Einfach branchen und mergen

Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung

Niels Stargardt

Verteilte Versionsverwaltung ist ein aktueller Trend und löst viele Probleme der bisher bevorzugten zentralen Versionsverwaltungssysteme. Damit stellt sich die Frage, ob sich ein Umstieg auf eine verteilte Versionsverwaltung lohnt. Der Artikel beleuchtet die Stärken und Schwächen der beiden Herangehensweisen und vergleicht die verteilten Versionsverwaltungssysteme Bazaar und git.

Für die Versionsverwaltung – eine der Grundlagen professioneller Softwareentwicklung – gibt es zahlreiche Produkte. Der Fokus dieses Artikels liegt auf Open-Source-Produkten und deren Nutzung in kommerziellen Java-Projekten eines mittelständischen Unternehmens. Dieser Text stützt sich auf Erfahrungen in kommerziellen Projekten mit CVS, Subversion (SVN) und Bazaar (bzr). Ferner beruht die Einschätzung von verteilten Versionsverwaltungssystemen auf zahlreichen Artikeln und Blogs über bzr, git und Mercurial (hg) sowie auf Erfahrungen mit bzr und git in Open-Source-Projekten.

Der Artikel geht davon aus, dass dem Leser das Arbeiten mit einem zentralen Versionsverwaltungssystem wie CVS oder SVN vertraut ist. Es werden die sprachlich nicht sauberen, aber in der Praxis verbreiteten Begriffe wie *commiten*, *mergen* usw. genutzt.

Nachteile von zentralen Versionsverwaltungssystemen

Der Wechsel von einem bewährten zentralen System wie SVN zu einem anderen lohnt sich nur, wenn das Altsystem Nachteile hat oder ein neues System große Vorteile verspricht. Zentrale Systeme haben alle folgende wesentliche Einschränkungen:

- *Skalierung*: Zentrale Versionsverwaltungssysteme skalieren sehr schlecht. Überschreitet ein Team eine gewisse Größe, gibt es Probleme, wenn zu viele Personen gleichzeitig ihre Arbeit zurückstellen wollen. Wenn vor dem Einstellen des eigenen Standes zunächst der Stand des Servers zu holen und ein Minimal-Test durchzuführen ist, wird der Zeitraum für das vollständige Zurückstellen verlängert. Die Wahrscheinlichkeit steigt, dass es zwischenzeitlich weitere Änderungen gibt und der lokale Stand erneut aktualisiert und getestet werden muss.
- *Umfang der Änderungen beim Commit*: Vor einigen Jahren war es übliche Praxis, dass Code commited wurde, sobald er fehlerfrei kompilierbar war. Inzwischen commiten viele Entwickler erst, wenn ein Feature im Wesentlichen abgeschlossen, implementiert und getestet ist. Dies führt zu relativ langen Entwicklungszeiten, in denen keine Zwischenstände archiviert werden. Es entstehen sehr umfangreiche Änderungen (changesets), die schwer zu reviewen sind.
- *Teilteams*: Häufig werden größere Probleme in Teilteams von zwei bis drei Entwicklern gelöst. Die Mitglieder eines Teilteams können Code nur schwer untereinander austauschen, denn die Zwischenstände sind oft nicht reif für den zentralen Server.
- *Sicherstellung eines getesteten Standes*: Der aktuelle Stand des Repositoriums sollte immer getestet sein, sodass aus diesem problemlos aktualisiert werden kann. Dies ist insbesondere bei großen Teams notwendig. Zentrale Versionsverwaltungssysteme haben nur einen „Head“-Stand, in den entweder ungetestete Stände eingestellt werden oder Stände, die vor dem Commit alle Unit-Tests durchlaufen haben. Letzteres ist im Allgemeinen nicht praktikabel.
- *Netzwerk*: Da eine Netzwerkverbindung notwendig ist, wird mobiles Arbeiten erschwert.

All diese Probleme werden von verteilten Versionsverwaltungssystemen adressiert. Wesentlicher Lösungsansatz ist dabei der vermehrte Einsatz von Entwicklungszweigen, sogenannten Branches. Dies ist theoretisch auch schon mit CVS oder SVN möglich, wird in der Praxis aber selten genutzt, weil bei CVS und SVN Branches schwer zu verwalten sind. Insbesondere das Zusammenführen von Branches, das sogenannte Mergen, ist zeitaufwendig.

Verteilte Versionsverwaltung

Bei verteilten Versionsverwaltungssystemen (DVCS – Distributed Version Control System) erhält jeder Entwickler ein lokales Repositorium. Die Anzahl der Repositorien ist praktisch nicht beschränkt. Es gibt auch keinen fest definierten Fluss von Informationen über einen zentralen Server. Technisch sind alle Repositorien gleichwertig. Einige Autoren präferieren daher den Begriff „Peer-to-Peer-Versionsmanagement“. Das Arbeiten mit Branches ist der Normalfall und das Anlegen von Branches sowie insbesondere das Mergen sind deutlich einfacher.

In der Praxis wird bei verteilten Versionsverwaltungssystemen organisatorisch eine Hierarchie zwischen den Repositorien angelegt. Üblicherweise erfolgt der Austausch über ein oder mehrere zentrale Repositorien. Anders als bei zentralen Versionsverwaltungssystemen haben Teilteams hier jeweils ein eigenes zentrales Repositorium.

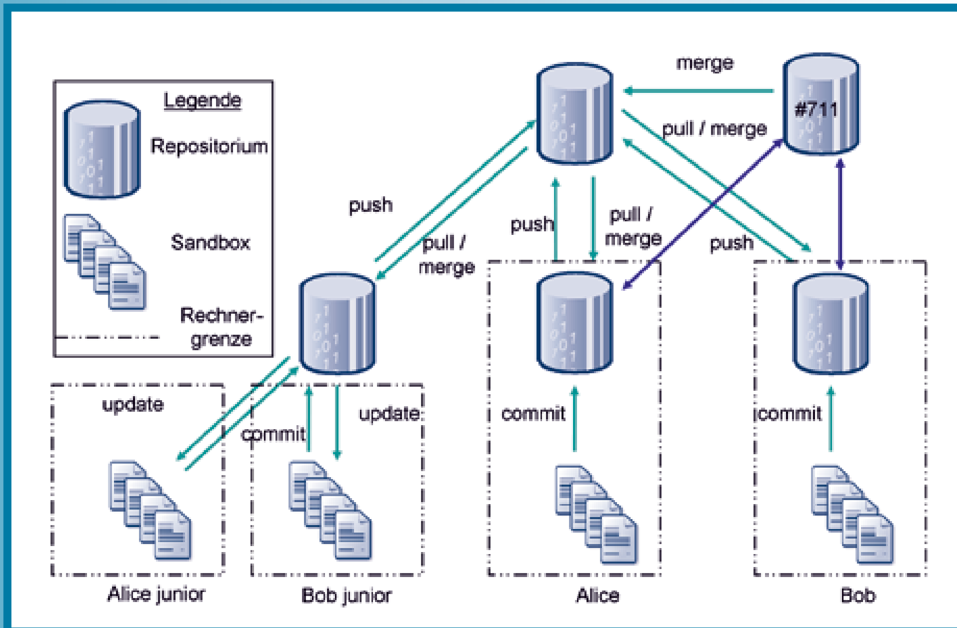

Abb. 1: Mögliche Workflows beim Arbeiten mit Bazaar

Die Arbeitsweise mit einem DVCS ist analog zu der mit einem zentralen System. Neben den gewohnten Befehlen `commit` und `update` werden für die tägliche Arbeit die Befehle `pull/merge` und `push` benötigt. Mit `pull` beziehungsweise `merge` wird das lokale Repositorium aktualisiert. Mit dem `push`-Befehl aktualisiert man das zentrale Repositorium mit den Revisionsinformationen aus dem lokalen Repositorium. Aktualisieren bedeutet, dass alle Commit-Informationen ausgetauscht werden und nicht nur der letzte Code-Stand abgeglichen wird.

In Abbildung 1 sind mögliche Workflows beim Arbeiten mit dem DVCS Bazaar dargestellt. Alice junior und Bob junior arbeiten gemeinsam auf einem zentral abgelegten Repositorium. Ihre Arbeitsweise entspricht der von SVN. Die erfahrenen Entwickler Alice und Bob haben lokal ihr eigenes Repositorium. Sie arbeiten auch durchaus einmal zusammen auf einem speziellen Repositorium, welches Ticket 711 behandelt. Sowohl die Änderungen aus dem Repositorium von Alice junior und Bob junior als auch die von Alice und Bob sowie die Änderungen des Feature-Branches #711 fließen in ein Hauptrepositorium ein. Denkbar wäre, dass aus diesem Hauptrepositorium von einem Build-Server alle Änderungen in ein „Clean“-Repositorium gepusht werden, sobald alle Tests erfolgreich durchlaufen sind.

Behebung der Nachteile zentraler Systeme

Verteilte Versionsverwaltungssysteme beheben die Nachteile zentraler Systeme und bieten folgende Lösungen:

- *Skalierung*: Das Problem der Skalierung wird bei verteilten Versionsverwaltungssystemen auf zwei Arten gelöst: Zum einen muss das zentrale Repositorium seltener aktualisiert werden, zum anderen ist es möglich, die Arbeit von Teilteams in einem jeweils eigenen Repositorium zu sammeln und das zentrale Repositorium nur aus diesen Teil-Repositorien zu aktualisieren (Integration Manager Workflow).
- *Umfang der Änderungen beim Commit*: Bei verteilter Versionsverwaltung kann zunächst lokal commited werden. An den einzelnen Commit müssen keine besonderen Anforderungen gestellt werden.
- *Teilteams:* Da jedes Teilteam ein eigenes Repositorium nutzen kann, ist der Austausch problemlos möglich.
- *Sicherstellung eines getesteten Standes:* Ein Continuously-Build-Server kann, nachdem alle Tests erfolgreich durchlaufen wurden, ein spezielles („Clean“-) Repositorium aktualisieren (Gatekeeper-Workflow). Die Entwickler aktualisieren nur ein „Input“-Repositorium.
- *Netzwerk:* Verteilte Systeme ermöglichen Workflows, bei denen nur zum Abgleich ein Netzwerk benötigt wird.

Weitere Vorteile der verteilten Systeme

Verteilte Systeme haben technische Vorteile, da das Netzwerk entlastet wird und aufgrund der Netzwerkunabhängigkeit auch eine bessere Performance erzielt wird. Die Anforderungen an die technische Administration sinken, da es nicht mehr einen Single-Point-Of-Failure gibt. Doch wesentlich wichtiger sind die erweiterten Möglichkeiten bei den Arbeitstechniken, die zahlreiche neue Workflows ermöglichen. Eine gute Übersicht darüber bietet die Dokumentation von Bazaar [BzrDoku].

Besonders hervorzuheben ist die Qualitätssteigerung, die durch eine besser lesbare Änderungshistorie und durch die bessere Unterstützung von Code-Reviews erreicht wird. Code-Reviews werden zum Beispiel durch Feature-Branches vereinfacht. Diese findet man in vielen Projekten mit verteilten Versionsverwaltungssystemen. Bei einem Feature-Branch wird für jedes Feature ein eigener Branch angelegt. Das Feature wird zunächst auf diesem Branch zu Ende entwickelt und dann in den Haupt-Branch eingepflegt. So kann das Feature ohne Seiteneffekte entwickelt werden und ein Code-Review ist erheblich einfacher. Es ist auch leicht zu erkennen, für welches Feature die Code-Änderungen vollzogen wurden.

Ein nicht zu vernachlässigender Aspekt ist die Zufriedenheit der Entwickler. Bei der Retrospektive eines Projektes mit Kollegen, die bislang nur Erfahrungen mit CVS/SVN hatten, zeigte sich: Alle Entwickler waren mit Bazaar zufrieden und wünschten den erneuten Einsatz im nächsten Projekt.

Nachteile der verteilten Systeme

Verteilte Versionsverwaltungssysteme haben folgende Nachteile:

- *Gesteigerte Komplexität*: Es gibt mehr Befehle und somit steigt die Komplexität der Bedienung.
- *Höhere Anforderungen an das Team*: Da es mehr Freiheiten bei der Arbeitsweise gibt, erfordert es Disziplin im Team, um die Möglichkeiten sinnvoll zu nutzen. Es muss ein Workflow im Projekt etabliert werden.
- *Umgang mit nicht zu mergenden Dateien*: Einige Dateien sind nur bedingt zu mergen, zum Beispiel Binär-Dateien oder komplexe XML-Dateien. Die Probleme mit solchen Dateien sind

bei verteilten Systemen größer, da es keinen Lock-Mechanismus gibt und Merge-Konflikte oft erst später bemerkt werden.
- *Geringere Verbreitung*: Verteilte Versionsverwaltungssysteme sind weniger verbreitet als zentrale. Dies macht sich in der Unterstützung durch Tools und bei der Anzahl an Entwicklern, die mit verteilter Versionsverwaltung vertraut sind, negativ bemerkbar.

Je nach Projektkonstellation kann einer dieser Nachteile ein Ausschlusskriterium zum Einsatz verteilter Systeme darstellen. Für das Problem der nicht zu mergenden Dateien gibt es allerdings Lösungsansätze oder zumindest Workarounds außerhalb der Versionsverwaltung. Zum Beispiel kann die Sandbox auf einem zentralen Laufwerk liegen. Wenn die Netzwerk-unabhängigkeit aufgegeben wird, ist es ferner möglich, ein verteiltes System (wie Bazaar oder git) um eine Lock-Möglichkeit zu erweitern. In den meisten Fällen lohnt es sich jedoch, auf Lock-Mechanismen zu verzichten, da diese ein Projekt leicht blockieren. Im Einzelnen ist zu prüfen, ob bestimmte Daten nicht in einem anderen Format abgelegt werden können, sodass für diese Dateien ein Merge wieder möglich ist. Ist dies nicht möglich, hat es sich bewährt, die Pflegearbeit an den kritischen Dateien auf einige wenige Personen zu beschränken und so das Problem ebenfalls zu umgehen.

Zusammenfassend ist festzustellen, dass verteilte Versionsverwaltungssysteme auch in einem kommerziellen Umfeld im Allgemeinen mehr Vorteile als Nachteile bieten.

Produkte für eine verteilte Versionsverwaltung im Überblick

Verteilte Versionsverwaltungssysteme gibt es seit 1999 in Form des kommerziellen Bitkeepers. Es folgten die Open-Source-Produkte Monotone (April 2003), darcs (November 2004), Bazaar (März 2005), Mercurial und git (beide April 2005). 2006 sind das kommerzielle Plasticscm und das freie Fossil-SCM hinzugekommen, die beide über die reine Versionsverwaltung hinausgehen. Wirkliche Bedeutung haben bei den Open-Source-Produkten die Programme Bazaar [Bzr], git [git] und Mercurial [hg].

Bazaar wird von der Firma Canonical entwickelt. git und Mercurial kommen aus dem Umfeld der Linux-Kernel-Entwicklung. Daraus ergeben sich unterschiedliche Zielsetzungen für die Produkte. Für die Linux-Kernel-Entwicklung ist es wichtig, Patches schnell und effektiv verwalten zu können, da diese Arbeit von wenigen Personen vollzogen wird. Aspekte wie Plattformunabhängigkeit und leichte Erlernbarkeit spielen hingegen eine untergeordnete Rolle.

Bei der Entwicklung von Bazaar stand der Werkzeug-Gedanke im Vordergrund. Das Ziel war, bei größtmöglichem Komfort mehr Workflows zu ermöglichen. In der Konsequenz lief Bazaar von Anfang an auf allen Plattformen* und hatte eine einfache, für CVS/SVN-Benutzer vertraute Schnittstelle. Ressourcenverbrauch und Geschwindigkeit standen nicht im Vordergrund.

Dieser historische Hintergrund ist wichtig, um bestimmte Eigenschaften der Produkte zu verstehen. Insbesondere erklärt er, warum in den älteren Vergleichen im Internet git hinsichtlich Usability und Bazaar hinsichtlich Performance so schlecht abschneiden. Insgesamt sind alle Produkte inzwischen ähnlich leistungsfähig. Dies betrifft sowohl die Bedienbarkeit als auch die Performance und den Funktionsumfang.

* Bazaar ist zum größten Teil in Python geschrieben.

git und Bazaar im Überblick

Im Folgenden werden die Produkte git und Bazaar vorgestellt. Der Überblick beschränkt sich auf diese beiden Produkte, da dem Autor für Mercurial die praktischen Erfahrungen fehlen. Bazaar und git unterscheiden sich beim Ressourcenverbrauch nicht. Bei der Geschwindigkeit ist Bazaar hinreichend schnell. Weder in der Projektarbeit noch bei der Arbeit an Open-Source-Projekten ergaben sich Wartezeiten.

Vorteile von git

- git erlaubt es, wahlweise nur die letzten n Einträge von einem zentralen Repositorium zu holen.
- git hat mit github [github] eine der besten Community-Plattformen.
- git ermöglicht es, Attribute für einzelne Dateien zu definieren, zum Beispiel für CRLF-Konvertierung.
- git hat die größte Verbreitung und die beste Tool-Unterstützung. Es gibt Python- und Java-Implementierungen (mit eingeschränkter Funktionalität). Mit EGit gibt es ein offizielles Eclipse-Plug-in, welches von den Eclipse-Entwicklern selbst genutzt wird. Allerdings ist die Merge-Funktion in EGit deutlich schlechter als die von git.

Nachteile von git

- git ist schwierig zu erlernen. Hierfür sollte großzügig Zeit eingeplant werden.
- Es gibt keine offizielle Windows-Version.
- Revisionen werden durch Hashwerte statt durch einfache Versionsnummern gekennzeichnet. Hashwerte lassen sich weder leicht merken oder kommunizieren noch ergibt sich daraus automatisch eine Reihenfolge.

Vorteile von Bazaar

- Bazaar ist in einer Stunde zu erlernen. In der täglichen Arbeit muss gelegentlich noch etwas nachgeschlagen werden.
- Bazaar kann auch wie SVN benutzt werden, das heißt jeder Commit erfolgt in das zentrale Repositorium. Dies ist insbesondere für Anfänger einfacher.
- Bazaar hat eine gute grafische Oberfläche integriert und läuft auf allen Plattformen.
- Bazaar unterstützt echte Moves.**
- Bazaar verwendet einfache Versionsnummern, die aufeinander aufbauen.

Nachteile von Bazaar

- Die Tool-Unterstützung ist eingeschränkt. Insbesondere gibt es nur eine rudimentäre Eclipse-Unterstützung, die nicht annähernd an die des CVS-Plug-ins heranreicht.
- Bazaar hat eine geringere Verbreitung.
- Die Einstellung zur Konvertierung des Zeilenumbruchs (CRLF vs. LF) muss auf jedem Rechner definiert werden.

Vergleich der Produkte: Umbenennen und Verschieben

Das Umbenennen und Verschieben von Dateien und Verzeichnissen sind in Java-Projekten typische Refactoring-Aufgaben,

** Im Vergleich zu Mercurial, git oder auch Subversion kennt Bazaar ein echtes Move. Bei allen anderen gibt es ein Copy (mit Historie) und ein Remove. Der Nachteil ist, dass man eine Datei nicht mit der Historie kopieren kann. Der Vorteil von einem echten Move ist ein besseres Verhalten beim Mergen, aber auch in der Historie. Siehe dazu auch [BzrMv] und [Bliz11].

die ein Merge erschweren. Diese Funktionalität soll deshalb detailliert betrachtet werden.

Bazaar und git gehen die Herausforderung auf fundamental unterschiedliche Weise an. Zum Verständnis muss zunächst das Datenmodell von git grob erläutert werden. git verwaltet Inhalte in „Blobs“, die in Form von „Trees“ organisiert sind, wobei jedes „Commit“-Objekt auf ein spezielles „Tree“-Objekt verweist. git speichert somit immer den kompletten Inhalt. Im Gegensatz dazu verfolgt Bazaar den klassischen Ansatz und speichert Veränderungen von Objekten. Für Bazaar sind sowohl Dateien als auch Verzeichnisse Objekte, die eine eindeutige ID bekommen.

Da git keine Veränderungen speichert, sondern immer den gesamten Inhalt, kann es eine Umbenennung nicht als solche ablegen. git löst dieses Problem mit einer „Rename-Detection“, welche davon ausgeht, dass Dateien umbenannt wurden, wenn der Inhalt zu 90 Prozent übereinstimmt. Dieses Vorgehen ist bequem, hat aber den Nachteil, dass das Verhalten nur schwer vorhersagbar ist. Im Gegensatz dazu merkt sich Bazaar explizit Umbenennungen – auch auf Verzeichnisebene – und hat dadurch mehr Informationen, die es beim Merge auswerten kann.

Zur Veranschaulichung ein einfaches Beispiel mit Autos: Jedes Fahrzeugmodell sei durch eine eigene Klasse repräsentiert, während Automarken durch Packages definiert sind. Das Projekt startet mit folgenden Dateien: `merzedes/AKlasse.java` und `seat/Sharan.java`. Ein Autokenner merkt schnell die Fehler und ändert die Benennung in einem eigenen Branch (s. Abb. 2) in `mercedes/AKlasse.java` und `vw/Sharan.java` (`seat` bleibt als leeres Verzeichnis zurück). git stellt beim Mergen fest, dass zwei Dateien „umbenannt“ wurden, zu beachten ist, dass der Verzeichnispfad bei git als Teil des Dateinamens gesehen wird. Bazaar merkt sich beim Commit, dass ein Verzeichnis umbenannt und eine Datei verschoben wurde. In der Folge wird Bazaar bei einem Merge alle Klassen, die zwischenzeitlich in dem Haupt-Branch im Verzeichnis `merzedes` angelegt wurden, ebenfalls verschieben. Bei git würden diese Klassen im Verzeichnis `merzedes` verbleiben. Eine im Verzeichnis `seat` angelegte Datei `Ibiza.java` bleibt bei einem Merge mit beiden Systemen unverändert.

Als Ergebnis bleibt festzuhalten, dass die Probleme des Umbenennens von beiden Systemen auf unterschiedliche Arten gelöst werden. Der prinzipiell aufwendigere Ansatz von Bazaar sorgt für ein besseres Verhalten beim Merge.

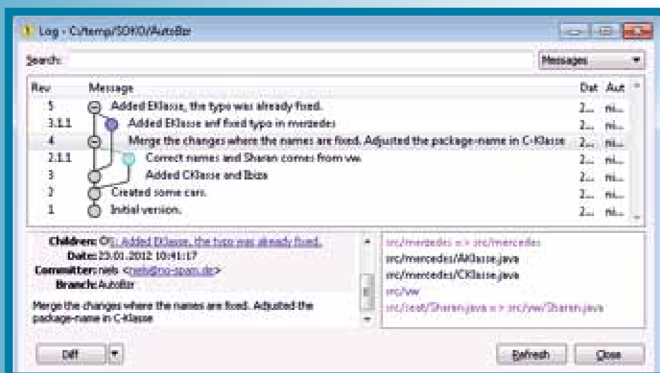

Abb. 2: Protokollierung in Bazaar

Fazit

Verteilte Versionsverwaltungssysteme eröffnen ganz neue Arbeitsweisen. Sie bieten gegenüber zentralen Systemen klare Vorteile, haben allerdings den Nachteil der geringeren Verbreitung. Alle betrachteten Produkte haben sich jedoch bereits in größeren Projekten bewährt und können als zuverlässig angesehen werden. Zudem werden die Produkte aktiv weiterentwickelt. Es gilt also, für jedes Projekt im Einzelnen zu prüfen, ob und auf welches Produkt sich ein Umstieg lohnt.

Bazaar empfiehlt sich vor allem aufgrund des geringen Einarbeitungsaufwands und der benutzerfreundlichen Schnittstelle. Es bietet die beste Unterstützung beim Verschieben und Umbenennen von Klassen oder Packages. git punktet mit der besseren Eclipse-Unterstützung und weiteren Verbreitung.

Für die Zukunft ist zu erwarten, dass verteilte Systeme verstärkt eingesetzt werden. Zu hoffen ist, dass die Hersteller von Entwicklungstools im Java-Umfeld Bazaar stärker berücksichtigen, sodass die Tool-Unterstützung noch besser wird.

Links

[Bliz11] M. Blizinzki, Directory renaming in SCM, Juni 2011, `http://automatthias.wordpress.com/2007/06/07/directory-renaming-in-scm/`
[Bzr] Bazaar Homepage, `http://bazaar.canonical.com`
[BzrDoku] Workflows, Bazaar User Guide, Juni 2011, `http://doc.bazaar.canonical.com/bzr.2.3/en/user-guide/bazaar_workflows.html`
[BzrMv] Rename Tracking and Smart Merging, Juni 2011, `http://doc.bazaar.canonical.com/migration/en/why-switch-to-bazaar.html#rename-tracking-and-smart-merging`
[git] git Homepage, `http://git-scm.com/`
[github] Github Community-Plattform, `https://github.com/`
[hg] Mercurial Homepage mit Tortoisehg, `http://mercurial.selenic.com/`

Weiterführende Information

http://bazaar.canonical.com

Niels Stargardt ist seit 14 Jahren bei der PPI AG Informationstechnologie als Projektleiter und Softwarearchitekt tätig. Er kümmert sich im Unternehmen um verschiedene Themen im Umfeld von Java und der Optimierung von Entwicklungsprojekten.
E-Mail: niels.stargardt@ppi.de