Quellcode auf CD!

Vom produktiven Arbeiten mit Git

Besser Gits nicht!

Ein Handwerker muss seine Werkzeuge kennen. Dabei reicht es aber bei Weitem nicht, nur theoretisch zu wissen, wie man beispielsweise einen Hammer verwenden könnte. Oft sind es einige spezielle Kniffe, die ein Werkzeug erst so richtig effektiv machen. Einige dieser Kniffe für das Werkzeug Git möchte dieser Artikel vorstellen. Eine kleine Anmerkung vorweg: Grundlegende Git-Kenntnisse werden vorausgesetzt.

von Martin Dilger

Versetzen wir uns in die Lage von M.. Er ist Java-Entwickler und Mitglied eines Scrum-Teams irgendwo in Nürnberg. Das Problem: M. stammt aus München und hat Familie. Das macht es unmöglich, die Woche in Nürnberg zu verbringen und so bleibt nur Pendeln. Das bedeutet, M. ist täglich unterwegs, und zwar 1,5 Stunden nach Nürnberg und abends wieder 1,5 Stunden zurück. Zeit, die effektiv genutzt werden will. M. hat Glück, denn in seinem Projekt wird seit Neuestem das dezentrale Versionskontrollsystem Git eingesetzt. Durch Git hat M. das komplette Repository immer dabei und so die Möglichkeit, auch von unterwegs und ohne Internetverbindung effektiv zu arbeiten, zu commiten oder die Historie zu betrachten. Im Folgenden begleiten wir M. durch einen typischen Arbeitstag und erhalten so einen Einblick in die effektive Arbeitsweise mit Git.

06:45 – Hauptbahnhof München

M. steigt am Hauptbahnhof in den ICE nach Nürnberg. Er hat sich am Abend zuvor noch einen wichtigen Produktions-Bug für die Fahrt mitgenommen, der heute dringend ausgeliefert werden muss. Hier kommt der

Mehr zum Thema

Mehr zum Thema Git finden Sie im entsprechenden Heftschwerpunkt der Ausgabe 1.2011. Dort erklärt Michael Johann die Git-Revolution: vom Klonen und Pushen in verteilten Welten. Die Ausgabe ist bestellbar via http://it-republik.de/jaxenter/java-magazin-ausgaben/.

Vorteil eines dezentralen Versionskontrollsystems voll zum Tragen, denn M. hat das komplette Projekt-Repository lokal auf seinem Laptop zur Verfügung.

Die Software, die von M. und seinem Team entwickelt wird, ist Swing-basiert und unterstützt die Personalabteilung bei der Berechnung des zu überweisenden monatlichen Gehalts für festangestellte Mitarbeiter (**Abb. 1**). Laut Bug-Beschreibung ist die Bedingung für die Anzeige eines Fehlertextes falsch, da die Fehlermeldung für korrekte Berechnungen angezeigt wird, im Fehlerfall jedoch nicht. Eine kleine Anmerkung des Autors: Was genau M. gemacht hat, um den Bug zu beheben, kann problemlos in der Historie im Git Repository nachvollzogen werden. Als sich der Zug in Bewegung setzt ist M. bereits dabei, den Bug zu fixen.

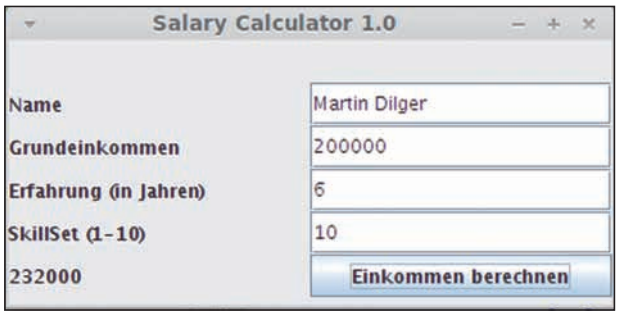

Abb. 1: Der Gehaltsrechner

Ein mögliches Branching-Modell

Zunächst verschafft sich M. einen Überblick über die aktuelle Repository-Struktur und die vorhandenen Branches:

```
git branch
* master
  stable-release
  next
```

Die größte Stärke von Git liegt in der einfachen und schnellen Möglichkeit, Branches zu erstellen und zwischen diesen zu mergen. Es gibt verschiedene Modelle, die eine kontinuierliche und stabile Lieferung in das Produktivsystem garantieren. Im Folgenden wird das Modell vorgestellt, mit dem das Team von M. sehr erfolgreich arbeitet.

Das Projekt ist in drei Branches unterteilt. Der *master* ist der aktuelle Entwicklungszweig, auf dem neue Features implementiert werden. Der *stable-release* zweigt vom *master* ab. Nur vom *stable-release* wird in das Produktivsystem geliefert. Auf dem *next*-Branch werden Features entwickelt, die zukünftig verfügbar sein, aber noch nicht in das Produktivsystem geliefert werden sollen. Da es sich bei M.s aktuellem Task um einen Bugfix handelt, erstellt er zunächst einen Branch vom *stable-release* (**Abb. 2**, Nr. 4). Vom (und nur vom) *stable-release* wird in das Produktivsystem geliefert:

```
#checkout stable-release branch
git checkout stable-release
#erstellen eines neuen bugfix-branches
git checkout -b bugfix-4711
# branch liste
git branch
* bugfix-4711
  master
  stable-release
  next
```

Die Ursache des Bugs ist schnell gefunden und behoben. Die Kondition für die Sichtbarkeit der Fehlermeldung ist einfach falsch gesetzt. M. commitet sehr (sehr) häufig. Für den Bugfix erzeugt er beispielsweise zwei Commits:

```
#print summary
git shortlog
initial project commit
Bugfix-4711 - correct condition for error label
Improved Javadocs a bit
```

Die beiden unteren Commits sind zwar physisch, aber nicht logisch getrennt. Besser wäre es, statt zwei nur einen Commit zu erzeugen. Das sollte bestenfalls geschehen, bevor der Bugfix Branch zurückgeführt wird. Git bietet hier ein hervorragendes Tool, und zwar den *interactive rebase*, der einfach mit *git rebase -i* gestartet wird. Der *interactive rebase* öffnet automatisch den voreingestellten Editor (in M.s Fall unter Ubuntu den vim, siehe Listing 1). Über einen *interactive rebase* hat M. die Möglichkeit, die lokale Historie zu ändern, solange noch nichts in das zentrale Repository eingecheckt wurde. Die Möglichkeiten, die sich M. hier bieten, sind u. a. das Ändern einer Commit Message im Nachhinein (*Option r - reword*) oder mehrere Commits zusammenzulegen (in Git auch als *squashen* bezeichnet):

```
git rebase -i HEAD~2
```

Listing 1

```
#start interactive rebase
git rebase -i HEAD~2
# in vim
pick fe8b38f Bugfix-4711 - correct condition for error label
pick fa227fb Improved Javadocs a bit
# Rebase 9950f83..fa227fb onto 9950f83
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
```

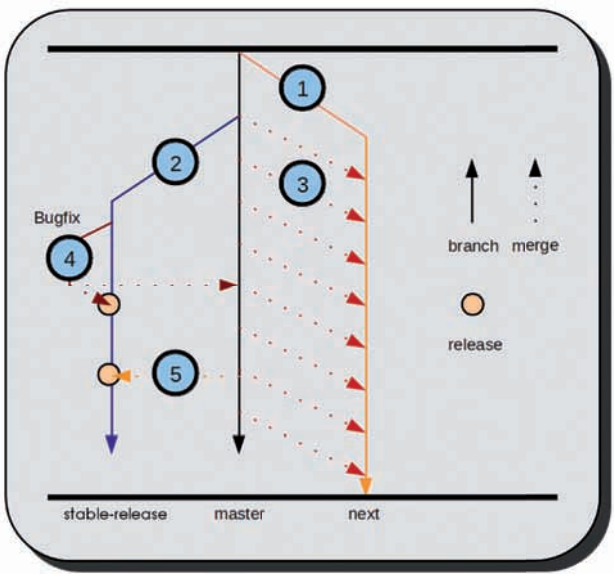

Abb. 2: Ein mögliches Branching-Modell

Die Syntax des *interactive rebase* ist sehr einfach zu verstehen, beispielsweise besagt der verwendete Ausdruck *HEAD~2*, dass vom obersten Commit aus die letzten beiden (~2) Commits bearbeitet werden sollen. Die exakte Beschreibung des mächtigen *rebase*-Befehls würde den Artikel sprengen, ist aber ausführlich unter [1] dokumentiert. Da M. einige Commits zusammenfassen möchte, ist der *interactive rebase* das Tool der Wahl. Hierzu editiert er lediglich die Zeilen der betroffenen Commits und macht beispielsweise direkt im Editor aus dem *pick* am Anfang der Zeile ein *f* für *fixup*. Hierdurch werden die beiden Commits „gesquashed" und ein neuer Commit entsteht, wobei die Commit Message des früheren Commits erhalten bleibt:

```
pick fe8b38f Bugfix-4711 - correct condition for error label
f fa227fb Improved Javadocs a bit
#now check the logs
git shortlog
initial project commit
Bugfix-4711 - correct condition for error label
```

Achtung, über einen *interactive rebase* wird die Historie verändert, da ggf. Commits verschwinden oder sich ändern können. Dieses Tool sollte also mit Bedacht verwendet werden und auch nur bei Commits, die nicht schon in einem öffentlichen Repository verfügbar sind.

Git Tags – Machen wirs gleich fix

Als Best Practice hat sich in M.s Team eingebürgert, dass sowohl Bugfix als auch Feature-Branches vor dem Zurückführen getaggt werden. Git verfügt über zwei Arten von Tags: *Annotated-* und *Lightweight*-Tags. Der Unterschied ist marginal, ein Lightweight-Tag ist lediglich eine Referenz auf einen Commit in der Historie, ein Annotated-Tag ist ein eigenes Objekt, das auf einen Commit in der Historie referenziert. M. arbeitet grundsätzlich mit Annotated-Tags (Option *-a*), da diese beispielsweise signiert werden können:

```
#create a tag with name Bugfix-4711
git tag -a -m "Bugfix-4711" Bugfix-4711
```

Durch den Tag lässt sich später sehr einfach nachvollziehen, ob ein bestimmter Bugfix oder ein bestimmtes Feature bereits reintegriert wurde, indem die vorhandenen Tags betrachtet werden:

```
#list all tags
git tag
Bugfix-4711
```

Der nächste Schritt besteht für M. nun darin, den Bugfix Branch sowohl auf den *stable-release* als auch den *master* zurückzuführen:

```
#reintegrate to stable-release
git checkout stable-release
git merge bugfix-4711
#reintegrate to master
git checkout master
git merge bugfix-4711
```

Warum aber auf beide Branches? Damit kommt exakt der Vorteil des dedizierten *stable-release* Branches zum Tragen. M. kann ein Bugfix-Release in das Produktivsystem veranlassen, der nur einen bestimmten Bugfix enthält, selbst wenn auf dem *master* eventuell schon weitere Features entwickelt wurden. Es ist sehr wichtig, dass der Bugfix sowohl auf den *stable-release* als auch den *master* zurückgeführt wird, da der *stable-release* bei jedem Major-Release gelöscht und neu vom *master* gezogen wird. Alternativ hätte M. den Bugfix auch nur auf den *master* zurückführen und diesen anschließend auf den *stable-release* mergen können (**Abb. 2**, Nr. 5), somit würden aber alle Features vom *master* mit ausgeliefert. Der letzte Schritt besteht für M. nun darin, den *master* auf den *next* Branch zu mergen, sodass der Bugfix auch in zukünftigen Versionen verfügbar ist (**Abb. 2**, Nr. 3):

```
#reintegrate to stable-release
git checkout next
git merge master
```

Damit hat M. alles vorbereitet und kann im Prinzip, sobald er im Büro ist, direkt eine neue Lieferung in das Produktivsystem veranlassen. Wohlgemerkt, M. sitzt noch immer im ICE in Richtung Nürnberg.

08:15 – Büro, Nürnberg

M. ist mittlerweile im Büro angekommen. Nach dem obligatorischen Plausch mit den Kollegen und nach-

dem er sich seine erste Tasse Kaffee geholt hat, fährt M. seinen Laptop zum zweiten Mal an diesem Tag hoch. Obwohl Git ein dezentrales Versionskontrollsystem ist, hat sich das Team von M. entschieden, ein zentrales Repository zu verwalten, auf das alle Entwickler einchecken und von dem aus der Continuous-Integration-Server (in M.s Fall Jenkins) baut und die entsprechenden Artefakte ausliefert. Der Zugriff auf das zentrale Repository erfolgt ganz einfach über SSH. M. freut sich, dass er heute Morgen bereits so produktiv war und möchte am liebsten sofort in das zentrale Repository einchecken:

```
#Push commits to Remote Master
git push origin master
! [rejected]        master -> master (non-fast-forward)
```

Was ist nun genau passiert?

```
git push origin master
```

bedeutet übersetzt: „Git, bitte übertrage die Commits auf meinem lokalen Branch, auf dem ich gerade bin, auf das Remote Repository mit dem Namen *'origin'*, und dort bitte auf den Branch mit dem Namen *master*“. *'origin'* ist lediglich der Name des Remote Repositories. Das wird sogar noch klarer, wenn man sich die Datei *config* im Verzeichnis *.git* im lokalen Git Repository betrachtet. Dort steht u. a. Folgendes:

```
[remote "origin"]
        url = ../central_git_repo/
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Was aber bedeutet folgender Satz, den M. auf seinen misslungenen Push-Versuch bekommen hat?

```
! [rejected]        master -> master (non-fast-forward)
```

Git erlaubt grundsätzlich nur Fast Forward Pushes auf entfernte Repositories. Das Prinzip eines Fast Forward Pushes ist in **Abbildung 3** dargestellt.

Bei einem Fast Forward Push sind bereits alle Commits des Remote Repositories im lokalen Repository enthalten (blaue Kreise). Das macht die Arbeit für Git natürlich extrem einfach. Es müssen lediglich die neuen Commits (weiße Kreise) entgegengeommen und ein Pointer vom alten HEAD (der Commit, auf dem der Branch aktuell steht) auf den neuesten Commit gesetzt werden. Dieser wird dann zum neuen HEAD. Die ganze Arbeit (z. B. Merge-Konflikte auflösen) muss bereits vorher lokal beim Entwickler erfolgt sein. Somit ist sichergestellt, dass das zentrale Repository immer einen „sauberen“ Stand hat. Kurz gesagt, Git hat den Push verweigert, weil noch nicht alle Commits aus dem Remote Repository im lokalen Repository vorhanden sind. M. bringt also zunächst sein lokales Repository auf den neuesten Stand:

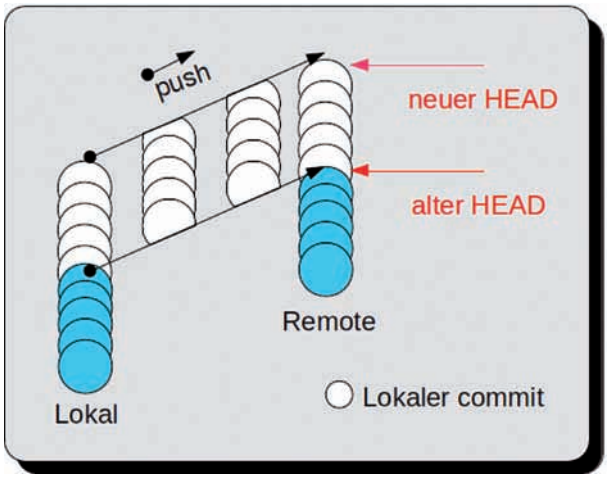

Abb. 3: Fast Forward Pushes

```
#pull from remote
git pull --rebase origin
Unpacking objects: 100% (9/9), done.
From ../central_git_repo
   9950f83..b3b6f11  master     -> origin/master
#check whats new
git shortlog -s
    1  Markus
    [..]
```

Offensichtlich war auch M.s Kollege Markus fleißig und hat ebenfalls schon eine Textänderung umgesetzt. Da M. jetzt alle Commits aus dem Remote Repository lokal zur Verfügung hat, versucht er erneut zu pushen (Listing 2).

Mit der Option *--all* pusht M. alle lokalen auf ihre entsprechenden Remote Branches. Die Option *--tags* stellt sicher, dass auch die gesetzten Tags übertragen werden, was standardmäßig nicht der Fall ist. Der Push war also erfolgreich, und alle Teammitglieder können auf den von M. gelieferten Bugfix zugreifen. Ein Release vom *stable-release* Branch kann in das Produktivsystem veranlasst werden. Zuletzt fällt M. jedoch noch auf, dass der Bugfix Branch ebenfalls unnötigerweise übertragen wurde. Da der Branch bereits reintegriert wurde,

Listing 2

```
#push to remote
git push --all --tags origin
Counting objects: 34, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (18/18), 1.33 KiB, done.
Total 18 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (18/18), done.
To ../central_git_repo/
 * [new branch]      bugfix-4711 -> bugfix-4711
 * [new branch]      stable-release -> stable-release
 * [new branch]      next -> next
```

kann dieser beruhigt gelöscht werden, um eine Branch-Müllhalde zu vermeiden:

```
#delete remote branch
git push origin :bugfix-4711
To ../central_git_repo/
 [deleted]         bugfix-4711
```

Der Befehl zum Löschen eines Remote Branches ist ein bisschen gewöhnungsbedürftig, macht aber Sinn, wenn man die Syntax genauer betrachtet. Die Syntax des *push*-Befehls sieht (vereinfacht) folgendermaßen aus:

```
git push <repository> <refspec>
```

Der Parameter *<repository>* wurde schon zuvor definiert und gibt an, auf welches Repository gepusht werden soll (in M.s Fall das Repository mit dem Namen *origin*). Der Parameter *<refspec>* ist schon interessanter und hat (ebenfalls vereinfacht) das Format *<src-branch>:<dest-branch>*. Hiermit kann angegeben werden, welcher lokale Branch (*src-branch*) auf welchen entfernten Branch (*dest-branch*) gepusht werden soll:

```
git push origin master:master
```

Daraus lässt sich jetzt auch sehr einfach herleiten, wieso die Refspec für das Löschen des Bugfix Branches das Format *:Bugfix-4711* hat. Vereinfacht könnte man interpretieren, dass M. einen „leeren" Branch auf das entfernte Repository schiebt und einfach den dort vorhandenen Branch „überschreibt". Nachdem M. den entfernten Branch gelöscht hat, zeigt ein kurzer Check aber, dass der Branch lokal immer noch vorhanden ist:

```
git branch
  bugfix-4711
  master
* stable-release
  next
```

Um diesen zu löschen, macht M. Folgendes:

Listing 3

```
git log --oneline
eef9feb A crappy Dummy Commit
1334204 Provide some better comment shit for this crappy class
#push to central repository
git push origin master
remote: Keine Kraftausdrücke!!
remote: error: hook declined to update refs/heads/master
To ../central_git_repo/
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to '../central_git_repo/'
```

```
git branch -D bugfix-4711
Deleted branch bugfix-4711 (was 039b68c)
```

Ein kleiner praktischer Hinweis: Hätte jemand anders den entfernten Branch gelöscht, hätte M. diesen immer noch in der Liste der Remote Branches gesehen. Diese werden nicht automatisch entfernt. Um aufzuräumen und alle Referenzen auf nicht mehr vorhandenen Remote Branches lokal zu löschen, macht M. regelmäßig Folgendes:

```
git remote prune origin
```

11:30 Kaffeeküche, Nürnberg

M. und Michael, ein weiteres Mitglied des Scrum-Teams, unterhalten sich gerade über die Resultate aus der letzten Retrospektive. Ein Punkt, der mehreren Teammitgliedern aufgefallen war, ist die relative Häufung von Flüchen und Kraftausdrücken in Commit Messages. Das Projekt ist schon weit fortgeschritten und alle Entwickler stehen unter Druck, also ein scheinbar typisches Phänomen [2]. M. verspricht Michael, sich eine Lösung einfallen zu lassen, denn aus rein professionellen Gesichtspunkten ist die Situation nicht tragbar. Zurück an seinem Arbeitsplatz überprüft M. die Liste an Commits der letzten Zeit und wird tatsächlich sofort fündig:

```
git log --oneline
1334204 Provide some better comment shit for this crappy class
[...]
```

Natürlich könnte M. versuchen, das Problem organisatorisch zu lösen, dieser Ansatz erscheint aber wenig erfolgversprechend. Viel besser scheint eine technische Lösung. Git bietet für derartige Fälle so genannte Hooks. Das sind Skripte, die bei bestimmten Ereignissen (beispielsweise Commits, Rebases oder Pushes) ausgeführt werden. Die möglichen Hook-Skripte befinden sich im Git Repository unter *.git/hooks*:

```
applypatch-msg.sample  post-update.sample       pre-rebase.sample
commit-msg.sample      pre-applypatch.sample    update.sample
post-commit.sample     pre-commit.sample
post-receive.sample    prepare-commit-msg.sample
```

Hook-Skripte können sowohl client- als auch serverseitig aktiviert werden. Um ein bestimmtes Skript zu aktivieren, muss lediglich das *.sample* aus dem Dateinamen entfernt werden.

Für M.s Problem bietet sich die Aktivierung des *update-Hook* im zentralen Repository an. Dieser Hook wird immer dann aktiviert, wenn ein Entwickler versucht, seine lokalen Änderungen mit *git push* in das zentrale Repository einzuchecken. M. (dessen *bash*-Kenntnisse in etwa so rudimentär sind wie die des Autors) investiert einige Minuten und schreibt ein entsprechendes *bash*-Skript, das die Commit Message auf Kraftausdrücke (aktuell die Ausdrücke *shit*, *damn*, *ugly* und *crap*) prüft:

```
#!/bin/bash
commit_msg=$(git log --pretty=%s $2..$3)
#list of expressions to deny
for string in shit damn ugly crap
do
 if [[ $commit_msg == *$string* ]]
  then
    echo "Keine Kraftausdrücke!!";
    exit 1;
  fi
done
exit 0;
```

Das Update-Hook wird von Git mit folgenden Parametern aufgerufen: *refname sha1-old sha1-new*. Interessant für M. sind die Parameter *sha1-old* und *sha1-new*, was den Hash-Werten des Commit-Objekts vor- bzw. nach dem Commit entspricht. Über folgende Expression kann M. die Commit Message auslesen:

```
commit_msg=$(git log --pretty=%s $2..$3)
```

Git akzeptiert einen Push nur dann, wenn das Update-Skript einen Wert zurückliefert, der nicht 1 ist. Zuletzt benennt M. das Updateskript von *update.sample* in *update* um, um das Skript zu aktivieren. Um das Skript zu testen, erstellt M. einen Dummy-Commit mit folgender Commit Message „A crappy Dummy Commit“ und versucht diesen anschließend zu pushen (Listing 3). Das Skript scheint zu funktionieren, das zentrale Repository verbietet den Commit. M. ist stolz, denn damit ist ein weiterer Schritt in Richtung „sauberer“ Arbeit getan. Vielleicht noch ein kleiner Hinweis: Git Hooks eignen sich auch ideal, um lokale Entwicklungsrichtlinien zu erzwingen. Der Phantasie eines Teams sind hier keine Grenzen gesetzt, beispielsweise könnte ein kompletter Build des Projekts angestoßen werden und ein Commit nur erlaubt, wenn dieser erfolgreich war – eine Art minimalistisches Continuous-Integration-System. Ein anderes interessantes Beispiel wäre die automatische Verlinkung von Commits mit Jira Issues [3].

17:00 – Auf dem Weg zum Hauptbahnhof, Nürnberg

M. befindet sich jetzt auf dem Weg in den verdienten Feierabend. Zuvor hat er jedoch noch die Fahrt nach München vor sich. Hierfür hat sich M. einen weiteren wichtigen Produktions-Bug mitgenommen, der unbedingt morgen in das Produktivsystem geliefert werden muss. Als sich der Zug in Bewegung setzt, ist M. bereits dabei, den Bug zu fixen.

Noch ein letzter Hinweis des Autors: Die Repositories von allen beteiligten Teammitgliedern befinden sich auf der beiliegenden CD, sodass die einzelnen Schritte sehr einfach nachvollzogen werden können. Ich hoffe, das Lesen des Artikels hat Ihnen genauso viel Freude gemacht, wie mir das Schreiben und dass der Artikel Ihren Erwartungen entspricht. Ich freue mich über Fragen, Anregungen und konstruktive Kritik.

Martin Dilger ist Senior Consultant bei der PENTASYS AG in München und beschäftigt sich intensiv mit den Themen Git, Wicket, OSGi, Spring und Scala. Er bloggt regelmäßig unter http://splitshade.wordpress.com.

Links & Literatur

[1] http://kernel.org/pub/software/scm/git/docs/git-rebase.html

[2] http://andrewvos.com/2011/02/21/amount-of-profanity-in-git-commit-messages-per-programming-language

[3] https://github.com/joyjit/git-jira-hook