

Halil-Cem Gürsoy

Über den Status quo verteilter Versionsverwaltungssysteme

Gratwanderung zwischen Flexibilität und Chaos



Verteilte Versionsverwaltungssysteme (DVCS, Distributed Version Control System) kommen zunehmend in Mode. Begnügten sich früher Entwickler noch mit einem zentralen CVS-Server, müssen es heute schon Git, Mercurial oder ein anderes verteiltes Versionsverwaltungssystem sein.

Der Autor kam das erste Mal mit einem Versionsverwaltungssystem in Form von PVCS in Berührung. Das war ein heute von Serena entwickeltes

System, das früher unter einem "miteinander entwickeln" eher ein "wer zuerst kommt, mahlt zuerst" verstand. Jeder Nutzer musste eine Datei erst exklusiv sperren, bevor er sie verändern durfte. Der weitere Weg ging über MKS (ähnliches Sperrverfahren) und Perforce zu CVS. Die letzten beiden Produkte waren damals eine Offenbarung: Nicht nur, dass sich mit ihnen "konkurrierend" arbeiten ließ, erstmals gab es auch Branches, also Entwicklungszweige. Das Mergen, das heißt das Zusammenführen solcher Entwicklungszweige stellte allerdings eine große Herausforderung dar, wenn mehrfach aus einem Branch in den Hauptentwicklungszweig zu "mergen" war.

Zudem führten die nicht atomaren Commits von CVS zu mancher Überstunde. Denn verabschiedete sich der CVS-Prozess auf dem Server mitten im Commit, war mehr oder weniger mühselig herauszufinden, welche Änderungen der Server aufgenommen hatte und welche nicht.

Mit Subversion bekamen Entwickler und Projektteams endlich ein Handwerkzeug, mit dem sich hinsichtlich der (endlich atomaren) Commits tatsächlich sicher arbeiten ließ. Von Version zu Version wurde die Unterstützung für Merges zwar etwas besser, sie stellt aber immer noch eine große Herausforderung dar: Welche Revisionen wurden bereits zusammengeführt? Was ist mit der Historie aus den Branches? Die gehen gerne verloren, zumal nach einem "Reintegrate" ein Branch stillgelegt wird.

Aber das Mergen ist nicht der einzige Stolperstein, mit dem ein Entwickler in Subversion zu kämpfen hat. Zu nennen wären etwa die External, das sind Verweise auf andere Bereiche im gleichen oder in einem anderen Repository: ein Tag, also eine Marke, die einen bestimmten Zustand markiert, auch über External zu erfassen ist einfach nicht möglich. Dadurch wird es schwierig, einen bestimmten Zustand zum Beispiel nach einem Release nachvollziehbar zu kennzeichnen.

Die Liste ließe sich fast beliebig fortführen. Diesen Systemen ist gemeinsam, dass sie einen zentralen Server aufweisen. Alle Informationen sind dort abgelegt, die Benutzer haben nur eine Arbeitskopie ohne weitergehende Informationen zu der Historie der versionierten Artefakte. Hat ein Benutzer eine Änderung vorgenommen, muss er sie auf den zentralen Server übermitteln, und die anderen Benutzer müssen eine Aktualisierung durchführen.

Aufbruch in eine neue Zeit

Seit dem Wechsel des Linux-Kernel-Projekts um Linus Torvalds auf das eigen entwickelte Git steigt die Sichtbarkeit verteilter Versionsverwaltungssysteme und natürlich von Git selbst. Die Entscheidung, ein eigenes Versionsverwaltungssystem für das Kernel-Projekt zu entwickeln, fiel 2005, als die Lizenz für das bis dahin im Kernel-Projekt eingesetzte kommerzielle System **BitKeeper[1]** geändert wurde. Anscheinend unterschieden sich die Anforderungen von Torvalds und dessen Umfeld so sehr von der Arbeitsweise von Subversion, dass Torvalds in der Mailingliste zum Linux-Kernel jegliche Diskussion um die Einführung von Subversion mit dem lapidaren Satz "Don't bother telling me about subversion" **zu unterbinden versuchte[2]**. Erfolgreich, wie die Geschichte gezeigt hat.

Sein Hauptaugenmerk galt dem verteilten, unabhängigen Arbeiten und der Effizienz. Eine klare Gliederung nach "Changesets" sollte her. Das heißt, Torvalds wünschte sich ein System, das ihm als Kernel-Maintainer das Leben vereinfachen sollte: Aus der großen Anzahl an Änderungen, die in den Kernel-Quelltexten stattfinden, kann und will er nur ausgewählte in ein kommendes Kernel-Release aufnehmen. Eine Aufgabenstellung, die zugegeben in einem "normalen Projekt" nicht immer aufkommt.

Getrieben durch eben diese Lizenzänderung wurden weitere Entwicklungen angestoßen: Etwa zur gleichen Zeit entstand das System Mercurial, das Matt Macall ebenfalls in der Kernel-Mailingliste **ankündigte[3]** und unter der GPL steht. Die Entwicklung von Mercurial schritt schnell voran und in der besagten Mailing-Liste konnte Torvalds es sich nicht verkneifen, darauf auf seine bissig süffisante Art **zu reagieren[4]**, insbesondere dann, wenn sich Macall erdreistete, Vergleiche zu Git zu ziehen.

Daneben fand die Entwicklung von Bazaar (siehe Kasten) statt, das auf die gleichen Konzepte wie Git und Mercurial setzt. Der Artikel konzentriert sich im Wesentlichen auf Mercurial und Git. Bazaar spielt eine eher untergeordnete Rolle und kommt in der "freien Wildbahn" selten vor.

Bazaar

Das unter der GPLv2 stehende **Bazaar[5]** entstand aus der gleichen Motivation heraus wie Mercurial und Git und steht unter der GPLv2-Lizenz. Größter Förderer ist **Canonical Ltd.[6]**, Hauptsponsor der bekannten Linux-Distribution Ubuntu. Aus dem Kontext heraus bezieht Bazaar seine größte Verbreitung, da die Entwicklung von Ubuntu sich auf Bazaar stützt. Auch Launchpad, die Plattform um Entwicklungen rund um Ubuntu, setzt Bazaar ein. Aber das System ist nicht nur auf Linux und Ubuntu beschränkt, es finden sich Installationspakete für andere gängige Betriebssysteme, neben Linux-Derivaten und Windows auch für Solaris, BSD und andere.

Die Konzepte von Bazaar unterscheiden sich nicht wesentlich von Mercurial oder Git, auch die Befehle sind sich recht ähnlich. Eine Integration in eine bestehende Subversion-Landschaft ist ohne größeren Aufwand möglich. Ein **Eclipse-Plug-in[7]** existiert ebenfalls und lässt sich über die **Update-Site[8]** installieren.

Verteilte Welt

Allen drei Systemen ist gleich, dass sie keinen zentralen Server benötigen, sondern nach dem Peer-to-Peer-Konzept arbeiten. Das bedeutet, dass auf jedem Peer ein Repository existiert, das eine Teilmenge aller Entwicklungszweige enthält. Es gibt aber nicht zwingend ein Repository, in dem alle Änderungen vorhanden sind. Jeder Benutzer entscheidet selbst, welche Änderungen er in sein Repository aufnimmt. In jedem Repository ist die Änderungshistorie der Artefakte enthalten, sodass hierdurch inhärent Backups vorliegen, vorausgesetzt, die Änderungen wurden bereits verteilt.

Das erscheint einem Betrachter, der aus der "alten Welt" der klassischen Versionsverwaltungssysteme kommt, recht chaotisch. Aber das Konzept ist weit von einem Chaos entfernt, denn alle Vorgänge sind nachvollziehbar.

Ein klarer Vorteil zeichnet sich ab, wenn es um die Geschwindigkeit und Arbeitsweise geht: Ein Entwickler ist unabhängig von einer Netzverbindung, sobald er ein lokales Repository hat. Auch in Zeiten von UMTS und fast flächendeckender Breitbandanbindung kommt es häufig genug vor, dass keine Netzverbindung besteht. Zwar ist auch ein Subversion-Nutzer nicht immer auf eine Netzverbindung angewiesen, aber sobald es darum geht, sich die Historie einer Datei im Detail anzuschauen, gestaltet sich die Arbeit mit dem Werkzeug schwierig. Ein Commit schließt sich damit verständlicherweise ganz aus. Nun mögen Kritiker anmerken, dass ein Commit in einem DVCS nicht ganz gleichzusetzen ist mit einem Commit im Kontext eines Versionskontrollsystems mit zentralem Server. Der Einwand ist nicht unberechtigt, da die Daten letztlich immer noch "nur" lokal vorliegen und zu verteilen sind, zudem existiert dadurch kein Backup der Daten. Aber es findet durch den Einsatz von Peer-to-Peer-Konzepten eine sinnvolle Entkoppelung statt.

Vollig losgelöst

Wie erfolgt nun die Synchronisation der Änderungen unter den Entwicklern, da verteilte Versionsverwaltungssysteme keinen zentralen Server benötigen? Das erfolgt nach dem Pull-und-Push-Prinzip. Ein Benutzer kann seine Änderungen, die er über ein Commit aus der Arbeitskopie in sein lokales Repository aufgenommen hat, in ein anderes Repository über einen Push übermitteln. Möchte der Benutzer Änderungen von einem anderen in sein lokales Repository ziehen, führt er einen "Pull" aus.

Möchte ein Projekt partout ein zentrales Repository einsetzen, steht diesem trotz allem "Verteilen" nichts entgegen: Sowohl Git als auch Mercurial bieten Funktionen an, einen eigenen Webserver zu starten oder eine Webserver-Installation zu nutzen. Ein schönes Beispiel ist hierfür der Server des **Linux-Kernel-Projekts**[9], aus dem heraus sich Torvalds die Snapshots für den nächsten Kernel in sein lokales Repository holt. Auch lassen sich Protokolle wie ssh für den Austausch der Daten einsetzen. Um das Ganze abzurunden, können Entwickler über E-Mails Changesets austauschen.

Das dezentrale Vorgehen führt dazu, dass fortlaufende Revisionsnummern wie bei Subversion nicht ausreichen. Git verwendet daher als Revisionsnummer einen SHA1-Hash, Mercurial eine Kombination aus einer lokalen und einer globalen Revisionsnummer, wobei Letztere ebenfalls ein SHA1-Hash ist.

Auf Basis der Revisionsnummern und des Verfolgens aller Branches und Merges können sowohl Mercurial als auch Git feststellen, welche Änderungen bereits von einem Branch in einen anderen oder den Hauptentwicklungszweig überführt wurden. Mehrfache Merges aus

einem Branch heraus sind dadurch fast zu einem Kinderspiel geworden. Konflikte, wenn vorhanden, sind natürlich dennoch zu beheben. Aber auch das "Zurückfallen" auf eine alte Version und das Erstellen von Patches oder das gezielte Auswählen dedizierter Änderungen aus einem Branch zur Aufnahme in den Hauptentwicklungszweig, das sogenannte "cherry picking", ist immer nur einen Kommandozeilenbefehl entfernt.

Schwere Entscheidung

Um die Arbeit mit einem DVCS beispielhaft im Artikel zu verdeutlichen, hat sich der Autor für Mercurial entschieden, wobei die Wahl zwischen Mercurial und Git schwerfällt. Mercurial ist mittlerweile vielleicht nicht so angesagt wie Git (siehe z.B. diese **Meldung auf heise Developer[10]**) und hat einen geringeren Nerd-Faktor, aber der Einstieg ist gerade für Umsteiger und Nicht-Nerds mit Mercurial wesentlich einfacher. Allein die schiere Befehlsflut von Git erschlägt einen armen Subversion-Nutzer (es müssten inzwischen um die 180 sein). In Mercurial ist die Anzahl der Basisbefehle überschaubar. Der Benutzer kommt dort mit den komplexeren, in Extensions gekapselten Funktionen nur bei Bedarf in Berührung. Viele Erweiterungen werden bereits mit Mercurial ausgeliefert und stehen sofort zur Verfügung, wie *gpg* für das Signieren der Changesets oder *patchbomb* für das Versenden von Changesets als Patch-E-Mails. Andere Extensions wiederum sind herunterzuladen, während ein Git-Benutzer, wie oben erwähnt, den vollen Befehlsumgang sofort zur Verfügung hat.

Alles in allem bieten beide Systeme ähnliche Funktionen und die Konzepte sind weitgehend deckungsgleich (siehe eine Übersicht unter **selenic.com[11]**), für den Autor ist Mercurial aber transparenter und verständlicher.

Technisch gibt es in den Tiefen der Systeme die einen oder anderen größeren Unterschiede, die aber ein normaler Benutzer in einem Projekt vermutlich überhaupt nicht wahrnehmen wird. Der Einstieg für einen Anwender, der aus der CVS-/Subversion-Welt kommt, geschieht in Mercurial einfach. Insbesondere ist die Mercurial-Dokumentation gut ausgearbeitet, während bei Git der Benutzer vielleicht auch mal zu einer Suchmaschine greifen muss.

Die Infrastruktur der beiden Systeme ist ebenfalls relativ ähnlich, wobei die Sichtbarkeit des Git-Ökosystems durch den derzeitigen Hype (oder besser: den Fanboyismus) größer ist. Da gibt es größere Hoster wie **GitHub[12]**, die das "soziale Coden" propagieren und unter bestimmten Bedingungen kostenlos Repositories hosten. Auch bietet inzwischen Google Code, viele meinen viel zu spät, neben einer Subversion- und Mercurial-Unterstützung auch die für Git an. Analog wird der Leser mit **Bitbucket[13]** für Mercurial fündig. Über solche Dienste liegen wieder zentrale Server vor, die als Dreh- und Angelpunkt zur Synchronisation der Änderungen dienen können.

Beide Systeme sind ausgereift und befinden sich teilweise in sehr großen Projekten im Einsatz. Erwähnenswert wären bei Git das Linux-Kernel-Projekt, Gnome oder auch Eclipse, dessen kommende Version, Juno, **mit Git versioniert wird[14]**. Mercurial steht dem nicht nach und kann ebenfalls auf namhafte Projekte verweisen. Besonders sichtbar sind die Plattformen Google Code und Microsofts CodePlex sowie OpenJDK und NetBeans.

Der Einstieg in Mercurial

Die Installation von Mercurial gestaltet sich einfach. Unter Windows stehen Installer für 32- und 64-bit zur Verfügung, für verschiedene Linux-Distributionen RPM- und Deb-Pakete; auch an die Benutzer von Mac OS X **wurde gedacht**[15]. Unter Ubuntu sollte der Leser, um möglichst die letzte stabile Version auf dem System zu haben, zuerst das Paket-Repository einbinden und anschließend das Mercurial-Paket installieren:

```
$ sudo add-apt-repository ppa:mercurial-ppa/releases
$ sudo apt-get install mercurial
```

Mercurial bedankt sich dann nach der Installation:

```
$ hg --version
Mercurial Distributed SCM (version 1.8)
```

Nebenbei bemerkt ist "Hg" das Symbol für Hydrargyrum, dem gemeinen deutschsprachigen Mitmenschen auch bekannt als Quecksilber (englisch Mercury).

Soll TortoiseHg ebenfalls installiert werden, stehen hierfür Windows-Binaries auf der Download-Seite des Projekts **zur Verfügung**[16]. Unter Ubuntu sollten hierzu erneut Repositories eingebunden werden. Dafür **stehen drei**[17] zur Verfügung, wobei der Autor *Releases* und *Stable Snapshots* verwendet hat.

```
sudo add-apt-repository ppa:tortoisehg-ppa/releases
sudo add-apt-repository ppa:tortoisehg-ppa/stable-snapshots
sudo apt-get install tortoisehg
sudo apt-get install tortoisehg-nautilus
```

Mit dem letzten Befehl werden die Erweiterungen für Nautilus, den Standard-Dateibrowser von Ubuntu, installiert. (Wichtiger Hinweis für Nautilus-User: Durch die Installation von TortoiseHg ist eine parallele Nutzung von RabbitVCS aufgrund einer Inkompatibilität **nicht möglich**[18]. Eine Deinstallation von TortoiseHg sollte RabbitVCS wieder zum Leben erwecken, ebenso der Umweg über Thunar für RabbitVCS, sodass in diesem analog zu TortoiseSVN im Kontextmenü des Dateibrowsers alle Befehle zur Verfügung stehen.)

Da hier mit der Kommandozeile gearbeitet wird, ist es sinnvoll, in einem ersten Schritt Mercurial den eigenen Benutzernamen mitzuteilen. Hierzu ist unter Windows die Datei *mercurial.ini* beziehungsweise unter Linux *.hgrc* zu bearbeiten:

```
[ui]
merge = meld
username = hcguersoy <guersoy@lendinarax>
```

Wie zu sehen, wurde eine zusätzliche Zeile für ein Merge-Tool, in diesem Fall das 3-Wege-Merge-Werkzeug von Gnome2 *meld*, eingetragen.

Um ein lokales Repository einzurichten, reicht ein Befehl aus:

```
$ hg init project1
```

Hiermit initialisiert man das neue Repository in dem Verzeichnis *project1*. Falls dieses nicht vorhanden war, legt Mercurial es an. In dem folgenden Beispiel erfolgt ein Wechseln in das

Projektverzeichnis, um anschließend eine Datei namens *datei1.txt* anzulegen. Abschließend wird sie mit einem "add" der Versionsverwaltung hinzugefügt:

```
$ cd project1
$ vi datei1.txt
$ hg add datei1.txt
```

Das ist bisher vergleichbar mit den Aktionen, die ein Nutzer unter CVS oder Subversion durchgeführt hätte. Die Datei *datei1.txt* ist zwar in der Arbeitskopie als zu versionieren markiert, aber tatsächlich noch nicht in das Repository aufgenommen. Das geschieht analog Subversion erst mit einem Commit.

```
hg commit -m "Datei hinzugefügt"
```

Wird beim Commit kein Kommentar angegeben, öffnet sich automatisch ein Editor, in dem der Benutzer einen Kommentar eingeben kann und sollte. Nun hat der Entwickler die Datei zu ändern oder noch weitere Dateien hinzuzufügen. Änderungen nimmt er, wie oben geschehen, mit einem Commit in das Repository auf. Da Mercurial über Transaktionen arbeitet, ist der Nutzer in der Lage, den letzten Commit (im Gegenteil zu einem Commit auf einer Datenbank) rückgängig zu machen:

```
$ hg rollback
repository tip rolled back to revision 1 (undo commit)
working directory now based on revision 1
```

Angeschaut sei nun die Historie der Datei:

```
$ hg log
Änderung:      1:f0f560a3ae27
Marke:         tip
Nutzer:        hcguersoy <guersoy@lendinarax>
Datum:         Fri Mar 11 10:35:13 2011 +0100
Zusammenfassung: Kleine Änderung vorgenommen

Änderung:      0:3ce8efc342ae
Nutzer:        hcguersoy <guersoy@lendinarax>
Datum:         Fri Mar 11 10:10:12 2011 +0100
Zusammenfassung: Datei hinzugefügt
```

Deutlich zu sehen ist, wie Mercurial die Revisionsnummer vergeben hat: eine fortlaufende lokale und eine zwölfstellige Repräsentation des SHA1-Hashes. Soll es den vollständigen Hash-Wert anzeigen, kann das mit *hg --debug log* erfolgen. Die lokale Revision 1 enthält als zusätzliche Information das Attribut *Marke*. Es handelt sich um nichts anderes als einen "Tag". Die Marke beziehungsweise der Tag "tip" liegt dabei immer auf der aktuellen Version. Der Hauptzweig hat übrigens den Namen "default" (vergleichbar mit "trunk" in Subversion).

Ein bedeutender Vorteil der verteilten Versionsverwaltungssysteme ist das einfache Branches und vor allem das Mergen der Änderungen zurück in den Hauptentwicklungszweig. Ein Branch wird angelegt mit dem Befehl *hg branch*:

```
$ hg branch proj1-branch1
Arbeitsverzeichnis wurde als Zweig proj1-branch1 markiert
```

Nimmt ein Entwickler nun Änderungen vor und legt diese vor, erfolgt das in dem Branch *proj1-branch1*:

```
$ hg log
Änderung:      3:913d42a61e38
Zweig:         proj1-branch1
Marke:         tip
Nutzer:        hcguersoy <guersoy@lendinarax>
Datum:         Fri Mar 11 11:21:11 2011 +0100
Zusammenfassung: Änderungen im Branch vorgenommen#
```

Sollen die Änderungen in den Hauptzweig integriert werden, geschieht das mit *hg merge* aus dem Hauptzweig heraus.

```
$ hg update default
1 Dateien aktualisiert, 0 Dateien zusammengeführt, 0 Dateien entfernt, 0 Da
$ hg merge proj1-branch1
1 Dateien aktualisiert, 0 Dateien zusammengeführt, 0 Dateien entfernt, 0 Da
(Zusammenführen von Zweigen, vergiss nicht 'hg commit' auszuführen)
```

Mercurial erkennt bei einem Update, ob ein Konflikt vorliegt. Konflikt bedeutet hier, dass gleiche Bereiche einer Datei parallel verändert wurden. Falls ein Merge-Tool, wie im Beispiel, in der Mercurial-Konfiguration hinterlegt wurde, öffnet Mercurial die lokale Datei und deren Remote-Version in diesem, und der Benutzer kann den Konflikt beheben.

Das erfolgt alles noch lokal, und es findet noch keine Berührung mit einer anderen Kopie des Repository statt. Eine Kopie oder, um in der Lingua Mercurial zu bleiben, ein Klon lässt sich mit einem Befehl erzeugen. Änderungen, die in dem Klon stattfinden, sind erst einmal unabhängig von den Änderungen, die in dem ursprünglichen Repository stattfinden. Wie erwähnt, erfolgt eine Synchronisation der Änderungen zwischen Repositories mit Push und Pull. Das sei etwas näher betrachtet. Hierzu ist ein lokaler Klon anzulegen:

```
$ hg clone project1 proj1-clon1
Aktualisiere auf Zweig default
1 Dateien aktualisiert, 0 Dateien zusammengeführt, 0 Dateien entfernt, 0 Da
```

Nach einem Wechsel in die Arbeitskopie des Klons, *proj1-clon1*, lassen sich nun Änderungen unabhängig von dem ursprünglichen Repository ausführen. Der Klon ließe sich aber auch auf einem anderen File-System legen und von einem anderen Benutzer bearbeiten.

Um kurzfristig Zugriff auf das lokale Repository zu gewährleisten, steht der Befehl *hg serve* zur Verfügung, mit dem man einen lokalen Webserver startet, wodurch ein anderer Benutzer einen Klon abrufen kann. Der Webserver ist in den Default-Einstellungen unter der Portnummer 8000 zu erreichen, zum Beispiel sollte *http://localhost:8000* den Zugriff auf das lokale Repository ermöglichen. Push ist im Übrigen per Default bei einem auf die Weise gestarteten Webserver ausgeschaltet, da keine Zugriffskontrolle möglich ist. Daher wäre für längerfristiges Bereitstellen eines Repository-

Mercurial-Repository zum Artikel

Damit der Leser sofort loslegen kann, hat der Autor bei Bitbucket ein Mercurial-Repository mit dem Namen "HalloMercurial" für Testzwecke eingerichtet. Ein Klonen des Repositorys erfolgt mit dem Befehl

```
hg clone
https://bitbucket.org/hcguers
```

Servers ein "richtiger" Webserver zu verwenden. Ausführliche Anleitungen hierzu finden sich in der Mercurial-Dokumentation.

Angenommen, ein Anwender hat Änderungen in dem Klon vorgenommen, lassen sich diese nun mit einem Pull abrufen (zuvor hat er mit *hg serve* in dem Klon den Webserver gestartet):

Falls der Leser ssh bevorzugt, geschieht das Klonen via

```
hg clone
ssh://hg@bitbucket.org/hcguer
```

Das Pullen und Pushen von Änderungen passiert wie im Artikel beschrieben mit *hg pull* und *hg push*. Die rege Teilnahme ist natürlich erwünscht.

```
$ hg pull http://localhost:8000
Hole von http://localhost:8000
Suche nach Änderungen
Füge Änderungssätze hinzu
Füge Manifeste hinzu
Füge Dateiänderungen hinzu
Fügte 1 Änderungssätze mit 1 Änderungen an 1 Dateien hinzu (+1 Köpfe)
("hg heads" zeigt alle Köpfe, nutze "hg merge" um sie zusammenzuführen)
```

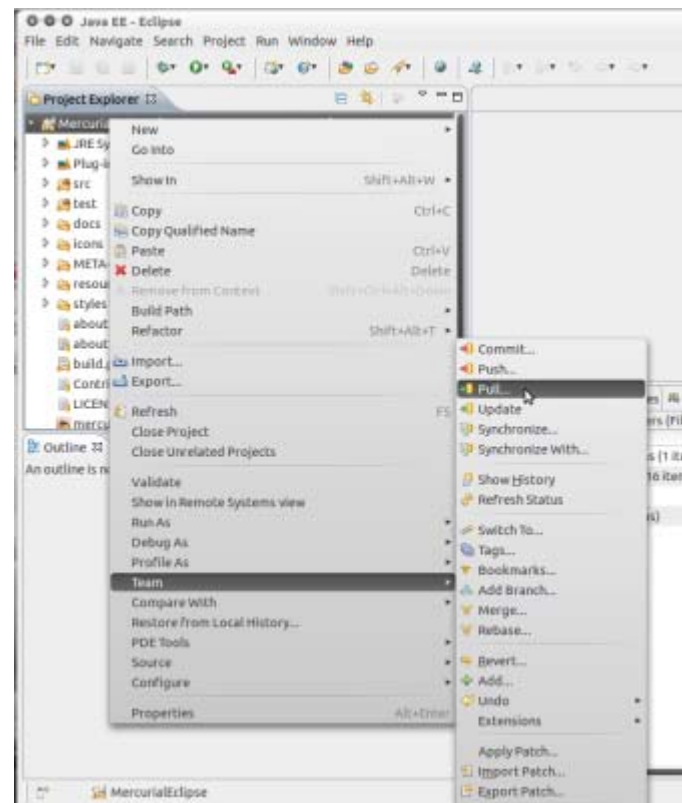
Die Änderungen sind allerdings noch nicht sichtbar, erst mit dem Befehl *hg merge* nimmt man sie in die aktuelle Arbeitskopie auf und bestätigt sie abschließend mit einem Commit. Dadurch ist ein Benutzer auch nach einer Synchronisation frei in seiner Entscheidung, ob Änderungen in die lokale Arbeitskopie einfließen oder auch nicht.

Das war ein kleiner Ausschnitt der Möglichkeiten, die Mercurial bietet, wobei vieles analog für Git gilt. Viele weitere Kernfunktionen erleichtern das Leben als Entwickler, insbesondere im Zusammenspiel mit dem Branches und Mergen. Hat er beispielsweise vergessen, mit dem Befehl *copy* oder *move* eine Datei im Kontext von Mercurial zu kopieren oder zu verschieben, lässt sich das mit *hg addremove -similarity 100* nachholen (wobei die 100 für eine hundertprozentige Übereinstimmung steht). Dadurch geht nicht, wie bei Subversion, die Historie einer Datei verloren.

Ökosystem der Handwerkzeuge

Kurz sei nun das Ökosystem um Mercurial und Git etwas näher angeschaut. Viel zur Verbreitung von Subversion hat im Übrigen die gute Tool-Unterstützung sowohl unter den Unix/Linux-Derivaten sowie Windows als auch in diversen Entwicklungsumgebungen wie Eclipse beigetragen. TortoiseSVN (<http://tortoisesvn.tigris.org/>), ein Subversion-Client für Windows, setzt immer noch Maßstäbe, an dem sich viele andere Clients orientieren. In die Fußstapfen möchte **TortoiseHg[19]** treten, was dem Tool teilweise gelingt. Anzumerken ist, dass TortoiseHg "fast" plattformunabhängig daherkommt: Pakete für diverse Linux-Distributionen und Windows-Installer sind vorhanden, auch eine Portierung auf Mac OS X **ist vorhanden[20]**.

TortoiseHg unterstützt Entwickler durch eine gelungene Visualisierung der Historie mit zusätzlichen Informationen. Hier sind die Entwicklungszweige von MercurialEclipse und die Merge-Vorgänge zu sehen (Abb. 1)



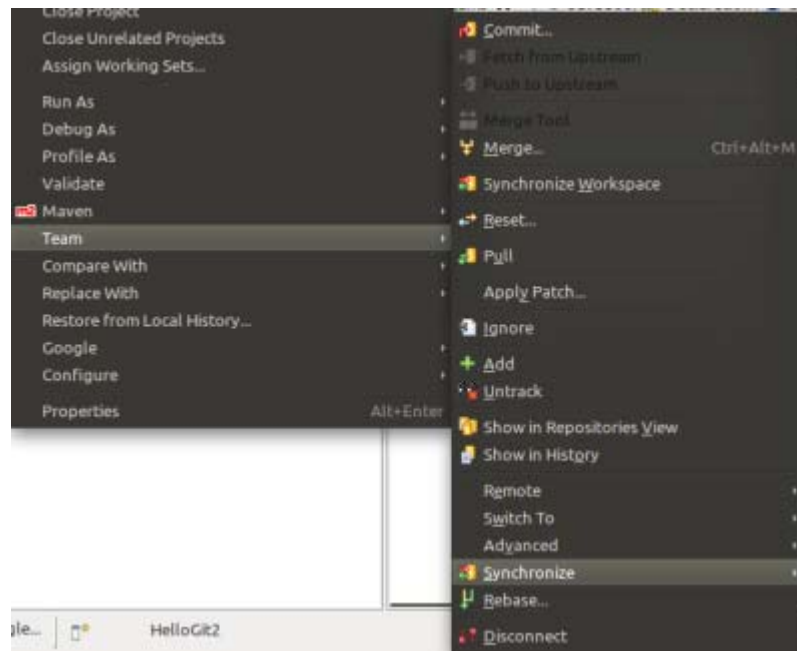
Zugriff auf Mercurial kann bequem über MercurialEclipse erfolgen (Abb. 2)

EGit, das Plug-in zur Integration von Git in die Entwicklungsplattform Eclipse, hat inzwischen mit Eclipse Indigo die Weihen der Version 1.0 erreicht. Die Installation erfolgt am besten, wie auf der **Projektseite**[26] empfohlen, über die **Update-Site**[27]. Eine Installation über den Eclipse Marketplace bricht mit Keybinding-Konflikten ab. Alle wichtigen Funktionen von Git sind über das Menü von EGit erreichbar, wenn auch viele Funktionen von Git nicht zur Verfügung stehen (siehe auch das passende *Release and Graduation Review*, **PDF**[28]). Kein Wunder bei deren Vielzahl. Das hindert einen Entwickler bei der täglichen Arbeit nicht.

EGit 1.0 wirkt aufgeräumt, wenn auch sich das Arbeiten mit EGit schon etwas von dem mit einem der Subversion-Plug-ins oder MercurialEclipse unterscheidet. Alleine die Tatsache, dass man ein Projekt umkopiert, (vorgeschlagen wird `~/git/[Projektname]`), dürfte manchen Neu- und Umsteiger etwas irritieren, die Einarbeitungszeit ist dadurch etwas länger und die Dokumentation hinkt oft hinterher.

Erwähnenswert ist die Unterstützung der "sozialen Entwicklungsplattform" GitHub, sodass sich das Einbinden von Repositories aus ihr heraus recht einfach gestaltet.

Das Menü von EGit 1.0 wirkt aufgeräumt, und die wichtigsten Funktionen stehen mit einem Mausklick zur Verfügung (Abb. 3).



Das Menü von EGit 1.0 wirkt aufgeräumt (Abb. 3)

Bleibt noch zu erwähnen, dass auch die Integration von Git und Mercurial in NetBeans und anderen Entwicklungsumgebungen gegeben ist. Selbst die Visual-Studio-Welt ist nicht vergessen worden: Es existieren mehrere Add-ins sowohl für Mercurial als auch für Git.

Umsteigen?

Eine Empfehlung zum Umsteigen in einem laufenden Projekt auf eines der vorgestellten verteilten Versionsverwaltungssysteme lässt sich nur bedingt aussprechen. Es empfiehlt sich abzuwägen, wie stark in einem Projekt tatsächlich verteilt entwickelt wird. Open-Source-Projekte haben es sicherlich mit ihrer Entscheidung einfacher, da sie häufig einen hohen "Verteilungsgrad" aufweisen. Diverse Werkzeugen erleichtern ein Umsteigen, sodass beispielsweise die Versionshistorie nicht verloren geht.

Subversion-Nutzer werden bei den Basisfunktionen viele Analogien vorfinden, müssen aber in einigen Punkten umdenken: Ein Commit reicht eben nicht mehr aus, um die lokalen Änderungen auch den Mitentwicklern zu übermitteln. Aber dafür hat ein Entwickler gegenüber Subversion viele Freiheiten, insbesondere im einfachen Umgang mit Branches oder der Möglichkeit, Änderungen auch als Mail verschicken zu können. Das und die immensen Geschwindigkeitsvorteile, die gerade in größeren Projekten zu spüren sind, sind ein guter Grund, sich für eines der vorgestellten Systeme im Rahmen eines neuen Projekts zu entscheiden.

Projekte, die eines der vorgestellten Systeme nutzen, sind weit entfernt von Chaos, auch wenn der eine oder andere Projektleiter die Befürchtung haben dürfte, dass seine Entwickler die gewonnenen Freiheiten exorbitant ausnutzen. Vielmehr erleichtern die Freiheiten die tägliche Arbeit und bauen unnötige Hürden ab. (ane)

Halil-Cem Gürsoy

ist als Senior Software Engineer bei der adesso AG tätig. Sein Schwerpunkt liegt auf Enterprise-Java-Entwicklungen (Java EE, Spring), bei denen er sich vor allem auf Event-getriebene Architekturen auf der Basis von SOA sowie der Definition und Implementierung von Entwicklungs- und Deployment-Umgebungen für SOA-Projekte konzentriert.

URL dieses Artikels:

<http://www.heise.de/developer/artikel/Ueber-den-Status-quo-verteilter-Versionsverwaltungssysteme-1285649.html>

Links in diesem Artikel:

- [1] <http://www.bitkeeper.com/>
- [2] <http://lkml.org/lkml/2005/4/6/121>
- [3] <http://lkml.org/lkml/2005/4/20/45>
- [4] <http://lkml.org/lkml/2005/4/25/278>
- [5] <http://bazaar.canonical.com>
- [6] <http://www.canonical.com>
- [7] <https://launchpad.net/bzr-eclipse>
- [8] <http://verterok.com.ar/bzr-eclipse/update-site/>
- [9] <http://git.kernel.org/>
- [10] <http://heise.de/-1280369%20und%20http%3A/heise.de/-1255416>
- [11] <http://mercurial.selenic.com/wiki/GitConcepts>
- [12] <http://www.github.com>
- [13] <https://bitbucket.org/>
- [14] http://wiki.eclipse.org/Platform-releng/Juno_Git_Migration
- [15] <http://mercurial.selenic.com/downloads/>
- [16] <http://tortoisehg.bitbucket.org/download/>
- [17] <https://launchpad.net/~tortoisehg-ppa>
- [18] <http://code.google.com/p/rabbitvcs/issues/detail?id=261>
- [19] <http://tortoisehg.bitbucket.org/>
- [20] <https://bitbucket.org/tortoisehg/thg/wiki/developers/MacOSX>
- [21] <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>
- [22] <http://andrei.gmxhome.de/mercurialeclipse/index.html>
- [23] <http://code.google.com/a/eclipseelabs.org/p/mercurialeclipse/>
- [24] <http://cbes.javaforge.com/update>
- [25] http://mercurialeclipse.eclipseelabs.org.codespot.com/hg.wiki/update_site/stable
- [26] <http://eclipse.org/egit>
- [27] <http://download.eclipse.org/egit/updates>
- [28] http://m1.archiveorange.com/m/att/OkDct/ArchiveOrange_nXAvhhpL9zFoyk0cPbTcHsFfuhka.pdf