Kurzanleitung SVN

1 Grundlagen

Was ist SVN? SVN ist ein freies Versionskontrollsystem, welches seit Beginn 2000 entwickelt wird. Es handelt sich dabei um freie Software, die unter der Apache-Lizenz 2.0 vertrieben wird. Ein Versionskontrollsystem ist eine Software, die Personen aller Art (nicht nur Entwicklern) dabei hilft, Dateien und Verzeichnisse über einen längeren Zeitraum hinweg zu verwalten. Dabei geht es unter anderem darum, jeden möglichen Stand einer Datei festzuhalten und (falls notwendig) wiederherstellen zu können.

Warum SVN? Die Entwicklung einer Software ist oftmals nicht trivial. Vielleicht hat man eine Änderung gemacht, die sich im Nachhinein als falsch herausstellt. Oder aber, man möchte den Verlauf eines Projektes im Auge behalten. So kann man den Stand eines Projektes rückblickend zum Beispiel wochenweise betrachten. In den meisten Fällen werden Versionskontrollsysteme hauptsächlich genutzt, um mit mehreren Personen gemeinsam an Dateien zu arbeiten. Damit dies möglich ist, müssen die Dateien zentral auf einem Rechner verwaltet werden. Eine verwaltete Sammlung von Dateien wird als Repository bezeichnet. Jedes Gruppenmitglied, welches an diesen Dateien mitarbeiten möchte, wird sich mit diesem Repository verbinden. In größeren Teams ist es nicht immer einfach, den Überblick über die Verantwortungsbereiche der Entwickler zu behalten. Die Frage „Welcher Entwickler hat eine Änderung in der Methode xy gemacht?“ lässt sich mit Hilfe eines Versionskontrollsystems sehr leicht beantworten. Solche Systeme verpflichten den Entwickler, zu jeder eingepflegten Änderung einen Kommentar hinzuzufügen (*commit-message*), der beschreibt, was er genau gemacht hat. Außerdem melden sich die Entwickler beim Verbinden mit dem Repository mit einer Benutzerkennung und dem zugehörigen Passwort an. So kann man genau sehen, wer was wann eingepflegt hat. Versionskontrolle stellt somit ein wichtiges Werkzeug im Softwareentwicklungsprozess dar.

1.1 Wie funktioniert SVN?

Ein *Repository* in SVN speichert die Informationen in Form eines Dateisystembaums. Man kann sich also SVN wie einen Fileserver mit Zusatzfunktionalität vorstellen. Um die Zusammenarbeit mehrerer Leute an einer Datei zu ermöglichen, nutzt SVN eine Lösung, die als „Kopieren-Verändern-Zusammenführen“ bezeichnet wird. Eine andere Lösung der Problematik ist als „Sperren-Verändern-Entsperren“ bekannt. Das erste Modell wird nun detaillierter erläutert.

Kopieren Jeder Benutzer kopiert sich die Dateien aus dem Repository (*checkout*, *auschecken*). Man bezeichnet diese Kopien als *Arbeitskopien*. Besagte Kopien liegen dann lokal auf dem Rechner des Benutzers.

Verändern Die Arbeitskopie kann vom Benutzer beliebig modifiziert werden.

Zusammenführen Die Kopie des Benutzers wird, wenn er sie in das Repository einfügt (man spricht von einem *commit*) mit der vorhandenen Version zusammengefügt.

Achtung! Man sollte darauf achten, bei einem *commit* immer nur Code einzupflegen, welcher „compile-clean“ ist. Ansonsten bekommt man mit ziemlicher Sicherheit Ärger. Als Beispiel sei Microsoft aufgeführt. Dort ist es so, dass derjenige, der Schuld daran ist, dass ein Nightly-Build scheitert, den nächsten beaufsichtigen muss.

1.2 Probleme beim Zusammenführen

Nehmen wir an, zwei Benutzer - Angua und Nobby - haben das Repository ausgecheckt. Nun arbeiten beide an einer lokalen Kopie der selben Datei `A`. Angua ist mit ihren Änderungen der Datei schneller fertig als Nobby, weshalb sie ihre Änderungen vor ihm einspielt (*commit*). Wenn Nobby nun versucht seine Änderungen in das Repository einzupflegen, wird er beim *commit* eine Fehlermeldung bekommen, dass seine Version der Datei nicht mehr aktuell („out-of-date“) ist. Das liegt daran, dass Angua eine neuere Version der Datei eingespielt hat. SVN erkennt dies anhand von Informationen, die es auf der Festplatte des Benutzers ablegt (`.svn`-Verzeichnis). Diese Informationen werden mit denen im Repository verglichen und geben Aufschluss darüber, wann sich wo etwas geändert hat. Nun muss Nobby SVN anweisen, die Änderungen von Angua (also die aus dem SVN) in seine Datei zu überführen. Man spricht dabei auch von der Operation *merge*. Hier können nun zwei Fälle eintreten:

Erfolg Anguas Änderungen überschneiden sich nicht mit denen von Nobby, somit kann SVN die Dateien eigenständig ohne ein Zutun der Benutzer *mergen*. Nobby führt mit der neu erstellten Datei ein *commit* durch, somit liegt die aktuelle Version mit allen Änderungen von Angua und Nobby im SVN.

Konflikt Anguas und Nobbys Änderungen überschneiden sich, somit kann SVN nicht entscheiden, wie die Zusammenführung passieren soll. Dies wird als *Konflikt* bezeichnet. Nobby erhält eine Meldung, dass ein automatisches Zusammenführen nicht möglich ist, danach wird die Datei von SVN entsprechend gekennzeichnet (*File xy.z is in a conflicted state*).

Wenn ein Konflikt vorliegt, muss Nobby ihn selbst auflösen. Entweder kann er dies allein tun (wenn ihm klar ist, was zu tun ist), oder aber er muss Rücksprache mit Angua halten, um die richtigen Entscheidungen treffen zu können. Hat er den Konflikt beseitigt, so kann er die konfliktfreie Version der Datei per *commit* in das Repository einpflegen.

1.3 Revisionen

Ein Benutzer kann beliebig viele Änderungen an einer oder mehreren Dateien und Verzeichnissen seiner Arbeitskopie des Repositorys vornehmen. Diese Änderungen kann er mit einem *commit* dann dem Repository bekannt machen. Ein *commit* ist immer atomar, das bedeutet, dass entweder alle oder gar keine Änderungen in das Repository geschrieben werden. Kommt es also bei einer einzigen Datei zu einem Problem, so wird keine einzige Datei / kein Verzeichnis verändert. Wenn beim *commit* keine Fehler auftreten, so wird von SVN für diese neue Version des gesamten Repositorys eine sogenannte *Revisionsnummer* angelegt. Zu dieser neuen Revisionsnummer wird der Zustand des kompletten Repositorys hinterlegt. Betrachten wir ein kleines Beispiel zu Revisionen.

Beispiel Wenn das Repository angelegt wird, wird dem leeren Verzeichnis von SVN die Revisionsnummer 0 zugeordnet. Nun verbindet sich Angua mit dem Repository und lädt eine Arbeitskopie herunter. Ihr lokales Verzeichnis verbleibt vorerst leer, es hat die Revisionsnummer 0. Nun legt sie im Verzeichnis eine Quellcode-Datei `Programm.java` an. Wenn sie nun ihre Arbeitskopie per *commit* an das Repository sendet, wird dort die Datei aufgenommen und der gesamte Inhalt des Repositorys wird mitsamt der neuen Revisionsnummer 1 gespeichert. Nun gibt es also zwei Revisionen im Repository, eine leere (0) sowie eine mit einer Datei `Programm.java` als Inhalt (1).

1.4 Ein Konfliktfall im Detail

Wenn Nobby ein SVN *update* durchführt, so kann es dabei zu einem Konflikt kommen. Angenommen, er führt ein *update* auf eine Datei `main.c` aus, die er seit dem letzten *update* oder *checkout* lokal verändert hat. In der Zwischenzeit hat Angua die gleiche Datei verändert und per *commit* eingepflegt. Nun überschneiden sich seine Änderungen mit denen von Nobby. Entsprechend wird Nobby beim *update* eine Meldung erhalten, dass es bei der Datei `main.c` zu einem Konflikt kam. SVN meldet Konflikte auf verschiedene Arten und Weisen.

1. Nobby wird während des *updates* die Ausgabe erhalten, dass sich die Datei `Programm.java` in einem Konfliktzustand befindet.
2. SVN wird in der Datei selbst sogenannte *Konfliktmarker* setzen. Das sind spezielle Strings die den Bereich der Datei kennzeichnen, in dem es zu einem Konflikt gekommen ist.
3. SVN wird mehrere Dateien in Nobbys Arbeitskopie erzeugen, die im Folgenden aufgeführt werden.

`dateiname.mine` ist die Datei so, wie sie in der lokalen Arbeitskope vorhanden war. Hierin werden keine Konfliktmarker gesetzt.

`dateiname.r<OLDREV>` ist die Version der Datei, die Nobby als letztes ausgecheckt hat. Also bevor er seine Änderungen vorgenommen hat. `OLDREV` stellt nur einen Platzhalter für die Revisionsnummer dar.

`dateiname.r<NEWREV>` ist die aktuelle Version der Datei im Repository, also die, die Angua per *commit* eingepflegt hat. Diese Datei stellt also die *HEAD*-Revision[1] des Repositorys dar.

Um nun zu vermeiden, dass Nobby versehentlich seine Version der Datei einpflegt, erlaubt SVN einen *commit* erst, wenn die temporären Dateien (`.mine, .rxx, .rxy`) gelöscht wurden. Natürlich ist es für Nobby ein Leichtes, diese vermeintliche Absicherung zu umgehen und keine Rücksprache mit Angua zu halten. An dieser Stelle muss SVN auf die Vernunft der Benutzer vertrauen.

1.5 commit-messages

Wie bereits beschrieben muss zu jedem *commit* eine Notiz in Form einer *commit-message* vorliegen. Darin wird beschrieben, was man verändert und eingepflegt hat. Es gibt jedoch keinen einheitlichen Standard, was genau in einer *commit-message* zu stehen hat. Blödsinnige oder nichtssagende Nachrichten der Art „Bug

[1] Die HEAD-Revision ist die aktuellste Revision des Dateisystembaums im Repository.

gefixt“ sind inakzeptabel! Eine *commit-message* muss aussagekräftig sein. Je nach Entwicklungsteam, Projekt und natürlich auch in Abhängigkeit von der Änderung, kann der Text variieren. Eine saubere *commit-message* könnte zum Beispiel wie folgt aussehen:

```
Änderungen an der Klasse ErlangKVerteilung

Add: Konstruktor ErlangKVerteilung(ZufaZaGenerator zzg)
     Da der Generator nur einmal gesetzt wird, kann er direkt übergeben werden.

Fix: Funktion getErlangKDistribution()
     Die Funktion verwendet statt direkten Zufallszahlen jetzt negativ
     exponentiell verteilte Zufallszahlen, da der Graph sonst nicht korrekt
     gezeichnet wird.
```

1.6 Aufbau eines Repositorys

In Repositories von Entwicklungsteams hat sich folgender Aufbau bewährt. Er wird daher oftmals verwendet und soll an dieser Stelle ein wenig erläutert werden. Die Beschreibung erfolgt Verzeichnis für Verzeichnis, zuerst jedoch ein Blick auf die Verzeichnisstruktur.

```
repository/trunk
          /branches
          /tags
```

`repository` stellt hier die Wurzel des Repositorys dar. Darin ist also der gesamte Repository-Inhalt abgelegt.

`trunk` ist das Verzeichnis, in dem aktuell entwickelt wird. Hier wird also eine (wahrscheinlich noch unfertige) Version der Software abgelegt. Man spricht dabei von der *Hauptentwicklungslinie*.

`branches` ist das Verzeichnis, in dem sogenannte *Nebenläufige Entwicklungszweige* abgelegt werden[2]. Hier werden Duplikate des trunks abgelegt, die dann zum Beispiel dazu dienen, andere Features etc. zu entwickeln. Später, nach dem Abschluss der Entwicklung, kann dann der entstandene Branch (genauer: die darin entstandenen Änderungen) mit dem trunk zusammengeführt werden.

`tags`[3] enthalten meist bestimmte Revisionen der Software, die man in irgendeiner Form festhalten möchte. Auch hier wird entsprechend ein bestimmter Stand der Hauptentwicklungslinie kopiert und dann als unveränderlich festgehalten.

1.7 Beispiel: Einen Zweig anlegen

Angenommen, eine Firma schreibt ein Warenwirtschaftssystem für mehrere Kunden. Angua und Nobby arbeiten beide an der Hauptentwicklungslinie (trunk). Der Kunde „Mumm“ wünscht sich im Verlauf der

[2] *engl. branch = Verzweigung*
[3] engl. *to tag sth. = etwas auszeichnen*

Entwicklung eine Funktionalität, die alle anderen Kunden nicht benötigen. Daraufhin wird Nobby beauftragt einen Zweig[4] anzulegen und das Feature gesondert darin zu entwickeln. Nobby entscheidet sich, den Zweig an der Konsole zu erzeugen. Dafür nutzt er den Befehl `svn copy`. Der Aufruf sieht wie folgt aus:

```
$ svn copy http://pfad/zum/repository/trunk \
           http://pfad/zum/repository/branches/feature_xy \
       -m "Neuer Zweig für Feature xy, angefordert von Kunde Mumm"

   Committed revision 145.
```

Wie das Beispiel zeigt, kann Nobby den Zweig einfach durch das Kopieren des trunks erzeugen. Da mit URLs gearbeitet wird, ist nicht einmal eine lokale Arbeitskopie erforderlich. Ein *commit* passiert ebenfalls automatisch. Auf anderem Wege wäre es auch möglich gewesen den trunk auszuchecken, ein Verzeichnis für den Zweig von Hand anzulegen und dann lokal mit `svn copy` zu arbeiten. Dann hätten die neu hinzugefügten Dateien (= der Zweig) allerdings auch manuell per *commit* in das Repository eingepflegt werden müssen.

1.8 Zweige zusammenführen

Angenommen, Nobby hat seine Arbeiten am erstellten Zweig fertiggestellt. Auch Angua war fleißig und hat in der Hauptentwicklungslinie vieles verändert und hinzugefügt. Nun soll Nobbys Feature in den trunk eingefügt werden. Man spricht bei diesem Vorgang vom Zusammenführen (*mergen*) zweier Zweige. Um den Zweig mit dem trunk zusammenzuführen, muss Nobby zuerst herausfinden, welche Revisionsnummer der Zweig bei seiner Erstellung besaß. Dies kann er mit dem Befehl `svn log --verbose --stop-on-copy` tun. Dieser wird, ausgeführt im Verzeichnis, in dem der Branch liegt, eine Ausgabe der Art

```
  ...
  ------------------------------------------------------------------
  r145 | nobby | 2010-03-01 ..........
  ...
  A /pfad/zum/repository/branches/feature_xy (from /pfad/zum/repository/trunk:144)
```

erzeugen. Hierbei stellt `r145` die Revisionsnummer des Zweiges beim Erstellen dar. Nun muss Nobby die Änderungen ab dieser Revision mit dem trunk zusammenführen. Dazu nutzt er den Befehl `svn merge`. Diesen muss er im trunk ausführen.

```
$ svn merge -r 145:HEAD http://pfad/zum/repository/branches/feature_xy
  ...
  U main.c
  A button.c
  D obsolete.c
```

[4] *engl. branch*

Kam es nicht zu Konflikten (oder nur zu solchen, die SVN automatisch beheben konnte), so kann Nobby das Ergebnis der *merge*-Operation per *commit* in den trunk einpflegen. Was er im Konfliktfall tun muss, wurde bereits in Kapitel 1.2 auf Seite 2 beschrieben. Sollte Nobby in seinem Zweig weiterarbeiten und später seine Änderungen erneut mit dem trunk zusammenführen wollen, so verändert sich die Startrevision der *merge*-Operation entsprechend. Das Zusammenführen startet nun bei der Revisionsnummer des *commits* seiner *merge*-Änderungen. Diese kann man mit dem Befehl `svn log` herausfinden. Im Log sollte eine Zeile der Form

```
r156 | nobby | ........

Feature von Mumm mit dem trunk zusammengeführt (r145:155)
```

zu finden sein. Achtung: Bei der zweiten Zeile handelt es sich um die *commit-message* von Nobby. Hätte er sich nicht an die Regeln gehalten und keine angegeben, so käme nun die Arbeit auf ihn zu, die Revision von Hand herauszusuchen.

1.9 Eigenschaften

SVN bietet die Möglichkeit zu allen versionierten Dateien und Verzeichnissen Metadaten hinzuzufügen, verändern oder auch zu entfernen. Diese Metadaten sind ebenfalls versioniert und werden als *Eigenschaften*[5] bezeichnet. Diese Eigenschaften kann man sich als Paare von Namen und Werten vorstellen. Name sowie den Wert einer Eigenschaft kann man nach Belieben wählen. In [1] wird dies wie folgt beschrieben:

„Ganz allgemein ausgedrückt können Eigenschaften alles sein, was Sie nur wollen. Die einzige Einschränkung besteht darin, dass es sich bei den Namen um vom Menschen lesbaren Text handeln muss.“

Wann verwendet man Eigenschaften? Auch hier wird auf ein Beispiel aus [1, Seite 159] zurückgegriffen, da es sich dabei um ein schönes und leicht verständliches handelt.

Angenommen, man entwickelt eine Webseite. Auf dieser sollen dem Besucher Fotos in digitaler Form präsentiert werden. Zu jedem Foto soll eine Bildunterschrift sowie ein Zeitstempel angezeigt werden. Außerdem soll der Besucher der Seite die Bilder im Kleinformat als thumbnails angezeigt bekommen können, damit er sich eine Übersicht verschaffen kann. Dies alles soll mit traditionellen Dateien erreicht werden. Was bedeutet das? Zu einem Bild würde es zunächst eine Datei `image123.jpg` geben. Außerdem zum Beispiel eine Datei `image123-thumbnail.jpg`. Oder aber man legt die thumbnail-Bilddatei in einem anderen Verzeichnis ab (`/thumbnails/image123.jpg`). Die Bildunterschriften sowie die Zeitstempel werden auf die gleiche Art und Weise abgelegt, auch getrennt von der Bilddatei. Es ist offensichtlich, dass hier schon nach kurzer Zeit Chaos im Verzeichnisbaum herrscht. Dieser Umstand wird mit der Zeit immer schlimmer. Wie können SVN Eigenschaften diese Situation nun vereinfachen oder verbessern? Mit ihrer Hilfe kann man nun zu jeder Bild-Datei Metadaten ablegen. Zum Beispiel kann SVN zu jeder Datei eine Eigenschaft mit dem Namen Timestamp mit dem Zeitstempel als Wert ablegen. Dies funktioniert analog für die Bildunterschrift und sogar das thumbnail-Bild! Damit existiert trotz der zusätzlichen Informationen nur eine einzige Datei pro Bild im Verzeichnis.

[5] *engl. properties*

1.10 Besondere Eigenschaften

In SVN gibt es einige spezielle Eigenschaften, von denen zwei an dieser Stelle näher beschrieben werden sollen. Diese Eigenschaften haben einen eigenen definierten Namensraum, ihre Namen beginnen alle mit dem Präfix `svn:`. Entsprechend dürfen selbstdefinierte Eigenschaften nicht mit diesem Präfix versehen werden.

svn:ignore Der Wert der `svn:ignore` Eigenschaft ist eine Liste von Dateimustern, die bei bestimmten SVN-Operationen (`svn status, svn add, svn import`, etc.) ignoriert werden. So kann recht einfach sichergestellt werden, dass bestimmte Dateien nicht im Repository auftauchen (zum Beispiel temporäre Dateien). Ein `svn:ignore` welches für ein Verzeichnis gilt, gilt nicht automatisch für die darin enthaltenen Unterverzeichnisse. Jedoch werden beim Befehl `svn:copy` die Eigenschaften ebenfalls kopiert.

svn:externals Mit der Eigenschaft `svn:externals` kann man SVN anweisen, ein versioniertes Verzeichnis mit ausgecheckten SVN-Arbeitskopien zu bestücken. Was bedeutet dies? In einer *external*-Eigenschaft kann man SVN anweisen, zusätzlich zu dem Verzeichnis, auf dem sie definiert ist, noch weitere Verzeichnisse auszuchecken. Diese können an einer beliebigen Stelle in der Arbeitskopie abgelegt werden. Der Pfad dazu wird in der Eigenschaft abgelegt. Nachdem die Eigeschaft also auf dem Verzeichnis gesetzt ist, wird bei jedem *checkout* auch die *external*-Eigenschaft abgearbeitet und das darin enthaltene Verzeichnis ausgecheckt.

Vorsicht ist beim *commit* geboten: SVN wird Verzeichnisse, die per *external*-Eigenschaft erzeugt wurden, nicht *committen*, solange es nicht explizit dazu aufgefordert wird (*commit* auf eben jenes Verzeichnis).

2 Verwendung

Betrachten wir nun, wie SVN verwendet wird. Dabei werden beispielhaft drei Tools beschrieben. Zunächst die SVN Befehle auf der Konsole (im Beispiel eine Linux-Shell), dann die Einbindung von SVN in Eclipse (*Subclipse*) und zuletzt das Tool *Tortoise SVN* (Windows).

2.1 Konsole

SVN kann man grundsätzlich recht einfach über die Konsole bedienen. Dazu installiert man unter Linux die entsprechenden Pakete. Wir werden hier nur die gebräuchlichsten Befehle betrachten, genauere Informationen dazu findet man in [1, Seite 6 ff.]. Ein erster *checkout* funktioniert an der Konsole wie folgt:

```
$   svn checkout http://repository.mni.fh-giessen.de/svn/dba/
A   dba/jdbc
A   dba/DIS_TableViewer_mit_ADO.NET.sln
A   dba/zugriffsystem

...
```

```
Checked out revision 219.
```

Wie man sieht, listet SVN jede Datei und jedes Verzeichnis auf, welche(s) ausgecheckt wurde. Zuletzt zeigt es die Revisionsnummer der ausgecheckten Objekte an. Das `A` steht für *add*, was bedeutet, dass die Datei der lokalen Arbeitskopie neu hinzugefügt wurde.

Nun kann man die Dateien und Verzeichnisse beliebig verändern. Der Befehl

```
svn commit --message "commit message"
```

schreibt die Änderungen ins Repository. *commit-message* ist natürlich nur ein Platzhalter für eine Nachricht. Man darf **niemals** ein *commit* ohne entsprechende Informationen für andere durchführen. Eine *commit-message* ist Pflicht! Möchte man nur eine einzelne Datei einpflegen so kann man

```
svn commit main.c --message "Funktion getRandomNumber() gefixt"
```

verwenden. Hierbei wird jetzt nur die Datei `main.c` mit der entsprechenden Nachricht im Repository aktualisiert. Achtung: Es wird trotzdem die Revisionsnummer aller Dateien und Verzeichnisse erhöht.

Möchte man seine lokale Arbeitskopie auf den neuesten Stand bringen, so verwendet man den Befehl `svn update`. Dieser sieht ausgeführt wie folgt aus:

```
$ svn update
U   dbarch.tex
U   anfragebearb.tex
Updated to revision 220.
```

In diesem Fall wurden beim Update zwei Dateien aktualisiert (`U`). Es hätte auch sein können, dass Dateien gelöscht (`D`) oder ersetzt (`R`) werden. Dies hängt davon ab, was sich im Repository seit dem letzten *Update* getan hat.

2.1.1 Eclipse Plugin: Subclipse

Für Eclipse existieren zwei verschiedene Plugins, mit denen sich SVN Funktionalität in die Enwicklungsumgebung integrieren lässt. Sie heißen *Subclipse* und *Subversive*. Der Vorteil liegt auf der Hand, kein Fenster- bzw. Anwendungswechsel ist notwendig, um SVN-Operationen durchzuführen. Als Fallbeispiel wird hier die gängige Vorgehensweise mit *Subclipse* gezeigt.

Nachdem man das Plugin installiert hat (Informationen dazu findet man auf `http://subclipse.tigris.org/`), kann man in Eclipse über das Menü *Window* → *Open Perspective* → *Other* → *SVN Repository Exploring* in die SVN-Ansicht wechseln.

Im nächsten Schritt muss die `URL` zum Repository eingegeben werden. Dann wird der Inhalt des Repositorys angezeigt und man kann per Kontextmenü einen Teil oder auch den gesamten Inhalt auschecken.

Nachdem man einen Teil zum Auschecken ausgewählt hat, kann man sich für ein Standardprojekt oder den Projekt-Assistenten von Eclipse entscheiden. Je nach Wahl ergeben sich entsprechend weitere Optionen. Nach dem erfolgreichen *checkout* wechselt man zurück in die Java-Perspektive, dort findet man das Projekt. In den Kontextmenüs der Dateien und Verzeichnisse findet man unter dem Punkt *Team* nun die SVN Operationen, die für diese zur Verfügung stehen (siehe Abbildung 1). Detaillierte Ausführungen zur Arbeit mit Subclipse findet man auf [1].

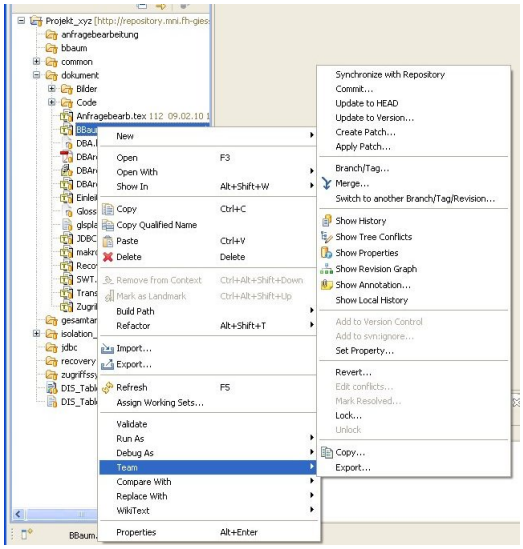

Abbildung 1: Die SVN-Operationen in Eclipse.

2.2 Tortoise SVN

Tortoise SVN ist ein Tool für Windows, welches SVN Funktionalität direkt im Windows Explorer integriert. Man kann es also hervorragend nutzen, wenn die Entwicklungsumgebung keine Möglichkeit zur Arbeit mit SVN bietet.

Nach der Installation des Programms stehen in den Kontextmenüs des Explorers verschiedene Befehle für SVN zur Verfügung. Dort wählt man den Punkt *SVN Checkout...* aus, wodurch sich das in Abbildung 2 auf der nächsten Seite dargestellte Fenster öffnet.

Den erfolgreichen *checkout* der Arbeitskopie quittiert Tortoise SVN mit dem einem Fenster, in dem alle ausgecheckten Inhalte aufgelistet werden. Zurück im Explorer fallen die grünen Haken an den Verzeichnis- und Dateisymbolen auf. Sie deuten darauf hin, dass die Objekte auf dem gleichen Stand wie die neueste Revision im Repository sind. Ein rotes Ausrufezeichen zeigt, dass sich an der Arbeitskopie etwas geändert hat. Ein gelbes Ausrufezeichen deutet auf einen Konflikt bei dieser Datei hin. Mehr Informationen zu Tortoise SVN kann man auf der Homepage des Tools finden (`http://tortoisesvn.tigris.org`).

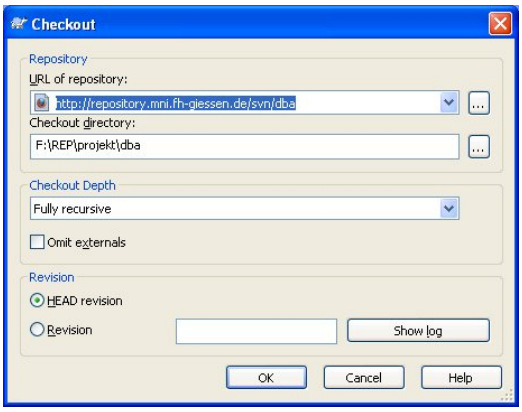

Abbildung 2: Ein Repository mit Tortoise SVN auschecken

2.3 Begriffsdefinitionen in aller Kürze

Repository Das zentrale Konstrukt, in dem SVN alle Dateien mit all ihren Revisionen ablegt.

Commit Als *commit* wird die Operation bezeichnet, mit der der Benutzer Veränderungen in das Repository einpflegt.

Checkout Wenn ein Benutzer den Inhalt eines Repositorys das erste Mal auf seinem Rechner abspeichert (Arbeitskopie), so spricht man von einem *checkout*.

Update Mit der *update*-Operation kann der Benutzer SVN anweisen, seine lokale Version auf den neuesten Stand zu bringen.

Merge Das oben beschriebene Zusammenführen wird im Englischen *merge* genannt. Eine *merge*-Operation entspricht also dem Zusammenführen zweier Revisionsstände.

Literatur

[1] Ben Collins-Sussman, Brian W. Fitzpatrick & C. Michael Pilato. *Versionskontrolle mit Subversion*. O'Reilly, Koeln, 2005.

[2] Ben Collins-Sussman, Brian W. Fitzpatrick & C. Michael Pilato. *Versionskontrolle mit Subversion*. Technischer Bericht, http://svnbook.red-bean.com, 2009.

Autor: Sebastian Philippi, Institut für SoftwareArchitektur.