisch, dass jemand Commits auf den Master gepusht hat, die wir bei uns noch nicht vorliegen haben. In diesem Fall ist ein Update notwendig. Oder in ganz, ganz seltenen Fällen ein **forced-push**. Ich zeige dir das jetzt nur, damit du es mal gesehen hast. Bitte immer vorher mit einem Kollegen absprechen.

```
git push -f test-remote master  
Total 0 (delta 0), reused 0 (delta 0)  
To ../testremote.git  
+ 6e5a1d7...8b46f14 master -> master (forced update)
```

⁹ <https://www.kernel.org/pub/software/scm/git/docs/git-rev-parse.html>

» Ein **forced-push** mit der Option **-f** überschreibt den Commit-Zeiger im Remote-Repository ohne Rücksicht. Es gibt nur ganz wenige Fälle, in denen das Verhalten gewünscht ist.

» Ist das nicht ganz schön gefährlich?

» Genau, das ist es. Bei uns sind in allen Remote-Repositories **forced-pushes** generell ausgeschaltet und werden nur bei Bedarf aktiviert. Per Default sind forced-pushes leider erlaubt und müssen manuell deaktiviert werden.

```
git config receive.denyNonFastForwards true
```

» Aber haben wir jetzt nicht alle unsere Commits verloren?

» Erstmal ja, denn wir haben unser Repository lokal zurückgesetzt und jetzt auch das Remote-Repository überschrieben. Auf den ersten Blick gibt es keine Möglichkeit mehr, die fehlenden 3 Commits wieder zurückzubekommen, richtig?

Aus dem Projektalltag

Es gab in einem Projekt einmal einen Entwickler, der sich immer scheute, mit anderen Entwicklern zu kommunizieren. Diesem Entwickler wurde vom Product-Owner ein Task zugewiesen. Der Entwickler war hoch motiviert und hatte die Aufgabe schnell umgesetzt. Der Product-Owner entschied sich aber, dass die Änderung zunächst nicht zurück auf den **master** gebracht werden sollte, da sie den Kunden erst in ein paar Monaten zur Verfügung gestellt werden sollte. Der Entwickler beschäftigte sich also in der Zwischenzeit mit anderen Aufgaben und so gingen einige Wochen ins Land.

Irgendwann, der Entwickler hatte natürlich schon längst vergessen, dass es diesen Branch überhaupt noch gab, kam vom Product-Owner die Ansage, den Branch doch jetzt bitte zurückzuführen.

Der Entwickler (**mit git zwar vertraut, aber nicht erfahren**) versuchte den **master** in diesen Branch zu mergen. Der **master** war aber in der Zwischenzeit so weit fortgeschritten, dass sehr viele Merge-Konflikte entstanden. Leider war genau an der Stelle, an der der Entwickler seine Änderungen gemacht hatte ein Refactoring vorgenommen worden. Auch ein Push auf den **master** funktionierte natürlich nicht, da dies schon lange kein **fast-forward**-Push mehr war.

Üblicherweise wäre das Vorgehen, jetzt schnell Hilfe zu holen und gemeinsam die Merge-Konflikte aufzulösen.

Nicht so dieser Entwickler - nachdem klar war, dass die Merge-Konflikte nicht ohne Hilfe aufzulösen waren und ein Push nicht funktionieren würde wählte der Entwickler die drittbeste Option - **die MAN-Page für git push**.

```
git push --help
```

Nach einem Stöbern entdeckte der Entwickler hier die **--force**-Option.

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. This flag disables these checks. [...]

Weiter hat er wahrscheinlich nicht gelesen, denn der nächste Satz sagt:

and can cause the remote repository to lose commits; use it with care.

Der Entwickler sah offensichtlich das Licht am Ende des Tunnels.

```
git push -f origin feature/4711-great-customer-feature:master  
+ f71c5b9...109e0ef feature/task-4711-new-great-feature -> master  
(forced update)
```

Zufrieden ging der Entwickler in die Mittagspause, mit dem Wissen, dass die Änderungen seines Branches jetzt auf dem master verfügbar sein würden.

Einige Minuten später gingen die ersten erstaunten Ausrufe durch die Gänge. Builds auf allen CI-Servern fingen an fehlzuschlagen. Integrations-Tests meldeten Fehler an unterschiedlichsten Stellen der Applikation.

Das Albtraum-Szenario in jedem Projekt - mit seinem **Forced Push** hatte der Entwickler ohne es zu ahnen die Arbeit von fast 6 Wochen überschrieben. Der **master** war auf einen alten Stand zurückgesetzt und alles was bis dorthin entwickelt wurde einfach überschrieben.

Natürlich konnte der ursprüngliche Zustand des Repositories wieder hergestellt werden. Je später dieser fatale Fehler allerdings entdeckt worden wäre, desto komplizierter wäre ein Recovery gewesen.



Verbieten Sie **Forced Pushes** grundsätzlich.

4.8. All is Lost - Reflog

» Keine Sorge Karl, Git hat ein Sicherheitsnetz genau für diese Fälle. Das sogenannte **Reflog**. Das Reflog ist quasi ein Logbuch über alle Aktionen, die du in deinem lokalen Repository durchgeführt hast. Schau dir am besten mal dein Reflog an.

```
git reflog
8b46f14 HEAD@{2}: reset: moving to HEAD~3 ①
6e5a1d7 HEAD@{3}: rebase finished: returning to refs/heads/master ②
6e5a1d7 HEAD@{4}: rebase: committing file 5
0d01e62 HEAD@{5}: rebase: committing file 4
22ea185 HEAD@{6}: rebase: committing file 3
8b46f14 HEAD@{7}: rebase: committing file 2
932e981 HEAD@{8}: rebase: committing file 1
4ef00cd HEAD@{9}: rebase: Konflikt
[....]
```

① Dies ist der Stand nach dem **reset**

② Dies war der Stand vor dem **reset**

» Das Einzige was wir brauchen um unseren alten Stand wieder herstellen zu können sind die Hash-Werte der verlorenen Commits. Git löscht keine Daten. Nur weil die Commits nicht mehr in der Historie sind, sind sie trotzdem noch eine ganze Weile verfügbar.



Git führt regelmässig **Garbage Collections** durch. Commits die nicht verankert sind, also keine Parents haben sind nach einer Garbage-Collection tatsächlich unwiderbringlich verloren.

» Karl, du siehst dass der Hash-Wert des Commits vor dem **reset** "6e5a1d7" war. Den alten Stand wieder herzustellen ist jetzt ganz einfach.

Übung

- Stellen Sie den alten Stand im Repository wieder her.
- Bringen Sie das Remote-Repository wieder auf den alten Stand

```
git reset --hard 6e5a1d7
HEAD is now at 6e5a1d7 committing file 5
```

```
#log
git log --oneline
6e5a1d7 committing file 5
0d01e62 committing file 4
22ea185 committing file 3
8b46f14 committing file 2
932e981 committing file 1
4ef00cd Konflikt
5b90802 Adjusted Readme
b93516d Remote Commit
ad261f2 Initial commit

#recover remote
git push test-remote
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 864 bytes | 0 bytes/s, done.
Total 9 (delta 2), reused 0 (delta 0)
To ../testremote.git
  8b46f14..6e5a1d7 master -> master
```

» Wow, das ist Klasse, ich kann mir gut vorstellen, dass dieser Tipp schon oft für zu Seufzern der Erleichterung geführt hat.

» *Ja, das Reflog als Recovery-Tool solltest Du dir wirklich merken. Man braucht es nicht jeden Tag, aber wenn man es braucht ist man jedesmal heilfroh, dass es da ist.*

4.9. Best Practices

» *In Ordnung Karl, bist du noch aufnahmefähig?*

» Ja klar, ich finde das alles total interessant.

» *Sehr gut, wir haben auch noch einiges vor. Bevor wir dazu kommen, wie wir hier im Team mit Branches arbeiten und wie wir unsere Releases machen würde ich dir gern noch einige Best-Practices zeigen, die sich einfach im Lauf der Zeit herauskristallisiert haben.*

» Super, ich bin ganz Ohr.

Interactive Rebase

» *Du hast schon gesehen, dass Git dir wirklich viele Möglichkeiten bietet, Commits lokal zu verändern.*

» Ja, das ist wirklich Klasse.

» *Es geht sogar noch besser, denn jeder Entwickler hier bei uns im Team sollte das Ziel haben, eine möglichst saubere Historie zu hinterlassen. Wir wollen es allen so einfach wie möglich machen zu verstehen, was wann von wem und wieso implementiert wurde.*

» Da ich bin voll und ganz bei dir, Lars.

» *Zumindest mir geht es aber oft so, dass ich meine Zwischenergebnisse gerne sichern möchte. Wozu haben wir schliesslich ein dezentrales System?*

» Ok, ich verstehe. Das war etwas, was mich bei Subversion immer gestört hat. Ich hatte nie die Möglichkeit, einen Stand quasi einzufrieren ohne ihn gleich committen zu müssen.

» *Ha, siehst du! Genau das meine ich, mit Git kannst du lokal so viele Commits machen wie du möchtest. Dein lokales Repository gehört allein dir und niemand kann sehen, was du für Experimente machst.*

» Ja schon, aber das Problem ist, dass ich manchmal auch gerne einen unfertigen Stand einfrieren möchte, weil ich vielleicht gerne etwas ausprobieren möchte. Ich will aber nicht, dass dieser Stand dann später im Repository als Commit erscheint. Commits sollten ja immer funktionsfähig sein, oder?

» *Ja, da hast du vollkommen recht. Trotzdem sollst und darfst du lokal so viel committen wie du nur möchtest. Es gibt eine schöne Regel - **Commit often, Polish later, Push once.***



Commit often, Polish later, Push once.

» Klingt interessant, kannst du das näher ausführen?

» *Erinnerst du dich an das Skript, das wir vorhin verwendet haben um automatisch eine Menge von Commits zu erzeugen als wir über Rebase gesprochen haben? Das machen wir jetzt einfach nochmal. Erzeuge doch mal bitte so 10 Commits in deinem Repository mit beliebigem Inhalt.*

```
git log --oneline
8104dbc committing file 10
66e8695 committing file 9
```

```
fe2164f committing file 8
c3dd9d3 committing file 7
c772573 committing file 6
976bd5a committing file 5
2c8772a committing file 4
fcb8147 committing file 3
b1e3c66 committing file 2
696c994 committing file 1
6e5a1d7 committing file 5
0d01e62 committing file 4
22ea185 committing file 3
8b46f14 committing file 2
932e981 committing file 1
```

» Nehmen wir an, Commit "file 10" und "file9" sowie "file 5", "file 4" und "file 3" sind nur Zwischenstände. Die würden wir also gerne zusammenfassen.

Nehmen wir weiterhin an, Commit "file 2" ist zwar in Ordnung, soll aber eigentlich im Nachhinein eine andere Commit-Message haben.

Ok, wie würdest Du das mit Subversion machen?

» Grins Machst du Witze?

» Ja.

Das was wir jetzt machen werden machst du typischerweise kurz bevor du deinen Stand pushen möchtest. **Commit often** hast du bereits getan, jetzt sind wir quasi im **Polish later**-Schritt. Am besten wir machen es einfach der Reihe nach. Zunächst fassen wir die Commits "file 10" und "file 9", also die obersten beiden Commits zusammen. Das Tool was wir hierfür brauchen ist ein **interactive rebase**.

```
git rebase -i HEAD~2 ①
```

» Die Notation **HEAD~2** kennst du schon. Damit geben wir an, dass wir die letzten beiden Commits bearbeiten möchten. Ein **interactive rebase** ist quasi ein rebase eines Branches mit sich selbst. Wir haben vorhin darüber gesprochen, wie ein rebase funktioniert.

» Ja genau, Git legt die lokalen Commits zur Seite, macht ein Update und packt die zur Seite gelegten Commits dann einfach wieder oben auf.

» Exakt, das funktioniert nicht nur mit entfernten Repositories, sondern auch mit dem Branch selbst. Du kannst quasi einen Rebase auf einem Branch gegen sich selbst machen.

» Wow, klingt ziemlich abgefahren.

» *Ja vielleicht, ist aber eines der besten Tools, die Git zu bieten hat. Schau dir doch die Ausgabe des interactive rebase mal an.*

```
pick 66e8695 committing file 9
pick 8104dbc committing file 10

# Rebase fe2164f..8104dbc onto fe2164f
#
# Commands:
#   p, pick = use commit ①
#   r, reword = use commit, but edit the commit message ②
#   e, edit = use commit, but stop for amending ③
#   s, squash = use commit, but meld into previous commit ④
#   f, fixup = like "squash", but discard this commit's log message ⑤
#   x, exec = run command (the rest of the line) using shell ⑥
```

①① Behalte den Commit so wie er ist. → Default.

② Commit soll erhalten bleiben, aber die Commit-Message kann verändert werden.

③ Git wird anhalten und bietet die Möglichkeit, Dateien in der Working-Copy zu editieren.

④ Bietet die Möglichkeit, zwei aufeinanderfolgende Commits zusammenzufassen. Die Commit-Message bei beiden bleibt erhalten.

⑤ Ähnlich wie Squash, jedoch bleibt nur die Commit-Message des letzten Commits erhalten.

⑥ Gibt erlaubt es, ein Skript auszuführen. Wird nur selten gebraucht.

» *Du siehst, Git öffnet dir einen Editor und ganz oben stehen die beiden betroffenen Commits 66e8695 und 8104dbc. Vor jedem Commit siehst du das Wort "pick" geschrieben. Solange dort "pick" steht, bleibt der Commit so erhalten wie er ist. Du siehst aber weiter unten die verschiedenen Optionen.*

» Ich glaube ich verstehe, Git fordert uns quasi auf, festzulegen, was mit den Commits geschehen soll?

» *Ja, das kann man so sehen. Wir möchten "file 10" und "file 9" zusammenfassen. Wie würdest Du das jetzt machen?*

» Ich würde im Editor folgendes ändern.

```
pick 66e8695 committing file 9
```

```
s 8104dbc committing file 10
```

» Perfekt, ich bin beeindruckt, Karl. Dadurch dass du für den Commit "file 10" die Option **pick** im Editor auf **s** änderst, wird Git diesen Commit durch den Rebase mit "file 9" verschmelzen. Da du dich für einen **Squash** entschieden hast, bleiben beide Commit-Messages erhalten.

» Ich habe den rebase jetzt gestartet. Es geht nochmal ein Editor auf?

```
Rebasing (2/2)
```

```
# This is a combination of 2 commits.  
# The first commit's message is:
```

```
committing file 9
```

```
# This is the 2nd commit message:
```

```
committing file 10
```

» Genau, Git zeigt dir an, was passieren wird. Für einen Squash sind wir bereits fertig. Wir könnten jetzt aber auch die Commit-Messages noch beliebig ergänzen und ändern, indem wir einfach beispielsweise eine weitere Zeile hinzufügen.

```
git log -n 1
```

```
Commit: bca4d8a021ef805d1403479ba31bd770eeafb9a3  
Author: dilgerm <martin@effectivetrainings.de>  
Date:   (27 minutes ago) 2014-01-17 17:17:18 +0100  
Subject: committing file 9
```

```
committing file 10
```

```
Eine neue Zeile 3
```

» Wow, das ist wirklich praktisch, kann ich den zweiten Fall versuchen?

» Ja klar, leg los, wir möchten jetzt die Commits "file 5", "file 4" und "file 3" zusammenfassen.

Übung

Fassen Sie 3 beliebige Commits in Ihrer Historie zusammen.

```
git rebase -i HEAD~10
pick 696c994 committing file 1
pick b1e3c66 committing file 2
pick fcb8147 committing file 3
pick 2c8772a committing file 4
pick 976bd5a committing file 5
pick c772573 committing file 6
pick c3dd9d3 committing file 7
pick fe2164f committing file 8
pick bca4d8a committing file 9
```

» Ja, ich sehs schon, das muss ich nur ändern in folgendes.

```
git rebase -i HEAD~10
pick 696c994 committing file 1
pick b1e3c66 committing file 2
pick fcb8147 committing file 3
f 2c8772a committing file 4
f 976bd5a committing file 5
pick c772573 committing file 6
pick c3dd9d3 committing file 7
pick fe2164f committing file 8
pick bca4d8a committing file 9
```

» Genau, Karl, sehr gut. Der Commit "file 3" muss erhalten bleiben, die anderen beiden sollen einfach zusammenschmelzen.

```
git log --oneline
eb76c89 committing file 9
7d3b543 committing file 8
15e4590 committing file 7
bbfeb8d committing file 6
3656b4d committing file 3 ①
b1e3c66 committing file 2
696c994 committing file 1
```

① Die Commits "file 4" und "file 5" sind im Commit "file 3" aufgegangen.

» Ok, Karl, zuletzt wollen wir noch die Commit-Message von Commit "file 2" ändern und nur diese. Welche Option wäre hierfür die richtige?

Übung

Welche Interactive Rebase Option erlaubt das Ändern einer Commit Message wobei der Commit vollständig erhalten bleibt?

```
git rebase -i HEAD~7
pick 696c994 committing file 1
pick b1e3c66 committing file 2
pick 3656b4d committing file 3
pick bbfeb8d committing file 6
pick 15e4590 committing file 7
pick 7d3b543 committing file 8
pick eb76c89 committing file 9
```

» Ich glaube, das muss ich jetzt ändern in folgendes.

```
git rebase -i HEAD~7
pick 696c994 committing file 1
r b1e3c66 committing file 2
pick 3656b4d committing file 3
pick bbfeb8d committing file 6
pick 15e4590 committing file 7
pick 7d3b543 committing file 8
pick eb76c89 committing file 9
```

» Genau, Karl. Du hast es verstanden, **r** ist die richtige Wahl für **reword**. Damit änderst Du nur die Commit-Message, der Commit bleibt erhalten.

» Aha, es öffnet sich wieder der zweite Editor, wie bei Squash.

» Genau, das Prinzip ist dasselbe. Hier kannst du jetzt die Message beliebig ändern.

```
git log --oneline
85df518 committing file 9
f0e3d46 committing file 8
1608f8e committing file 7
2dcb0f3 committing file 6
7ef9186 committing file 3
cedf325 committing file 2 - changed by Karl in interactive rebase
696c994 committing file 1
```

```
6e5a1d7 committing file 5  
0d01e62 committing file 4  
22ea185 committing file 3  
8b46f14 committing file 2  
932e981 committing file 1
```



Achten Sie darauf, dass sogar beim Ändern der Commit-Message sich der Hash-Wert des Commits vollständig verändert.

» Wow, Lars, ich bin begeistert. **Interactive Rebase** ist eine wirklich tolle Sache.

» *Ja, für mich das wichtigste Tool, dass ich täglich sehr oft verwende. Mein typischer Workflow ist der, pass auf.*

```
#hack,hack,hack  
#Stand ist gut, also commit  
git add .  
git commit -m "First draft"  
  
#hack, hack, hack  
#Stand ist wieder gut  
git add .  
git commit -m "egal"  
git rebase -i HEAD~2 ❶  
  
#hack, hack, hack  
#Idee war gut, aber nicht brauchbar, also zurück zum letzten Stand  
git checkout -f ❷
```

❶ Sehr oft fasst Lars einfach die letzten beiden Commits zusammen

❷ git checkout -f geht zurück zum letzten Commit, also dem letzten stabilen Stand.

Cherry-Pick

» Ein weiteres wichtiges Tool was ich wirklich sehr oft verwende ist **cherry-pick**.

» Cherry-Pick? Klingt interessant.

» *Ja, mit Cherry-Pick hast du die Möglichkeit, dir von beliebigen Branches einzelne Commits auf deinen aktuellen Branch zu holen. Stell dir vor du arbeitest an einem Feature, und auf dem Master wurde ein Bugfix gemacht, den du jetzt gut brauchen könntest. Du willst dir aber jetzt nicht den ganzen master in deinen Feature-Branch*

mergen, weil du weißt, dass gerade ein Feature zurückgeführt wurde, das in Konflikt mit deinem steht. Das Auflösen dieser Merge-Konflikte willst du später machen, jetzt willst du erst mal dein Feature fertig bekommen. Das ist der perfekte Use-Case für Cherry-Pick, weil du dir damit nur den einen Commit holen kannst, der den Fix enthält, die anderen holst du dir später mit dem regulären Merge.

» Aber führt das dann nicht zu Merge-Konflikten beim nächsten Merge?

» *Wir versuchen das einfach mal, oder? Ich würde vorschlagen, du ziehst einen neuen Feature-Branch, ein Feature schaffen wir heute noch. Anschließend machst du ein update auf deinem master in der Hoffnung, dass einige neue Commits kommen. Und dann machen wir einfach ein Cherry-Pick einzelner Commits damit Du ein Gefühl dafür bekommst.*

» Klingt gut.

Übung

- Erzeugen Sie einen neuen Feature Branch (Erzeugen und Switch auf den Branch in einem Kommando)
- Erzeugen Sie 5 Commits auf dem master
- Holen Sie sich den 2. Commit vom master und nur diesen auf Ihren Feature-Branch

```
git checkout -b feature-4911
Switched to a new branch 'feature-4911'

#zurück zum master
git checkout master
Switched to branch 'master'

#erzeuge commits
makeCommits 5
[master e4fbace] committing file 1
 1 file changed, 1 insertion(+)
[master 678af97] committing file 2
 1 file changed, 1 insertion(+)
[master ae3003b] committing file 3
```

```
1 file changed, 1 insertion(+)
[master fab0445] committing file 4
1 file changed, 1 insertion(+)
[master 3753cdc] committing file 5
1 file changed, 1 insertion(+)
```

```
#wieder zum Feature Branch
git checkout feature-4911
Switched to branch 'feature-4911'
```

```
#Was ist auf dem master passiert?
git log --oneline master
3753cdc committing file 5
fab0445 committing file 4
ae3003b committing file 3
678af97 committing file 2
e4fbace committing file 1
[...]
```

» Ok, Karl, wir haben jetzt genau das Szenario für einen Cherry-Pick bei dir. Wir möchten gerne den Commit **678af97** ("committing file 2") auf unserem Feature-Branch haben aber nicht die ganzen anderen Commits. Kriegst du das hin mit **cherry-pick**?

» Ich denke schon.

```
git cherry-pick 678af97
[feature-4911 2621a70] committing file 2
1 file changed, 1 insertion(+)

#log
git log --oneline
2621a70 committing file 2 ❶
```

❶ Der Commit ist da, hat aber natürlich einen komplett anderen Hash-Wert
» Das geht auch mit mehreren Commits, hol dir jetzt mal bitte die Commits **ae3003b** ("committing file 3") und **fab0445** ("comitting file 4") auf deinen Feature-Branch.

Übung

Holen Sie die beiden Commits **ae3003b** ("committing file 3") und **fab0445** ("comitting file 4") mit nur einem Cherry-Pick auf Ihren Feature-Branch.

```
git cherry-pick ae3003b fab0445
[feature-4911 b685085] committing file 3
 1 file changed, 1 insertion(+)
[feature-4911 6c5474b] committing file 4
 1 file changed, 1 insertion(+)

#log
git log --oneline
6c5474b committing file 4
b685085 committing file 3
[...]
```

» *Karl, es ist übrigens interessant was passiert, wenn jetzt nochmal den ersten Cherry-Pick durchführst. Kannst du mir erklären, was passieren müsste?*

Übung

Was müsste passieren, wenn Sie den ersten Cherry-Pick erneut ausführen?

```
git cherry-pick 678af97
# On branch feature-4911
# You are currently cherry-picking.
#   (all conflicts fixed: run "git commit")
#
nothing to commit, working directory clean
The previous cherry-pick is now empty, possibly due to conflict
resolution.
If you wish to commit it anyway, use:
```

» Aha, das ist interessant. Lars, ich versuche dir das mal zu erklären und du korrigierst mich, wenn ich falsche liege, in Ordnung? Ich glaube, dadurch, dass wir zuerst den cherry-pick auf den Commit 678af97 gemacht haben, und anschließend auf die beiden Commits ae3003b und fab0445 bekommt der Commit 678af97 beim erneuten Cherry-Pick wieder einen anderen Hash-Wert.

» *Perfekt, Karl, genauso ist es. Der Hash-Wert basiert unter anderem auf dem Parent-Commit. Wir haben jetzt zweimal den gleichen Commit mit jeweils einem anderen Parent. Der Commit muss also nach dem Cherry-Pick einen anderen Hash-Wert bekommen. Da Git aber Änderungen trackt und nicht Commits versuchst du jetzt quasi*

einen leeren Commit zu machen, weil alle Änderungen aus diesem Commit bereits vorhanden sind. Git ist so klug und fragt wenigstens nach. In den meisten Fällen willst du keine leeren Commits haben und machst dann einfach git reset.

» Ok, das macht Sinn, so langsam glaube ich wirklich, dass Git total einfach ist, wenn man versteht wie es gedacht ist.

» *Genauso ist es. Die Architektur ist wirklich durchdacht und wenn wir ehrlich sind, Linus Torvalds ist nicht unbedingt bekannt dafür, schlechte Tools zu entwickeln.*

» Ach, Git ist von Linux Torvalds entwickelt?

» *Ja, so ist es.*

» Interessant, was ich mich jetzt noch frage ist, was passiert wenn bei einem Cherry-Pick ein Merge-Konflikt auftritt?

» *Gute Frage! Das probieren wir einfach aus.*

Übung

Bereiten Sie die Datei "file5.txt" auf dem Feature-Branch so vor, dass es einen Merge-Konflikt mit den Änderungen auf dem master gibt.

```
git log --oneline master
3753cdc committing file 5
[...]

#cherry pick
git cherry-pick 3753cdc
error: could not apply 3753cdc... committing file 5
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'

#merge
git mergetool

#add
git add file5.txt
git commit -m "cherry picking"
```

```
[feature-4911 9882a14] cherry picking  
1 file changed, 1 insertion(+)
```

» Ok Karl, du hast den Merge-Konflikt aufgelöst. Soweit so gut. Das wird uns aber später noch Probleme machen sobald wir das Update gegen den master machen.

» Ach ja?

» Ja, du kannst dir schonmal überlegen warum.

Übung

Was würde passieren, wenn jetzt ein Rebbase gegen den master gemacht wird?

» Cherry-Pick ist ein gutes Tool, sollte aber nur in Ausnahmefällen verwendet werden. Das Problem haben wir vorher schon erkannt. Wir erzeugen für jeden Cherry-Pick einen neuen Commit mit neuem Hash-Wert. Damit machen wir uns das Leben manchmal unnötig schwer.

» Kannst Du mir hierfür ein Beispiel sagen?

» Klar, oft interessiert dich beispielsweise, in welchen Commits sich dein Feature-Branch vom master unterscheidet. Das Tool der Wahl hierfür ist **cherry**.

» Cherry wie Cherry-Pick?

Cherry

» Nein, einfach nur cherry. Die Anwendung ist ganz einfach, du bist noch auf deinem Feature-Branch?

» Ja.

» Ok, dann mach einfach mal folgendes.

```
git cherry master  
- 2621a700381577e930e2633a9c06b997018ec832  
- b6850850fc163738732236496653f7e2946621d7  
- 6c5474bafa098ddf7d8f1244ef57cc2a1df6a766  
+ 2e2f7da17c173a95787cb1c0a5f38d6fb223c030
```

+ 2c597887ffd990ee27543f33f17da26d55c5cf65

» Du siehst hier Commits, von denen Git weiß, dass sie nur auf deinem Feature-Branch sind an dem vorangestellten "+", Commits deren Änderungen bereits auf dem master aber in einem anderen Commit enthalten sind erkennst du an einem vorangestellten "-". Betrachten wir die Commits nochmal ein wenig genauer.

```
git log 2621a700381577e930e2633a9c06b997018ec832 -n 1
Commit: 2621a700381577e930e2633a9c06b997018ec832
Author: dilgerm <martin@effectivetrainings.de>
Date:   (43 minutes ago) 2014-01-18 07:43:49 +0100
Subject: committing file 2
```

» Der Commit **2621a70** entspricht dem Commit **678af97** vom master, das vorangestellte "-" ist also richtig.

Welcher Branch hat welchen Commit

» Ein echtes Problem haben wir dann, wenn wir herausfinden möchten, auf welchem Branch welche Commits verfügbar sind. Commits werden nunmal durch ihre Hash-Werte identifiziert.

```
git branch --contains 678af975b23f
  master
```

» Siehst du, eigentlich würde ich hier gerne sehen, dass der Commit sowohl auf dem master als auch auf meinem Feature-Branch ist. Das funktioniert aber leider nicht, das sich die Hash-Werte unterscheiden.

Konflikte

» Karl, wir hatten vorher einen Konflikt provoziert.

» Ja, du meintest, da würden wir in Probleme laufen, sobald wir ein Update gegen den master machen.

» Ich hab dir schon erklärt, dass wir updates mit Rebase machen. In diesem Fall möchten wir aber den master in den Feature-Branch mergen.



Updates vom eigenen Branch werden mit Rebase gemacht. Updates gegen andere Branches grundsätzlich mit Merge.

» Merge doch einfach mal den master in deinen Feature-Branch.

```
git merge master
Auto-merging file5.txt
CONFLICT (content): Merge conflict in file5.txt
Automatic merge failed; fix conflicts and then commit the result.
```

» Tatsächlich haben wir den gleichen Konflikt nochmal?

» Ja, das Problem ist dass wir Git nicht genügend Hilfestellung bieten können um zu erkennen, dass wir diesen Konflikt bereits gelöst haben. Durch den cherry-pick und den Merge-Konflikt haben wir einen völlig neuen Commit erzeugt. Git kann keine Verbindung zwischen diesen beiden Commits herstellen. Es bleibt also leider nichts anderes übrig, als den Merge-Konflikt nochmals zu lösen.



Merge-Konflikte bei Cherry-Pick ziehen fast immer Konflikte und Probleme nach sich, in diesem Fall sollte ein Merge wenn möglich bevorzugt werden.

4.10. Hooks

» Warte Karl, bevor du wieder nach Hause fährst, sollten wir uns unbedingt noch kurz über **Hooks** unterhalten. Hooks kennst du vielleicht sogar aus Subversion, da gibt es sie auch. Hooks sind Shell-Skripte, die an bestimmten Zeitpunkten von Git automatisch ausgeführt werden. Beispielsweise wenn du einen Commit machst, nachdem du erfolgreich gemerged hast oder wenn du deine Änderungen in das **Remote-Repository** veröffentlichtst.

» Interessant, ja ich weiß, dass es so etwas in Subversion auch gibt. Welche Hooks habe ich denn bei Git genau zur Verfügung?

» Das siehst du ganz einfach in deinem **.git**-Verzeichnis.

```
ls .git/hooks/
applypatch-msg.sample ①
post-update.sample ②
pre-commit.sample ③
pre-rebase.sample ④
update.sample ⑤
commit-msg.sample ⑥
```

pre-applypatch.sample ⑦

pre-push.sample ⑧

prepare-commit-msg.sample ⑨

- ① Skript wird ausgeführt wenn ein Patch eingespielt wird.
- ② Skript wird ausgeführt, nachdem ein Update (fetch) erfolgreich durchgeführt wurde.
- ③ Skript wird ausgeführt, bevor bei einem Commit die Objekte ins **objects** Verzeichnis geschrieben werden.
- ④ Skript wird ausgeführt, bevor ein **rebase** gemacht wird.
- ⑤ Skript wird ausgeführt, wenn ein Update (**fetch**) durchgeführt wird.
- ⑥ Skript wird ausgeführt, wenn ein Commit gemacht wird und direkt bevor die Commit-Message geschrieben wird.
- ⑦ Skript wird ausgeführt, direkt bevor ein **Patch** eingespielt wird
- ⑧ Skript wird ausgeführt, direkt bevor ein **Push** die Daten in ein Remote-Repository schreibt
- ⑨ Skript wird ausgeführt, wenn die Commit-Message geparsed wird.

» Mir fallen sofort einige Use-Cases ein, die uns das Leben erleichtern können. Habt ihr denn aktuell bereits Hooks im Einsatz?

» Nur teilweise. Leider arbeiten nicht alle Entwickler im Team auf der Konsole. Wir haben die Wahl der Tools bewusst nicht begrenzt. Hooks werden leider nicht von allen Tools gleichermaßen unterstützt. Ehrlich gesagt, ist die Unterstützung in den meisten Tools sogar ziemlich schlecht. Eclipse zum Beispiel unterstützt sie derzeit überhaupt nicht. Es macht also keinen Sinn, sich darauf zu verlassen.

» Aber Lars, du zum Beispiel, du arbeitest auf der Konsole. Mit welchen Hooks arbeitest du?

Task-Nummer in jeder Commit-Message

» Ich habe einige Hooks im Einsatz. Der praktischste kümmert sich darum, dass in jeder Commit-Message die Task-Nummer steht, für den dieser Commit gemacht wurde.

» Oh ja, sehr praktisch, kannst du mir erklären, wie das funktioniert? Das würde mich tatsächlich sehr interessieren.

» Natürlich, eine einfache Commit-Message könnte beispielsweise so aussehen.

#Feature

```
echo "Besseres File-Name Handling" >> file-handling.txt
#add
git add file-handling.txt
#Commit
git commit -m "Correct handling of Uploaded File names."
[master 68c2a3c] Correct handling of Uploaded File names.
 1 file changed, 2 insertions(+)
 create mode 100644 file-handling.txt

#log
git log --oneline -n1
68c2a3c Correct handling of Uploaded File names.
```

» *Stell dir vor, du siehst nur diesen Commit und die Commit-Message in der Historie und möchtest wissen, wieso wir diese Änderung überhaupt gemacht haben. Es ist relativ schwer, das ohne Kontext nachzuvollziehen. Was wir eigentlich brauchen, ist die Task-Nummer in Jira, richtig? Was ich persönlich gerne hätte wäre eine Commit-Message, die so aussieht:*

```
68c2a3c [4711] Correct handling of Uploaded File names.
```

» *Es ist natürlich relativ umständlich, für jeden Commit die Task-Nummer mit aufzunehmen. Es wird viel zu oft vergessen und ich bin ein Fan davon, so wenig manuelle Tasks wie möglich in der täglichen Arbeit auszuführen. Mit **Hooks** lässt sich das relativ einfach automatisieren. Wir gehen davon aus, dass unsere Branches die jeweils passende Bezeichnung haben, etwas wie "fb-4711-beschreibung". Das heisst, wir können für jeden Commit die Task-Nummer aus dem jeweils ausgecheckten Branch extrahieren.*

Übung

Können Sie aus der Liste der Hooks erschliessen, welcher Hook geeignet ist, um die Commit-Message anzupassen, bevor der Commit in der Historie landet?

applypatch-msg.sample
post-update.sample
pre-commit.sample
pre-rebase.sample
update.sample
commit-msg.sample

```
pre-applypatch.sample
pre-push.sample
prepare-commit-msg.sample
```

» Der passenden Hook ist **prepare-commit-msg**, da dieser ausgeführt wird genau bevor die Commit-Message in den Commit aufgenommen wird. Wir haben also die Möglichkeit, die Message nochmal anzupassen. Ich habe mir hierfür ein kleines Shell-Skript geschrieben.

```
#!/bin/sh

#author Martin Dilger - EffectiveTrainings.de
#get the original commit msg
orig_msg=$(cat $1) ❶
#expects branchnames in the form fb_task-4711_some_description
branchName=$(git rev-parse --abbrev-ref HEAD) ❷
#branches may start with fb_task and then have "-1234"
regexpForBranches='^fb-[0-9]-' ❸

branchMatches=$(echo $branchName | grep -E $regexpForBranches) ❹
if [ "$branchMatches" ] ;
then
echo "matched branch";
#split branchname by underscore and take the second chunk
task=$(echo $branchName | cut -f 2 -d '-') ❺
#prepend task name to original msg
msg="$task - $orig_msg" ❻
echo "$msg" > "$1" ❼
else
echo "[ATTENTION] - branch name does not match, no task number in branch
but committing"; ❽
fi
```

- ❶ Lade die Original-Commit-Message (funktioniert nur wenn der Befehl "git commit -m *Commit Message*") verwendet wird.
- ❷ Lade den aktuellen Namen des ausgecheckten Branches
- ❸ Definiere eine passende Regular-Expression für Branch-Namen. Für unseren Fall gilt fb-<Task-Nummer>-<Beschreibung>
- ❹ Prüfe ob der aktuelle Branch-Name dem Schema entspricht und ob die Task-Nummer extrahiert werden kann

- ⑤ Extrahiere Task-Nummer aus Branch-Namen
- ⑥ Erzeuge neue Commit-Message in der Form [4711] Original Message
- ⑦ Überschreibe den ersten Parameter, so dass Git die neue Commit-Message verwendet
- ⑧ Gib eine Meldung aus, wenn der Name des Branches nicht dem Schema entspricht und die Task-Nummer nicht extrahiert werden kann.

*» Du musst das Skript gar nicht im einzelnen verstehen. Um es zu verwenden kopierst du es einfach in die Datei **prepare-commit-msg.sample** im .git/hooks-Verzeichnis deines Git-Repositories. Um das Skript zu aktivieren benennst du es dann noch von **prepare-commit-msg.sample** in **prepare-commit-msg** um. Git ignoriert alle Skripte, die auf **.sample** enden.*

Übung

Entfernen Sie zunächst alles aus der Datei **.git/hooks/prepare-commit-msg.sample**.

Kopieren Sie das Skript in die nun leere Datei

Benennen Sie die Datei von **prepare-commit-msg.sample** in **prepare-commit-msg**

Erzeugen Sie einen neuen Branch mit dem Namen "fb-0815-new-feature"

Machen Sie einen Commit mit der Message "New Feature"

Prüfen Sie die Commit-Message, sie sollte "[0815] New Feature" lauten.

```
vi .git/hooks/prepare-commit-msg.sample
#Datei leeren und Skript einfügen

mv .git/hooks/prepare-commit-msg.sample .git/hooks/prepare-commit-msg

#erzeuge Branch
git checkout -b fb-0815-new-feature
Switched to a new branch 'fb-0815-new-feature'

#Feature
echo "new Feature" >> new-feature.txt
git add new-feature.txt
```

```
git commit -m "New Feature"  
matched branch ①  
[fb-0815-new-feature ed3e17a] [0815] - New Feature ②  
1 file changed, 1 insertion(+)  
create mode 100644 new-feature.txt  
  
#log  
git log --oneline -n 1  
ed3e17a [0815] - New Feature
```

① Name des Branches konnte erfolgreich geparsed werden

② Die neue Commit-Message

» *Karl, ich glaube, damit hast du alle Tools die du brauchst um hier bei uns produktiv zu arbeiten. Das war wirklich ein sehr produktiver Tag, ich habe viel erklärt und du hoffentlich viel gelernt.*

» Ja klar, Lars. Das war wirklich Klasse.

4.11. Daily Alias

» *Eins zeige ich dir vielleicht noch. Ich habe mir im Lauf der Zeit einige Git-Shortcuts zugelegt, mit denen ich noch produktiver bin.*

» Was ist denn ein Git-Shortcut?

» *Du kennst sie wahrscheinlich unter **Alias**. Je länger du mit Git arbeitest, desto öfter wirst du feststellen, dass du bestimmte Befehle immer und immer wieder ausführst. Beispielsweise das Wechseln zwischen verschiedenen Branches.*

```
git checkout "mein_branch"
```

» *Meistens macht es Sinn, für Befehle die du oft verwendest Shortcuts festzulegen. Dafür lässt sich ganz einfach ein **Alias** definieren.*

```
git config alias.go "git checkout" ①
```

① **go** ist kürzer als **checkout**

```
#checkout fb-branch  
git checkout fb-0815-new-feature
```

```
Switched to branch 'fb-0815-new-feature'
```

```
#git go  
git go master  
Switched to branch 'master'
```

» Das ist interessant, wahrscheinlich sammeln sich da im Lauf der Zeit ziemlich viele Shortcuts an, oder?

» *Manche Entwickler haben hunderte Shortcuts. Ich persönlich konzentriere mich auf einige wenige und arbeite wo immer möglich und produktiv mit den Standard-Befehlen.*

» Welche Shortcuts hast du denn noch definiert?

Übung

Alle Shortcuts lassen sich anzeigen mit

```
git config --get-regexp alias
```

Definieren Sie einen Alias *shortcuts* um Shortcuts anzuzeigen.

```
git config alias.shortcuts "git config --get-regexp alias"  
git shortcuts
```

```
alias.logtree log --graph --oneline --decorate --all ❶  
alias.gc git commit ❷  
alias.gcm git commit -m ❸  
alias.st git status ❹  
alias.clearfile git reset --hard -- ❺  
alias.sq git rebase -i HEAD~❻ ❻  
alias.msg git commit --amend ❼  
alias.stage git commit -am ❽  
alias.go checkout ❾
```

❶ Log mit Graph-Funktion

❷ Commit

❸ Commit mit Message

- ④ Status
- ⑤ File zurücksetzen
- ⑥ Die obersten beiden Commits zusammenfassen
- ⑦ Commit-Message neu schreiben
- ⑧ Alles hinzufügen und Committen
- ⑨ Branches wechseln

Git-Extras

» Es gibt ein interessantes Projekt [Git Extras](#)¹⁰, das einige Alias-Definitionen mitbringt, die tagtäglich sehr hilfreich sein können.

4.12. Tag 1 endet

» Nachdem wir heute die Grundlagen besprochen haben werden wir uns morgen mit einigen fortgeschrittenen Themen beschäftigen. Wir müssen uns unbedingt noch über die verschiedenen Branching-Modelle unterhalten. Außerdem würde ich dir gerne zeigen, mit welchen Tools wir hier arbeiten. Viele kennst du wahrscheinlich sowieso schon. Aber ich denke, für heute ist es wirklich genug. Ich sehe, du hast dir sehr viele Notizen gemacht, das finde ich gut. Wir haben ein Wiki für unser Team, vielleicht kannst du ja das eine oder andere ins Wiki übertragen.

» Das ist ein hervorragende Idee, ich werde die Zugfahrt zurück nach München heute nutzen, um die Notizen nochmal zusammenzufassen. Wars das dann für heute?

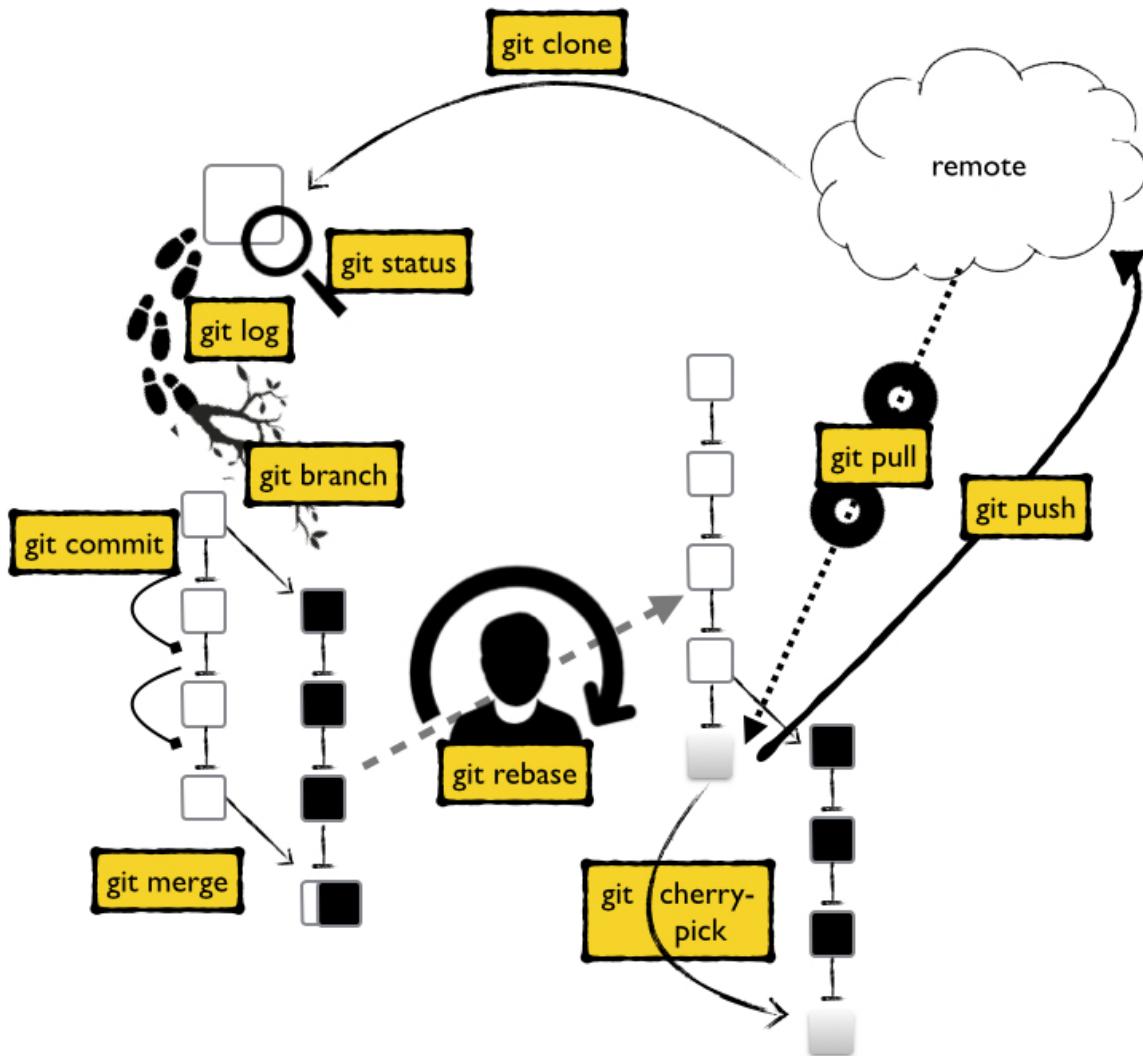
» Ja Karl, wir sehen uns morgen, komm gut nach Hause.

Karl macht sich also auf den Heimweg nach München und freut sich bereits auf seine Familie. Der Tag war anstrengend doch äusserst lehrreich. Karl hatte heute morgen noch keine Ahnung von Git. Jetzt kennt er bereits die wichtigsten Befehle.

Er nutzt die Zeit im Zug um seine Notizen neu zu strukturieren. Er verwendet hierfür die Technik des [Sketchnoting](#)¹¹ und erstellt folgendes Diagramm.

¹⁰ <https://github.com/visionmedia/git-extras>

¹¹ <http://de.wikipedia.org/wiki/Sketchnotes>



Er weiß jetzt, wie man

- ein Repository klont (**git clone**)
- sich die Historie betrachtet (**git log**)
- Branches erzeugt (**git branch**)
- Branches merged (**git merge**)
- die Historie linearisieren kann (**git rebase**)
- mit entfernten Repositories kommuniziert (**git remote, git push, git fetch, git pull**)
- einzelne Commits zwischen Branches austauschen kann (**git cherry-pick**)
- verlorene Commits wiederfindet (**git reflog**)
- mit Hooks arbeitet

Für einen Tag ist das bereits eine ganze Menge. Karl ist aber schon sehr gespannt auf den nächsten Tag im Team.

Übung

Betrachten Sie alle Befehle, die Karl am ersten Tag gelernt hat.
Verstehen Sie grundsätzlich, wozu jeder dieser Befehle verwendet werden kann?
Könnten Sie jeden dieser Befehle anwenden?

Aus dem Projektalltag

Durch meine Arbeit als Git Trainer komme ich in viele Unternehmen und sehe viele Projekte und Entwickler. Gerade durch meine Arbeit als Trainer ist es für mich eine Herzensangelegenheit, Entwickler bei der Arbeit mit Git zu unterstützen. Ich mache das nicht, weil es mein Beruf ist sondern aus purer Überzeugung, das Git tatsächlich das derzeit beste Versionskontrollsystem ist, mit dem Sie arbeiten können. In meinen ganzen Jahren, in denen ich jetzt mit Git arbeite habe ich zwar des öfteren ein wenig Überzeugungsarbeit für Git leisten müssen, ich kenne aber nicht einen Entwickler, der heute sagen würde, dass früher alles besser war und dass der Umstieg auf Git keine Vorteile gebracht hat.

Es entspricht leider nicht ganz der Wahrheit, denn tatsächlich gab es in meiner ganzen Laufbahn einen einzigen Entwickler, der nicht zu überzeugen war. Es war eine Migration von ClearCase (einem sehr angegrauten System) hin zu Git notwendig, da Lizenzen ausgelaufen waren und schlichtweg ein neues System her musste. Und dieser Entwickler war so vertraut mit der Art und Weise mit ClearCase zu arbeiten dass er für jedes Argument **für** Git ein passendes Argument **für** ClearCase und damit **gegen** Git zu bieten hatte. Bis zuletzt konnte ich diesen Entwickler nicht überzeugen, dass der Umstieg etwas Gutes ist und viele Vorteile mit sich bringt.

Leider weiß ich nicht, wie die Geschichte schlussendlich ausgegangen ist und wie der Entwickler heute zur Arbeit mit Git steht, aber ich bin sicher, wenn schon nicht meine Worte überzeugen konnten dann doch mit Sicherheit die Praxis.

5. Tag 2

Karl hat einen entspannten Abend mit seiner Familie verbracht und befindet sich jetzt wieder auf dem Weg nach Nürnberg. Er nutzt die Fahrzeit um sich weiter mit Git vertraut zu machen. Er kann dank dem dezentralen System problemlos auch ohne Verbindung ins Büro *offline* arbeiten. Die Zeit verfliegt und somit steht Karl schon bald wieder rechtzeitig zum *Daily Scrum* im Büro und wird herzlich von seinen neuen Teamkollegen begrüßt.

» *Hello Karl, schön dass du wieder da bist. Ich hoffe mit der Rückfahrt hat alles soweit geklappt?*

» Hallo Lars, ja, alles problemlos. Was ist der Plan für heute?

» *Wir werden uns heute zunächst mal mit unserem Branching Modell befassen. Hast du bereits von Git Flow¹² gehört?*

» Ja, Git Flow hab ich schon öfter gehört. Ich kann dir aber nicht erklären, was sich dahinter verbirgt.

» *Kein Problem, denn genau das erkläre ich dir jetzt.*

5.1. Git Branching Modelle

» *Zunächst stellt sich jeder Entwickler normalerweise die Frage, wieso man überhaupt etwas wie ein Branching-Modell braucht. Wie habt ihr denn bisher gearbeitet?*

» Naja, in den Projekten in denen ich bisher gearbeitet habe war durchweg Subversion als Versionskontrollsystem im Einsatz. Über konkrete Branching-Modelle habe ich mir bisher ehrlich gesagt wenig Gedanken gemacht. Wir haben eigentlich immer direkt auf dem **trunk** entwickelt.

» *Ja, das ist meistens so. Wie war denn dein Eindruck davon?*

» Prinzipiell habe ich damit keine Probleme gehabt. Jeder Entwickler checkt seinen Stand ein, wenn er denkt, dass er eine Art Meilenstein in seinem aktuellen Feature erreicht hat. Das Schöne ist, man sieht sofort im Code, woran die Kollegen gerade arbeiten. Das nennt sich dann wohl auch *Continuous Integration*.

¹² <http://www.effectivetrainings.de/blog/2012/04/22/git-flow-einfaches-arbeiten-mit-dem-perfekten-git-workflow/>

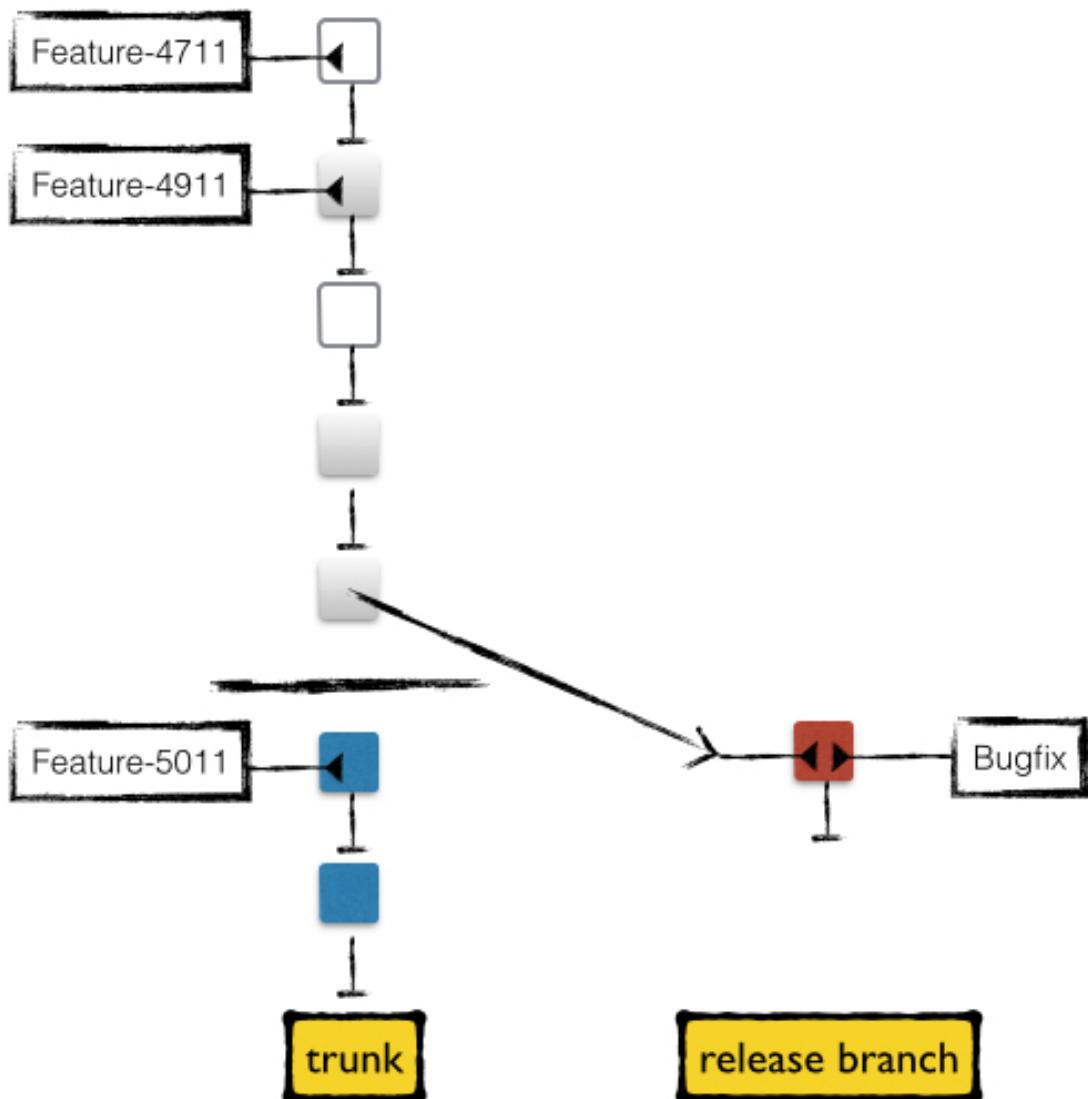
» Konflikte zwischen verschiedenen Features werden möglichst früh entdeckt und bereits aufgelöst. Das ist ein echter Vorteil. Hattet ihr Probleme mit der Stabilität des **trunk**?

» Das ist die Kehrseite der Medaille. Da die Entwickler natürlich ständig auf dem **trunk** einchecken sind dort natürlich auch halbfertige Features, die zwar bauen aber nicht funktional abgeschlossen sind.

» Von welchem Branch habt ihr üblicherweise eure Releases gemacht?

» Es gab bestimmte Zeitpunkte, an denen ein **Release Branch** gezogen wurde.

» Ok, jetzt kommen wir der Sache schon näher. Lass und das am Whiteboard aufzeichnen.



» So ähnlich sieht das üblicherweise in Projekten aus. Alle arbeiten auf dem trunk, irgendwann entscheidet jemand der die Befugnis hat, dass es an der Zeit für das nächste Release ist. Es wird ein Release-Branch gezogen, der alle Features beinhaltet, die bis zu diesem Zeitpunkt abgeschlossen und auf dem trunk eingechekkt sind.

» Ja genau, ein Problem ist immer, den richtigen Zeitpunkt abzupassen. Normalerweise wird hierzu ein **Code-Freeze** ausgerufen. Es müssen alle eingechekkten Features fertig implementiert sein.

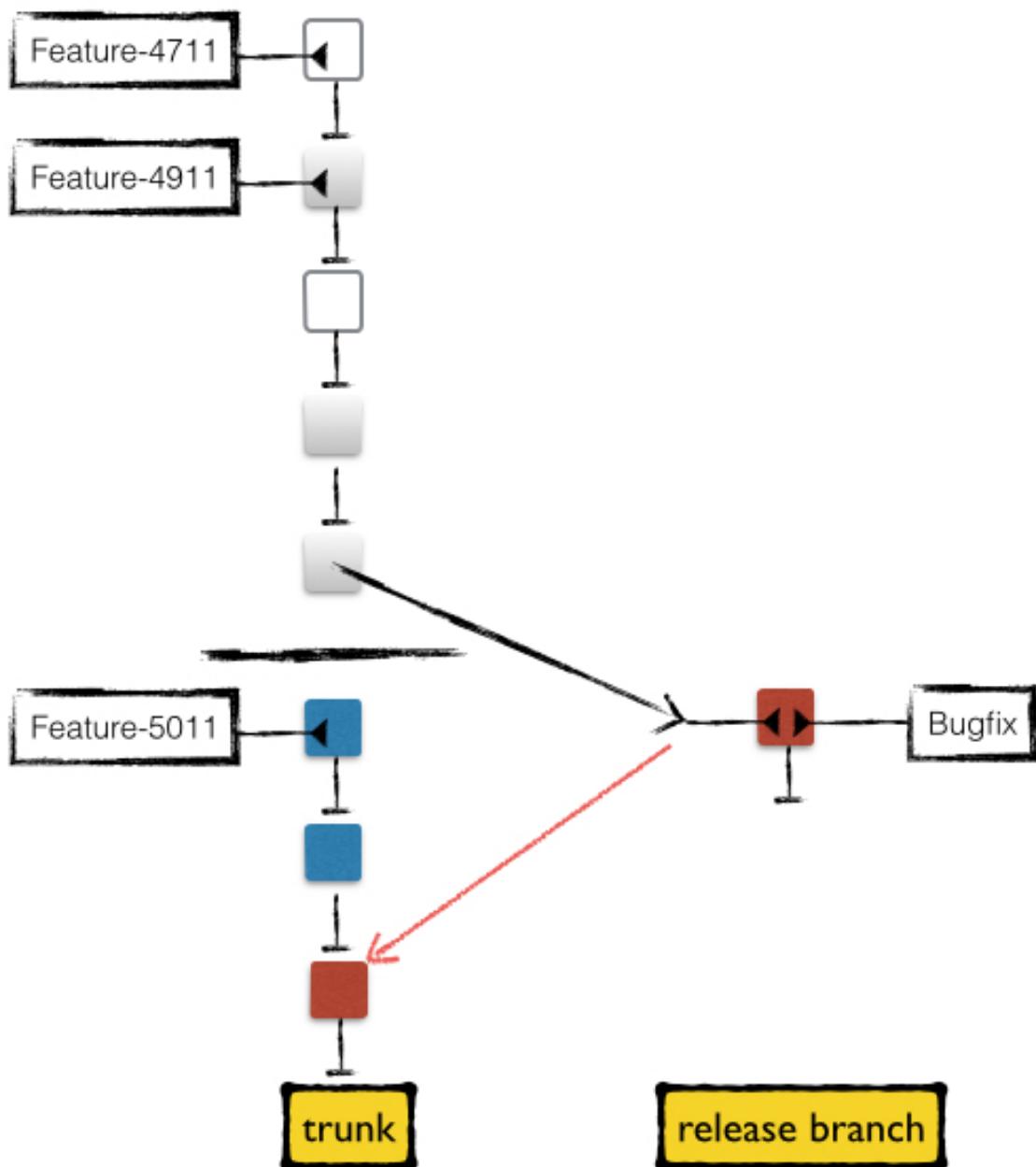
» *Habt ihr dann auf dem Release-Branch die Features noch stabilisiert?*

» Ja, üblicherweise wird die Version vom Release-Branch auf eine Testumgebung installiert, die zumindest ähnlich zur Produktionsumgebung ist. Dort wird sie dann von den Entwicklern und der QA-Abteilung verifiziert.

» *Das ist das Standardvorgehen. Ich gehe davon aus, dass ihr Bugfixes und Stabilisierungen auf dem Release Branch sofort wieder zurück auf den trunk gemerged habt?*

» Oh ja, das wurde oft vergessen, sollte aber natürlich gemacht werden. Der Release-Branch lebt genauso lange wie das Release. Sobald das nächste Release gemacht wird wird der Release Branch hierfür neu gezogen.

» Ok, lass uns das am Whiteboard ergänzen.



» Wie war das bei euch, Karl? Wer hat sich beispielsweise darum gekümmert, dass Bugfixes auf den trunk zurückmerged wurden?

» Naja, im idealfall kümmert sich der Entwickler darum, der den Fehler behoben hat, oder?

» Im Idealfall, ja. Was ist beispielsweise, wenn beim Zurückmergen ein Konflikt auftritt? Den kann eigentlich nur der Entwickler selbst beheben.

» Ja, trotzdem gab es eigentlich in fast jedem Projekt jemanden, manchmal in einer eigenen Rolle **Release-Manager**, oft auch nur ein Entwickler, der eben zufällig dafür verantwortlich war.

» *Es ist auf jedenfall ziemlich viel manueller Aufwand notwendig um Releases vorzubereiten, zu stabilisieren und zu finalisieren, richtig?*

» Ich glaube, das kann man so sagen, ja. Soweit ich weiß, gibt es hier in der Subversion-Welt eigentlich kein standardisiertes Modell. Die meisten Entwickler wissen einfach intuitiv, was getan werden muss um das Release auszurollen.

5.2. Git Flow

Im Jahr 2010 wurde die Idee des bis heute wahrscheinlich am meisten in Projekten eingesetzten Branching-Modells vorgestellt.

Git-Flow¹³.

Der Einsatz von Git-Flow garantiert:

- Der master ist und bleibt stabil. Releases können jederzeit gemacht werden.
- Features werden getrennt voneinander auf Feature-Banches entwickelt.
- Es können beliebig viele Releases parallel gewartet werden.
- Neue Features für das übernächste Release können isoliert entwickelt werden.

» *Karl, jetzt haben wir prinzipiell über Software-Entwicklung und Releases gesprochen. Ich würde dir jetzt gerne zeigen, wie wir wirklich arbeiten.*

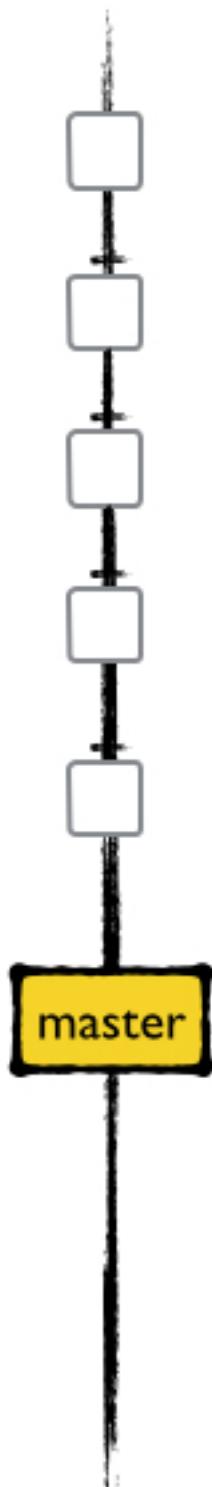
» Ja bitte, das würde mich sehr interessieren.

» *Ich sage dir lieber gleich, es ist keine Magie was wir tun. Wir arbeiten nach dem Git-Flow Modell. Wenn du mich fragst ist Git-Flow nichts anderes als "Einmal richtig nachdenken, aufschreiben und dann machen". Das Modell macht Sinn und sorgt für produktives und strukturiertes arbeiten.*

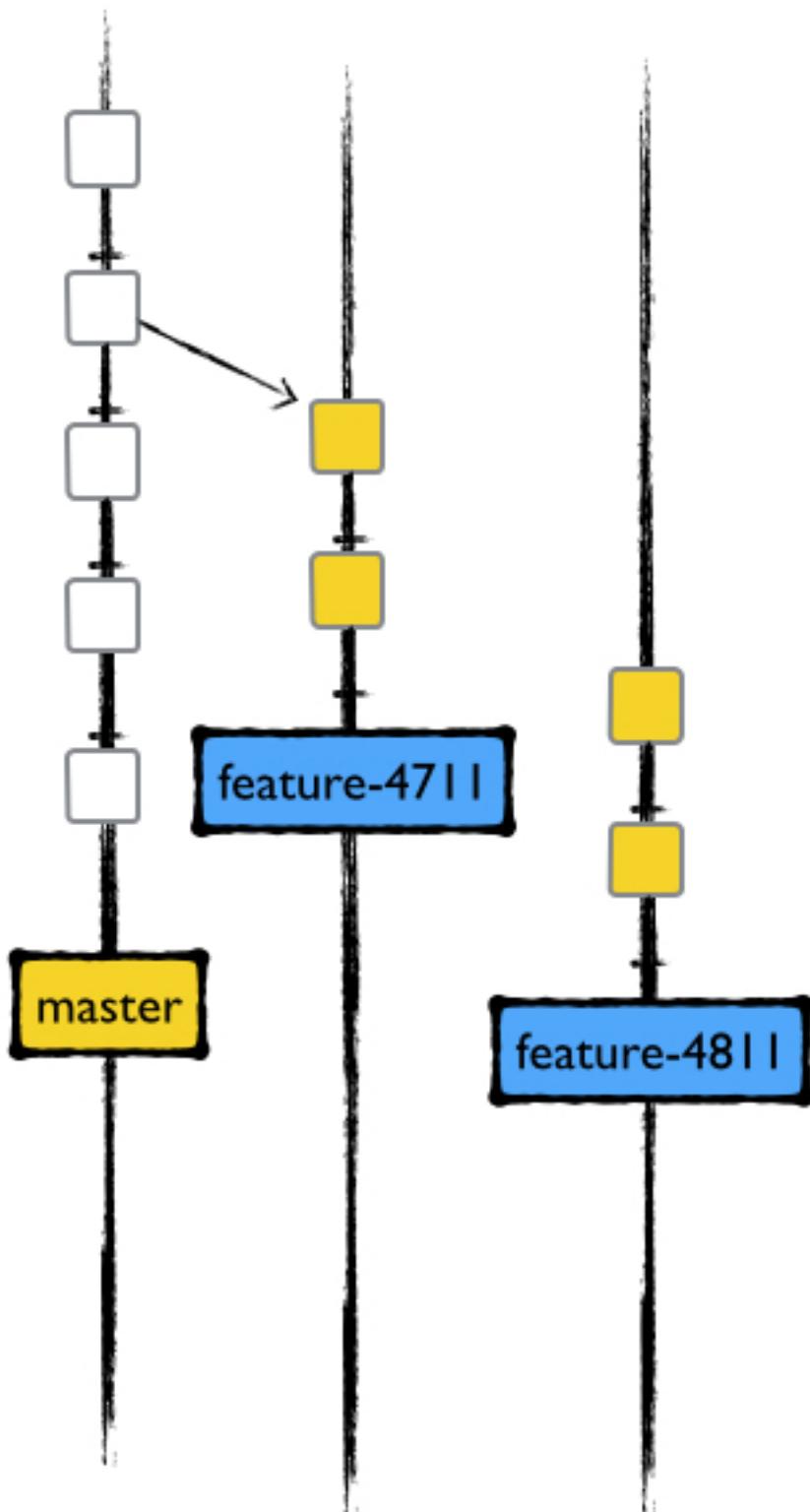
Das Arbeiten auf Feature Branches

» *Ich hatte dir schon erklärt, dass wir alle neuen Features auf Feature-Banches entwickeln. Vielleicht versuchen wir, unsere Projektstruktur kurz zu skizzieren. Das was in Subversion der trunk ist, ist in Git der master. Jedes Projekt hat einen master.*

¹³ <http://nvie.com/posts/a-successful-git-branching-model/>



» Für jedes Feature, das wir entwickeln ziehen wir einen neuen Feature-Branch vom master. Auf dem **master** selbst wird eigentlich nicht gearbeitet.



```
git checkout -b "fb-4711-single-sign-on-customer-login"  
Switched to a new branch 'fb-4711-single-sign-on-customer-login'
```

» Wir achten hierbei darauf, dass wir uns an ein einheitliches Namensschema halten. Feature-Banches starten mit dem Kürzel "fb". Auf das Kürzel folgt grundsätzlich die Jira-Task Nummer, in diesem Fall **4711**. Es sollte keine Aufgabe ohne zugeordnete Jira-Tasknummer geben. Anschließend kommt eine kurze Beschreibung, so dass man problemlos bereits am Namen des Feature-Banches erkennen kann, woran dort gearbeitet wird.

» Die Features sind also komplett getrennt voneinander?

» Genau, alle Features für ein bestimmtes Release starten vom Stand des **master-Banches**. Die Features sind unabhängig voneinander. Ein Feature wird erst dann zurückgeführt, wenn die Entwickler es für abgeschlossen erachten und das Zurückführen mit dem Team besprochen ist.

```
#branches
git branch
* fb-4711-single-sign-on-customer-login
  master

#feature implementation
git makeCommits 3
[fb-4711-single-sign-on-customer-login 11a6e37] committing file 1
  1 file changed, 1 insertion(+)
[fb-4711-single-sign-on-customer-login 9d2ee96] committing file 2
  1 file changed, 1 insertion(+)
[fb-4711-single-sign-on-customer-login 730eac4] committing file 3
  1 file changed, 1 insertion(+)
```

» Ok, das bedeutet, es kann sein, dass ein fertig umgesetztes Feature eine ganze Weile auf einem Feature-Branch liegt, bevor es in den master zurückgeführt wird?

» Theoretisch ja, in der Praxis kommt das eher selten vor, da wir die meiste Zeit an Features arbeiten, auf die unsere Kunden warten. Die Entwickler haben also ein berechtigtes Interesse daran, dass ihre Arbeit möglichst schnell zurückgeführt und produktiv wird.

» Nehmen wir an, Felix hat ein Feature abgeschlossen und fährt anschließend in Urlaub, hat aber vergessen seinen Teamkollegen Bescheid zu geben, dass dieses Feature fertig aber noch nicht zurückgeführt ist. Die Entwickler möchten das Feature einige Tage später zurückführen, sind sich aber nicht sicher, dass Felix alles abgeschlossen hat. Sie haben erfolglos versucht, Felix telefonisch zu erreichen. Woher wissen die Entwickler jetzt, ob das Feature fertig ist oder nicht?

» Schönes Szenario! Das ist ein guter Punkt. Es wäre äusserst ineffizient, jedesmal die Anwendung zu starten und sich zu vergewissern, dass ein Feature tatsächlich abgeschlossen ist und funktioniert bevor es zurückgeführt wird. Zumal das bereits hoffentlich vom zuständigen Entwickler erledigt wurde. Wir verwenden hierfür Tags.

» Interessant, ich glaube wir haben noch gar nicht über Tags gesprochen, oder?

Tags

» Genau, dann wird es höchste Zeit. Tags sind ein wichtiges Werkzeug. Tags in Git funktionieren vom Konzept her genauso wie in jedem anderen Versionskontrollsystem. Was ist ein Tag? Ein Tag ist immer eine Möglichkeit, einen bestimmten Punkt in der Historie für später zu markieren um ihn leicht wieder auffindbar zu machen.

» Ich dachte bisher, Tags werden beispielsweise nur für Releases verwendet?

» Dazu kommen wir gleich. Wir verwenden Tags auch um fertig implementierte Features zu "markieren". Am besten wir versuchen das einfach mal. Wir sind immer noch auf dem Feature-Branch für das Feature 4711, richtig?

```
git branch
* fb-4711-single-sign-on-customer-login
  master
```

» Dieses Feature ist fertig implementiert, also taggen die Implementierung. Ein Tag in Git referenziert immer einen Commit, oder besser gesagt den Hashwert eines Commits. Git bietet leichtgewichtige und annotierte Tags. Über annotierte Tags sprechen wir später, für das Markieren von Features verwenden wir leichtgewichtige Tags, weil sie viel einfacher sind und eigentlich alles bieten was wir brauchen. Ein leichtgewichtiger Tag ist nichts anderes als eine Referenz auf den Hashwert eines Commits mit einem frei wählbaren Namen. Einen Tag erzeugst du einfach mit `git tag <tag-name>`. Damit taggst du den aktuell obersten Commit auf deinem Branch.

```
git tag fb-4711
```

» Nichts passiert?

» Doch, am besten du öffnest kurz dein .git-Verzeichnis unter `refs/heads/` sollte es jetzt ein neues Verzeichnis `tags` geben.

```
ls .git/refs/
```

```
heads  remotes  tags
```

```
#inhalt von refs/tags  
ls .git/refs/tags/  
fb-4711
```

» Siehst du, Git hat eine neue Datei erzeugt mit dem Namen deines Tags. Schau dir bitte den Inhalt der Datei an.

```
cat .git/refs/tags/fb-4711  
730eac42835edcce3a431cca4dafc0ad275994c5
```

» Fällt dir etwas auf?

```
git log --oneline -n 1 ①  
730eac4 committing file 3 ②
```

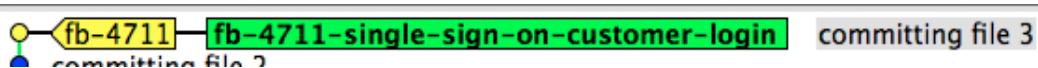
- ① Zeige letzten Commit auf dem Branch
- ② Der letzte Commit ist 730eac4

Übung

Das Konzept von leichtgewichtigen Tags sollten Ihnen bereits unter anderem Namen bekannt sein. Leichtgewichtige Tags sind identisch mit ... ?

» Warte mal, das kommt mir bekannt vor. Kann es sein, dass ein Tag nichts anderes ist als ein Branch?

» Hervorragend, Karl! Ich hatte gehofft, dass dir das auffällt. Das Konzept von Tags ist identisch mit Branches. Nur das Verzeichnis, in dem sie abgelegt werden ist eben nicht `.git/refs/heads` sondern `.git/refs/tags`. Du kannst dir das auch in der Historie betrachten. Die meisten grafischen Tools zeigen Tags an.



» Jetzt lässt sich sehr einfach überprüfen, ob ein bestimmtes Feature fertig implementiert ist, indem einfach nach dem Tag für die Task-Nummer gesucht wird.

```
git tag -l '*4711' ❶  
fb-4711
```

- ❶ git tag -l erwartet eine Regular Expression
» *Es gibt allerdings ein Problem mit diesem Ansatz.*

» Ich glaube ich weiß, worauf du hinauswillst. Der Tag referenziert einen Hashwert, richtig? Wir haben gestern bereits besprochen, dass wir nach Möglichkeit eine lineare Historie behalten möchten. Bevor ich also einen Feature-Branch zurückführe kann es passieren, dass ein Entwickler einen **rebase** gegen den master macht, richtig?

» *Sehr gut Karl, genau darauf wollte ich hinaus. Es ist kein Problem, wenn ein fertiges Feature getaggt und anschließend direkt zurückgeführt wird. Normalerweise machen wir unmittelbar bevor wir ein Feature zurückführen einen rebase gegen den master. Was aber passiert, wenn der Entwickler das Feature abschließt, taggt und dann nicht direkt zurückführen kann? Nehmen wir an, in der Zwischenzeit sind auf dem master einige Commits gemacht worden.*

```
git checkout master  
Switched to branch 'master'  
  
#in der Zwischenzeit gab es neue Commits auf dem master  
git makeCommits 3  
[master d9e44e8] committing file 1  
 1 file changed, 1 insertion(+)  
[master 292e21f] committing file 2  
 1 file changed, 1 insertion(+)  
[master 60ceee4] committing file 3  
 1 file changed, 1 insertion(+)
```

- » *Spulen wir in Gedanken einige Tage nach vorne. Der Entwickler hat endlich Zeit und möchte sein bereits getaggtes Feature zurückführen. Was macht er zunächst?*
- » Er wird ein **rebase** gegen den master machen, damit der Feature-Branch auf einem aktuellen Stand ist.
- » *Genau.*

```
# einige Tage später  
git checkout fb-4711-single-sign-on-customer-login  
Switched to branch 'fb-4711-single-sign-on-customer-login'
```

```
#Zeige alle Tags, die den obersten Commit referenzieren
git tag --points-at HEAD ①
fb-4711
```

```
#update Feature Branch
git rebase master
First, rewinding head to replay your work on top of it...
```

```
#Finde Tags
git tag --points-at HEAD
#keine Ausgabe
```

- ① git tag --points-at HEAD zeigt den Tag an, der auf den aktuell obersten Commit zeigt. Es zeigt nichts an, wenn es keinen Tag gibt, der diesen Commit referenziert.

Übung

Wieso ist der Tag verschwunden, der auf den letzten Commit auf dem Feature-Branch referenziert?

» Ich glaube, das kann ich erklären. Durch den rebase hat sich der Hashwert des Commits geändert, oder?

» *Hervorragend, genau.*

```
git log --oneline -n 1
60ceee4 committing file 3 ①
```

- ① Der Hashwert **730eac4** hat sich geändert in **60ceee4**

» *Der Tag selbst ist aber immer noch da und referenziert den alten Commit.*

```
#zeige alle Tags
git tag
fb-4711

#betrachte Tag
git show fb-4711 ①
Commit: 730eac42835edcce3a431cca4dafc0ad275994c5 ②
Author: Markus <Markus@effectivetrainings.de>
```

```
Date:      (20 hours ago) 2014-02-08 17:18:24 +0100
Subject: committing file 3
```

```
diff --git a/file3.txt b/file3.txt
index 68d1ef3..3720c95 100644
--- a/file3.txt
+++ b/file3.txt
@@ -1 +1,2 @@
 commit 3
+commit 3
```

- ① fb-4711 ist der Name des Tags
- ② Der Tag referenziert nach wie vor den alten Commit mit dem Hashwert 730eac



Sind auf dem master neue Commits vorhanden kann nach dem rebase der Stand des Feature-Branches nicht mehr gepusht werden (non-fast-forward push Problem) Der Rebase kann / darf nur direkt vor dem Zurückführen durchgeführt werden. Eine sichere Alternative ist ein Merge, mit dem Nachteil der nicht-linearen Historie. Wird ein Rebase gemacht, muss in diesem Fall auch der Tag aktualisiert werden. **Achtung**, diese Operation sollte im Idealfall nur in absoluten Ausnahmefällen notwendig sein.

```
#remote Tag löschen
git push origin :refs/tags/fb-4711 ①
```

```
#Tag aktualisieren
git tag -f fb-4711 ②
Updated tag 'fb-4711' (was 730eac4)
```

- ① Bevor ein Tag aktualisiert werden kann muss er im Remote-Repository gelöscht werden, da ansonsten ein **Push** nicht möglich ist.
- ② **git tag -f** aktualisiert einen bereits bestehenden Tag.
» Jetzt können wir den Feature-Branch zurückführen.

```
git checkout master
Switched to branch 'master'
```

```
#Feature abschliessen
git merge fb-4711-single-sign-on-customer-login
Updating ad261f2..60ceee4
Fast-forward
```

```
15 files changed, 18 insertions (+)
```

» *Sehr gut Karl, damit haben wir das Feature abgeschlossen und der Tag ist auf dem master verfügbar. Es ist jetzt für jeden in der Historie sehr einfach zu sehen, wann das Feature zurückgeführt wurde.*

» Ok, werden die Tags automatisch mit übertragen, wenn wir einen **push** machen?

» Nein, das müssen wir explizit mit dem Parameter **--tags** angeben.

```
git push --tags origin master
* [new tag] fb-4711 -> fb-4711
```

» *Es ist wichtig zu verstehen, dass wir hier nur mit einem leichtgewichtigen Tag arbeiten. Diese Art von Tag bietet im Gegensatz zu annotierten Tags keinerlei kryptografische Sicherheit und könnten jederzeit verändert werden. Wir verwenden diese Art von Tags hier nur aus praktischen Gründen. Für Releases sind leichtgewichtige Tags eigentlich ungeeignet. Das werden wir aber später sehen.*

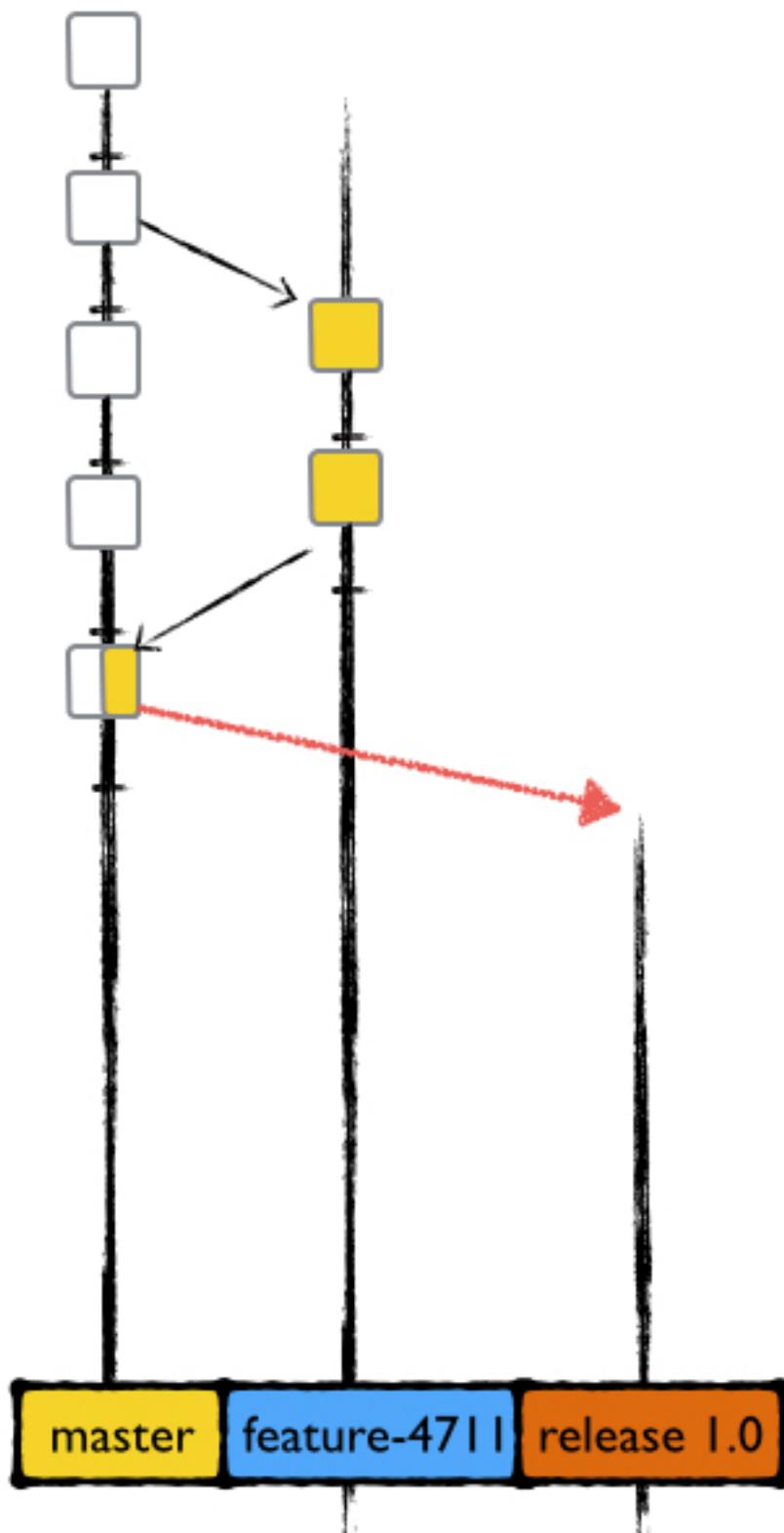
» Lars, ich habe mir jetzt folgendes aufgeschrieben um ein Feature zu implementieren.

```
#Feature Branch erzeugen
git checkout -b "feature-4711"
#hack hack hack
#update gegen master
git rebase master
#Feature taggen
git tag feature-4711
#branch zurückführen
git checkout master
git merge feature-4711
#Stand inkl. Tags pushen
git push --tags origin master
```

» *Genau, sehr gut Karl. Damit schauen wir uns an, wie wir **Releases** machen.*

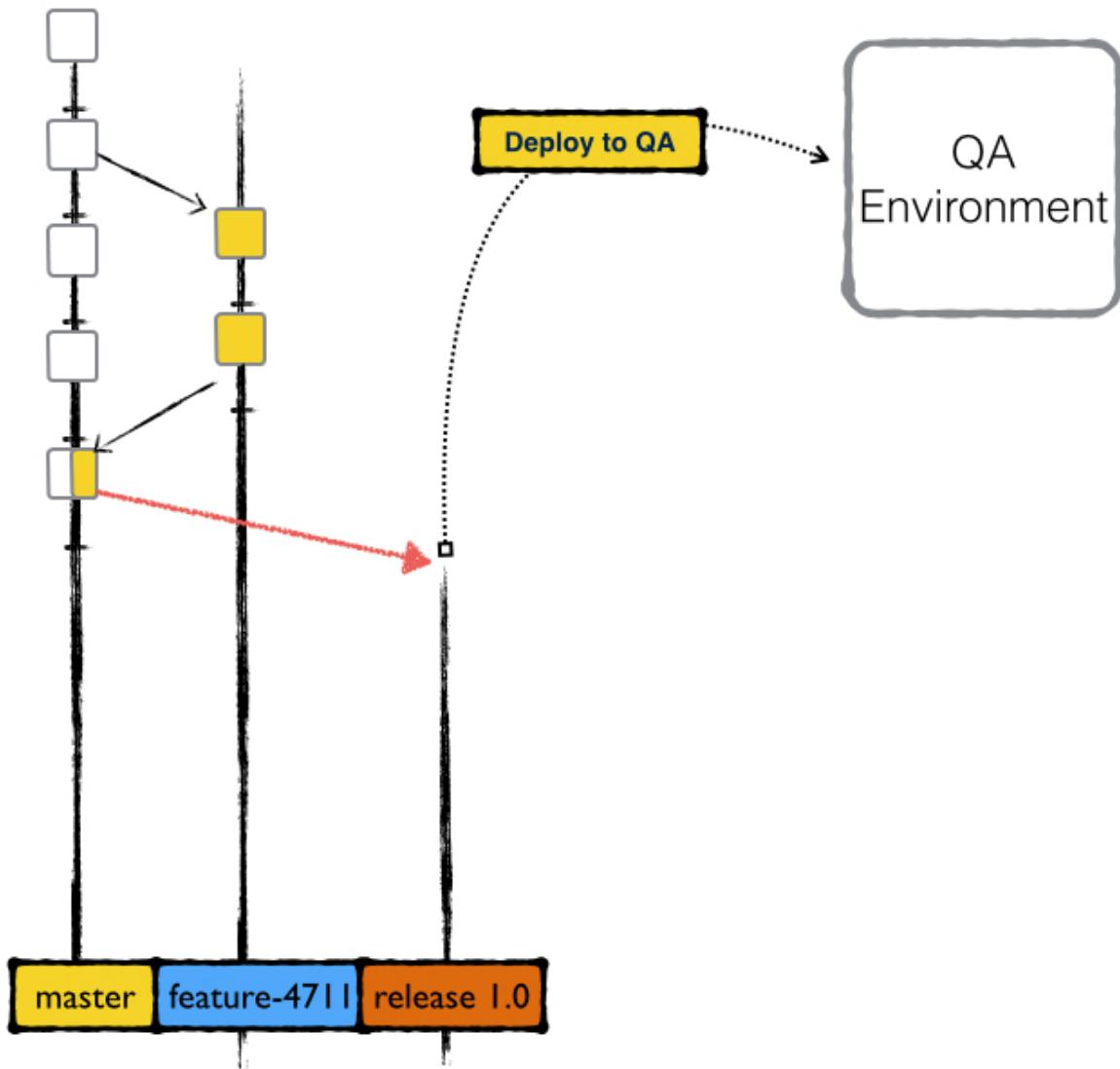
Release it!

» *Releases werden bei uns auf einem eigenen Branch vorbereitet. Wenn es an der Zeit für eine neue Release-Version ist wird ein neuer Branch für die Stabilisierung des Releases vom master gezogen. Vielleicht gehen wir kurz nochmal am Whiteboard durch, wie wir vom master zu einem fertigen Release in Produktion kommen.*

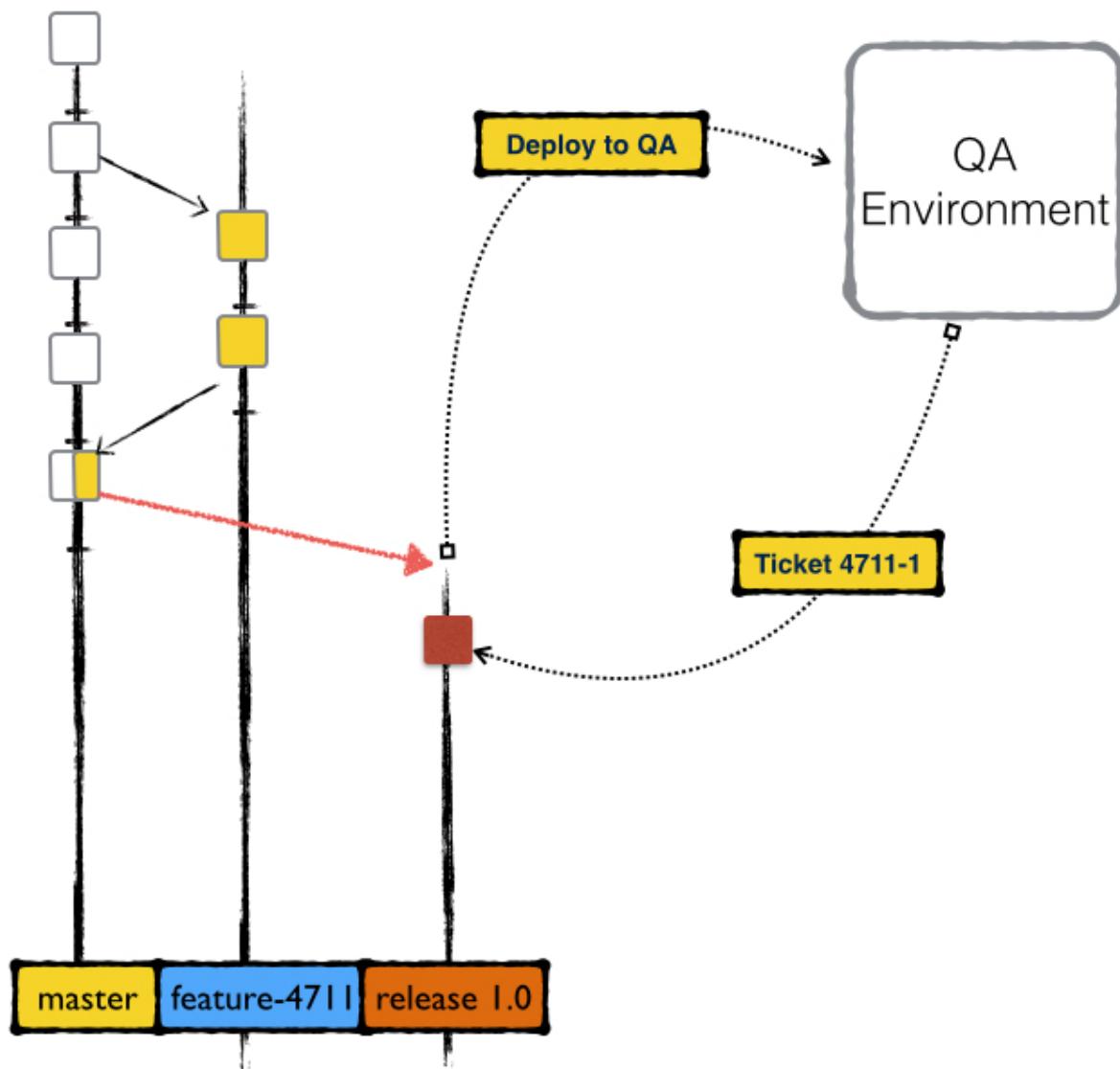


```
git checkout -b 'release-1.0'  
Switched to a new branch 'release-1.0'  
#deploy nach QA
```

» Vom Release-Branch wird ein Build direkt auf eine eigene QA-Umgebung installiert. Auf dieser Umgebung wird nochmal verifiziert, dass alles funktioniert und korrekt ist. Diese Verifikation nehmen sowohl unsere Entwickler selbst als auch die QA Abteilung vor.



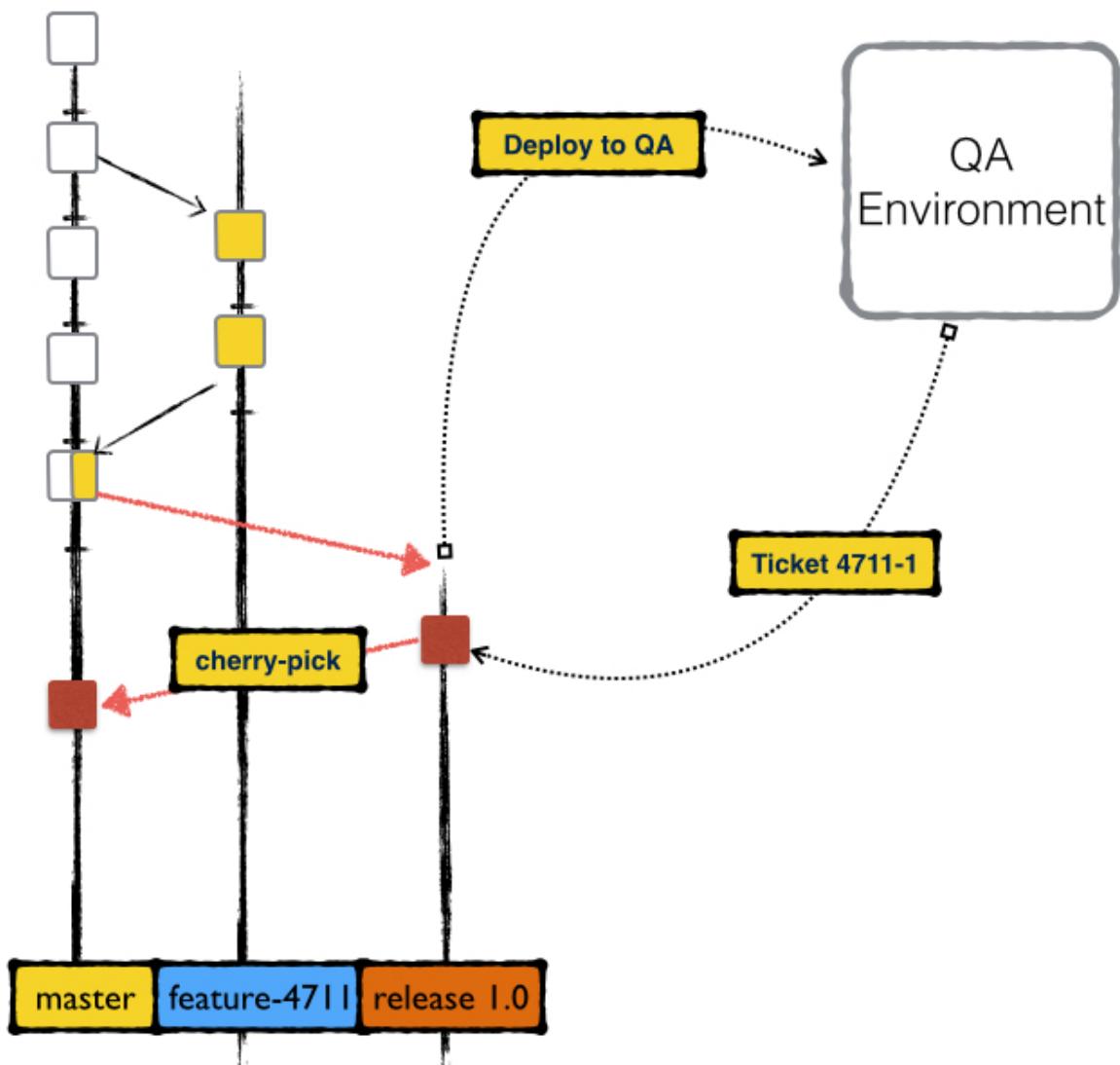
» Soweit ich mich erinnere ist es bisher noch niemals vorgekommen, dass wir während der Vorbereitung auf ein Release keine Bugfixes oder Improvements umzusetzen hatten. Üblicherweise wird von der QA Abteilung ein Bug- oder Improvement-Ticket eingestellt, das von uns bearbeitet wird. Bugfixes und Improvements werden direkt auf dem Release-Branch gefixt. Sobald der Bugfix auf den Release-Branch gepusht worden ist, müssen wir sicherstellen, dass der Commit auch vom Release-Branch auf den master gebracht wird. Auch auf dem master möchten die Entwickler natürlich von den Bugfixes vom Release-Branch profitieren.



```
#bugfix
git makeCommits 1
[release-1.0 1cea586] committing file 1
 1 file changed, 1 insertion(+)
#bugfix commit
git log --oneline -n 1
8708402 bugfix 4711
#deploy nach Prod
```

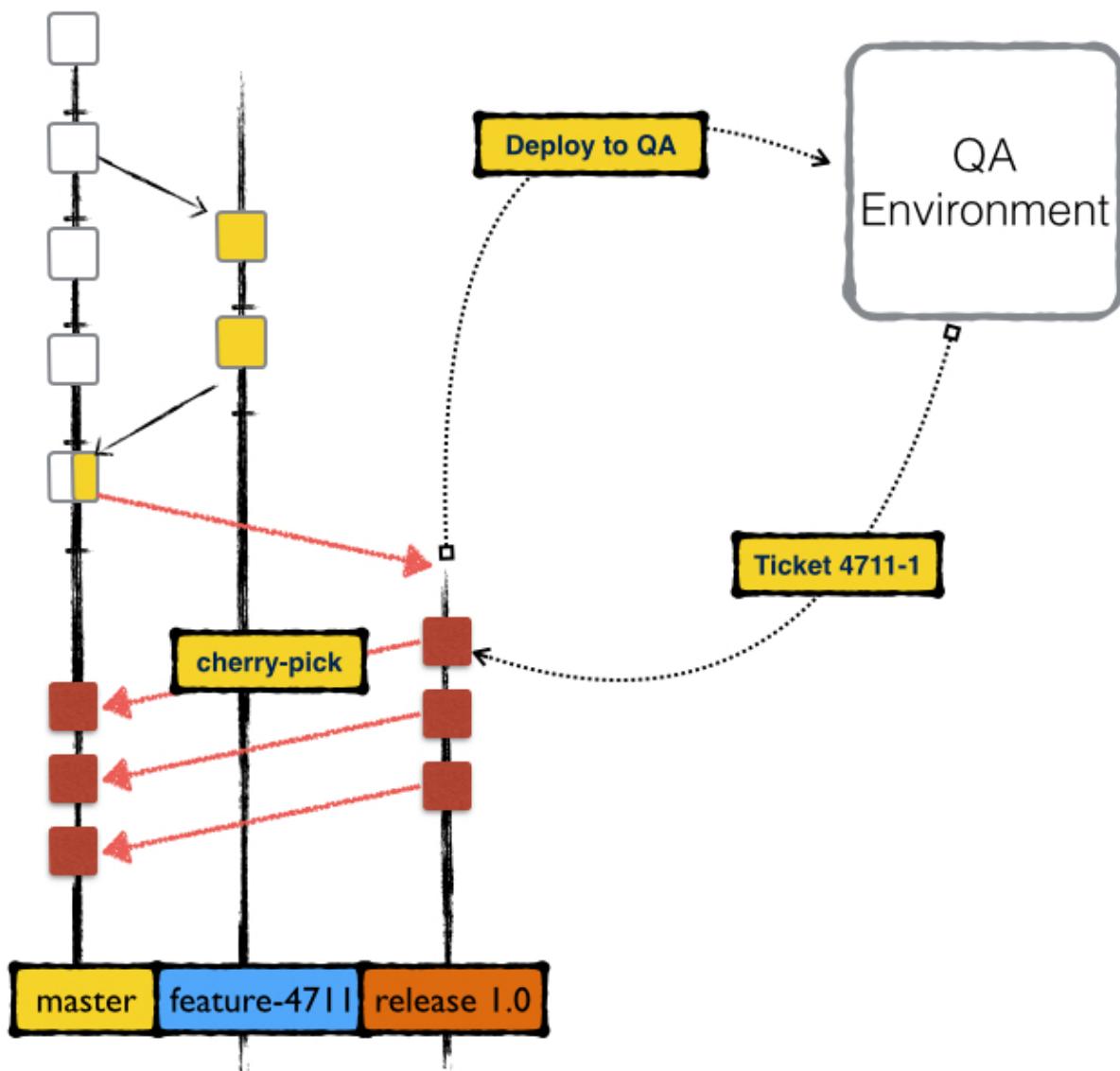
» Üblicherweise versuchen wir, Bugfixes in einem Commit zu machen, damit problemlos mit **Cherry-Pick** gearbeitet werden kann. Der Bugfix wird direkt nach dem Push vom Release-Branch mit einem **Cherry-Pick** oder einem **Merge** auf den master gebracht. Natürlich gibt es auch Bugfixes, die ein wenig größer sind. Dann macht es

natürliche Sinn, nicht direkt auf dem Release-Branch zu arbeiten, sondern einen eigenen Hotfix-Branch zu ziehen.

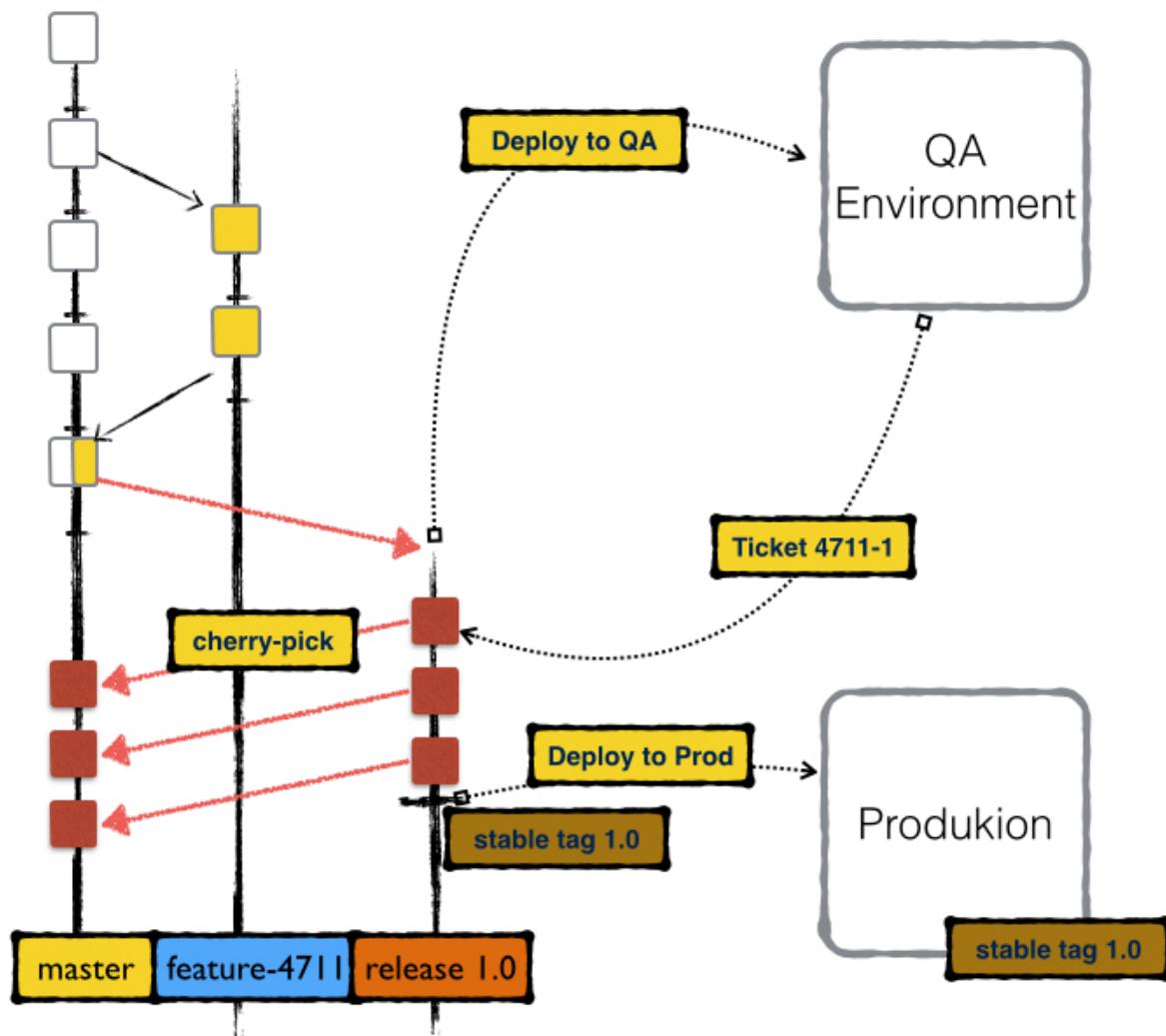


```
git checkout master
Switched to branch 'master'
#cherry pick bugfix
git cherry-pick 8708402
[master 31d841b] bugfix 4711
 1 file changed, 1 insertion(+)
On branch master
```

» Diese Prozedur wiederholt sich für alle Tickets, die während der Stabilisierung des Releases auftreten.



» Geben alle grünes Licht für das Release wird ein annotierter Tag erstellt und die Version live gestellt.



```

git checkout release-1.0
Switched to branch 'release-1.0'
#Tag für Deployment
git tag -a -m "release-1.0" release-1.0 ①
#Push Tag
git push --tags origin release-1.0
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 388 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ../Projekt.git
 * [new tag]           release-1.0 -> release-1.0

```

- ① Erstellung des annotierten Tags - git tag -a -m <message> <tag name>

» Der Release-Branch lebt genauso lange wie das Release selbst. Auch im laufenden Betrieb können natürlich Bugs auftreten, die bisher unentdeckt waren. Das passiert bei uns glücklicherweise sehr selten. Passiert es jedoch trotzdem einmal, gehen wir genauso wie in der Stabilisierungsphase vor.

Annotierte Tags

» Wir taggen Releases nicht leichtgewichtig, sondern mit annotierten Tags. Ein leichtgewichtiger Tag ist nur ein Zeiger auf einen Hashwert bzw. einen Commit. Das haben wir bereits beim Taggen von abgeschlossenen Features gesehen. Ein annotierter Tag hingegen ist ein eigener Objekt-Typ, der erstellt und in `.git/objects` abgelegt wird. Karl, lass uns mal genauer untersuchen, was jetzt passiert ist.

```
#tag datei wurde erstellt
ls .git/refs/tags/
fb-4711
release-1.0

#tag details
git show release-1.0 ①
tag release-1.0 ②
Tagger: Markus <Markus@effectivetrainings.de> ③

release-1.0
Commit: 8708402f583612dea23cef5d8d32711da93a26cb ④
Author: Markus <Markus@effectivetrainings.de>
Date: (13 minutes ago) 2014-02-09 15:09:54 +0100
Subject: bugfix 4711
```

- ① git show <tag-name>
- ② Tag Information
- ③ Entwickler der den Tag erstellt hat
- ④ Commit auf den sich der Tag referenziert.

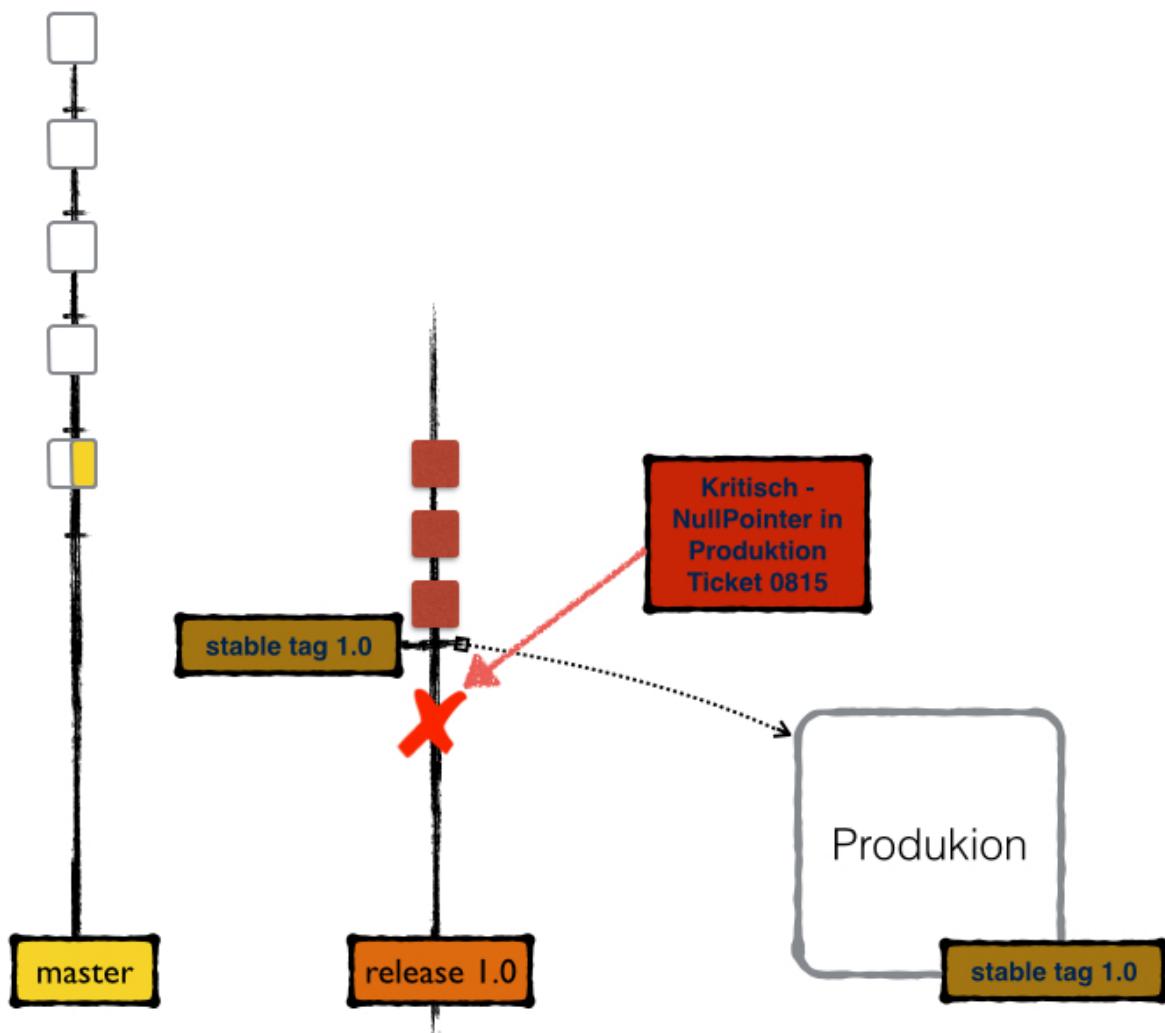
Fehler gibt es immer - Bugfixes

» Für mich klingt das alles ziemlich logisch.

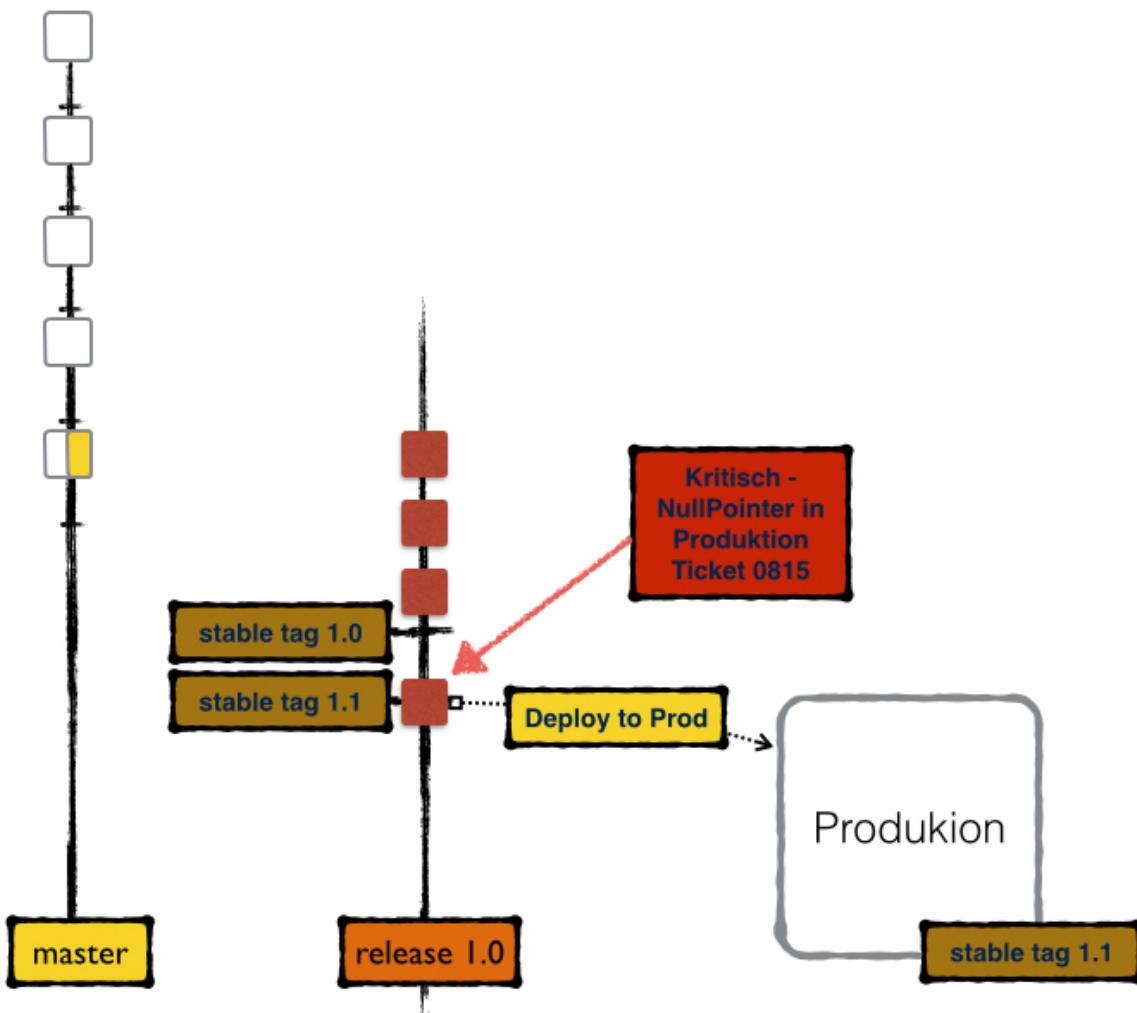
» Genau, das habe ich dir ja bereits gesagt, Git-Flow ist keine Magie. Wenn man genau darüber nachdenkt ist es ganz natürlich so zu arbeiten. Wir haben jetzt unser Release ausgerollt. Wie würdest du jetzt vorgehen, wenn ein kritischer Bug im Live-Betrieb auftritt?

» Du hast ja vorhin bereits erwähnt, dass das Vorgehen für Bugfixes immer gleich ist. Ich gehe also davon aus, dass wir den Bug wieder auf dem aktuellen Release-Branch fixen?

» Genau, stellen wir uns vor, es tritt ein klassischer **NullPointerException** in Produktion auf. Sehr hässlich. Die QA Abteilung stellt uns natürlich hierfür sofort das Ticket **0815** ein. Mir fällt leider gerade keine bessere Nummer ein.



» Ein Entwickler nimmt sich natürlich sofort der Sache an. Der Bug wird direkt auf dem Release-Branch gefixt und getaggt.



```
#Checkout Release Branch
git checkout release-1.0
Switched to branch 'release-1.0'

#Bugfix
git makeCommits 1
[release-1.0 eeeef873] committing file 1
 1 file changed, 1 insertion(+)

#Bugfix verifizieren und taggen
git tag -a -m "Bugfix - Ticket 0815" fix-0815

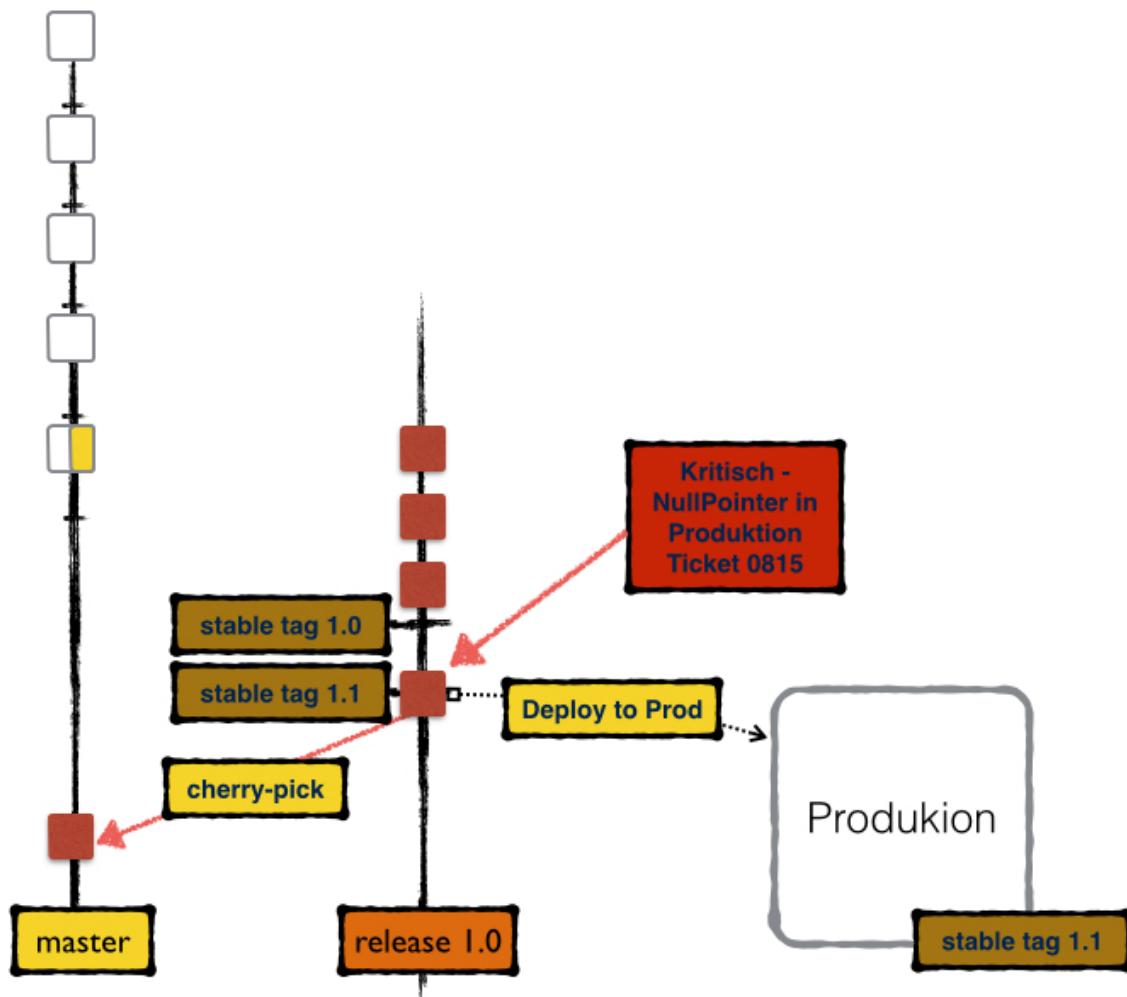
#Pushen
git push --tags origin release-1.0
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 417 bytes | 0 bytes/s, done.
```

```
Total 4 (delta 1), reused 0 (delta 0)
To https://github.com/dilgerma/effective-git-workshop
  8708402..eeef873 release-1.0 -> release-1.0
* [new tag]      fix-0815 -> fix-0815
```

Übung

Ist damit der Bugfix komplett? Welcher Schritt fehlt?

- » Der Bugfix muss natürlich direkt auch zurück auf den **master** gebracht werden.



```
git checkout master
Switched to branch 'master'
```

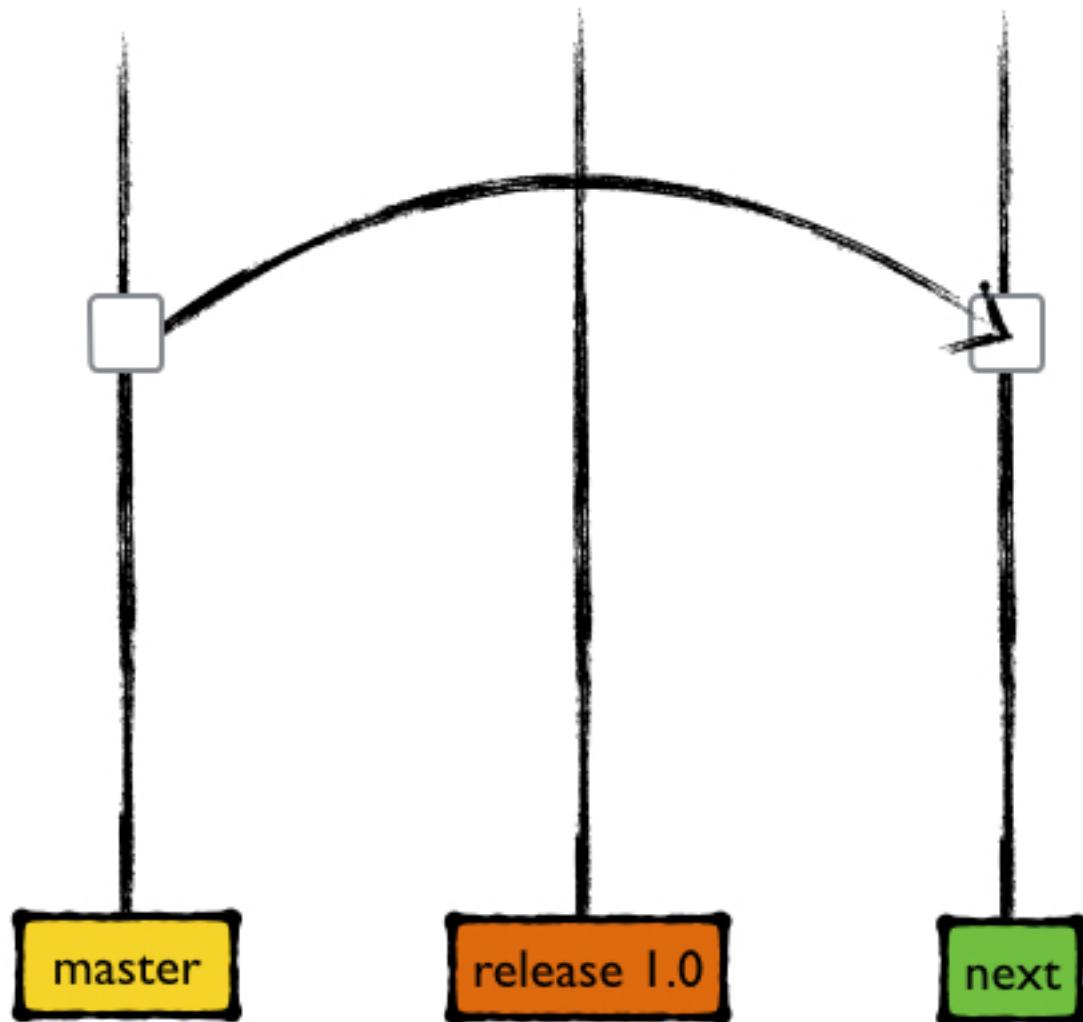
```
#cherry pick bugfix
```

```
git cherry-pick fix-0815 ①
[master abceefb] committing file 1
 1 file changed, 1 insertion(+)
```

- ① Cherry-Pick funktioniert auch direkt mit Tag-Referenzen.

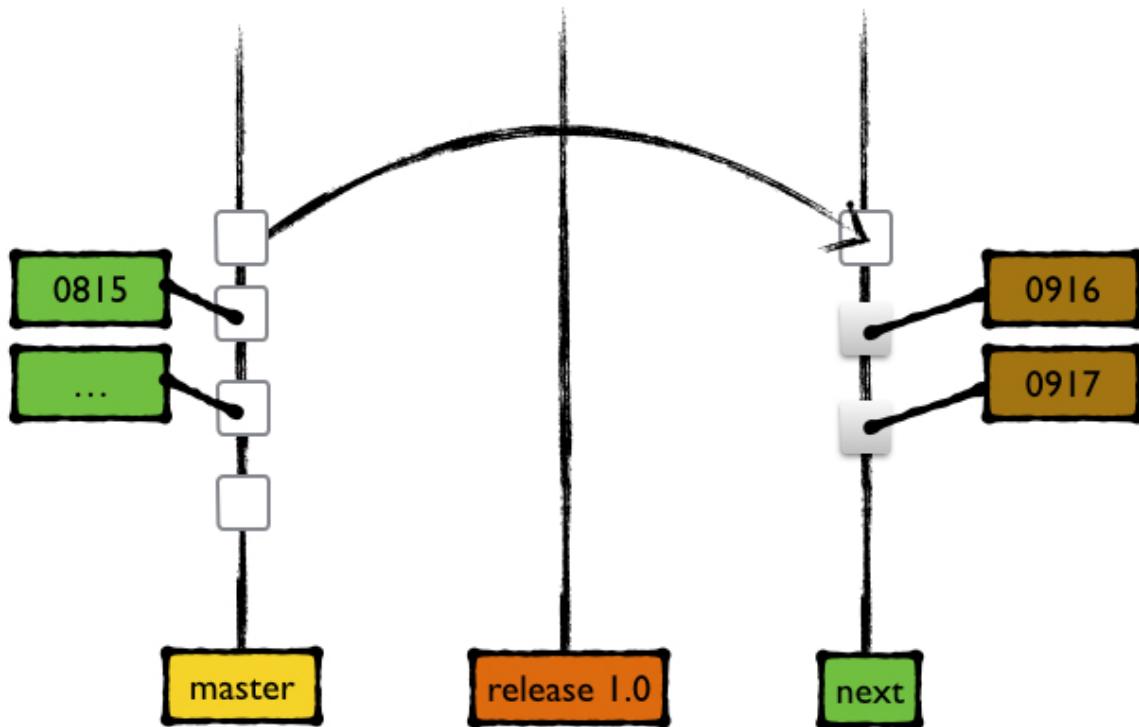
Next-Features - wir denken bereits an morgen

- » Eine Sache fehlt noch. Manchmal gibt es Features, die "vorsorglich" entwickelt werden. Das sollte nicht die Regel sein, denn meistens haben wir genug mit den aktuellen Features zu tun. Manchmal haben wir aber Zeit, um an der Zukunft unseres Produktes zu arbeiten. Das können wir natürlich nicht direkt auf dem **master** machen, denn diese Features sollen ja eben gerade nicht mit dem nächsten Release ausgerollt werden.
- » Wenn ich jetzt logisch denke, dann würde ich sagen, wir brauchen dafür noch einen Branch, auf dem parallel zum master entwickelt wird, richtig?
- » Genau, wir nennen diesen Branch **next**. Wir ziehen den Branch direkt vom **master**, also von der aktuellen Entwicklung.



```
git checkout -b "next"  
Switched to a new branch 'next'
```

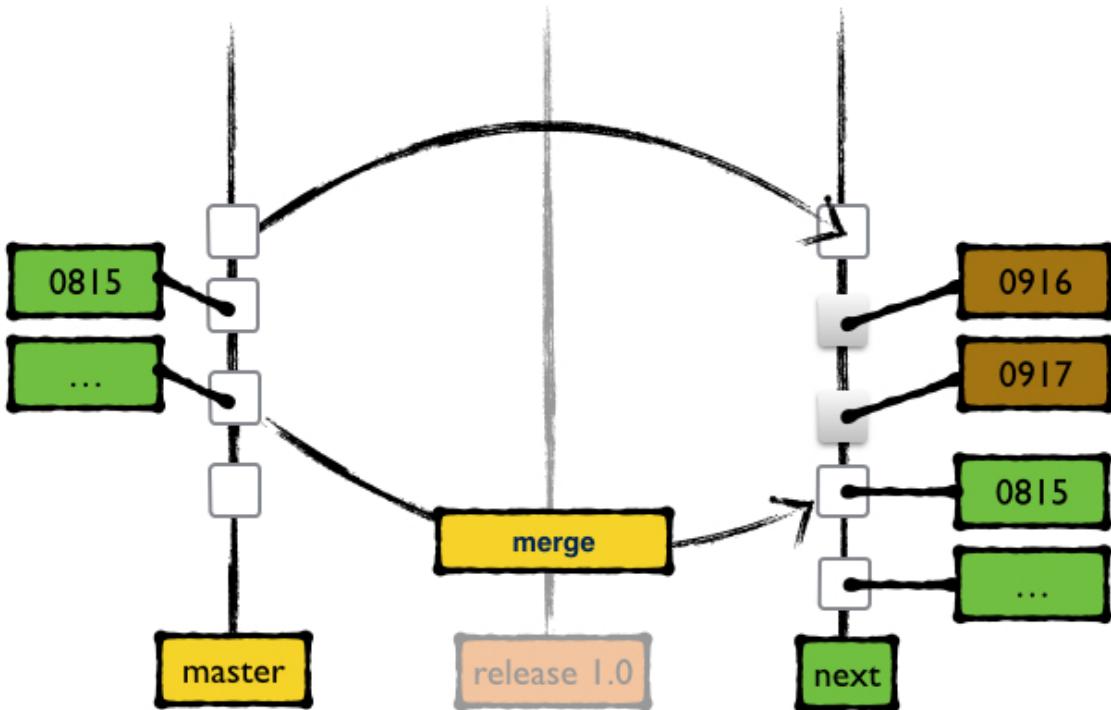
» Wir entwickeln auf beiden Branches parallel. Auf dem **master** werden die Features für das nächste, auf dem **next**-Branch Features für das übernächste Release.



» Hierbei ist nur enorm wichtig, dass in regelmäßigen Abständen vom **master** auf den **next**-Branch gemerged wird. Für die Version vom **next**-Branch haben wir eine eigene Integrations-Umgebung, die wir nur für Developer-Smoke-Tests verwenden. Wir wollen natürlich auch auf diesem Branch alle Entwicklungen, die in der Zwischenzeit auf dem **master** gemacht worden sind.

» Wer macht diese Merges?

» Die werden automatisiert von unserem Repository-Management-System durchgeführt. Das zeige ich dir morgen. Hätten wir dieses System nicht würde ich wahrscheinlich jeden Entwickler diese Arbeit gelegentlich machen lassen. Jeder Entwickler sollte die Prozesse kennen, wir wollen keine Wissensinseln.

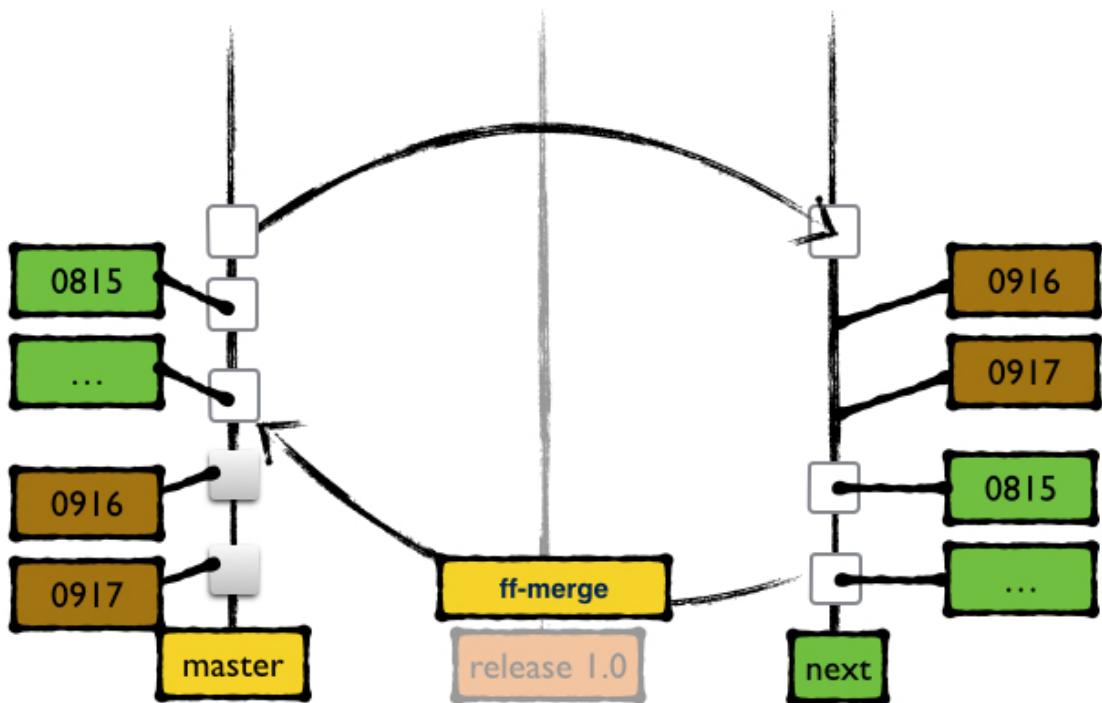


» Ok, das habe ich verstanden. Irgendwann werden aber die Features vom **next**-Branch für das nächste Release relevant. Wie wird das gemacht?

» *Das ist dann sehr einfach. Dadurch dass wir den **next**-Branch mit dem **master** synchron halten muss nur noch ein Merge vom **next** zurück auf den **master** gemacht werden. Dieser merge kann nur ein **fast-forward-merge** sein.*



Treten beim Merge vom **next**-Branch zurück auf den **master** Mergekonflikte auf liegt ein Problem vor, das untersucht werden muss.



» Nach diesem Merge kann der **next**-Branch einfach gelöscht und wenn nötig neu gezogen werden.

» Danke für die Erklärung Lars. Das war sehr aufschlussreich.

» Ich würde dir gerne noch das Tooling für das Git-Flow-Modell zeigen. Wir verwenden es derzeit nicht, trotzdem kann es nicht schaden, dass ich es dir mal zeige. Das Problem ist, dieses Tooling funktioniert nur auf der Konsole. Wir haben aber einige Entwickler, die mit grafischen Tools arbeiten. Mit grafischen Tools beschäftigen wir uns später noch. Wir haben im Team beschlossen, dass wir einheitlich arbeiten möchten, deswegen verwenden wir Git-Flow manuell.

Git Flow Tooling

» Das Tooling haben wir nicht selbst entwickelt, sondern es ist [hier¹⁴](#) ausführlich beschrieben. Im Prinzip handelt es sich um eine kleine Anzahl Shell-Skripte, die auf der Konsole verwendet werden können. Am besten ich erkläre dir das in einem leeren Repository, dann siehst du am besten was passiert.

```
#neues repository
```

¹⁴ <https://github.com/nvie/gitflow>

```
mkdir git-flow-example
cd git-flow-example
git init
```

» Damit haben wir ein leeres Repository und können anfangen. Was wir ausserdem brauchen werden ist das Git-Flow-Tooling. Am besten du lädst es dir schnell von [hier¹⁵](#) herunter und befolgst die Installationsanleitung.

Git Fow Tooling

Karl lädt sich das Tooling herunter und initialisiert es auf seinem System. Das Tooling funktioniert auf allen gängigen Systemen auf denen eine Konsole verfügbar ist.

» Karl, um jetzt Git-Flow auch für dein neues Repository zu initialisieren führst du auf der Konsole einfach folgendes aus.

```
git flow init
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master] release 1
Branch name for "next release" development: [develop] master 2
How to name your supporting branch prefixes? 3
Feature branches? [feature/] fb- 4
Release branches? [release/] release- 5
Hotfix branches? [hotfix/] hotfix- 6
Support branches? [support/] support- 7
Version tag prefix? [] 8
```

- 1** Welcher Branch wird für das nächste Release verwendet. Branches für Releases heissen bei uns **release-<Release-Nr>**
- 2** Welcher Branch wird für **next**-Development verwendet. Git-Flow schlägt **develop** vor, wir entwickeln aber auf dem **master**.
- 3** Das Prefix für Feature-Banches, die von Git-Flow automatisch erstellt werden. Git-Flow schlägt hier **feature/** vor, wir haben aber die Erfahrung gemacht, dass der "/" im Namen auf einigen Systemen Probleme verursacht, wir bevorzugen also **fb-**.

15 <https://github.com/nvie/gitflow>

- ④ Welches Prefix sollen **Release-Banches** haben. Wir verwenden **release-**.
 - ⑤ Welches Prefix sollen Hotfix-Banches haben. Hotfix-Banches werden vom Release-Branch gezogen und werden verwendet, um einen Hotfix direkt für eine Release zu erstellen.
 - ⑥ **Support** Branches verwenden wir für unsere Entwicklung nicht. Sie werden aber normalerweise ähnlich wie **Hotfix**-Branches behandelt.
 - ⑦ Git-Flow taggt automatisch Releases. Wir verwenden hierfür kein Prefix, sondern lassen den Standard stehen.
- » Interessant, ist jetzt schon was im Repository passiert?
-

```
git branch
* master
  release
```

» *Ja, wie du siehst wurden bereits die beiden wichtigsten Branches erzeugt. **master** für die tägliche Entwicklung und der initiale **release**-Branch.*

» Ok, interessant. Wie starte ich jetzt mit der Arbeit?

» *Ganz einfach, gehen wir davon aus, du möchtest ein neues Feature entwickeln.*

```
git flow feature start 4711 ❶
Switched to a new branch 'fb-4711'

Summary of actions:
- A new branch 'fb-4711' was created, based on 'next'
- You are now on branch 'fb-4711'
```

Now, start committing on your feature. When **done**, use:

```
git flow feature finish 4711
```

» Das ist wirklich praktisch! Sehe ich das richtig, dass Git-Flow gerade automatisch den Feature-Branch für mein Feature erzeugt hat?

» *Genau, so ist es. Schau dir doch mal die Liste an Branches an.*

```
git branch
* fb-4711
  master
  release
```

» *Git-Flow hat den Branch erzeugt und direkt ausgecheckt. Du kannst also direkt anfangen zu arbeiten. Am besten, du machst einen Commit und wir simulieren, dass wir damit das Feature abgeschlossen haben.*

```
echo "working with git flow is so easy" >> feature.txt
git add feature.txt
git commit -m "4711 - finished"
[fb-4711 18cf53e] 4711 - finished
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
```

» *Wunderbar, damit haben wir das Feature abgeschlossen. Was wäre jetzt der nächste Schritt, erinnerst du dich noch?*

Übung

Erinnern Sie sich noch an die einzelnen Schritte im Workflow?

Was muss passieren, nachdem ein Feature abgeschlossen ist?

Zeichnen Sie ein Diagram!

» Natürlich erinnere ich mich, wir führen den Feature-Branch zurück. Ich schätze, auch dafür gibt es ein passendes Kommando?

» *Genau. Es ist ganz einfach.*

```
git flow feature finish 4711
Switched to branch 'master'
Updating acafbbc..61c0b63
Fast-forward
  feature.txt | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 feature.txt
Deleted branch fb-4711 (was 61c0b63).
```

Summary of actions:

- The feature branch 'fb-4711' was merged into 'master'
 - Feature branch 'fb-4711' has been removed
 - You are now on branch 'master'
-

» Ok, ich glaube ich verstehe. Wir schliessen das Feature ab, d.h. Git-Flow wechselt auf den **master**, unseren aktuellen Entwicklungs-Branch. Git-Flow macht automatisch einen Merge des Features und entfernt anschließend den Feature-Branch. Der Branch wird nach der Rückführung nicht mehr gebraucht.

```
git branch
* master
  release

#log
git log
Commit: 61c0b63830ddefaee7cdf7c7c224b7ba171f69da
Author: dilgerm <martin@effectivetrainings.de>
Date:   (5 minutes ago) 2014-02-13 18:11:50 +0100
Subject: 4711 - finished

Commit: acafbbce40cdc602b74879135a1ac8e8e239532c
Author: dilgerm <martin@effectivetrainings.de>
Date:   (10 minutes ago) 2014-02-13 18:05:54 +0100
Subject: Initial commit
```

» Versuchen wir das Ganze nochmal?

Übung

Implementieren Sie ein zweiten Feature 4911.

Schliessen Sie das Feature mit Hilfe von Git-Flow ab.

```
git flow feature start 4911

echo "feature-4911 - finished" >> 4911.txt
git add 4911.txt
git commit -m "4911 - finished"
[fb-4911 fd75f84] 4911 - finished
 1 file changed, 1 insertion(+)
 create mode 100644 4911.txt

#Feature abschliessen
git flow feature finish 4911
```

» Wir haben schon viel über **Rebase** und **Merge** gesprochen. Was macht Git-Flow?

» *Gute Frage, Karl. Git-Flow arbeitet standardmäßig mit Merges. Wir haben ja bereits besprochen, dass ein **Rebase** unter Umständen sinnvoll sein kann, bevor ein Feature zurückgeführt wird. Dafür bietet Git-Flow aber keine Unterstützung. Es ist Aufgabe des jeweiligen Entwicklers bevor er das Feature abschliesst.*

» Ich verstehe, ich hätte also vor dem **git flow feature finish** einfach ein **Rebase** gegen den **master** machen müssen?

» *Ja genau. Bereiten wir ein Release vor?*

Übung

Kennen Sie noch die Schritte um ein Release vorzubereiten?

Was ist zu tun?

Zeichnen Sie ein Diagramm!

» Ich versuche das mal zusammenzufassen. Um jetzt ein Release vorzubereiten würden wir einen Release-Branch vom **master** ziehen. Ab diesem Zeitpunkt wird auf dem **master** dann für das nächste Release entwickelt. Alles für das aktuelle Release wird auf dem neuen **Release-Branch** gemacht. Der **Release-Branch** dient uns zur Stabilisierung. Die Version vom Release-Branch baut ihr separat und deployt sie in eine QA-Umgebung, auf der das Release getestet wird. Aber wie mache ich das jetzt mit **Git-Flow**?

» *Am besten du findest es selbst heraus. Schau dir am besten mal die Hilfe von Git-Flow an.*

```
git flow
usage: git flow <subcommand>
```

Available subcommands are:

```
init      Initialize a new git repo with support for the branching
model.
feature   Manage your feature branches.
release   Manage your release branches.
hotfix    Manage your hotfix branches.
support   Manage your support branches.
```

```
version Shows version information.
```

```
Try 'git flow <subcommand> help' for details.
```

» Ok, ich vergesse immer, die Hilfe zu verwenden. Git-Flow bietet also separate Kommandos für die einzelnen Branch-Typen?

» Ja genau, jetzt interessieren wir uns hauptsächlich für **Releases**. Am besten du lässt dir die Hilfe für Releases anzeigen.

```
git flow release  
No release branches exist.
```

You can start a new release branch:

```
git flow release start <name> [<base>]
```

Übung

Können Sie mit den Informationen, die Sie jetzt haben ein Release vorbereiten?

» Ich denke, das schaffe ich. Wie nennen wir das Release?

» *Das wird eine 1.0!*

```
git flow release start 1.0  
Switched to a new branch 'release-1.0'
```

Summary of actions:

- A new branch 'release-1.0' was created, based on 'master'
- You are now on branch 'release-1.0'

Follow-up actions:

- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When **done**, run:

```
git flow release finish '1.0'
```

» Ok, das habe ich erwartet. Git-Flow hat einen neuen Branch erzeugt, den wir für die Release-Stabilisierung verwenden können.

```
git branch
  master
  release
* release-1.0
```

» Wie geht es jetzt weiter?

» Am besten wir machen einige Commits auf dem Branch und machen das Release dann fertig, oder?

» Ich versuche das.

```
git makeCommits 4
[release-1.0 13bdbd1] committing file 1
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
[release-1.0 683cc21] committing file 2
 1 file changed, 1 insertion(+)
 create mode 100644 file2.txt
[release-1.0 5461a94] committing file 3
 1 file changed, 1 insertion(+)
 create mode 100644 file3.txt
[release-1.0 e30ac56] committing file 4
 1 file changed, 1 insertion(+)
 create mode 100644 file4.txt
```

```
#log
git log --oneline
e30ac56 committing file 4
5461a94 committing file 3
683cc21 committing file 2
13bdbd1 committing file 1
fd75f84 4911 - finished
61c0b63 4711 - finished
acafbbc Initial commit
```

» Ok, und jetzt machen wir das Release fertig.

```
git flow release finish 1.0

#wechsel auf release branch
Switched to branch 'release'
Merge made by the 'recursive' strategy.
```

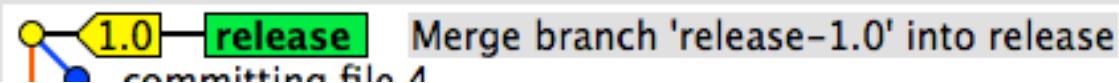
```
[...]
Switched to branch 'master'
Merge made by the 'recursive' strategy.
[...]
#Tag release, Message eingeben
```

» Ok, jetzt ist aber einiges passiert. Ich versuche das nochmal zusammenzufassen.

Übung

Können Sie anhand des obigen Listings erklären, was passiert ist und warum?

» Wir sind auf dem **Release-Stabilisierungs-Branch** und weisen Git-Flow an, das Release fertig zu machen. Zuerst wechselt Git-Flow also auf den **Release-Branch** und merged den Stabilisierungsbranch hierher. Damit bringen wir quasi einen neuen Stand auf den bereits bestehenden **Release-Branch**. Der **Release-Branch** ist nicht für Stabilisierung gedacht, sondern hier ist der echte Release-Stand für unsere Kunden. Anschließend setzt Git-Flow einen Tag auf dem Release-Branch, richtig?



» Genau, denn jetzt ist klar, welcher Stand für das nächste Release final ist. Das hält Git-Flow mit Hilfe eines Tags fest.

» Weiter, der nächste Schritt besteht nun darin, den Stand vom **Stabilisierungs-Branch** auch auf den master, also die aktuelle Entwicklung zu bringen. Ganz einfach deswegen, weil vielleicht für die Stabilisierung wichtige Bugfixes oder beispielsweise Performance-Improvements gemacht worden sind, die wir natürlich auch auf dem **master** haben möchten. Den Stabilisierungs-Branch brauchen wir anschließend nicht mehr und er wird von Git-Flow gelöscht.

» Perfekt Karl, genauso funktioniert ein Release mit Git-Flow. Einfach oder? Du siehst übrigens auch an der Ausgabe in der Konsole, was Git-Flow jeweils gemacht hat.

Summary of actions:

- Latest objects have been fetched from 'origin'
- Release branch has been merged into 'release'
- The release was tagged '1.0'
- Release branch has been back-merged into 'master'

- Release branch '`release-1.0`' has been deleted

» Ok, sind wir damit dann fertig?

» *Noch nicht ganz, das Release ist jetzt draussen, unsere Kunden sind glücklich bis zum Tag X. Wir haben einen kritischen Bug übersehen, der jetzt natürlich gefixt werden muss. Wie würdest du jetzt vorgehen?*

» Zunächst mal würde ich mir wieder die Hilfe von Git-Flow anschauen.

» *Na das ist doch zumindest ein Anfang.*

```
git flow
usage: git flow <subcommand>
```

Available subcommands are:

```
init      Initialize a new git repo with support for the branching
model.

feature   Manage your feature branches.
release   Manage your release branches.
hotfix    Manage your hotfix branches.
support   Manage your support branches.
version   Shows version information.
```

Try '`git flow <subcommand> help`' **for** details.

» Da der Bug kritisch zu sein scheint würde ich auf das Hotfix-Kommando tippen, richtig?

» *Exakt, Fehler in Produktion werden immer als Hotfixes behandelt.*

```
git flow hotfix
No hotfix branches exist.
```

You can start a new hotfix branch:

```
git flow hotfix start <version> [<base>]
```

» Habt ihr eine Versionierungstrategie für Hotfix-Branhes?

» *Hotfixes verhalten sich genauso wie Features. Die Version ist immer die Tasknummer aus der QA-Abteilung.*

```
git flow hotfix start 5011
Switched to a new branch 'hotfix-5011'
```

Summary of actions:

- A new branch '`hotfix-5011`' was created, based on '`release`'
- You are now on branch '`hotfix-5011`'

Follow-up actions:

- Bump the version number now!
- Start committing your hot fixes
- When `done`, run:

```
git flow hotfix finish '5011'
```

» *Karl, das ist extrem wichtig, versteht du was Git-Flow hier macht?*

Übung

Lars hat recht. Es ist sehr wichtig zu verstehen wie **Hotfixes** behandelt werden.
Können Sie erklären, was jetzt passiert ist?

» Ich versuche es mal zu erklären. Git-Flow erstellt einen neuen Branch vom `release`-Branch weg, denn der Bug ist in Produktion aufgetreten.

```
git branch
* hotfix-5011
  master
  release
```

» Auf diesem Hotfix-Branch lösen wir das Problem.

```
echo "hotfix" >> hotfix.txt
git add hotfix.txt
git commit -m "5011 - Hotfix"
[hotfix-5011 9a1f5f0] 5011 - Hotfix
 1 file changed, 1 insertion(+)
 create mode 100644 hotfix.txt
```

» *Sehr gut, ich würde sagen, das Problem ist behoben und wir versuchen schnellstmöglich, den Fix in Produktion zu bringen.*

```
git flow hotfix finish 5011
```

[...]

Summary of actions:

- Latest objects have been fetched from '`origin`'
 - Hotfix branch has been merged into '`release`'
 - The hotfix was tagged '`5011`'
 - Hotfix branch has been back-merged into '`master`'
 - Hotfix branch '`hotfix-5011`' has been deleted
-

» Git-Flow verhält sich hier ähnlich wie der Beendigung des Release-Stabilisierungs-Banches. Der Hotfix-Branch wird zunächst in den **Release**-Branch gemerged, damit die Release-Version stabil wird. Anschließend setzt Git-Flow einen Tag auf dem Release-Branch. Zu guter Letzt wird der Stand wieder zurück in den **master** gemerged, damit auch die Entwickler etwas davon haben. Alles vollautomatisch, ich bin wirklich begeistert. Der Hotfix-Branch wird automatisch gelöscht.

```
git branch
* master
  release
```

» Lars, kannst du mir erklären, wie genau sich das mit **Support**-Branches verhält, was genau ist das?

» *Erklärt ist das sehr schnell. Stell dir vor, einige Kunden kaufen unser Produkt in der Version 1. Diese Version haben wir gerade eben Released und zum Kauf angeboten. In drei Monaten releasesen wir eine neue Version, die die Kunden für einen kleinen Aufpreis kaufen können. Es gibt immer wieder Kunden, die diesen Aufpreis nicht zahlen möchten oder aber einfach nicht können. Diese Kunden sind an die Version 1 gebunden. Falls jetzt aber im laufenden Betrieb kritische Bugs auftreten, dann möchten wir diese trotzdem auch für alte Versionen fixen können. Einen **Support**-Branch kannst du dir vorstellen wie einen alten **Release**-Stand, auf dem noch Bugs gefixt werden können.*

Wie gesagt, wir arbeiten derzeit nicht mit Support-Branches und die Unterstützung in Git-Flow ist hier, naja, rudimentär. Aber du kannst dir gerne mal die Hilfe dazu anschauen.

```
git flow support
note: The support subcommand is still very EXPERIMENTAL!
note: DO NOT use it in a production situation.
No support branches exist.
```

You can start a new support branch:

```
git flow support start <name> <base>
```

» Kann ich das einfach mal schnell ausprobieren?

» *Klar, um einen Support-Branch zu starten brauchst du einen Namen, das wird meistens die Release-Version sein, die du supporten möchtest und eine Basis. Die Basis ist der Tag des Releases, das du supporten möchtest.*

» Ok, wenn ich dich richtig verstehre müsste der Befehl folgender sein.

```
git flow support start 1.0 1.0
note: The support subcommand is still very EXPERIMENTAL!
note: DO NOT use it in a production situation.
Switched to a new branch 'support-1.0'
```

Summary of actions:

- A new branch '`support-1.0`' was created, based on '`1.0`'
 - You are now on branch '`support-1.0`'
-

» Sehr schön, genau wie ich vermutet habe. Git-Flow checkt nicht einen bestimmten Branch aus, sondern den **Tag** den ich als **Basis** angegeben habe.

» *Interessant ist, es ist nicht vorgesehen, dass Supporting-Banches je wieder gelöscht werden, denn du möchtest deine Version normalerweise für immer Supporten, oder zumindest die einfache Möglichkeit haben, die Version nochmals zu bauen.*

5.3. Tag 2 endet

» *Genug für heute, Karl. Ich hoffe du hast einiges zum Thema Workflows gelernt.*

» Danke Lars, mir raucht zwar der Kopf aber ich denke, das meiste habe ich verstanden. Ich denke, für die meisten Projekte ist **Git-Flow** tatsächlich das richtige Modell. Ich werde mir auf dem Weg nach München nochmal genau anschauen, wie sich die Modelle unterscheiden.

» *Es gibt übrigens noch mehr Modelle, mit denen du dich beschäftigen kannst, wenn du möchtest. Beispielsweise hat GitHub¹⁶ das Modell GitHub-Flow¹⁷. GitHub-Flow ist im Prinzip ein vereinfachtes Modell von Git-Flow. Lies dir das gerne mal durch.*

¹⁶ <http://www.github.com>

¹⁷ <http://scottchacon.com/2011/08/31/github-flow.html>

» Das werde ich, danke Lars. Was ist für morgen geplant?

» *Morgen werden wir endlich ein wenig produktiv arbeiten. Wir haben ein Legacy-Projekt hier, das derzeit noch mit Subversion arbeitet. Da wir jetzt das erste Mal seit längerer Zeit an diesem Projekt wieder Änderungen vornehmen möchten wir die Gelegenheit nutzen und das Projekt von Subversion nach Git migrieren.*

» Sieht aus als hätten wir noch viel zu tun?

» *Natürlich! Glaub mir, die Arbeit wird uns nicht ausgehen.*

» Danke Lars, bis morgen!

6. Tag 3 - Die Migration nach Git

Karl hat sich die ganze Rückfahrt nach München Gedanken über Git-Flow und die Arbeit mit dem Branching-Modell gemacht. Das einfache Arbeiten mit Branches und die Hilfe, die Git beim Mergen zwischen den Branches bietet haben ihn schwer beeindruckt. In Karls bisherigen Projekten war eigentlich fast immer **Subversion** (und in den schlimmsten Fällen **CVS** und **ClearCase**) im Einsatz. Für Karl ist es undenkbar, ein Branching-Modell wie Git-Flow mit einem dieser Systeme zu realisieren. Langsam fängt Karl an, die Idee von dezentraler Versionskontrolle so richtig zu verstehen und fragt sich tatsächlich, warum nicht schon mehr Projekte auf dieses System migriert sind. Ist es tatsächlich so kompliziert, den Sprung von Subversion hin zu Git zu machen? Er freut sich schon sehr auf den neuen Arbeitstag mit Lars, da genau diese Frage offenbar heute beantwortet werden soll. Wieder pünktlich zum Daily-Scrum findet sich Karl im Büro in Nürnberg ein und kann es kaum erwarten, sich mit Lars an den Rechner zu setzen und seine erste Subversion-Git-Migration zu machen.

» *Hallo Karl! Ich hoffe, Du hast die Informationen von gestern gut verdaut?*

» Guten Morgen, Lars. Ja danke, ich habe mir unterwegs noch sehr viele Gedanken gemacht und je länger ich über diese Art und Weise zu arbeiten nachdenke, desto mehr ärgere ich mich, dass ich mich nicht schon viel früher mit dem Thema befasst habe.

» *Siehst du, ich hatte dich ja vorgewarnt, wenn Du einmal angefangen hast, wirklich mit Git zu arbeiten kannst du nicht mehr zurück.*

» Ich frage mich tatsächlich, warum nicht mehr Unternehmen bereits auf den fahrenden Zug aufgesprungen sind und die Migration nach Git längst erledigt haben. Ist es so kompliziert?

» Überhaupt nicht, wenn man auf einige Dinge achtet, dann ist es tatsächlich sogar sehr einfach und problemlos. Aber gut, dass Du dir genau die richtigen Fragen schon selbst gestellt hast. Ich hatte dir ja schon gesagt, dass wir heute eines unserer letzten auf Subversion basierten Projekte nach Git migrieren möchten. So siehst du gleich, was es eigentlich bedeutet, ein Projekt inklusive der kompletten Historie nach Git zu migrieren. Am besten du holst dir noch schnell einen Kaffee und dann fangen wir direkt an.

5 min später

» Ok, Karl. Das Projekt das wir migrieren ist aktuell auf Google-Code gehostet. Und zwar hier: <https://code.google.com/p/git-subversion-migration/>. Wir haben schon mehrere unserer Projekte von Subversion nach Git migriert. Dabei haben wir eigentlich immer die gleichen Schritte durchgeführt. Hierbei sind einige Skripte entstanden, die wir für jede Migration wiederverwenden. Diese Skripte werden selbstverständlich wieder in Git verwaltet und finden sich auf GitHub¹⁸. Am besten wir checken uns diese Skripte zunächst mal aus. Die Skripte enthalten überhaupt keine Magie und sind sehr, sehr einfach. Man kann die Migration auch problemlos ohne diese Skripte machen.

Übung

Klonen Sie sich das Repository unter <https://github.com/dilgerma/git-migrations>

```
git clone https://github.com/dilgerma/git-migrations
Cloning into 'git-migrations'...
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 0), reused 7 (delta 0)
Unpacking objects: 100% (7/7), done.
Checking connectivity... done.

#
svn-user-map.sh ①
show-svn-revisions.sh ②
svn-clone-repository.sh ③
apply-svn-ignore.sh ④
```

① Skript zum Erstellen des User-Mappings von Subversion hin zu Git

18 <http://www.github.com>

- ② Zeigt die letzten **n** - Revisions aus dem Subversion-Repository.
- ③ Klont das Subversion-Repository und überführt es in ein äquivalentes Git-Repository.
- ④ Überführt die svn-ignores in eine äquivalente .gitignore-Datei
 - » Lars, das verstehe ich nicht, wieso brauchen wir ein Mapping der User von Subversion hin zu Git?
 - » *Gute Frage, Karl. Die Lösung kennst du aber eigentlich bereits, überleg nochmal genau.*

Übung

Es gibt einen fundamentalen Unterschied zwischen dem User-Management in Subversion und Git. Kennen Sie ihn und können ihn in wenigen Sätzen erklären?

» Ich bin mir nicht ganz sicher. Wenn ich mich an meine letzten Projekte erinnere, dann sieht man in Subversion eigentlich immer nur das Benutzerkürzel. In Git haben wir den Benutzernamen und eine E-Mailadresse, korrekt? Ist das Usermapping dafür gedacht? Um aus dem Benutzerkürzel die korrekten Git-Daten zu bekommen? Also einen Namen und eine E-Mailadresse?

» *Perfekt Karl, das ist genau die richtige Erklärung. Am besten wir betrachten uns einmal die Original-Sourcen im Subversion-Repository. Ich hoffe, ich habe noch irgendwo meinen mittlerweile angestaubten Subversion-Client installiert.*

Übung

Auch wenn es weh tut, installieren Sie sich einen **aktuellen (hüstel)** Subversion-Client auf Ihrem Rechner.

» *Wir checken uns zunächst das Subversion-Repository aus, um schnell zu prüfen, was überhaupt zu migrieren ist.*

```
svn checkout https://git-subversion-migration.googlecode.com/svn/
A    svn/wiki
A    svn/trunk
[...]
A    svn/branches
```

```
A    svn/branches/feature-1
A    svn/branches/feature-1/branch-file.txt
A    svn/branches/feature-1/another-test-file.txt
A    svn/branches/feature-1/test-file.txt
A    svn/branches/another-branch
A    svn/branches/another-branch/another-test-file.txt
A    svn/branches/another-branch/test-file.txt
A    svn/branches/another-branch/another-file-on-another-branch.txt
A    svn/tags
A    svn/tags/release-v1
A    svn/tags/release-v1/another-test-file.txt
A    svn/tags/release-v1/release-1.0.txt
A    svn/tags/release-v1/test-file.txt
[...]
```

» Ok, wir sehen hier schon, es ist alles dabei. Wir haben mehrere Branches, einen Tag und natürlich den **Trunk**, den wir migrieren möchten.

» Ist es tatsächlich notwendig, dass wir alles migrieren? Du meintest, das Projekt wäre jetzt schon eine ganze Weile nicht mehr angepasst worden. Die Chancen sind also groß, dass zumindest die Feature-Branches mittlerweile hoffnungslos veraltet sind?

» Du hast völlig recht, mit großer Wahrscheinlichkeit ist das der Fall. Würden wir die Subversion-Repositories weiterhin behalten wäre es auch eine valide Option, beispielsweise nur den **Trunk** zu migrieren. Wir sind allerdings bestrebt, in naher Zukunft überhaupt keine Subversion-Repositories mehr vorzuhalten. Wir migrieren also normalerweise zunächst alles aus dem Subversion-Repository nach Git und löschen anschließend das Subversion-Repository komplett. Es schadet also nicht, einfach das komplette Repository zu migrieren. Zunächst prüfen wir mit **svn log**, welche Revisions wir zu migrieren haben und welche User als Committer im Repository aktiv waren. Wenn ich mich richtig erinnere, war in diesem Projekt nur ein Entwickler aktiv, und der ist schon lange nicht mehr bei uns.

```
At revision 76.
~/development/.../svn/trunk$ svn log
-----
r76 | martin.dilger@gmail.com | 2014-06-21 17:05:22 +0200 (Sa, 21
Jun 2014) | 1 line

committing file 68
-----
r75 | martin.dilger@gmail.com | 2014-06-21 17:05:16 +0200 (Sa, 21
Jun 2014) | 1 line
```

```
committing file 67
-----
r74 | martin.dilger@gmail.com | 2014-06-21 17:05:10 +0200 (Sa, 21
Jun 2014) | 1 line
[...]
```

» Ich hätte gerne, dass Du tatsächlich verstehst, was Git beim Klonen des Repositories macht, deswegen spielen wir jetzt verschiedene Szenarien durch. Im ersten Schritt klonen wir uns das Repository so wie es ist, ohne dass wir besondere Flags oder Einstellungen übergeben.

6.1. Eine erste Migration

Übung

Der Befehl zum Klonen eines Repositories lautet `git svn clone`. Versuchen Sie, den Befehl um das Repository unter <https://git-subversion-migration.googlecode.com/svn> zu klonen zu erschliessen.

```
git svn clone https://git-subversion-migration.googlecode.com/svn
#
Initialized empty Git repository in ../migrations/svn/.git/
W: +empty_dir: branches
W: +empty_dir: tags
W: +empty_dir: trunk
W: +empty_dir: wiki
r1 = 5b24dc003b91a109b54b0bd8f4b72484ac6b97a4 (refs/remotes/git-svn)
 A trunk/test-file.txt
r2 = 346ae0f3d5168455ad79a7dd93871759d7dd8b9b (refs/remotes/git-svn)
 A trunk/another-test-file.txt
r3 = c18222217842af253cbcf01632fb637a95679db2 (refs/remotes/git-svn)
 M trunk/test-file.txt
r4 = 06a9f2509994376b7b43bab66ea78f5ae894f941 (refs/remotes/git-svn)
 A branches/feature-1/branch-file.txt
 A branches/feature-1/another-test-file.txt
 A branches/feature-1/test-file.txt
```

» Karl, du siehst hier eigentlich schon ganz gut, was **Git-SVN** eigentlich macht. Da die Struktur eines Subversion-Repositories fundamental anders ist als die eines Git-

Repositories muss Git-SVN einiges an Arbeit erledigen. Git-Svn geht die SVN-Historie Commit für Commit durch und überführt jeden Commit einzeln nach Git.

» Jeden Commit einzeln? Üblicherweise haben Subversion-Repositories in Projekten tausende wenn nicht hunderttausende von Commits!

» *Ja, da hast du völlig Recht, deswegen macht es gerade für sehr große Projekte kaum Sinn, das komplette Repository zu klonen, da es schlichtweg zu lange dauert. Die aufwendigste Migration die wir hier bisher gemacht haben hat mehr als 4 Tage gedauert, allerdings nur weil das Repository so groß war. Leider bricht Git-SVN in den meisten Fällen nach einer bestimmten Zeit einfach ab, so dass die Migration erneut angestossen werden muss. Git-SVN ist zwar so klug, genau an der Stelle weiterzumachen wo es zuletzt aufgehört hatte, trotzdem sind große Migrationen nervenaufreibend. Unser Projekt ist glücklicherweise relativ klein, die Migration ist bereits abgeschlossen, siehst du?*

```
cd svn/
ls
branches +
tags +
trunk +
wiki +
```

» Das ist allerdings nicht was ich erwartet hätte. Die Struktur des Subversion Repositories spiegelt sich direkt in der Verzeichnisstruktur wider.

» *Das war genau, was ich Dir gerne zeigen wollte. Betrachte doch mal die letzten Commits in der Historie mit git log.*

```
git log -n 3
Commit: c4c4b8f6bf29619b508e04b76ef358266d62bc7c
Author: martin.dilger@gmail.com <martin.dilger@gmail.com@e7ff270f-
ff10-540e-3a92-2a19eadc0c21>
Date:   (29 minutes ago) 2014-06-21 15:05:22 +0000
Subject: committing file 68
```

```
git-svn-id: https://git-subversion-migration.googlecode.com/svn@76
e7ff270f-ff10-540e-3a92-2a19eadc0c21
```

```
Commit: 6fa61a2a746e5e9523970654ee434a5189232ff4
Author: martin.dilger@gmail.com <martin.dilger@gmail.com@e7ff270f-
ff10-540e-3a92-2a19eadc0c21>
```

```
Date:      (30 minutes ago) 2014-06-21 15:05:16 +0000
Subject: committing file 67
```

```
git-svn-id: https://git-subversion-migration.googlecode.com/svn@75
e7ff270f-ff10-540e-3a92-2a19eadc0c21
```

```
Commit: e27e81137830a32df78801e7d2fe85ced6cf002c
Author: martin.dilger@gmail.com <martin.dilger@gmail.com@e7ff270f-
ff10-540e-3a92-2a19eadc0c21>
Date:      (30 minutes ago) 2014-06-21 15:05:10 +0000
Subject: committing file 66
```

```
git-svn-id: https://git-subversion-migration.googlecode.com/svn@74
e7ff270f-ff10-540e-3a92-2a19eadc0c21
```

» *Du siehst hier gleich mehrere Seltsamkeiten. Beispielsweise der Autor.*

```
martin.dilger@gmail.com <martin.dilger@gmail.com@e7ff270f-
ff10-540e-3a92-2a19eadc0c21>
```

» *Git-SVN berechnet für jedes Subversion Repository eine eindeutige UUID. Da Git eine Kombination aus Benutzername und E-Mailadresse braucht, verwendet es die Repository-UUID einfach als Domäne. Nicht unbedingt das, was wir erwarten würden. Zumal in diesem speziellen Fall der Benutzername bereits eine E-Mailadresse ist. Genau hierfür brauchen wir das Usermapping, aber dazu kommen wir gleich. Die nächste Auffälligkeit ist folgende Meta-Information in der Commit-Message.*

```
git-svn-id: https://git-subversion-migration.googlecode.com/svn@74
e7ff270f-ff10-540e-3a92-2a19eadc0c21
```

» *Über diese Meta-Information schafft es Git, eine Brücke zwischen Git und Subversion zu schlagen, so dass eine eindeutige Zuordnung von Commit zu Subversion-Revision möglich ist. Du hast ja hoffentlich noch im Hinterkopf, dass Git Hashwerte als Revisions verwendet, die rein gar nichts mit den Subversion-Revisions gemein haben. Git braucht also eine Möglichkeit, eine eindeutige Zuordnung zu machen. Das ist allerdings nur für den Fall interessant, dass tatsächlich auch im Subversion-Repository weiter gearbeitet wird. Oder dass Git nur als verbessertes Frontend für Subversion verwendet wird. Wir machen aber eine komplette Migration, das bedeutet, sobald das Subversion-Repository einmal sauber nach **Git** migriert wurde, wird diese Meta-Information völlig wertlos und macht die Historie nur schwerer lesbar. Das werden wir uns auch gleich im Anschluss betrachten.*

- » Ich sehe schon, Git-SVN scheint ein wirklich mächtiges Tool zu sein.
- » *Die relativ einfache Migration von Subversion hin zu Git hat entscheidend zur schnellen Verbreitung des Systems beigetragen. Jetzt vergessen wir erstmal das eben geklonte Repository und fangen nochmal von vorne beim ersten Schritt an. Das erste Problem das wir haben ist, dass Git-SVN die komplette Repository-Struktur in das Git-Repository überführt hat, mit den Unterverzeichnissen für Branches und Tags. In der Git-Welt gibt es diese Zuordnung von Branches zu Verzeichnissen nicht. Wir brauchen eigentlich ein sauberes Mapping vom Subversion-Verzeichnis "/branches" auf einen Git-Branch, korrekt?*
- » Naja, das wäre schön, aber ist das so ohne weiteres möglich?

6.2. Der bessere Weg

- » *Glücklicherweise ja, denn wir haben eine wichtige Option beim Klonen des Repositories vergessen. Wir müssen Git-SVN explizit mit der Option --stdlayout mitteilen, dass unser Subversion-Repository dem Standard-Layout entspricht. Wir versuchen das Klonen nochmal und lösen das erste Problem in diesem Schritt.*

```
git svn clone --stdlayout https://git-subversion-migration.googlecode.com/
svn
#
WARNING: --prefix is not given, defaulting to empty prefix.
This is probably not what you want! In order to stay compatible
with regular remote-tracking refs, provide a prefix like
--prefix=origin/ (remember the trailing slash), which will cause
the SVN-tracking refs to be placed at refs/remotes/origin/*.
NOTE: In Git v2.0, the default prefix will change from empty to 'origin/'.
Initialized empty Git repository in /Users/martindilger/development/git/
workshops/migrations/svn/.git/
#
r1 = dd97f592342a1e869b40e48266b6bc839e30074c (refs/remotes/trunk)
 A test-file.txt
r2 = f41c9a451bc24bd90b53b30397636ab5d7b26e00 (refs/remotes/trunk)
 A another-test-file.txt
r3 = d9d4266375f9e4b640f1e190c7133b5eeaf94d49 (refs/remotes/trunk)
 M test-file.txt
[...]
```

- » *Git-SVN gibt uns hier schon einen wichtigen Hinweis, den wir gleich noch brauchen werden. Aber zunächst betrachten wir uns, wie das geklonte Repository aktuell aussieht.*

```
ls  
file1.txt +  
file16.txt +  
file22.txt +  
file29.txt +  
[...]
```

» Du siehst, das Repository sieht mittlerweile schon viel besser aus. Dadurch das wir für den Klon des Repositories die Option **--stdlayout** oder **-s** angegeben haben weiß Git-SVN, dass es die Strukturen des Repositories korrekt in Git-Banches überführen kann. Du siehst das auch, wenn Du dir die aktuellen Branches im Repository betrachtest.

```
git branch  
* master  
# remote branches  
git branch -r  
another-branch  
feature-1  
tags/release-v1  
trunk
```

» Ach, so einfach ist das? Das sieht jetzt doch genauso aus, wie ich das erwarten würde. Jeder Subversion-Branch wurde in einen korrekten Git-Branch überführt. Aber Moment, auch die Subversion-Tags wurden in eigene Git-Banches überführt. Das ist doch eigentlich nicht, was wir wollen, oder?

SVN / Git Tags

» Du hast völlig Recht, wir sind auch noch lange nicht fertig. Wir arbeiten uns Schritt für Schritt voran, damit Du wirklich verstehst, was die Migration bedeutet. Eine deiner zukünftigen Aufgaben wird nämlich auch darin bestehen, unsere letzten Subversion-Repositories nach Git zu überführen. Um die Subversion-Tag-Banches korrekt in Git-Tags zu überführen brauchen wir ein wenig Skript-Arbeit. Das ist allerdings genau das, was eines unserer Skripte (**svn-to-git-tags.sh**) für uns erledigt. Am besten führen wir das direkt aus, damit Du siehst, was passiert.

```
./../git-migrations svn-to-git-tags.sh  
ref=767d62bb3975fd6d878b77d6a48842f070838186  
parent=cc79ad67b3947d027b06166076939dd9823194d4 tagname=release-v1  
body=committing file 1
```

```
git-svn-id: https://git-subversion-migration.googlecode.com/svn/tags/
release-v1@9 e7ff270f-ff10-540e-3a92-2a19eadc0c21
Deleted remote branch tags/release-v1 (was 767d62b).
```

» Schau dir mal jetzt die Liste an Branches an.

```
git branch -r
another-branch
feature-1
trunk
```

» Interessant! Der Branch für die Tags ist verschwunden?

» Genau. Und noch viel besser wird es, wenn Du dir die Tags im Repository anschaugst.

```
git tag
release-v1
#
git log -n 1 release-v1
Commit: cc79ad67b3947d027b06166076939dd9823194d4
Author: martin.dilger@gmail.com <martin.dilger@gmail.com@e7ff270f-
ff10-540e-3a92-2a19eadc0c21>
Date:   (76 minutes ago) 2014-06-21 14:47:20 +0000
Subject: tagging release 1
```

» Ich bin beeindruckt, der Tag scheint korrekt angelegt worden zu sein. Kannst Du mir erklären, was das Skript genau macht?

» Das Skript sieht komplizierter aus, als es eigentlich ist.

Übung

Das Skript stammt von [hier¹⁹](#). Es ist nicht schwer zu lesen. Entziffern Sie das Skript und beschreiben Sie die Funktionalität.

```
git for-each-ref \
--format="% (refname:short) %(objectname)" refs/remotes/tags
| while read BRANCH REF
```

¹⁹ <http://stackoverflow.com/questions/2244252/importing-svn-branches-and-tags-on-git-svn>

```
do
  TAG_NAME=${BRANCH##*/}
  BODY=$(git log -1 --format=format:%B $REF)

  echo "ref=$REF parent=$(git rev-parse $REF^) tagname=
$TAG_NAME body=$BODY" >&2

  git tag -a -m "$BODY" $TAG_NAME $REF^ && \
  git branch -r -d $BRANCH
done
```

» Zunächst iterieren wir mit Hilfe von **for-each-ref** über alle Branches und lassen uns den Hashwert des jeweils obersten Commits im Branch ausgeben. Ich zeige dir das einfach mal im aktuellen Repository.

```
git for-each-ref --format="% (refname:short) %(objectname)"
master f52c3d082c2812937abd68050790f09549bf61fd
another-branch f44ade7c7dabea8cfa9ff31d4f392d7c08062ee0
feature-1 9ee327c3005ae0bb39583da170417509ee5112be
tags/release-v1 767d62bb3975fd6d878b77d6a48842f070838186
trunk f52c3d082c2812937abd68050790f09549bf61fd
```

» Wir beschränken die Auswahl anschließend nur auf die Tags.

```
git for-each-ref --format="% (refname:short) %(objectname)" "refs/remotes/
tags"
#
tags/release-v1 767d62bb3975fd6d878b77d6a48842f070838186
```

» Damit bekommen wir den Hashwert, des Commits bzw. der Revision, die ursprünglich in Subversion getaggt wurde. Denk daran, jede Revision in Subversion entspricht einem Commit-Hashwert in Git. Im nächsten Schritt iterieren wir über diese Liste und lassen uns mit Hilfe von **while read BRANCH REF** den jeweils ausgewählten Branch und den ausgewählten Hashwert als lokale Variable speichern.

```
git for-each-ref --format="% (refname:short) %(objectname)" refs/remotes/
tags
| while read BRANCH REF
do
  echo "$BRANCH $REF"
```

done

```
#  
tags/release-v1 767d62bb3975fd6d878b77d6a48842f070838186
```

» Jetzt holen wir uns den Tagnamen, indem wir den Refnamen ("tags/release-v1") parsen und alles nach dem "/" nehmen. Wir bekommen also "release-v1".

```
TAG_NAME=${BRANCH##*/}
```

» Im nächsten Schritt lassen wir uns die eigentliche Commit-Message des Tag-Commits ausgeben.

```
BODY=$(git log -1 --format=format:%B $REF)"  
#  
committing file 1 git-svn-id: https://git-subversion-  
migration.googlecode.com/svn/tags/release-v1@9 e7ff270f-  
ff10-540e-3a92-2a19eadc0c21
```

» Jetzt haben wir alles, was wir brauchen. Wir erzeugen einen neuen Tag, setzen diesen auf den richtigen Commit mittels \$REF, übernehmen die Commit-Message aus \$BODY und den Tag-Namen aus \$TAG_NAME.

» Toll, Lars. Ich bin ehrlich beeindruckt. Das ist relativ simpel, wenn man sich einmal die Mühe macht, sich genauer einzulesen.

SVN / Git User Mapping

» Genau, es ist so wie fast immer in Git, wenn man sich die Mühe macht und versucht, zu verstehen wie es tatsächlich funktioniert wird alles plötzlich ganz einfach. Wir haben aber noch einige Aufgaben, die wir für eine erfolgreiche Migration erfüllen müssen. Zunächst sollten wir sicherstellen, dass wir die richtigen Commiter in der Historie haben. Hierfür verwenden wir das Skript **svn-user-map.sh** noch im Subversion-Repository. Im Prinzip lassen wir uns den kompletten SVN-Log ausgeben und parsen für jede Revision den Autor und werfen am Ende die Duplikate raus.



Das Skript muss für die meisten Fälle leicht angepasst werden. Für die aktuelle Migration beispielsweise ist der Benutzername bereits eine E-Mailadresse. Das Skript im Repository erwartet die Übergabe einer Domain, die in den Benutzer übernommen wird (z.B. myName@<domain>).

```
# svn-user-map.sh <url> <filename>
svn log $1 | sed -ne "s/^r[^|]*| \([^\ ]*\) |.*$/\1 = \1 <\10>/p" | sort -u
> $2
```

» Der Skriptaufruf sieht also so aus.

```
./svn-user-map.sh https://git-subversion-migration.googlecode.com/svn
usermap.map
```

Was hierbei entsteht ist eine Datei usermap.map mit folgendem Inhalt.

```
martin.d@gmail.com = martin.d@gmail.com <martin.d@gmail.com>
```

» Für jeden User, der jemals etwas in diesem Projekt committed hat entsteht eine Zeile in der Usermap. Der nächste Schritt besteht jetzt darin, das Repository zu klonen. Wir unterstützen Git-SVN, indem wir mit der Option **--authors-file** angeben, wo die Datei mit den generierten User-Mappings zu finden ist.

```
# <authors-file> <prefix> <svn url> <git repository directory name>
git svn clone --stdlayout --authors-file usermap.map --prefix=svn/
https://git-subversion-migration.googlecode.com/svn svn.git
#
Author: (no author) not defined in usermap.map file
```

» Hier haben wir einen interessanten Fall. Anscheinend wurde ein Commit im Repository ohne Usernamen gemacht. Die Migration bricht ab, weil kein korrektes User-Mapping für diesen User erstellt werden kann. Wir müssen die **usermap.map**-Datei manuell anpassen, damit die Migration durchgeführt werden kann.



Bevor Sie mit der Migration starten sollten Sie die Usermap nochmal ganz genau studieren, ob sie korrekt ist. Fehler können zwar auch noch später korrigiert werden aber nur sehr umständlich und invasiv.

» Wir fügen einfach diese Zeile (**no author**) = **no author <unknown>** in die Datei ein. Damit sollte für die Migration alles fertig sein.

```
# <authors-file> <prefix> <svn url> <git repository directory name>
git svn clone --stdlayout --authors-file usermap.map --prefix=svn/
https://git-subversion-migration.googlecode.com/svn svn.git
```

» Es scheint zu funktionieren, Lars, sehr schön. Wir haben hier jetzt aber noch zusätzlich das Flag **--prefix** verwendet. Kannst Du mir erklären, was es damit auf sich hat?

» *Natürlich, stell dir vor, wir migrieren ein sehr großes Subversion-Repository mit allen Branches, die darin enthalten sind. Das können unter Umständen hunderte sein, korrekt?*

» Ich habe schon solche Projekte gesehen, ja.

» *Genau, jetzt stell dir vor, wir arbeiten in diesem Repository weiter. Du hättest keine Chance zu sehen, welche Branches jetzt aus Subversion konvertiert wurden und welche Branches seit der Migration im Git-Repository neu angelegt wurde. Genau dafür ist die Option **--prefix** da. Jeder Branch der aus Subversion migriert wird erhält ein Prefix, so dass sie später einfach auseinanderzuhalten sind. Das siehst du auch ziemlich gut, wenn Du dir im konvertierten Repository die Branches anschaugst.*

```
git branch -r
  svn/another-branch
  svn/feature-1
  svn/tags/release-v1
  svn/trunk
```

» Siehst Du? Jeder konvertierte Branch hat das Prefix "svn/". » Zuletzt führen wir nochmals die Migration der Tags durch. Da alle Branches und Referenzen jetzt ein Prefix haben müssen wir das Skript leicht anpassen.

```
git for-each-ref --format="% (refname:short) %(objectname)" refs/remotes/
svn/tag ...
#
ref=de5f2219344b941d330f077c67c34ea54711ea45
parent=97c74c5ffe439f93c937c62bd0313fb55b95a410 tagname=tags/release-v1
body=committing file 1

git-svn-id: https://git-subversion-migration.googlecode.com/svn/tags/
release-v1@9 e7ff270f-ff10-540e-3a92-2a19eadc0c21
Deleted remote branch svn/tags/release-v1 (was de5f221).
```

» Karl, betrachte die Historie mit **git log**, was fällt dir auf?

```
git log -n 2
```

```
#  
Commit: c8d7cbdd124f2376b74248408358eea6a86f99f9  
Author: martin.dilger@gmail.com <martin.dilger@gmail.com>  
Date:   (69 minutes ago) 2014-06-22 09:09:39 +0000  
Subject: change by Martin  
  
git-svn-id: https://git-subversion-migration.googlecode.com/svn/trunk@77  
e7ff270f-ff10-540e-3a92-2a19eadc0c21  
  
Commit: e5a240072483638676d88d43d585b205c8ec432a  
Author: martin.dilger@gmail.com <martin.dilger@gmail.com>  
Date:   (19 hours ago) 2014-06-21 15:05:22 +0000  
Subject: committing file 68  
  
git-svn-id: https://git-subversion-migration.googlecode.com/svn/trunk@76  
e7ff270f-ff10-540e-3a92-2a19eadc0c21
```

» Das Usermapping scheint geklappt zu haben, das ist schonmal gut. Damit sind wir eigentlich fertig, oder?

6.3. SVN / Git Ignores

» *Noch nicht ganz, ein wichtiger Schritt fehlt noch, denn wir müssen noch die svn-ignores in das Git-Repository überführen. Wir möchten ja nicht plötzlich in Git irgendwelche kompilierten Sourcen einchecken.*

» Natürlich, du hast ja völlig recht! Git hat ja einen eigenen Mechanismus um Dateien und Änderungen zu ignorieren.

Übung

Erinnern Sie sich wie Ignores in Git funktionieren?

» *Git arbeitet mit einer .gitignore Datei in der einfach alles definiert wird, was von Git nicht getrackt werden soll. Wir müssen als letzten Schritt noch die SVN-Ignores in eine .gitignore-Datei überführen. Natürlich bietet Git-SVN hierfür bereits ein passendes Tool.*

```
git svn show-ignore >> .gitignore  
less .gitignore
```

```
# /  
/target
```

» Anscheinend wird im Projekt lediglich das Verzeichnis **/target** exkludiert. Git-SVN hat diese Änderung aber korrekt in die .gitignore Datei überführt.

» *Genau, damit ist die Migration abgeschlossen und wir können ein weiteres Subversion-Repositotry löschen.*

Übung

Es ist heutzutage (leider?) bereits schwierig Projekte zu finden, die überhaupt noch auf Subversion gehostet sind. Suchen Sie sich einige Test-Repositories von Open-Source Projekten und migrieren Sie diese nach Git. Dies ist eine sehr gute Übung, bevor Sie sich an die Migration Ihrer eigenen Projekte machen, um ein Gefühl dafür zu bekommen, welche Schritte in welcher Reihenfolge am meisten Sinn ergeben und wie lange eine Migration für etwas größere Repositories dauern kann.

Beispielprojekte die aktuell noch auf Subversion gehostet sind sind beispielsweise:

<https://svn.apache.org/repos/asf/oltu/trunk/>
<http://primefaces.googlecode.com/svn/primefaces/>
<http://jmockit.googlecode.com/svn/>

6.4. Tag 3 endet

» Danke Lars, das war sehr beeindruckend. Die Migration von Subversion zu Git ist tatsächlich sehr einfach und ohne Probleme vollständig möglich. Ich finde es wirklich beeindruckend, dass sogar Branches problemlos nach Git überführt werden können obwohl sich die Systeme technisch so sehr unterscheiden.

» *Ja, nicht wahr? Subversion ist mit Sicherheit das System das bis vor ein paar Jahren noch am meisten Verbreitung in der IT-Welt hatte. Ich würde mal behaupten in den letzten 4 Jahren hat sich dieses Bild extrem gewandelt und heutzutage dürfte kaum mehr ein neues Projekt mit Subversion zur Versionskontrolle gestartet werden. Die meisten Projekte, die heute noch auf Subversion basieren würden wahrscheinlich gerne migrieren und haben noch nicht die Zeit gefunden. Subversion funktioniert*

teilweise ganz gut, aber ich denke Du stimmst mit mir überein, dass Git wirklich einige sehr große Vorteile bietet, die man nutzen sollte. Und da die Migration relativ problemlos und auch in einem überschaubaren Zeitrahmen machbar ist, gehe ich davon aus, dass früher oder später fast alle Projekte auf Git basieren werden. Es gibt zwar noch andere Systeme, wie beispielsweise Mercurial²⁰, allerdings haben diese System bei weitem keine so große Verbreitung wie Git. Bei uns im Team ist noch relativ viel Git Know-How vorhanden, es gibt jedoch Teams, die sich hierfür externe Unterstützung einkaufen müssen. Es gibt einige wirklich fähige Leute, die genau wissen, wie Git funktioniert. Bei anderen Systemen tut man sich schwerer, geeignete Consultants zu finden, die unterstützen können.

» Was ist denn für morgen geplant? Gibt es bereits konkrete Tasks, die wir zusammen angehen können?

» *Tatsächlich haben wir für morgen eine hochinteressante Aufgabe. Wir haben einige Projekte, die voneinander abhängig sind. Ein Projekt ist unser unternehmensweites Utility-Projekt. Hier finden sich dutzende Utilities, die eigentlich in jedem Projekt gebraucht werden. Aktuell werden die Abhängigkeiten allein über unser Build-System gesteuert. Wir wollen evaluieren, ob wir nicht auf das Versionskontrollsystem hierfür einsetzen können.*

» Das klingt wirklich interessant! Bietet Git hierfür Support?

» *Es gibt mehrere Möglichkeiten, wir werden uns morgen die aus meiner Sicht praktikabelste genauer betrachten. Die Lösung nennt sich Submodul. Über Submodule lassen sich mehrere Repositories miteinander verbinden.*

» Toll, ich freue mich schon, das genauer kennenzulernen.

Den Rest des Tages verbringt Karl damit, weitere Projekte von Subversion nach Git zu migrieren. Mit der Einführung von Lars ist dies ohne Probleme möglich. Karl fährt abends sehr zufrieden zurück nach München, denn wieder hat er eine Menge gelernt und freut sich, dieses Wissen auch zukünftig in seinen Projekten einzusetzen.

Aus dem Projektalltag

In einem Projekt, in dem ich als Entwickler gearbeitet habe war leider von der Organisation ganz klar Subversion als das präferierte System vorgegeben. Im

²⁰ <http://mercurial.selenic.com/>

Team waren wir uns allerdings einig, dass wir mit Git arbeiten wollten. Die meisten Entwickler hatten schon sehr viel Erfahrung mit Git und wir wollten die Vorteile des Systems nicht missen.

Eine einfache Möglichkeit wäre gewesen, die Git-Subversion-Bridge zu verwenden und mit Git zu arbeiten. Wir wollten aber mehr. Wir wollten mit Branches arbeiten, wir wollten mit Workflows arbeiten, **wir wollten alles**.

Wir haben uns also einen Rechner organisiert und einen eigenen **Team-Server** eingerichtet. Natürlich alles unter der Hand, und genau deswegen hat es so viel Spaß gemacht.

Der **master** des Team-Servers wurde einmal täglich mit Hilfe der Git-Subversion-Bridge mit dem **Trunk** des Subversion-Repositories synchronisiert. Für die tägliche Arbeit hatte das Team den vollen Funktionsumfang von Git zur Verfügung. Besonders die Möglichkeit mit Feature-Banches zu arbeiten und Code-Reviews für Branches durchzuführen hat dem Team sehr viel Geschwindigkeit gebracht. Mittlerweile ist dieses Unternehmen komplett auf Git migriert und es gibt Gott-sei-Dank keine Notwendigkeit mehr für Schatten-Git-Server, die irgendwo unter Tischen versteckt sind.

6.5. Links

[Migrating to Git - git-scm.com²¹](http://git-scm.com)

[Subversion to Git Tags²²](http://stackoverflow.com/questions/2244252/importing-svn-branches-and-tags-on-git-svn)

7. Tag 4 - Verteile Repositories

Als Karl am nächsten Tag in Nürnberg am Hauptbahnhof aussteigt, schwirrt ihm bereits ein wenig der Kopf. Es waren fast schon zu viele Informationen, die in den letzten drei Tagen auf ihn eingeprasselt sind. Aber Karl erinnert sich noch an einen Satz von Lars, den er direkt am ersten Tag erwähnt hatte. **Git ist nicht immer einfach, aber immer logisch**. Es hilft tatsächlich ungemein, sich mit den Details des Systems auseinanderzusetzen und zu verstehen, wie Git intern funktioniert. Es gibt viel zu lernen und es dauert üblicherweise eine ganze Zeit, bis sich ein Entwickler an diese neue Art zu arbeiten gewöhnt hat. Aber keine Frage - durch die intensive Schulung und die

²¹ <http://git-scm.com/book/en/Git-and-Other-Systems-Migrating-to-Git>

²² <http://stackoverflow.com/questions/2244252/importing-svn-branches-and-tags-on-git-svn>

zahlreichen Erklärungen hat Karl bereits den schwierigsten Teil des Weges geschafft. Ein Grundverständnis für Git ist bereits da, genauso wie die Lust und der Drang noch mehr zu lernen und zu verstehen.

» *Guten Morgen Karl. Wie war die Anreise, du siehst müde aus!*

» Hallo Lars, guten morgen. Ja, ich habe tatsächlich relativ wenig geschlafen, weil ich gestern abend nochmal meine Notizen der letzten Tage durchgegangen bin. Es war so unglaublich viel Information!

» *Ja, das tut mir sehr leid, Git scheint teilweise tatsächlich ein wenig kompliziert auf den ersten Blick, und manchmal leider auch auf den Zweiten. Aber soweit ich sehe machst du wirklich unglaubliche Fortschritte. Wenn ich bedenke, dass Du bis vor wenigen Tagen noch nie mit Git gearbeitet hast, bin ich ehrlich beeindruckt. In den meisten Teams würdest Du mittlerweile als der Git-Experte gelten.*

» Wollen wirs nicht übertreiben. Ich hoffe, es gibt heute die Möglichkeit, etwas zu lernen?

» *Darauf kannst du wetten. Bisher haben wir uns ja unter anderem mit den Grundlagen von Git beschäftigt, mit dem Git-Flow Branching Modell und gestern sogar schon mit der Migration einiger einfacher Projekte von Subversion nach Git. Heute betrachten wir eines unserer etwas komplexeren Projekte, die leider auch migriert werden müssen. Hierbei geht es aber nicht so sehr um die technische Migration, denn das kannst und kennst du bereits. Nein, für dieses Projekt haben wir es gleich mit mehreren Repositories zu tun.*

» Mehrere Repositories? Du meinst, die Projekt-Sourcen sind auf verschiedene Repositories verteilt?

» *Noch nicht. Wir haben ein sehr großes Projekt, das wir bisher immer in einem Repository entwickelt haben. Mittlerweile ist es aber so, dass Teile aus diesem Projekt auch in anderen Repositories Verwendung finden könnten und es teilweise auch schon tun. Wir haben für einige Komponenten bereits den falschen Weg eingeschlagen und diese einfach in andere Projekte kopiert. Dass das nicht der ideale Weg ist ist dir wahrscheinlich klar.*

» Code zu kopieren ist nie eine Lösung. Für jede Kopie verdoppelt sich der Wartungsaufwand. Muss an der Komponente etwas geändert werden, muss das für alle Kopien nachgezogen werden. Und leider passiert es schnell, oder besser immer, dass irgendwo etwas vergessen wird.

» *Ganz genau. Das Projekt, über das wir heute sprechen besteht rein logisch aus den verschiedenen Teilprojekten **Middleware**, **Web** und **Tools**. Diese Teilprojekte sollen jetzt in unterschiedlichen Repositories abgelegt werden.*

» Interessant, wir haben sowas eigentlich immer über das Build-System gelöst. Also ein Projekt wie Middleware hätte beim Build eine jeweils neue Version bekommen, beispielsweise **1.2.33**, die dann von der Web-Komponente referenziert werden kann.

» *Ja, gutes Beispiel Karl. Abhängigkeiten zwischen Repositories müssen nicht zwangsläufig über die Versionsverwaltung gelöst werden. Das funktioniert natürlich auch über das Buildsystem, beispielsweise **Maven**²³. Wir haben auch lange überlegt, welchen Weg wir einschlagen möchten. Ich bin der Meinung, dass die Lösung über das Versionskontrollsystem die bessere ist, da das genau die Aufgabe unserer Versionsverwaltung ist. Warum sollen wir uns die Mühe machen, Versionen doppelt zu pflegen? Wenn wir über das Build-System Artefakte releasen, müssen die neuen Versionen trotzdem nochmal in der Versionsverwaltung versioniert werden. Wir versuchen uns diese doppelte Arbeit zu sparen und verwenden gleich ausschließlich Git zur Versionsverwaltung.*

» Welche Möglichkeiten bietet uns Git denn, mit verteilten Repositories zu arbeiten?

» *Es gibt mehrere Ansätze, die teilweise sehr kontrovers diskutiert werden. Fangen wir aber am besten vorne an. Zunächst möchten wir ein bestehendes Repository in mehrere wiederverwendbare Komponenten oder Teilprojekte aufteilen. Dazu klonen wir uns das Projekt, damit wir direkt anfangen können damit zu arbeiten.*

7.1. Repository Splitting

Übung

Das Projekt ist auf GitHub gehostet.

Klonen Sie sich das Projekt, um lokal damit arbeiten zu können.

<https://github.com/dilgerma/Daily-Git-Distributed-Repositories-Example.git>

```
git clone https://github.com/dilgerma/Daily-Git-Distributed-Repositories-Example.git Daily-Git-Sample.git
```

²³ <http://maven.apache.org>

```
Cloning into 'Daily-Git-Sample.git'...
remote: Counting objects: 51, done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 51 (delta 12), reused 51 (delta 12)
Unpacking objects: 100% (51/51), done.
Checking connectivity... done.
```

» *Karl, wenn Du dir das Projekt genauer anschaust, dann siehst du schon die logische Unterteilung in der Ordnerstruktur im Projekt.*

```
README.md
services
tools
web
```

» Ich verstehe, **Services**, **Tools** und **Web**?

» *Genau, bisher sind wir ganz gut damit gefahren alles in einem Repository zu belassen. Ich würde das auch jederzeit wieder so machen, zumindest für den Anfang. Alles in einem Repository zu halten ist auf jedenfall der einfachste Weg. Ein Aufteilen in verschiedene Repositories sollte nur dann passieren, wenn es einen zwingenden Grund gibt und nicht, weil es vielleicht in irgendeinem Buch so steht.*

» Aber was wäre denn ein zwingender Grund? Was zwingt uns genau jetzt, das Repository aufzuteilen?

» *Wir haben sogar mehrere Gründe. Den ersten hab ich dir schon genannt, es gibt Teile in der Applikation, speziell die **tools**, die wir auch in anderen Projekten wiederverwenden möchten.*

» Also würde es Sinn machen, nur die Tools herauszuschneiden?

» Wäre das der einzige Grund, ja, definitiv.

» Gibt es denn noch andere?

» *Mittlerweile arbeiten sehr viele Entwickler an diesem Projekt. Es landen sehr viele Commits dort. Es ist schwer, allein durch einen Blick in die Historie einen Überblick zu bekommen was in letzter Zeit im Repository passiert ist. Das allein ist aber noch kein Grund für eine Aufteilung. Auf lange Sicht soll aus den **services** eine eigenständige Applikation entstehen, die als Plattform für weitere Anwendungen dient. Es werden*

also neue Services hinzukommen, die mit der jetzigen Anwendung nichts mehr zu tun haben. Sobald absehbar ist, dass ein Teil eines Repositories wie eine eigene Anwendung behandelt werden kann ist meistens der Zeitpunkt zumindest darüber nachzudenken, die Repositories aufzuteilen.

» Ich denke, ich verstehe und ich glaube, eine Aufteilung macht Sinn. Wie gehen wir jetzt aber vor?

» *Das Ziel ist ganz einfach. Zuerst zerschneiden wir das Repository in drei Teile, eben genau **web**, **services** und **tools**. Alles unter **web** bleibt in diesem Repository, denn das ist alles was von der Anwendung übrig bleibt. Anschließend verbinden wir die Repositories wieder über einen der beiden Mechanismen die uns Git bietet. Die **Services** und auch die **Tools** können von jetzt an unabhängig von unserer Anwendung entwickelt werden. Nur wenn ein Update notwendig ist aktualisieren wir die Version.*

» Bleibt die Historie der Teilprojekte dabei komplett erhalten?

» *Das sollte so sein, denn es wäre mehr schade, wenn das alles verlieren würden. Ich würde sagen, bevor wir lange reden, lass uns einfach anfangen. Zuerst schneiden wir die **tools** aus dem Repository. Das Git-Kommando hierfür nennt sich **Subtree**. Vor einigen Versionen war es übrigens noch relativ kompliziert, Teile eines Repositories herauszutrennen und dabei die Historie komplett zu erhalten. Mittlerweile ist es sehr einfach geworden.*

```
git subtree split --prefix=tools -b repo/tools
-n 1/    14 (0)
-n 2/    14 (1)
-n 3/    14 (2)
-n 4/    14 (3)
-n 5/    14 (4)
-n 6/    14 (5)
-n 7/    14 (6)
-n 8/    14 (7)
-n 9/    14 (8)
-n 10/   14 (9)
-n 11/   14 (10)
-n 12/   14 (11)
-n 13/   14 (12)
-n 14/   14 (13)
Created branch 'repo/tools'
```

» Interessant, was ist jetzt passiert?

» Es ist ganz einfach. Mit **git subtree split** geben wir Git einen Hinweis, dass wir eigentlich nur an einem bestimmten Unterverzeichnis im Repository interessiert sind. Dieses Verzeichnis geben wir mit der Option **--prefix** an.

» Ach so, **--prefix=tools** sagt Git, das wir alles aus dem *tools*-Unterverzeichnis haben möchten?

» Exakt. Mit der Option **-b repo/tools** sagen wir Git, dass alles was sich aktuell im Unterverzeichnis **tools** befindet auf einem eigenen Branch untergebracht werden soll. Der ursprüngliche Branch ändert sich hierdurch nicht. Git durchsucht also die Historie nach jedem Commit, der eine der Dateien im **tools** Verzeichnis beinhaltet und nimmt die relevanten Teile dieses Commits in den neuen Branch mit auf. Git geht hierbei sehr schlau vor und erstellt neue Commits, die nur die Klassen und Libraries aus dem *tools*-Verzeichnis beinhalten.

» Was ist mit Commits, die beispielsweise sowohl **tools** als auch **services** betreffen?

» Eine ganz hervorragende Frage! Schauen wir uns den **master**-Branch doch mal genauer an.

```
git log a596000 --stat --oneline -n 1
a596000 Service and Library
  services/service6 | 1 +
  tools/Library4    | 2 ++
2 files changed, 2 insertions(+), 1 deletion(-)
```

» Mit **--stat** und **--online** bekommen wir einen sehr übersichtlichen Ausgabe eines Commits, und der Commit mit der ID **a596000** betrifft sowohl **services** als auch die **tools**. Werfen wir also einen Blick auf unseren neuen Branch.

```
git checkout repo/tools
Switched to branch 'repo/tools'
ls

Library1
Library2
Library3
Library4
```

» Toll, auf dem Branch sind ja tatsächlich nur die Library-Sourcen! Und auch der Commit mit der Datei **Library4** wurde korrekt gesplittet. Es ist nur die Library-Datei vorhanden, nicht jedoch die service6-Datei aus dem gleichen Commit.

» Damit haben wir die Tools bereits in einem eigenen Branch. Im nächsten Schritt erzeugen wir uns ein neues lokales Repository, in das wir diesen Branch pushen. Dieses Repository können wir dann anschließend veröffentlichen und von da ab können die Tools schon unabhängig vom alten Repository entwickelt werden.

Übung

Erzeugen Sie ein neues Bare-Repository und fügen Sie dieses als Remote **tools-origin** in das aktuelle Repository hinzu.

```
cd..
mkdir tools-remote.git
cd tools-remote.git
git init --bare
Initialized empty Git repository

#new remote
cd Daily-Git-Sample.git/
git remote add tools-remote ../tools-remote.git
git checkout repo/tools

# push branch
git push tools-remote master
Counting objects: 55, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (55/55), 4.13 KiB | 0 bytes/s, done.
Total 55 (delta 14), reused 0 (delta 0)
To ../tools-remote.git
 * [new branch] master -> master
```

» Damit haben wir alle, was zuvor im Tools-Verzeichnis war in einem eigenständigen Repository untergebracht.

```
cd tools-remote.git/
git log --oneline
e0aade8 Service and Library
b9c1e53 Library 4
0582150 Library 3
```

```
abb40a0 Library2  
f6205a5 Library1
```

» Jetzt löschen wir den Branch und das Verzeichnis aus dem ursprünglichen Repository, denn beiden brauchen wir nicht mehr.

```
cd Daily-Git-Sample.git/  
git checkout master  
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.  
  
#delete branch  
git branch -D repo/tools  
Deleted branch repo/tools (was e0aade8).  
  
# directory  
git rm -r tools/  
rm 'tools/Library1'  
rm 'tools/Library2'  
rm 'tools/Library3'  
rm 'tools/Library4'  
  
git commit -m "removed tools, as moved to other repository"  
[master c11e74c] removed tools, as moved to other repository  
 4 files changed, 4 deletions(-)  
 delete mode 100644 tools/Library1  
 delete mode 100644 tools/Library2  
 delete mode 100644 tools/Library3  
 delete mode 100644 tools/Library4  
  
ls  
README.md  
services  
web
```

» Wir haben die Tools jetzt extrahiert. Die Anwendung ist aber so nicht mehr lauffähig. Wir müssen das neu erzeugte Remote-Repository wieder hinzufügen, damit die Anwendung weiterhin Zugriff auf die Libraries hat.

» Geht das denn so einfach? Ich meine, das klingt relativ komplex.

» Git bietet uns zwei Wege, getrennte Repositories miteinander zu verknüpfen. Wir werden uns beide ein wenig genauer betrachten um zu entscheiden, welcher Weg für uns der Richtige ist.

7.2. Distributed Repositories

Submodules

» Der erste und auch der bedeutend ältere Weg ist die Arbeit mit Submodulen. Über ein Submodule kann ein Repository als Unterverzeichnis in einem anderen Repository abgelegt werden.

» Das klingt ein bisschen wie die inverse Operation zu **subtree split**, was wir gerade gemacht haben.

» In gewissem Sinne ja, am besten wir versuchen es einfach mit Submodules und prüfen, ob wir damit zurecht kommen, oder?

```
git submodule add <PATH>/tools-remote.git/ tools ❶
Cloning into 'tools'...
done.
```

❶ Achtung - Submodule lassen sich nur absolut klonen, nicht mit einem relativen Pfad.

» Das war schon? Ziemlich unspektakulär...

» Moment, du weißt ja gar nicht, was jetzt passiert ist. Lass uns mal einen genaueren Blick in das Repository werfen.

```
ls
README.md
services
tools
web
```

» Wie du siehst ist das tools-Verzeichnis wieder da.

```
cd tools/
ls
Library1
Library2
Library3
Library4
```

» Auch alle Dateien sind im Tools-Ordner.

```
git log --oneline
e0aade8 Service and Library
b9c1e53 Library 4
0582150 Library 3
abb40a0 Library2
f6205a5 Library1
```

» Interessant, das tools-Verzeichnis verhält sich genau wie das Tools-Repository das wir zuvor erstellt haben. Ich kann mir sogar die Historie anschauen.

» *Genau, das liegt daran, dass das tools-Verzeichnis tatsächlich eine Referenz auf das tools-Repository ist. Werfen wir einen Blick in das Applikations-Repository, um zu sehen was sich hier verändert hat.*

```
git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   tools
```

» Wir haben zwei neue Dateien? Einmal das tools-Verzeichnis und einmal eine Datei .gitmodules? Ich dachte das tools-Verzeichnis wäre eine Repository und nicht einfach ein Verzeichnis?

» *Das ist korrekt. Die Datei .gitmodules gibt Git einen Hinweis darauf, dass im Repository **Submodule** im Einsatz sind. Öffne die Datei am besten einfach.*

```
less .gitmodules
#.gitmodules

[submodule "tools"]
    path = tools
    url = /Users/martindilger/development/git/playground/tools-
remote.git/
```

» *In der Datei hält Git die Information vor, welches Verzeichnis ein Submodule ist und wo das entsprechende Repository zu finden ist.*

» Ach jetzt verstehe ich, Git behandelt das Verzeichnis als Repository und im Dateisystem ist die jeweils ausgecheckte Working-Copy?

» So kann man es betrachten. Die Datei `.gitmodules` ist aber nicht die einzige Informationsquelle für das Submodul. Am besten du öffnest die config-Datei des Repositories, damit wir uns die zweite Stelle anschauen können, die für Submodule relevant ist.

```
less .git/config

...
[submodule "tools"]
    url = /Users/martindilger/development/git/playground/tools-
remote.git/
```

» Das verstehe ich nicht. Das ist doch genau die gleiche Information wie in der `.gitmodules`?

» Richtig, die Information ist redundant. Es ist aber wichtig zu wissen, dass beide Stellen für Git trotzdem relevant sind. Über die Sektion "submodule" erkennt Git, dass Submodule im Repository vorhanden sind und weiß, dass es das Repository des Submoduls als **Remote** behandeln soll. Was aber ist jetzt genau ein Submodul? Bevor wir committen solltest Du das verstanden haben. Du siehst das ziemlich gut, wenn du ein Diff machst, um zu sehen, was wir gleich Committen werden.

```
git diff HEAD
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..e8b921f
--- /dev/null
+++ b/.gitmodules ❶
@@ -0,0 +1,3 @@
+[submodule "tools"]
+    path = tools
+    url = /Users/martindilger/development/git/playground/tools-
remote.git/
diff --git a/tools b/tools
new file mode 160000
index 0000000..e0aade8
--- /dev/null
+++ b/tools
@@ -0,0 +1 @@

```

+Subproject commit e0aade8a72cdbf218024f5175446d9231646ee63 ②

- ❶ Die Datei .gitmodules haben wir bereits gesehen
- ❷ Das eigentliche Submodule ist lediglich ein Hash auf einen Commit.
 - » Das ist seltsam, und ehrlich gesagt verstehe ich das nicht so ganz. Das Submodule ist nur ein Hashwert?
 - » *Gut erkannt. Das stimmt prinzipiell. Ein Submodule in Git steht nicht auf einem Branch, wie man es vielleicht erwarten würde, sondern auf einem Commit, genauer gesagt dem Hashwert eines Commits. Klingt erstmal kompliziert, ist aber eigentlich korrekt, wenn du genauer darüber nachdenkst.*
 - » Lass mich übelegen. Das Problem was wir mit Submodulen lösen ist doch, dass wir eine bestimmte Version eines anderen Repositories in unser Repository aufnehmen möchten. Wenn wir einen Branch referenzieren würden, dann hätten wir ein Problem. Sobald die Entwicklung im Submodule-Repository weitergeht wird auch der Branch sich verändern. Wir würden also eine neue Version des Submodules bekommen, immer wenn wir ein Update machen.
 - » *Sehr gut, du bist genau auf dem richtigen Weg. Es wäre natürlich bequemer den Branch zu referenzieren, aber wenn der Branch sich weiterentwickelt möchten wir trotzdem auf dem Stand des Branches bleiben, den wir ursprünglich als Submodule eingefügt haben. Wir möchten keinesfalls automatisch Änderungen untergejubelt bekommen. Es ist also genau richtig, dass Git den Commit referenziert. Committen wir die Änderung, damit haben wir die Repositories verbunden.*

```
git commit -m "adding submodule tools"
[master eb5f6c9] adding submodule tools
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 tools
```

- » Und damit sind wir fertig?
- » *Erstmal ja, die Repositories sind miteinander verknüpft. Es ergeben sich aber einige, naja, Unbequemlichkeiten bei der Arbeit mit Submodulen.*
- » Das wäre ja auch zu schön gewesen.
- » *Genau. Dadurch dass Submodule grundsätzlich auf Commits und nicht auf Branches stehen ist es ein wenig kompliziert, auf Ihnen zu arbeiten. Am besten wir spielen das Szenario einfach durch. Das übliche Vorgehen ist, dass im Tools-Repository gearbeitet*

wird und unser Team neue Features bereitstellt, von denen alle profitieren. Am besten wir gehen in das Tools-Repository und machen das direkt.

Übung

Klonen Sie sich das Tools-Remote Repository lokal um darin arbeiten zu können. Das Repository ist aktuell ein Bare-Repository, wir brauchen aber eine lokale Working-Copy.

```
git clone tools-remote.git/ tools-repo.git
Cloning into 'tools-repo.git'...
done.
```

```
cd tools-repo.git
```

```
#new library
echo "Library5" >> Library5
git add Library5
git commit -m "new Library"
[master 31d42f9] new Library
 1 file changed, 1 insertion(+)
 create mode 100644 Library5

# push the new library
git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 269 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
```

» Wir haben also eine neue Tools-Library-Klasse eingefügt. Diese möchten wir jetzt gerne in unserem Projekt verwenden. Also zurück ins Anwendungs-Repository.

```
cd Daily-Git-Sample.git/
ls tools/
Library1
Library2
Library3
Library4
```

» Die neue Library-Klasse Library5 ist noch nicht da. Wir müssen explizit ein Update anfordern. Zunächst müssen wir das tools-Submodule auf den aktuellsten Stand bringen.

```
cd tools
git pull --rebase origin master
From /Users/martindilger/development/git/playground/tools-remote
 * branch           master      -> FETCH_HEAD
First, rewinding head to replay your work on top of it...
Fast-forwarded master to 31d42f99ae27d97ec182d6c22aec8416a239c6d1.
```

» Jetzt hat sich im Repository etwas verändert.

```
cd..
git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

modified:   tools (new commits)
```

» Ach, interessant, Git gibt uns schon den Hinweis, dass es im Submodule neue Commits gibt?

» Genau. Das einzige was wir jetzt noch machen müssen ist, die neue Version zu Committen. Ab diesem Zeitpunkt ist der neue Commit für das Submodule fixiert und jeder, der das Repository auscheckt bekommt diese neue Version.

```
git add .
git commit -m "new Submodule - Library 5"
[master 8e82786] new Submodule - Library 5
 1 file changed, 1 insertion(+), 1 deletion(-)
```

» Vielleicht gleich ein Hinweis für dich, Karl. Wir sehen es nicht allzu gerne, wenn jemand beim Update eines Submoduls nur die Commit-Message "new Submodule" oder ähnliches verwendet. Bitte verwende immer eine erklärende Commit-Message,

beispielsweise welches neue Feature mit diesem Update aus dem Submodule bereit steht.

» Ja klar, danke für den Hinweis. Ich werde versuchen, mich daran zu halten. Ich finde aber nicht, dass die Arbeit mit Submodulen bisher sehr ungemütlich ist. Eigentlich ist doch alles relativ klar.

» *Das ist richtig, kompliziert wird es erst, wenn mehrere Repositories im Spiel sind. Wir klonen uns das Projekt-Repository, um den typischen Workflow durchspielen zu können, wenn mehrere Entwickler am Projekt arbeiten.*

```
git clone Daily-Git-Sample.git/ Another-Daily-Git-Sample.git
Cloning into 'Another-Daily-Git-Sample.git'...
done.
```

» *Schau bitte mal in das Repository.*

```
cd Another-Daily-Git-Sample.git/
cd tools
ls
```

» *Das Tools-Verzeichnis ist leer?*

» *Korrekt, Submodule werden bei einem Klon nicht automatisch befüllt. Das müssen wir explizit anfordern.*

```
git submodule update --init ①
Submodule 'tools' (/Users/martindilger/development/git/playground/tools-
remote.git/) registered for path '../tools'
Cloning into 'tools'...
done.

Submodule path '../tools': checked
out '31d42f99ae27d97ec182d6c22aec8416a239c6d1'

#
ls tools
Library1
Library2
Library3
Library4
Library5
```

- ① Ein Submodul wird mit **git submodule update --init** initialisiert.

» Ok, Git scheint das Submodul richtig initialisiert zu haben.

» *Eine wichtige Besonderheit haben wir jetzt. Schau dir doch bitte mal den Status des Submodul-Repositories an.*

```
cd tools  
git status  
HEAD detached at 31d42f9  
nothing to commit, working directory clean
```

» HEAD detached? Was bedeutet das denn jetzt?

» *Das ist es, was ich vorher meinte und was wahrscheinlich der schwierigste Teil bei der Arbeit mit Submodulen ist. Da Submodule auf Commit-Hashes stehen und nicht auf Branches checkt Git beim Initialisieren des Submodules natürlich auch den Hashwert und nicht einen Commit aus. Was passiert, wenn ein Hashwert direkt ausgecheckt wird?*

» Git weiß in diesem Fall wahrscheinlich nicht, ob und wenn ja welcher Branch diesem Commit entspricht.

» *Genau, Git denkt in diesem Fall nicht in Branches. Es könnte ja auch sein, dass mehrere Commits auf dem gleichen Branch stehen. Welchen sollte Git dir also auschecken. Git macht also das einzige richtige, es schickt dich auf den **Detached-HEAD**, was eigentlich nur bedeutet: Achtung, du stehst aktuell nicht auf einem Branch.*

» Ok, das ist zwar ungewöhnlich, aber ist das ein Problem?

» *Das wird genau dann zum großen Problem, wenn du direkt im Submodul Änderungen machen möchtest. Denn auf dem **Detached HEAD** kannst du nicht arbeiten. Commits die du dort machst landen im Nirgendwo, da sie nicht durch einen Branch referenziert werden.*

» Ich verstehe, das Problem ist also, wenn ich vergessen, vorher zurück auf beispielsweise den **master** zu gehen, mache ich meine Arbeit, denke alles ist fertig und übersehe dabei, dass meine Commits aktuell im Nirgendwo liegen?

» *Genau so ist es. Versuchen wir es doch einfach mal. Zunächst sollte immer überprüft werden, auf welchem Branch das Submodul aktuell steht. Idealerweise wird dort nicht*

mit verschiedenen Branches gearbeitet, denn wir wollen eigentlich nur stabile und offiziell verfügbare Versionen eines Submodules referenzieren. In den meisten Fällen steht ein Submodul also fest auf dem **master**.

```
cd tools  
git name-rev --name-only HEAD ❶  
master ❷
```

- ❶ Über **git name-rev --name-only <hashwert>** lässt sich der Branch-Name für einen bestimmten Hashwert ermitteln
 - ❷ Jetzt wissen wir, dass das Submodul tatsächlich auf dem **master** steht.
» Als nächstes erzeugen wir uns direkt im **Submodule** eine neue Library-Klasse. Vergiss nicht, das Submodul ist eine Referenz auf das Repository. Es spricht also nichts dagegen, direkt dort zu arbeiten, wenn es Sinn macht.
-

```
echo "Library 6" >> Library6  
git add Library6  
git commit -m "new Library 6"  
[detached HEAD 69b5929] new Library  
 1 file changed, 2 insertions(+)  
 create mode 100644 Library6  
  
#  
git status  
HEAD detached from 31d42f9  
nothing to commit, working directory clean
```

- » Interessant, Git hat den Commit akzeptiert, wir sind aber immer noch auf dem **Detached-HEAD**.
 - » Genau das ist die Schwierigkeit. Wir wechseln jetzt zurück auf den **master**.
-

```
git checkout master  
Warning: you are leaving 1 commit behind, not connected to  
any of your branches:  
  
 69b5929 new Library
```

If you want to keep them by creating a new branch, this may be a good time to **do** so with:

```
git branch new_branch_name 69b5929  
  
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.
```

» Git gibt uns wenigstens direkt eine Warnung aus, wenn wir Gefahr laufen einen Commit zu verlieren.

» *Das ist korrekt, wir haben den Commit bereits verloren. Das siehst du direkt, wenn du dir mit git log die Historie betrachtest. Der Commit, den wir gerade eben gemacht haben ist verschwunden.*

```
git log --oneline  
31d42f9 new Library  
e0aade8 Service and Library  
b9c1e53 Library 4  
0582150 Library 3  
abb40a0 Library2  
f6205a5 Library1
```

» Und jetzt? Müssen wir alles nochmal machen?

» *Nein, das wäre zu viel verlangt. Die Gefahr ist aber sehr groß, dass man die Warnung von Git übersieht und einfach weiterrchreibt, ohne zu merken, dass wir den Commit verloren haben. Wir holen uns den verlorenen Commit einfach mit einem Cherry-Pick auf den master zurück. Den Hashwert für den Commit erhalten wir aus der Warnung, die Git uns ausgegeben hat.*

```
git cherry-pick 69b5929  
[master 02e9f76] new Library 6  
 1 file changed, 2 insertions(+)  
 create mode 100644 Library6
```

» Ok, das ist beruhigend, also alles wieder gut?

» *Ja fast, nur noch ein kleiner Hinweis. Git erlaubt es zwar nicht, einen Detached HEAD zu pushen, sehr wohl aber eine Referenz auf einen Detached HEAD. Das ist deswegen so extrem gefährlich, weil bei dir lokal alles funktioniert, da der Commit im Repository ja da ist, nur nicht von einem Branch referenziert. Sobald aber einer deiner Team-Kollegen versucht, das Submodul auf einen aktuellen Stand zu bringen wird Git mit der Fehlermeldung abbrechen, dass es den Commit mit dem Hashwert X nicht*

finden kann. Das passiert leider sehr oft. Das gleiche Problem ergibt sich, wenn ein Entwickler im Submodule eine Änderung gemacht hat, alles committed und dann die Referenz im Submodul auf den neuen Commit aktualisiert. Schade ist dann nur, wenn der Entwickler vergisst, den Stand im Submodule zu pushen. Ergebnis ist wieder das gleiche, das Submodul steht auf einem Commit, der bei dir lokal zwar verfügbar ist aber leider bei keinem anderen Entwickler, weil ein **Push** vergessen wurde. Das meinte ich eben damit, dass es zeitweise ein wenig ungemütlich sein kann mit Submodulen zu arbeiten. Es gibt mehrere Schritte, die alle abgearbeitet werden müssen. Sobald einer vergessen wird funktioniert meistens lokal noch alles, aber leider bei keinem anderen Entwickler.

» Ok, können wir nochmal kurz durchgehen, was alles in welcher Reihenfolge gemacht werden muss um das Submodule korrekt auf einen neuen Stand zu bringen?

» Ja klar, bekommst du die Schritte noch zusammen?

Übung

Bekommen Sie die Schritte noch zusammen? Was muss gemacht werden um ein Submodul zu ändern und diese Änderung anderen Entwicklern zur Verfügung zu stellen? Wenn nein, überfliegen Sie die letzten Abschnitte am besten erneut.



Um ein Submodul auf einen aktuellen Stand zu bringen ist folgendes zu tun:

- git checkout <branch>, üblicherweise **master**
- change.. change..
- git commit
- git push
- Wechsel ins Root-Repository
- git add <submodule>
- git commit
- git push

» Das sind schon eine ganze Menge Schritte, ich kann gut verstehen das hierbei oft Fehler passieren. Du meintest, es gibt noch einen anderen Weg?

» Ja, und zwar mit einem Kommando was wir sogar heute schon verwendet haben - **git subtree**.

Subtree

- » Damit haben wir doch das Repository gesplittet, können wir es damit auch wieder vereinen?
- » *Genau, Subtree funktioniert in beide Richtungen, wenn auch mit einem komplett anderen Ansatz als Submodule.*
- » Das klingt spannend. Wie gehen wir vor?
- » *Wir haben bis jetzt das Tools Verzeichnis in ein eigenes Repository extrahiert und anschließend wieder über Submodule eingebunden. Was hälst du davon, wenn wir dasselbe mit dem services-Verzeichnis machen, nur dass wir es anschließend über Subtree einbinden. Dann haben wir auch gleich den direkten Vergleich?*
- » Klingt toll.

Übung

Gehen sie genau gleich vor wie beim Tools-Verzeichnis und extrahieren Sie das **services**-Verzeichnis in ein eigenes Repository **services-remote.git**

```
git subtree split --prefix services -b "repo/services"  
-n 1/      17 (0)  
-n 2/      17 (1)  
-n 3/      17 (2)  
-n 4/      17 (3)  
-n 5/      17 (4)  
-n 6/      17 (5)  
-n 7/      17 (6)  
-n 8/      17 (7)  
-n 9/      17 (8)  
-n 10/     17 (9)  
-n 11/     17 (10)  
-n 12/     17 (11)  
-n 13/     17 (12)  
-n 14/     17 (13)  
-n 15/     17 (14)  
-n 16/     17 (15)  
-n 17/     17 (16)  
Created branch 'repo/services'  
8eb2fc57575da7d93f01eb246bbbbf70fc4c01801
```

```
#  
cd ..  
mkdir services-remote.git  
git init --bare  
Initialized empty Git repository  
  
#  
cd Daily-Git-Sample.git  
git checkout repo/services  
git remote add services-origin ../services-remote.git/  
git push services-origin repo/services:master  
Counting objects: 57, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (44/44), done.  
Writing objects: 100% (56/56), 5.00 KiB | 0 bytes/s, done.  
Total 56 (delta 13), reused 0 (delta 0)  
To ../services-remote.git/  
 * [new branch] master -> master  
  
#  
git checkout master  
  
git branch -D repo/services  
Deleted branch repo/services (was 8eb2fc5).  
  
git rm -r services/  
git commit -m "removed services"  
[master b3fc360] removed services  
 6 files changed, 6 deletions(-)  
 delete mode 100644 services/service1  
 delete mode 100644 services/service2  
 delete mode 100644 services/service3  
 delete mode 100644 services/service4  
 delete mode 100644 services/service5  
 delete mode 100644 services/service6
```

» *Sehr gut Karl, ich bin beeindruckt, damit haben wir ein eigenes Repository für unsere Services und haben alle Services-Klassen sauber aus dem Anwendungs-Repository extrahiert. Gute Arbeit!*

» Danke Lars.

» *Als nächsten Schritt werden wir das Services-Repository als Subtree in das Anwendungs-Repsitory integrieren.*

```
git subtree add --prefix=services ../services-remote.git master --squash
git fetch ../services-remote.git master
From ../services-remote
 * branch master      -> FETCH_HEAD
Added dir 'services'
```

» Was ist jetzt passiert? Das ging mir ein wenig zu schnell.

» Klar, entschuldige. Das Subtree-Kommando ist immer gleich aufgebaut. Über ein Prefix legst du fest, welches Verzeichnis gerade bearbeitet werden soll. Über **--prefix=services** also beispielsweise das Verzeichnis **services**. Über **git subtree add** lässt sich jetzt die Historie eines kompletten Repositories als Snapshot in ein Unterverzeichnis übernehmen. Wir müssen Git lediglich sagen, welches Repository und welchen Branch aus diesem Repository wir übernehmen möchten. Zu guter Letzt stellen wir noch sicher, dass Git nicht die komplette Historie aus dem Subtree-Repository in unser Repository übernimmt sondern die komplette Historie in einen Commit zusammenpackt. Man nennt das Squashen. Würden wir den Commit nicht Squashen hätten wir plötzlich alle Commits so bei uns in der Historie, wie sie auch im services-Repository vorhanden sind. Das interessiert uns aber nicht, denn wir möchten nur einen bestimmten Snapshot haben. Klingt zunächst sehr komplex, ist aber eigentlich ganz simpel. Schau dir am besten mal das Repository an.

```
ls
README.md
services
tools
web
```

» Das services-Verzeichnis ist auf jedenfall wieder da?

» Genau, schau am besten einmal genauer hin, am besten in das services-Verzeichnis und anschließend in die Historie.

```
ls services/
service1
service2
service3
service4
service5
service6
```

» Sieht soweit eigentlich ganz korrekt aus, oder?

```
git log --oneline -n2
2edc28b Merge commit '6a1233a9ec26f0506d617fb667472f3e0e3454c9'
as 'services'
6a1233a Squashed 'services/' content from commit 8eb2fc5
```

» Interessant, durch das **Subtree**-Kommando sind zwei neue Commits entstanden?

» *Genau, ich habe dir ja schon gesagt, dass wir Git befohlen haben, das neue Services-Repository als einen Commit zu übernehmen. Ab jetzt ist dieser Stand eingefroren und fester Bestandteil des Repositories.*

» Aber das macht doch keinen Sinn? Wie sollen wir künftige Änderungen aus dem Services-Repository übernehmen?

» *Hier hilft uns wieder das Subtree-Kommando. Spielen wir das Szenario doch einmal durch und machen eine Änderung im Services-Repository.*

```
git clone services-remote.git/ services-repo.git
Cloning into 'services-repo.git'...
done.

#
cd services-repo.git

#new service
echo "service7" >> service7
git add service7
git commit -m "service7"
[master 5eb5e42] service7
 1 file changed, 1 insertion(+)
 create mode 100644 service7

#
git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 268 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
```

» *Damit haben wir eine Änderung im Services-Repo. Diese Änderung holen wir uns jetzt in das Anwendungs-Repository.*

```
git subtree pull --prefix=services ..../services-remote.git master --squash
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ..../services-remote
 * branch           master      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 services/service7 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 services/service7
```

» *Statt mit `git subtree add` arbeiten wir jetzt einfach mit `git subtree pull`. Ansonsten ist das Kommando exakt dasselbe. Wir geben also wieder an, welches Remote-Repository und welchen Branch wir gerne hätten.*

```
ls services/
service1
service2
service3
service4
service5
service6
service7
```

» Tatsächlich, der neue Service ist da. Das Subtree-Kommando ist zwar ein wenig komplizierter, dafür sind weniger Schritte notwendig wie bei Submodulen. Können wir jetzt auch direkt in dem Subtree-Verzeichnis Änderungen machen oder geht das nur im externen Repository?

» *Eine ausgezeichnete Frage. Probieren wir es aus.*

```
echo "service8" >> services/service8
git add services/service8
git commit -m "adding service 8"
[master 8d6bcf6] adding service 8
 1 file changed, 1 insertion(+)
 create mode 100644 services/service8
```

» *Um diese Änderung jetzt in das Services-Repository zu übernehmen gibt es das **Subtree Push**-Kommando.*

```
git subtree push --prefix=services ../services-remote.git master
git push using: ../services-remote.git master
[...]
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 279 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ../services-remote.git
  5eb5e42..e9e8fe0  e9e8fe00a65bfd04ad5d7b4a04f14d3d43a7a0c4 -> master
```

» *Achtung, hier ist es wichtig, dass nicht mit der Option **--squash** gearbeitet wird, wenn wir wollen die Commits ja nicht gepackt sondern so wie wir sie gemacht haben ins Repository übernehmen. Schauen wir mal, ob die Änderungen angekommen sind.*

```
git log --oneline -n1
e9e8fe0 adding service 8
```

» Tatsächlich, das hat funktioniert.

» *Das wars eigentlich schon. Damit haben wir beide Wege ausprobiert. Ich habe eine Präferenz, und du?*

» Das ist eine schwierige Frage, da ich die beiden Wege bisher nicht praktisch kenne. Rein intuitiv würde ich sagen, der Workflow mit Subtree ist einfacher, dafür ist das Kommando ein wenig komplizierter als bei Submodulen. Gibt es gravierende Nachteile bei einem der beiden Wege, die wir noch nicht besprochen haben?

» *Prinzipiell nicht, Submodule sind zwar etwas komplexer in der Handhabung, aber trotzdem relativ intuitiv, weil wir nicht die Historie zweier Repositories miteinander vermengen. Subtree ist toll, aber ich persönlich bin kein Fan davon. Für einen einfachen Fall, beispielsweise wir möchten lediglich ein Repository als Unterverzeichnis haben ohne dass wir planen dort regelmäßig Änderungen zu machen würde ich wahrscheinlich Subtree bevorzugen. In komplexeren Fällen, wenn beispielsweise viele Änderungen zu erwarten sind neige ich zu Submodulen, weil einfach eine klare Trennung zwischen den beteiligten Repositories besteht.*

Aus dem Projektalltag

Mit Submodulen zu arbeiten ist anfangs gewöhnungsbedürftig und durchaus gefährlich. Dies haben einige meiner Kollegen kürzlich in einem Projekt zu spüren bekommen.

In diesem Projekt gab es verschiedene Applikationen, die aber alle eine unternehmensweite gemeinsame Bibliothek verwenden. In dieser Bibliothek sind u.a. Anwendungsübergreifende Komponenten abgelegt. Diese *Shared-Library* wurde in jedem Projekt über ein *Submodule* eingebunden. So konnten wir auf aufwendige *Maven*-Releases verzichten und in der Git-Historie war jederzeit sichtbar, wenn ein Entwickler ein Update der *Shared*-Komponenten vorgenommen hat.

Wann immer eine Änderung an einer diesen Komponenten notwendig war, musste diese im *Shared-Repository* durchgeführt und anschließend zusätzlich noch die Referenz im jeweiligen Projekt auf den aktuellsten Commit im *Shared-Repository* gesetzt werden.

Grundsätzlich wurde auf *Branches* entwickelt. Da Qualität in diesem Projekt sehr wichtig war wurden zusätzlich alle Änderungen vor dem Zurückführen in den **master** einem Review unterzogen.

Ein Entwickler wollte eine Komponente um eine Methode erweitern, von der er wusste, dass sie auch in anderen Projekten nützlich wäre. Er erzeugt also einen Branch im *Shared-Repository*, macht seine Änderung und pusht den Branch.

```
cd shared
git checkout -b "feature/enhance-common-component"
// ...
// Notwendige Änderungen durchführen
// ...
git commit -m "Component enhanced"
git push origin feature/enhance-common-component
```

Anschließend macht er in seinem Projekt testweise ein Update auf die Version im Branch.

```
cd Project
cd shared
```

```
git fetch  
git checkout feature/enhance-common-component
```

Jetzt kann der Entwickler das Projekt bauen und testen, ob die Anwendung mit der angepassten Komponente noch funktioniert. Der Entwickler freut sich natürlich, dass die Änderungen kompatibel ist und verwendet die neue Methode direkt an einer Stelle im Projekt.

Anschließend stellt er zwei Merge-Requests an einen Reviewer. Ein Request für das *Shared-Repository* um den Branch **feature/enhance-common-component** auf den **master** des *Shared-Repositories* zu überführen. Ein zweiter Merge-Request für die neue *Shared*-Referenz und den neuen Code im Projekt. (Die meisten Entwickler haben selbst keine Rechte um auf den **master** zu pushen.)

Der Reviewer (zuständig sowohl für das Projekt, als auch für die *Shared-Bibliothek*) wiederum akzeptiert den Merge-Request für das Projekt, übersieht dabei aber den Merge-Request in der *Shared-Bibliothek*.

Nachdem die Änderungen auf dem Master gelandet sind baut der Jenkins (das eingesetzte CI-System) die Version und meldet nach kurzer Zeit einen erfolgreichen Build.

Ein weiterer Entwickler macht kurze Zeit später eine weitere Änderung im *Shared-Modul*. Er erzeugt einen neuen Branch vom **master**, macht seine Änderung und stellt beim Testen im Projekt fest, dass das Projekt mit dem aktuellen **master** aus dem *Shared-Modul* gar nicht baut.

Er wirft sofort einen Blick auf den Jenkins - alles grün?

Er baut nocheinmal, und der Build bricht wieder. Er startet zur Sicherheit den Jenkins-Job erneut, der nach kurzer Zeit einen erfolgreichen Build meldet.

Was war die Lösung des Rätsels?

Das Projekt auf dem Jenkins ist deshalb grün, da die Referenz im **master** derzeit direkt auf den obersten Commit im Branch **feature/enhance-common-component** zeigt. Der Branch ist nach wie vor verfügbar und **noch nicht** in den **master** überführt, da der Reviewer leider vergessen hat, den Merge-Request zu akzeptieren.

Der Entwickler, der die zweite Änderung machen wollte macht dies natürlich vom **master** weg. Um die neue Implementierung zu testen aktualisiert er die

Referenz auf die *Shared*-Bibliothek, die jetzt wieder auf dem **master** steht. Das Projekt verwendet aber bereits die neue Methode aus **feature/enhance-common-component**, so dass der Build aktuell nicht funktioniert. Würde der ***feature/enhance-common-component**-Branch jetzt gelöscht werden würde der Jenkins-Build fehlschlagen, mit der Fehlermeldung, dass die Referenz des Submodules nicht auffindbar wäre.



Submodule sind mächtig und hilfreich. Es sind aber leider immer mindestens zwei Schritte nötig (**Änderung im Submodule-Repository** und **Update der Referenz im Projekt-Repository**). Wird eines von beiden vergessen sind Probleme vorprogrammiert.

7.3. Tag 4 endet

Karl schwirrt der Kopf von den Möglichkeiten, die Git bietet. Von den Workflows die bereits besprochen wurden bis hin zur einfachen Trennung und Wiedervereinigung von Repositories mit Submodulen oder dem Subtree-Kommando. Karl macht heute etwas früher Schluss um das Gelernte zu verdauen - für einen Tag war das mehr als genug Information.

» Lars, ich muss heute ein wenig früher zurück nach München. Ich hoffe das ist in Ordnung. Ich bin wirklich beeindruckt von den Möglichkeiten, die wir mit Git haben. Jetzt bin ich schon fast eine Woche bei euch und es kommt mir wie eine Ewigkeit vor. So viel habe ich noch nie in einer einzigen Woche gelernt.

» *Einen Tag haben wir noch in dieser Woche, Karl. Ab nächster Woche müssen wir unbedingt anfangen produktiv zu arbeiten. Der Tag morgen ist nochmal für eine kurze Wiederholung und einige Tools reserviert. Dann bis morgen.*

8. Fazit

Karl ist mittlerweile ein wichtiger Bestandteil des Teams und sogar Ansprechpartner für Fragen rund um Git. Karl führt das auf das Coaching von Lars und sein grundsätzliches Verständnis über die Art und Weise wie Git funktioniert zurück.

Obwohl Karl jeden Tag von München nach Nürnberg und zurück pendelt lohnt sich der Weg für ihn. Karl hat mit seinem Team die Abmachung getroffen, dass er die Zeit im Zug abrechnen darf, wenn er sie produktiv nutzt. Karl kann mit Hilfe von Git problemlos unterwegs im Zug arbeiten. Hierfür nimmt er sich regelmäßig einige Tickets und Bugs mit auf den Weg.

Ich bin der Ansicht, dass Git für uns Softwareentwickler die wichtigste Neuerung der letzten 10 Jahre ist. Kein anderes Tool hat mich persönlich produktiver gemacht. Ich arbeite jetzt seit nunmehr fast 6 Jahren ausschließlich mit Git und unterstütze Firmen bei der Einführung, Migration von anderen Systemen und bei der Schulung von Mitarbeitern.

Ich hoffe, die Art und Weise wie dieses Buch geschrieben ist hat dazu beigetragen, dass Sie ein Verständnis dafür entwickeln konnten, wie Git arbeitet und wie die Arbeit mit Git gedacht ist. Nutzen Sie ihr jetzt erworbenes Wissen und geben Sie es weiter. Ich wünsche Ihnen viel Erfolg beim produktiven Arbeiten mit Git.

Übrigens komme ich auch gerne in Ihr Unternehmen und spreche mit Ihnen über das Thema Git. Ich halte kostenlose Tech-Talks über Git und biete natürlich Schulungen, Seminare und Consulting. Am besten Sie besuchen meine Homepage <http://www.effectivetrainings.de> und informieren sich, oder Sie schreiben mich einfach an: martin@effectivetrainings.de²⁴ Ich würde mich freuen von Ihnen zu hören.

Natürlich freue ich mich auch über Bewertungen zum Buch auf Amazon.

Im Buch finden Sie einige Geschichten aus dem Projektalltag. Vielleicht haben Sie auch eine Geschichte, die zum Thema passt. Ich würde mich freuen, wenn Sie mir dazu schreiben würden. Vielleicht landet Ihre Geschichte ja auch im Daily-Git-Buch.

Danke fürs Lesen!

Martin Dilger, München am 01. Dezember 2014

9. Appendix A

9.1. Branch Per Feature (BPF)

Das folgende Gespräch haben Lars und Karl in der Kaffeepause nach der Diskussion über Git-Flow geführt. Es gibt ein alternatives Branching-Modell zu Git-Flow, das sich BPF (Branch per Feature) nennt.

» *Wir machen uns derzeit Gedanken, ob es noch bessere Modelle als Git-Flow für unsere Arbeit gibt. Versteh mich nicht falsch, der Gedanke von Git-Flow ist super und bildet perfekt unsere jetzige Art zu arbeiten ab, aber ich habe kürzlich einen recht interessanten Artikel²⁵ gelesen, der noch einen Schritt weiter geht.*

²⁴ <mailto:martin@effectivetrainings.de>

²⁵ <http://www.effectivetrainings.de/blog/2013/12/26/enterprise-git-enterprise-workflows-branch-per-feature/>

» Was fehlt dir denn bei Git-Flow?

Unterschied zu Git-Flow

» Das Modell nennt sich BPF, was für "Branch per Feature" steht und wurde ursprünglich von Adam Dymitruk²⁶ beschrieben. Ich lese öfter Artikel von ihm, und dieses Modell klingt interessant. Soll ich es dir kurz erklären? Für unsere tägliche Arbeit spielt es derzeit keine Rolle, aber ich spreche immer gerne mit Teamkollegen über neue Themen und höre mir andere Meinungen an.

» Unbedingt, ich bin gespannt, was es an Git-Flow noch zu verbessern gibt.

» Die Grundidee von BPF ist schnell erklärt. Git-Flow zwingt uns praktisch, Features so schnell wie möglich zurückzuführen. Warum? Um Merge-Konflikte zu vermeiden. Auf dem master landen immer mehr Commits. Features die abgeschlossen aber noch nicht zurückgeführt sind entfernen sich deshalb immer weiter vom master-Stand. Die Wahrscheinlichkeit von Merge-Konflikten wächst also täglich. Git-Flow sieht es außerdem nicht vor, Features, die einmal zurückgeführt wurden wieder zu entfernen.

» Kommt das tatsächlich vor? Zurückgeführte Features wieder zu entfernen?

» Zugegeben, selten. Es treten natürlich gelegentlich Bugs auf, die gefixt werden müssen. Das ist aber normalerweise kein Grund, ein Feature komplett aus einer Version herauszunehmen. Manchmal stellen wir aber zu spät fest, dass ein Feature, das umgesetzt wurde nicht genau genug spezifiziert war. Oder der Markt ändert sich so rapide, so dass ein Feature plötzlich überhaupt keinen Sinn mehr macht. Oder unser Product-Owner möchte gerne ein Feature nur für eine bestimmte Zeit verwenden um zu sehen, wie es von den Kunden angenommen wird. Ist die Resonanz schlecht, soll das Feature wieder entfernt werden. Der für uns interessanteste Use-Case ist aber sicherlich, dass wir Features unserer QA-Abteilung zur Verfügung stellen möchten, die sich die Umsetzung anschaut und ausführlich testet. Fallen Features durch, erstellen wir einfach eine neue Version für QA und zwar komplett ohne das fehlerhafte Feature.

» Diese Möglichkeit ist in Git-Flow nicht vorgesehen. Wir haben nur die Möglichkeit, Features programmatisch oder über Konfigurationen, sogenannte Feature-Flags wieder zu entfernen. Im Prinzip bauen wir etwas in dieser Art ein.

```
if(newFeatureActive) {  
    #do something fancy with the feature
```

²⁶ <http://dymitruk.com/blog/2012/02/05/branch-per-feature/>

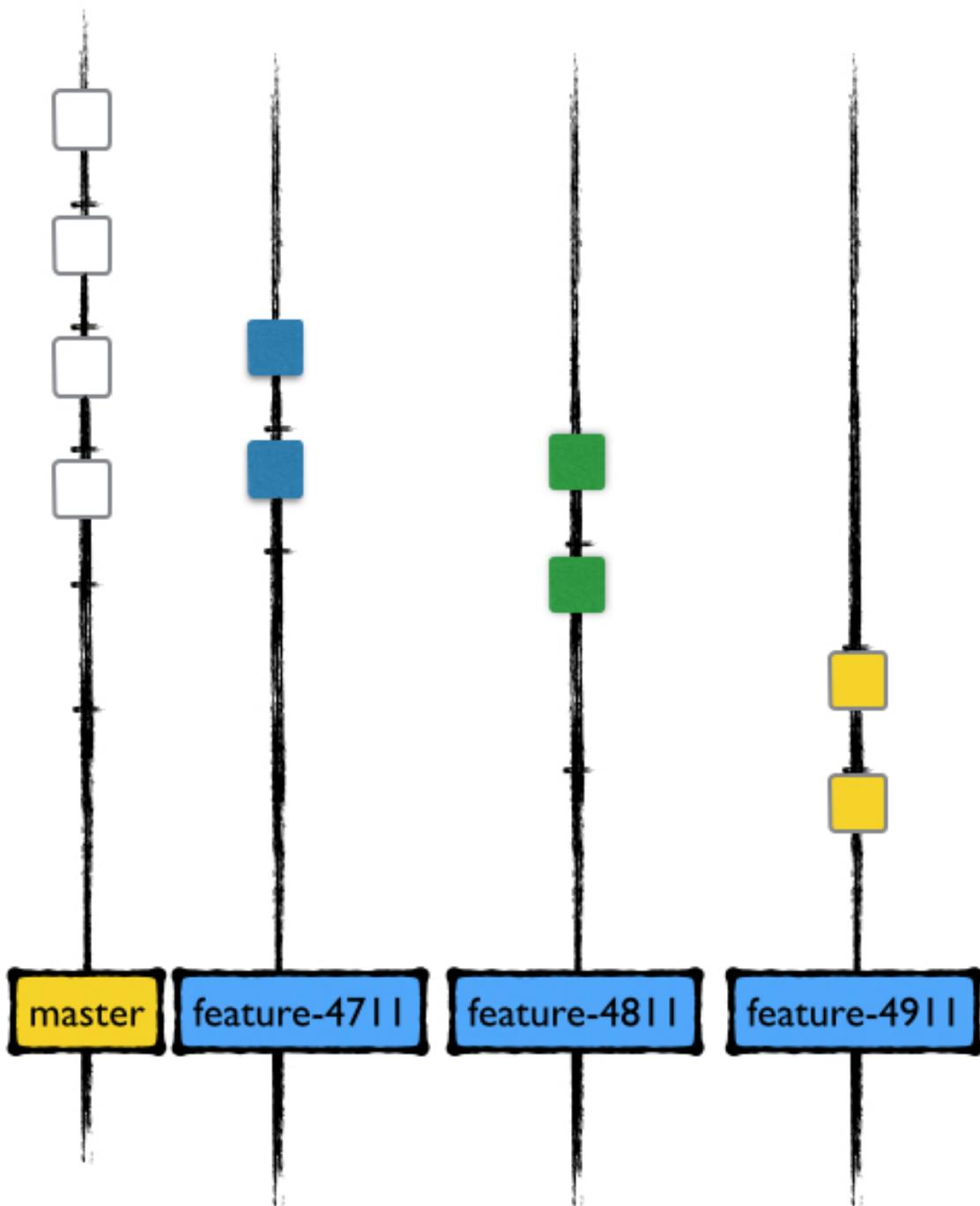
}

» Ich muss gestehen, ich bin wirklich kein großer Fan von Feature-Flags. Sie machen den Code fragil und wenn man es übertreibt, hat man überall *if / else* - Kaskaden, die nur schwer zu warten sind. Ich hatte das mal in einem Projekt, wo eigentlich in jeder Klasse diese Abfragen gemacht werden mussten und eigentlich die ganze Anwendung nur sehr schwer zu verstehen war. Es kam auch oft vor, dass Features versehentlich aktiviert wurden weil die Dokumentation unzureichend war. Es ist insgesamt einfach relativ gefährlich.

» *Ich denke, da sind wir uns einig. Deswegen finde ich auch das BPF-Modell ganz interessant. Damit könnten wir versuchen, das Thema Feature-Flags bereits zu erschlagen bevor die Version für das Deployment gebaut wird, und zwar ausschließlich mit Git und Branches. Die Idee hinter BPF ist ganz einfach und in wenigen Sätzen erklärt. Eine Version für ein Release wird nicht mehr statisch aus den Features erzeugt, die bereits auf den master zurückgeführt wurden sondern dynamisch auf Knopfdruck aus den vorhandenen Feature-Banches.*

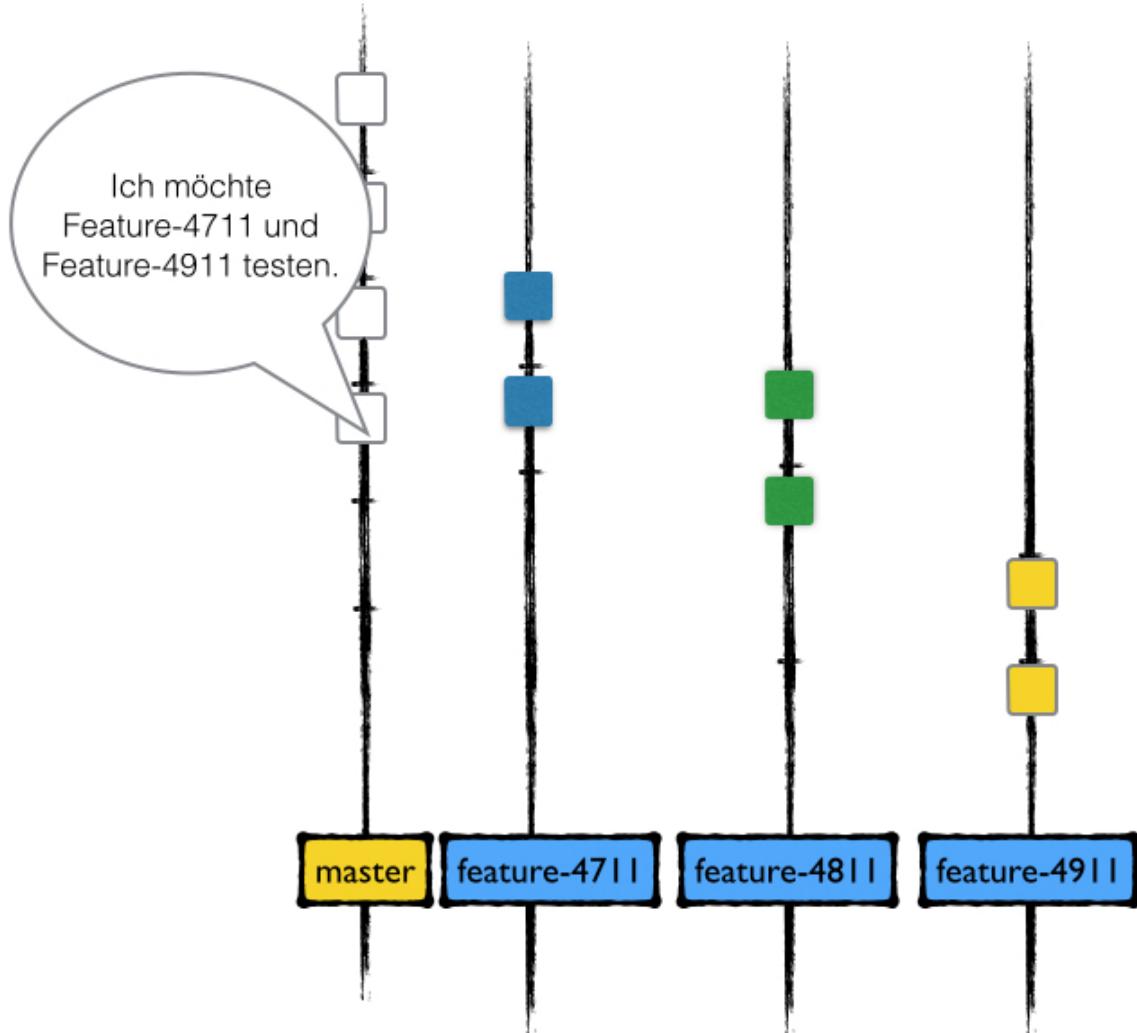
» Ganz ehrlich, das klingt für mich ein wenig abstrakt. Ist das überhaupt praktikabel? Macht es Sinn, dass wir uns das kurz zusammen am Whiteboard anschauen?

» *Definitiv. Wir haben nach wie vor unseren master-Branch sowie einen eigenen Feature-Branch für jedes Feature.*



» Karl, stell dir vor, ich bin der Product-Owner, ich hab das Sagen und bestimme. Ich entscheide, welche Features wir an unsere Kunden liefern und was diese zu sehen bekommen. Gutes Gefühl! Ich entscheide heute aus dem Bauch, dass wir die beiden Features **4711** und **4811** unseren Testern zur Verfügung stellen. Diese Features sind funktional abgeschlossen und sobald die Tester grünes Licht geben, werden wir sie installieren. Sollte aber eines der Features bei den Testern durchfallen, beispielsweise

wegen einem Bug, dann soll das Release wegen diesem Problem nicht verschoben werden.

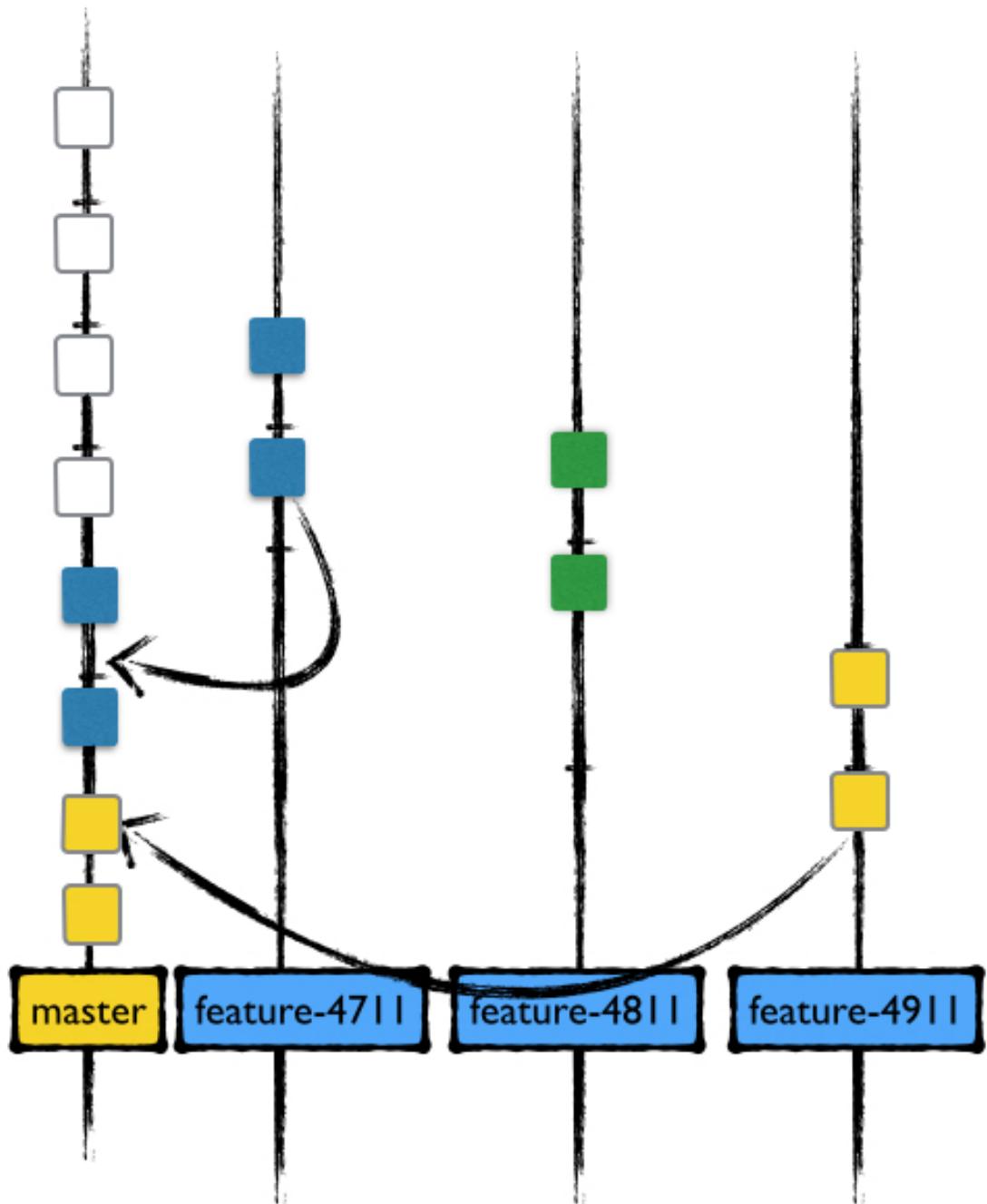


» Das sehe ich ein und macht Sinn. Normalerweise wäre das Release blockiert, bis entweder der Bug gefixt oder das Feature über einen Feature-Flag ausgeschaltet ist.

» *Was erfahrungsgemäß ziemlich lange dauern kann, nicht wahr?*

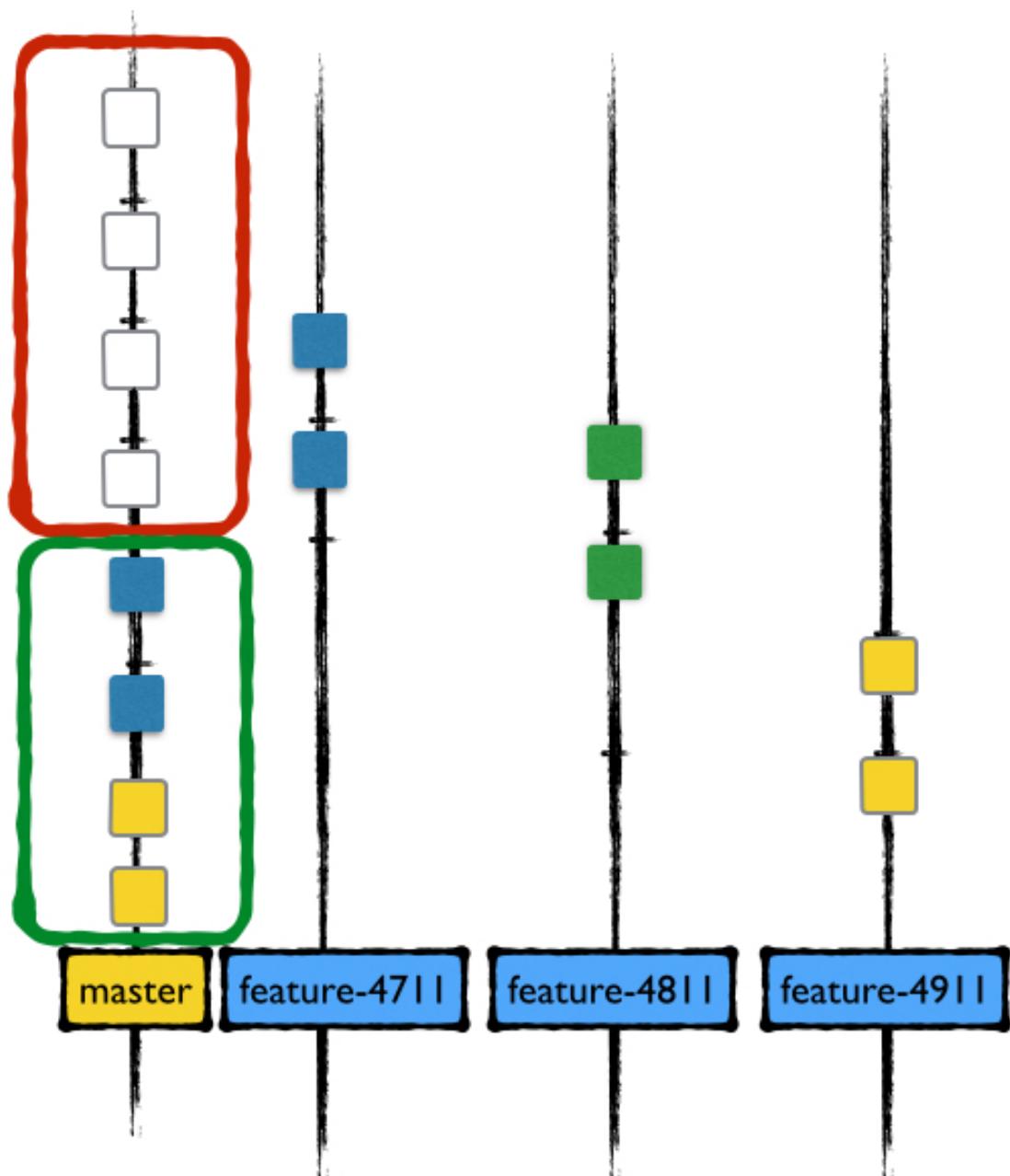
» Genau.

» *Karl, wenn du dir dieses Bild auf dem Whiteboard jetzt anschaust. Ich als Product-Owner möchte gerne mit einem Fingerschnippen entscheiden, dass beispielsweise Feature-4711 nicht deployt werden soll.*



» Ja klar willst du das, ich würde auch gerne fliegen können. Manches geht einfach nicht.

» *Das mit dem Fliegen machen wir später. Das mit dem Fingerschnippen sofort, pass auf.*



» Am besten wir schauen uns nochmal den **master** etwas genauer an. Die rot eingekreisten Commits sind "alte" Commits vom **master**, die bereits im letzten Release enthalten waren. Die jetzt grün eingekreisten Commits sind die neuen Commits, die wir gerade von den verschiedenen Feature-Banches gemerged haben, soweit klar? Ich entscheide mich jetzt also, dass das **Feature-4711** nicht reif ist und lieber nicht deployt werden soll. Erinnerst du dich noch an das Konzept von **Reset** und was es bedeutet einen Branch zurückzusetzen?

Übung

Zeit für eine kurze Wiederholung.

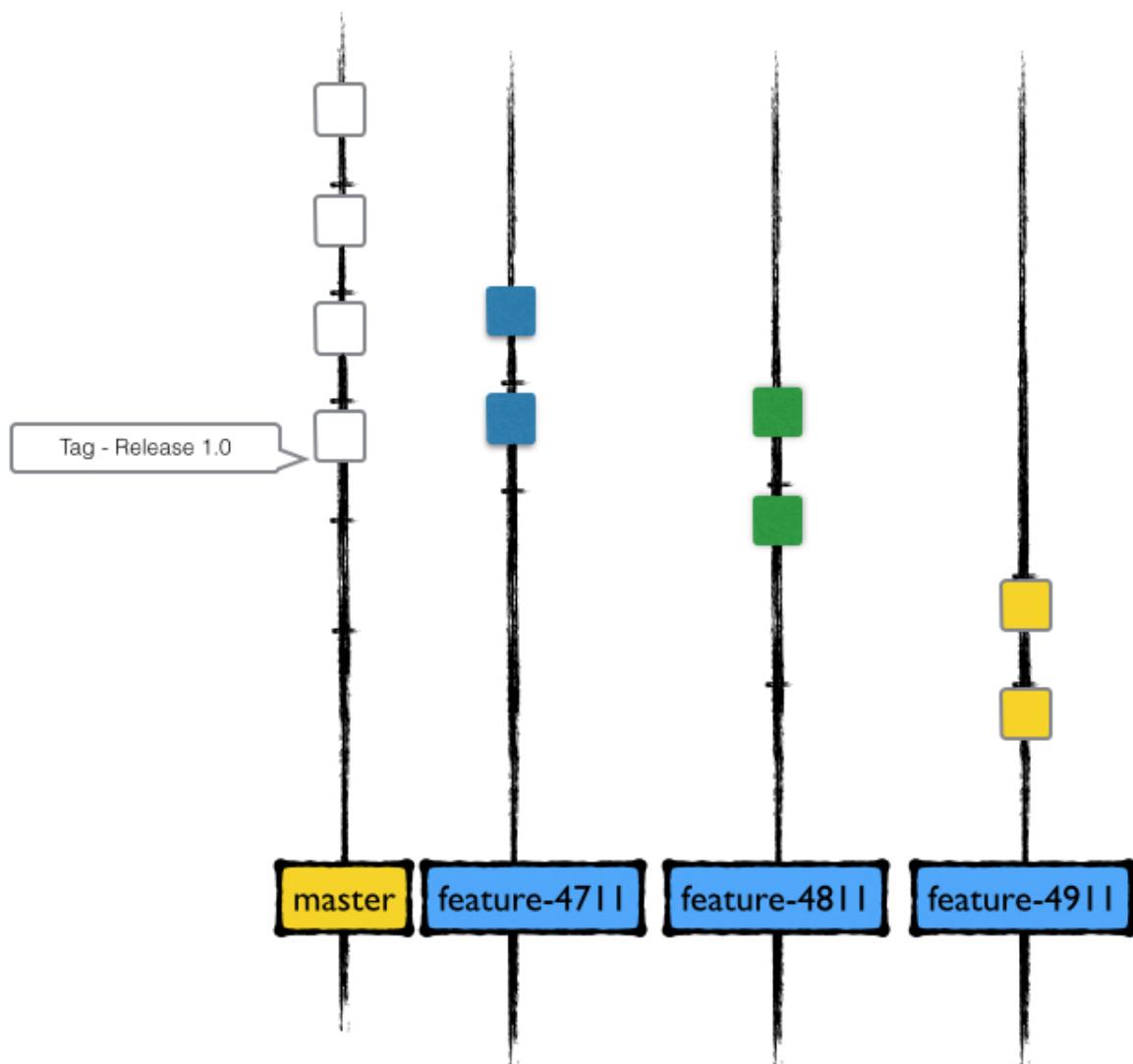
Lesen Sie nochmals kurz nach, was mit Hilfe von **git reset** möglich ist.

Erzeugen Sie ein neues leeres Repository

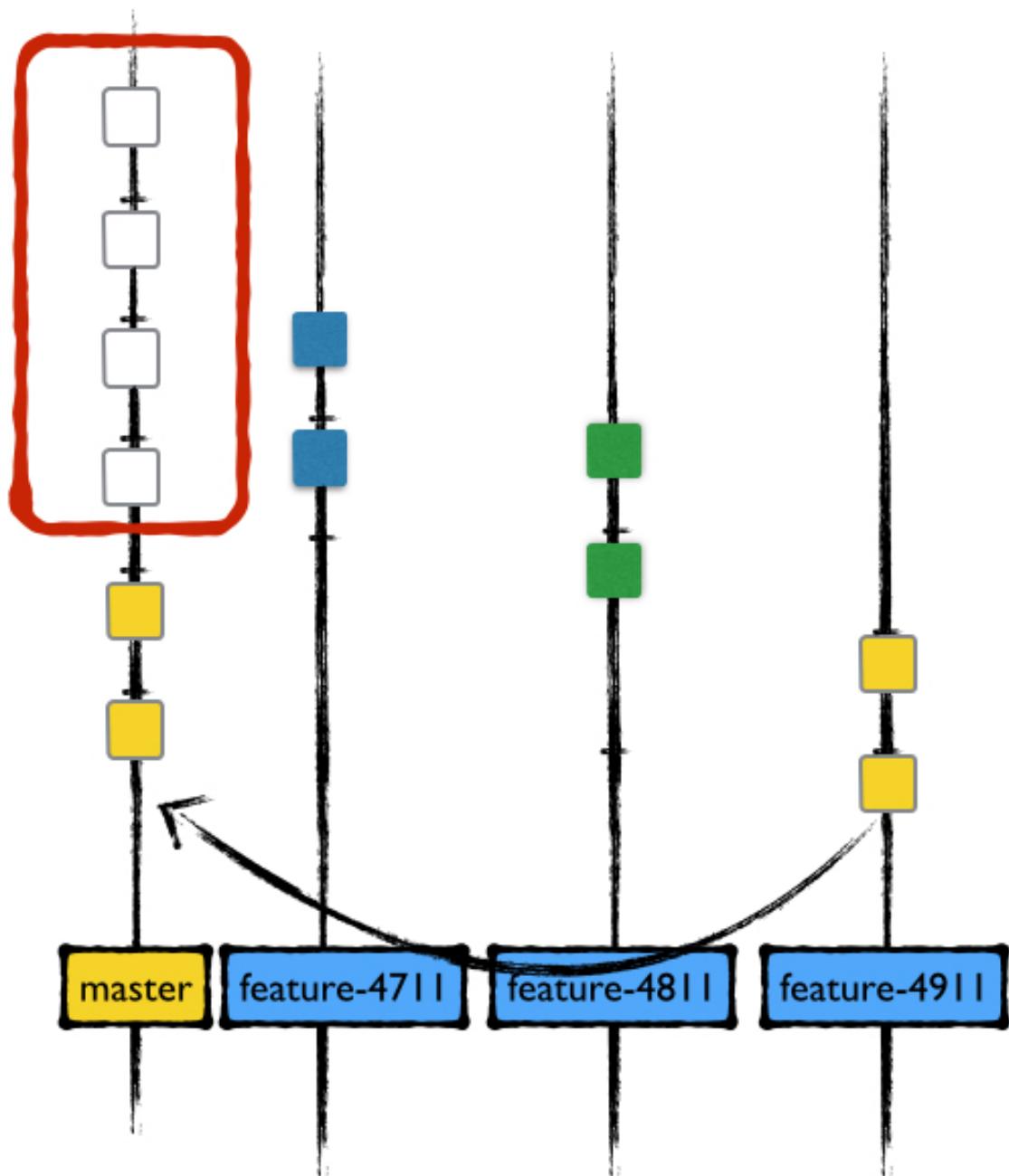
Erzeugen Sie 5 beliebige Commits

Springen Sie im Repository zum zweiten Commit

Springen Sie jetzt zum vierten Commit (Stichwort **Refllog**)



» Sobald der Product-Owner entscheidet, dass ein Feature nicht deployt werden soll, wird der Branch von dem wir Releases machen mit **reset** auf den letzten Tag des Releases zurückgesetzt. Das würde dann wieder so aussehen. Es ist so als wären die Feature-Banches nie gemerged worden.



» Lars, warte. Jetzt startet das Ganze wieder von vorne, richtig? Jetzt könnte ich das **Feature-4911** wieder in den master mergen, ohne aber vorher **Feature-4711** auch zu mergen oder?

» Sehr schön, Karl. Genau, du hast es verstanden. Das ist die Idee dahinter. Ich habe dir das hier alles ein wenig vereinfacht erklärt, in Wahrheit ist es ein klein wenig komplizierter. Die Idee an sich ist aber eigentlich recht einfach und genial, nicht wahr? Dadurch, dass wir **Feature-4711** einfach nicht ein zweites Mal mergen, wäre es so, als wäre es niemals zurückgeführt worden. Ich kann also tatsächlich mit Hilfe eines "Fingerschnippens" ein komplettes Feature entfernen.

» Aber das klingt wirklich ein wenig zu einfach. Lass mich nachdenken. Was passiert mit Merge-Konflikten?

» Ach, du hast direkt die Schwachstelle dieses Systems erkannt. Kannst du vielleicht schon erklären, wo wir wirklich Probleme haben werden?

» Moment, ich überlege. Am besten wir versuchen das direkt im Repository, das wir zuvor für Git-Flow verwendet haben. Ich lerne besser, wenn ich die Dinge praktisch anwende.

```
git branch
* master
  release
  support-1.0
```

» Zunächst taggen wir den **master**, wir tun also so, als hätten wir kürzlich ein Release gemacht.

```
git tag -a -m "release-1.0" release-1.0
git tag
1.0
5011
release-1.0
```

» Ok, wir entwickeln also die Features **4711**, **4811** und **4911**. Wieder mal...

» Ja, tut mir leid, für meine Kreativität bin ich nicht gerade bekannt. Aber ja, mach bitte weiter.

```
#vom master
git checkout master
Switched to branch 'master'
#branch
git checkout -b "fb-4711"
Switched to a new branch 'fb-4711'
```

```
echo "feature 4711 - Implementierung" >> feature.txt
git add feature.txt
git commit -m "feature-4711"
[fb-4711 b132dbb] feature-4711
 1 file changed, 1 insertion(+)
```

» Dasselbe machen wir für 4811. Aber bitte so, dass ein Merge-Konflikt zwischen den beiden Features besteht.

```
#wieder vom master
git checkout master
Switched to branch 'master'
#branch
git checkout -b "fb-4811"
Switched to a new branch 'fb-4811'

echo "feature 4811 - Implementierung" >> feature.txt
git add feature.txt
git commit -m "feature-4811"
[fb-4811 72d335a] feature-4811
 1 file changed, 1 insertion(+)
```

» Und ein letztes Mal für 4911.

```
#und wieder vom master
git checkout master
Switched to branch 'master'

#branch
git checkout -b "fb-4911"
Switched to a new branch 'fb-4911'

echo "feature 4911 - Implementierung" >> feature-4911.txt
git add feature-4911.txt
git commit -m "feature-4911"
[fb-4911 5f66eaa] feature-4911
 1 file changed, 1 insertion(+)
 create mode 100644 feature-4911.txt
```

» Wir gehen jetzt zurück auf den **master** und mergen die Features. Ich habe mir schon einige Gedanken zu diesem Modell gemacht. Ich denke, wir sollten das Zusammenführen der Features auf einem eigenen Branch **qa** machen, den wir direkt

vom **Tag** ziehen, den wir für das letzte Release gemacht haben. Auf diesen Branch mergen wir alle Features, die wir der QA-Abteilung zur Verfügung stellen möchten.

```
git checkout -b qa 'release-1.0' ①
```

```
#branches
git branch
fb-4711
fb-4811
fb-4911
master
* qa
  release
  support-1.0
```

① Erzeuge den Branch mit Namen **qa** direkt vom Tag mit dem Namen **release-1.0**
» Spielt die Reihenfolge dabei eine Rolle? Ich meine, welches Feature zuerst gemerged wird.

» *Prinzipiell ja, das werden wir aber gleich sehen. Wir fangen zunächst an mit 4711, mergen dann 4811 und zuletzt 4911.*

ReReRe-Cache

» Ich glaube ich sehe schon das Problem. Wir erzeugen den **qa**-Branch von wir die Version bauen immer wieder neu, richtig? Und mergen die Features jedesmal wieder zusammen. Müssen wir dann nicht die gleichen Merge-Konflikte immer und immer wieder lösen?

» *Genau! Das ist die große Schwierigkeit beim BPF Modell. Kein Entwickler löst einen Merge-Konflikt gerne zweimal. Das wäre auch ziemlich ineffizient.*

» Aber wie verhindern wir das?

» *Das die Merge-Konflikte auftreten gar nicht. Aber wir können Git sagen, dass es sich merken soll, wie ein Merge-Konflikt, der bereits einmal aufgetreten ist gelöst wurde. Tritt derselbe Merge-Konflikt erneut auf, kann Git den Konflikt automatisch lösen.*

» Wirklich, das funktioniert? Klingt ein wenig wie Magie.

» *Naja, das Wort hierfür ist ReReRe und steht für Reuse-Recorded-Resolutions. Das Feature ist wirklich sehr gut, leider kennen es die wenigsten Entwickler und es ist per*

*Default inaktiv. Ich würde empfehlen, **ReReRe** generell zu aktivieren, das machen wir jetzt direkt.*

```
git config rerere.enabled true  
git config rerere.autoupdate true
```

» Das ist alles?

» *Das ist alles. Mit der Konfiguration **rerere.enabled** aktivierst du generell den **ReReRe-Cache**. Mit **rerere.autoupdate** sorgst du dafür, dass Git Änderungen nach einem automatisch gelösten Merge-Konflikt automatisch dem Index hinzufügt, quasi ein automatisches **git add**. Du sparst dir jedesmal ein **git add**. Und jetzt arbeiten wir ganz normal weiter, als wäre nichts gewesen.*

» Ok, wir mergen als die Features auf den **qa**-Branch.

```
git merge fb-4711  
Updating 1f4d632..f0150b9  
Fast-forward  
 feature.txt | 1 +  
 1 file changed, 1 insertion(+)  
  
#und jetzt 4811  
git merge fb-4811  
Auto-merging feature.txt  
CONFLICT (content): Merge conflict in feature.txt  
Recorded preimage for 'feature.txt' ❶  
Automatic merge failed; fix conflicts and then commit the result.
```

❶ ReReRe speichert den Merge Konflikt

» Ok, wie erwartet haben wir einen Merge-Konflikt. Soll ich den auflösen?

» *Ja, sofort. Schau dir aber bitte vorher nochmal genau die Ausgabe an. Du hast etwas übersehen.*

» Warte mal, du hast Recht! Da steht **Recorded preimage for feature.txt**. Ist das der **ReReRe-Cache**?

» *Genau, du siehst genau was passiert, wenn du dir anschaugst, wo **ReReRe** seine Daten speichert. Schau am besten mal in dein **.git**-Verzeichnis.*

```
ls .git/rr-cache/
```

```
925cd9b95c7a5b830701d56480b9530de341afc8
```

» Es ist ein neues Verzeichnis **rr-cache** entstanden. Dieses Verzeichnis verhält sich ganz ähnlich zum **objects**-Verzeichnis das du ja bereits kennst.

```
less .git/rr-cache/925cd9b95c7a5b830701d56480b9530de341afc8/preimage

#Merge Konflikt
working with git flow is so easy
<<<<<
feature 4711 - Implementierung
=====
feature 4811 - Implementierung
>>>>>
```

» Du siehst Karl, in der Datei **preimage** hat Git den Merge-Konflikt abgelegt, so wie er ist bevor du ihn aufgelöst hast. Jetzt lösen wir den Konflikt.

Übung

Lösen Sie den Merge-Konflikt auf. Verwenden Sie der Übung halber ein anderes Merge-Tool als zuvor. Kennen Sie noch den Befehl um das Git-Mergetool zu starten?

```
git commit -m "merged features"
Recorded resolution for 'feature.txt'. ①
[qa 55767ed] merged features
```

① Git merkt sich die Lösung des Konfliktes

» Karl du siehst, dass **ReReRe** sich scheinbar auf die Lösung des Konfliktes merkt.

```
ls .git/rr-cache/
925cd9b95c7a5b830701d56480b9530de341afc8

#Konflikt Lösung
ls .git/rr-cache/925cd9b95c7a5b830701d56480b9530de341afc8/
postimage ①
preimage
```

#Postimage

```
less .git/rr-cache/925cd9b95c7a5b830701d56480b9530de341afc8/postimage
working with git flow is so easy
feature 4711 - Implementierung
feature 4811 - Implementierung
```

- ❶ Ein neues Verzeichnis **postimage** ist entstanden.
 » In der neu entstandenen Datei **postimage** steht ganz genau, wie die betroffenen Dateien nach der Lösung des Konfliktes auszusehen haben. Tritt der gleiche Konflikt erneut auf muss Git lediglich prüfen, wie die Lösung auszusehen hat.
- » Ok, Karl. Jetzt mergen wir noch **4911**. Wir wollen unserer Testabteilung schließlich alle Features zur Verfügung stellen, von denen wir glauben, dass sie fertig sind.

Übung

Mergen Sie das Feature 4911 in den master. Es sollte hierbei kein Merge-Konflikt auftreten.

```
git merge fb-4911
Merge made by the 'recursive' strategy.
 feature-4911.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature-4911.txt
```

- » Ok, Karl, wir spulen in Gedanken jetzt mal einige Tage vorwärts. Die Testabteilung hat die neue Version ausgiebig getestet und hat einige kritische Bugs im **Feature-4911** festgestellt. Wir möchten das Feature also temporär wieder herausnehmen, bis diese Fehler behoben sind. Dazu bauen wir eine komplett neue Version. Hier unterscheidet sich **BPF** grundsätzlich von beispielsweise **Git-Flow**. Mit **Git-Flow** würden wir jetzt einen Hotfix machen, den Fehler beheben und eine neue Version deployen.

In **BPF** ist das Vorgehen, dass wir das komplette Feature herausnehmen und eine neue Version ohne dieses Feature deployen. Um die neue Version zu bauen löschen wir den **qa-Branch**. Ja du hast richtig gehört, wir löschen den Branch und erzeugen ihn einfach neu. Das klingt radikal, aber denk mal darüber nach. Auf dem **qa-Branch** ist nichts, was sich nicht automatisch aus dem **master** und den **Feature-Banches** neu generieren lässt. Wir können ihn problemlos komplett neu erstellen mit den Features, die wir brauchen.

```
#zurück zum master  
git checkout master  
Switched to branch 'master'
```

```
#branch löschen  
git branch -D qa  
Deleted branch qa (was d0adf7b) .
```

```
#branch neu erzeugen  
git checkout -b qa 'release-1.0'  
Switched to a new branch 'qa'
```

```
#merge 4711 und 4811  
git merge fb-4711  
Updating 1f4d632..f0150b9  
Fast-forward  
 feature.txt | 1 +  
 1 file changed, 1 insertion(+)
```

```
git merge fb-4811  
Auto-merging feature.txt  
CONFLICT (content): Merge conflict in feature.txt ❶  
Staged 'feature.txt' using previous resolution. ❷  
Automatic merge failed; fix conflicts and then commit the result.
```

```
#statsu  
git status  
On branch qa  
All conflicts fixed but you are still merging.  
(use "git commit" to conclude merge)
```

```
Changes to be committed: ❸
```

```
modified:   feature.txt
```

-
- ❶ Der Konflikt tritt erneut auf
 - ❷ ReReRe löst den Konflikt automatisch
 - ❸ Der Konflikt ist gelöst, es muss nur noch Committed werden.
» *Karl, wir haben die gleiche Version wie zuvor, nur dass das Feature-4911 nie zurückgeführt wurde. Soweit verstanden?*

» Ja das Prinzip habe ich verstanden. Ist denn dieser **ReReRe-Cache** im ganzen Team verfügbar?

Shared ReReRe-Cache

» *Leider nein, **ReReRe** ist zunächst nur als eine lokale Merge-Hilfe gedacht. Git bietet von Haus aus keine Möglichkeit, den **ReReRe-Cache** zu teilen.*

» Ist das nicht problematisch? Was ist, wenn ich die Lösung für alle Merge-Konflikte bei mir lokale am Rechner habe und in Urlaub fahre? Was ist, wenn beispielsweise Felix den **qa-Branch** neu erzeugen möchte? Muss er dann die ganzen bereits gelösten Merge-Konflikte erneut lösen?

» *Das wäre fatal. Nein, es gibt einen Workaround, wie der **ReReRe-Cache** geteilt werden kann. Der Workaround ist aber nicht perfekt. Er arbeitet mit einem Hook und funktioniert bisher nur auf der Konsole richtig gut. Ich habe kürzlich ein wenig damit experimentiert.*

» Über Hooks haben wir uns ja bereits unterhalten. Wie genau arbeitest du funktioniert das Sharing des Caches?

» *Eigentlich ist die Idee ganz einfach. Ich würde den **ReReRe-Cache** in einem eigenen separaten Repository verwalten. Am besten wir spielen das Szenario direkt durch, dann verstehst du, was ich meine. Leg doch bitte einfach im gleichen Verzeichnis, wo unser Test-Repository aktuell liegt ein neues leeres Repository an.*

```
mkdir rerere-cache

#initialize repository
git init --bare
Initialized empty Git repository
```

» *Ich hatte dir schon erklärt, dass der **ReReRe-Cache** in deinem .git-Verzeichnis im Ordner **rr-cache** liegt. Diesen Ordner **rr-cache** machen wir einfach zu einem "lokalen" Repository im Repository.*

» Bitte? Ein Repository im Repository?

» *Denk daran, ein Git-Repository ist prinzipiell nichts anderes als eine vorgegebene Verzeichnisstruktur und einige Shell-Skripte. Es spricht überhaupt nichts dagegen, irgendwo .git-Verzeichnis ein weiteres Repository zu initialisieren. Alles im .git-Verzeichnis wird von Git standardmäßig nicht getrackt. Wir versuchen das einfach.*

```
cd .git/rr-cache
ls
925cd9b95c7a5b830701d56480b9530de341afc8

#initialize repository
git init
Initialized empty Git repository in ./git-flow-example/.git/rr-cache/.git/

#Repository verbinden
git remote add origin ../../../../rerere-cache/
git fetch origin

#bereits existierende Lösungen teilen
git add .
git commit -m "new cache"
[master (root-commit) 5383293] new cache
 3 files changed, 15 insertions(+)
create mode 100644 925cd9b95c7a5b830701d56480b9530de341afc8/postimage
create mode 100644 925cd9b95c7a5b830701d56480b9530de341afc8/preimage
create mode 100644 925cd9b95c7a5b830701d56480b9530de341afc8/thisimage

git push origin master
git push origin master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 481 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To ../../../../../../rerere-cache/
 * [new branch]      master -> master
```

» Die Idee ist ganz einfach. Alles was Git braucht, um mit ReReRe zu arbeiten ist der Inhalt des **rr-cache** Verzeichnisses. Wir machen das komplette Verzeichnis einfach zu einem **Git-Repository** und teilen die Inhalte so. Ein Entwickler kann sich alle Merge-Konflikt-Lösungen dann über ein einfaches **git pull origin/master** holen.

» Genial. Aber jeder Entwickler muss sich selbst darum kümmern, dass die neuen Inhalte *committed* und *gepusht* werden? Was ist, wenn ich zum Beispiel einen Merge-Konflikt lokal löse, aber vergesse die Lösung einzuchecken und zu pushen?

» Guter Punkt. Auch hier könnte wieder mit Hilfe von **Hooks** gearbeitet werden. Bevor wir das aber machen, wollen wir noch sicherstellen, dass unser bisheriger Workflow tatsächlich funktioniert. Wir **klonen** uns das Repository und initialisieren auch hier den **ReReRe-Cache**.

```
git clone git-flow-example/ git-flow-example-2
Cloning into 'git-flow-example-2'...
done.
```

» Wir müssen **ReReRe** für jedes Repository neu initialisieren. Auch in unserem geklonten Repository ist **ReReRe** nicht aktiv. **ReReRe** lässt sich entweder über die Konfiguration initialisieren, die wir vorher besprochen haben oder aber indem wir manuell das **rr-cache** Verzeichnis im **.git**-Verzeichnis anlegen.

```
mkdir .git/rr-cache/
cd .git/rr-cache

#initialisiere Repository
git init
Initialized empty Git repository in ./git-flow-example-2/.git/rr-
cache/.git/

#verbinde mit Repository
git remote add origin ../../../../rerere-cache/
git fetch origin
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From ../../../../../../rerere-cache
 * [new branch] master      -> origin/master

git pull --rebase origin master
From ../../../../../../rerere-cache
 * branch master      -> FETCH_HEAD
Current branch HEAD is up to date.

#Inhalte prüfen
ls
925cd9b95c7a5b830701d56480b9530de341afc8 ①
```

① Die Inhalte sind identisch

» Bevor wir jetzt wirklich mit **ReReRe** arbeiten stellen wir sicher, dass der Cache nach jedem Merge automatisch mit allen Teammitgliedern geteilt werden. Idealerweise teilen wir den **rr-Cache** einfach nach jedem Commit. Hierfür verwenden wir den **post-commit-Hook**, der nach jedem Commit ausgeführt wird. Perfekt für unsere Zwecke. Hierfür passen wir die Datei **.git/hooks/post-commit.sample** an.

Übung

Editieren Sie die Datei `.git/hooks/post-commit.sample` und entfernen Sie alle Inhalte

Fügen Sie folgendes Skript ein.

```
#!/bin/bash
cd .git/rr-cache
git pull --rebase origin master
git add .
git commit -m "new cache"
git push origin master
```

Benennen Sie die Datei von **post-commit.sample** um in **post-commit**. Ist die Datei nicht vorhanden legen Sie sie bitte einfach an.

Machen Sie diese Änderung in beiden lokalen Repositories.

» *Im Prinzip machen wir nach jedem Commit ein Update unseres geteilten rr-Caches ("git pull --rebase"), fügen alle neuen Inhalte hinzu, wenn es welche gibt ("git add ."), committen die Änderungen ("git commit -m ..") und teile alle neuen Inhalte ("git push origin .."). Jedesmal, wenn lokal ein neuer Merge-Konflikt gelöst wird landen die Pre- und Post-Images im rr-cache Verzeichnis und wir teilen diese neuen Inhalte mit allen Teammitgliedern.*

» Es ist tatsächlich einfach! Ich hätte gar nicht daran gedacht, dass es so einfach sein kann. Hast du das bereits ausprobiert?

» *Naja, wir arbeiten derzeit noch nicht mit BPF. Aber ich sehe keinen Grund, warum das nicht funktionieren sollte. Am besten wir spielen das BPF-Szenario einfach mal komplett durch.*

BPF in Action

» *Ok, wir haben aktuell zwei Repositories, mit denen wir arbeiten können. Wir implementieren ein neues Feature in einem Repository und provozieren dabei einen Merge-Konflikt. Anschliessend erzeugen wir den qa-Branch mit allen Features erneut. Anschließend machen wir ein Update im anderen Repository und erzeugen den*

qa-Branch erneut, wobei wir wieder ein Feature entfernen. Eigentlich sollte das funktionieren, ohne dass ein Merge-Konflikt gelöst werden muss.

» Ich bin gespannt, ob das tatsächlich so funktioniert.

```
#Im Repository 1
git checkout master
Switched to branch 'master'

#Lösche qa Branch
git branch -D qa
Deleted branch qa (was f0150b9) .

#Feature 5011
git checkout -b fb-5011
Switched to a new branch 'fb-5011'

#Commit / Update + Share RR-Cache
git commit -m "Feature in Konflikt mit 4711 und 4811"
From ../../../../rerere-cache
 * branch           master      -> FETCH_HEAD
Current branch master is up to date.
On branch master
nothing to commit, working directory clean
Everything up-to-date
[fb-5011 7fb64ab] Feature in Konflikt mit 4711 und 4811
 1 file changed, 1 insertion(+)

#erzeuge qa Branch
git checkout qa
Switched to branch 'qa'
```

» Ok, der nächste Schritt besteht jetzt darin, alle Features nach und nach auf den neuen "leeren" **qa-Branch** zu mergen.

```
#4711
git merge fb-4711
Updating 1f4d632..f0150b9
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)

#4811
git merge fb-4811
```

```
Auto-merging feature.txt
CONFLICT (content): Merge conflict in feature.txt ❶
Staged 'feature.txt' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

```
#commit merge
git commit -m "merged features"
From ../../../../rerere-cache
 * branch master      -> FETCH_HEAD
Current branch master is up to date.
On branch master
nothing to commit, working directory clean
Everything up-to-date
[qa dd3a949] merged features
```

```
#4911
git merge fb-4911
Merge made by the 'recursive' strategy.
 feature-4911.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature-4911.txt
```

```
#5011 mit Konflikt
git merge fb-5011
Auto-merging feature.txt
CONFLICT (content): Merge conflict in feature.txt
Recorded preimage for 'feature.txt' ❷
Automatic merge failed; fix conflicts and then commit the result.
```

```
#check ReReRe
ls .git/rr-cache
925cd9b95c7a5b830701d56480b9530de341afc8
f810bb4c35a6bafb4f5d48179b66a4bcc27dd2b3 ❸
```

❶ Automatische Lösung mit ReReRe

❷ Neues Pre-Image

❸ Neuer Eintrag in rr-cache

» *Den Merge-Konflikt haben wir ja erwartet. Karl, schaffst du es, den Konflikt zu lösen?*

Übung

Lösen Sie den Merge-Konflikt so auf, dass alle Änderungen erhalten bleiben und machen Sie anschließend den Commit.

```
git commit -m "merged features"
Recorded resolution for 'feature.txt'.
From ../../..../rerere-cache
 * branch           master      -> FETCH_HEAD
Current branch master is up to date.

[master ef6a683] new cache ①
 2 files changed, 11 insertions(+)

  create mode 100644 f810bb4c35a6bafb4f5d48179b66a4bcc27dd2b3/postimage ②
  create mode 100644 f810bb4c35a6bafb4f5d48179b66a4bcc27dd2b3/preimage ③
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 531 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To ../../..../rerere-cache/
  5383293..ef6a683 master -> master
[qa 6d66c4f] merged features
```

- ① Der ReReRe-Cache wird automatisch geteilt.
- ② PreImage
- ③ PostImage

» Soweit so gut. Was jetzt, Lars? Spielen wir das gleiche Szenario nochmal im zweiten Repository durch und tun so, als würde einfach ein zweiter Entwickler den **qa-Branch** einige Tage später erneut erzeugen?

» *Genau, das wäre die Idee. Sollen wir wieder ein Feature exkludieren? Lass uns den **qa-Branch** wieder ohne das Feature **4911** erzeugen. Am besten du wechselst in das andere Repository, in dem wir bisher noch nichts vom Feature **5011** wissen, machst ein Update und erzeugst den **qa-Branch** neu.*

```
#Wechsel in Repository #2
```

```
#Wechsle auf master
git checkout master
```

```
Switched to a new branch 'master'

#Update
git fetch origin
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (11/11), done.
From /Users/martindilger/development/git/playground/git-flow-example
 * [new branch] fb-5011      -> origin/fb-5011 ①
   f0150b9..6d66c4f qa      -> origin/qa
```

① Feature 5011 ist verfügbar

» Ok, Karl, bereit? Dann löschen wir jetzt den **qa-Branch**, erzeugen ihn neu und mergen dann die Features **4711**, **4811** und **5011**.

```
git branch -D qa
Deleted branch qa (was f0150b9).
```

```
#erzeuge qa Branch neu
git checkout -b qa
Switched to a new branch 'qa'
```

```
#merge features
git merge origin/fb-4711 ①
[...]
git merge origin/fb-4811 ②
[...]
git merge origin/fb-5011 ③
Auto-merging feature.txt
CONFLICT (content): Merge conflict in feature.txt
Resolved 'feature.txt' using previous resolution. ④
Automatic merge failed; fix conflicts and then commit the result.
```

- ①** Syntax "origin/4711", da fb-4711 lokal nicht verfügbar, da bisher kein **checkout** gemacht wurde.
- ②** Syntax "origin/4811", da fb-4811 lokal nicht verfügbar, da bisher kein **checkout** gemacht wurde.
- ③④** Der Merge-Konflikt wurde automatisch gelöst und der ReReRe-Cache geteilt.
» Sehr schön, Lars. Siehst du, es hat funktioniert! Der Merge-Konflikt wurde auch im zweiten Repository automatisch gelöst!

Die Reihenfolge

» Es kann jedoch noch zu Problemen kommen. Du hast mich vorhin gefragt, ob die Reihenfolge der Merges eine Rolle spielt. Ich würde sagen, teilweise. Fangen wir doch nochmal von vorne an, ich zeige dir was ich meine.

Übung

Spielen Sie das Szenario erneut durch.

Löschen Sie den qa-Branch und erzeugen Sie ihn neu.

Mergen Sie die Features in der Reihenfolge 5011, 4711, 4911, 4811.

Prüfen Sie, ob immer noch alle Konflikte automatisch gelöst werden können.

```
git checkout master

git branch -D qa
Deleted branch qa (was 6d66c4f).

git checkout -b qa
Switched to a new branch 'qa'

#merge features
git merge fb-5011
[...]

git merge fb-4711
Auto-merging feature.txt
CONFLICT (content): Merge conflict in feature.txt
Recorded preimage for 'feature.txt' ①
Automatic merge failed; fix conflicts and then commit the result.
```

① Merge-Konflikt

» Du siehst, Karl, ganz so einfach ist es leider nicht. Je nachdem, in welcher Reihenfolge die Branches gemerged werden können unterschiedliche Merge-Konflikte auftreten. Es ist aber egal, ob jetzt 4711 und dann beispielsweise 4811, oder erst 4811 und dann 4711 gemerged wird. Diese Operation ist **kommutativ**. Wenn aber erst beispielsweise 5011 gemerged wird und dann 4711, wird ein Konflikt auftreten. Wird

*anschliessend 4811 gemerged wird leider erneut ein Konflikt auftreten, da die Lösung für den Konflikt 4711/4811 nicht für die Konstellation 5011/4711/4811 gilt. Ebenso kann es passieren, dass Lösungen nicht mehr gelten, wenn ein Feature entfernt wurde. Stell dir vor, du löst den Konflikt 5011/4711 und anschließend den Konflikt 4711/4811. Stell dir jetzt vor, wir entfernen das Feature 4711. Jetzt haben wir beim Erzeugen des **qa-Branches** die Konstellation 5011/4811 für die bisher keine Lösung im **ReReRe-Cache** hinterlegt ist.*

» *Das ist doch total kompliziert, kann das in der Praxis überhaupt funktionieren?*

» Das ist der schwierige Teil. In der Praxis muss sichergestellt werden, dass die Branches möglichst immer in der gleichen Reihenfolge gemerged werden. Das ganze Modell macht nur unter der Annahme Sinn, dass Features nur selten entfernt werden. Es kann also durchaus passieren, dass plötzlich beim Erzeugen des **qa-Branches** Merge-Konflikte auftreten, das sollte allerdings nicht allzu oft vorkommen.

BPF Use Cases

» Ok, technisch ist mir jetzt schon klar, wofür wir BPF verwenden können. Was aber wäre ein mögliches Szenario in der Praxis?

» *Da fallen mir einige ein. Du weißt ja zum Beispiel, dass wir hier mit Jira als Task-Management-System arbeiten. Jira hat eine relativ mächtige **REST-API**, du kannst also sehr einfach über **HTTP-GET** Informationen aus Jira abrufen. Beispielsweise ist es möglich, dass Jira dir alle Tasks in einem bestimmten Status gibt. Ich denke beispielsweise an eine Liste von Tasks im Status "Ready for QA". Du könntest dann beispielsweise eine Liste in dieser Form bekommen.*

4711
4811
4911
5011
[...]

» *Jetzt musst du eigentlich nur noch über diese Liste iterieren und alle Branches automatisch mergen. Etwa so etwas in Pseudo-Code.*

```
git branch -D qa
git checkout -b qa
Jira-Task-List = getTaskListFromJira()
for(String taskNummer in Jira-Task-List)
```

Daily GIT.

```
git merge taskNummer_to_BranchName(taskNummer)
git push origin qa
```
