

Versionskontrolle mit Subversion

Ralph Thesen

Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn

Seminar: Technische Numerik
November 2009



Überblick

- 1 Einleitung
 - Warum?
 - Aber!
 - Konzepte
- 2 Erste Schritte
 - Zugriff auf ein Repository
 - import
 - list
 - checkout
- 3 Täglicher Arbeitsablauf
 - update
 - add
 - commit
 - status
 - diff
- 4 Hinter den Kulissen
 - Parameter
 - .svn
 - Properties
 - Keyword Substitution
- 5 Rocket Science
 - Konflikte
 - Branch
 - Merge
 - Blame und Log
- 6 Anhang
 - TortoiseSVN
 - RapidSVN
 - KDESvn
 - Subclipse
 - Netbeans



Warum Versionskontrolle?

Mein tägliches

```
zip -r megaprojekt.20091112.zip megaprojekt/
```

 reicht mir!

Nein.

Mit Subversion (kurz: **svn**) kann man

- einfach *kleinschrittig* versionieren
- einfach zwischen Versionen *springen*
- einfach verschiedene Versionen *vergleichen*
- einfach die Arbeit an verschiedenen Plätzen synchron halten (Notebook, Uni, Zuhause)



Aber ...

Das ist doch alles zuviel Aufwand! Mein Projekt ist riesig! Ich habe ganz viele Verzeichnisse und Dateien!

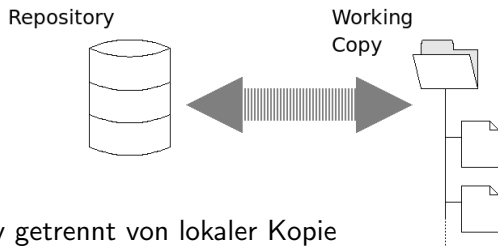
Nein. - Es ist nur einmal Aufwand.

Bevor man ein Projekt in ein Repository ablegt, muss/sollte man

- aufräumen. (*Sollte man vermutlich eh längst gemacht haben.*)
- das Projekt in einer geordneten Struktur ablegen. (*Sollte man vermutlich ...*)
- generierte/kompilierte Dateien rauswerfen.



Konzept (1): Lokale Kopie, zentrales Repository



- Repository getrennt von lokaler Kopie
- Änderungen werden zwischen dem Repository und lokalen Kopien übertragen
- Es können mehrere lokale Kopien existieren, auch bei verschiedenen Benutzern



Konzept (2): Revisionen

Jeder Übertragung von Änderungen wird eine Revisionsnummer zugeordnet.

- Import des Projekts.
→ Revision 1.
- Änderung an den Dateien `matrix.h` und `matrix.cpp`.
→ Revision 2.
- Löschen von `Test/UnitTestMatrixSymmetrisch.cpp`, Hinzufügen von `Test/UnitTestMatrix.cpp`, Änderung von `main.cpp`.
→ Revision 3.
- ...

Zu jeder Revision gehört ein Kommentar. Dieser sollte durch Menschen lesbar sein.

HEAD ist immer die aktuellste Revision, BASE immer die Revision, die die Basis der lokalen Kopie bildet.



Erste Schritte

checkout create
svnadmin cat list
SVN import



Ein Repository erstellen

Muss man ein Repository selber lokal erstellen, nimmt man den `svnadmin` zur Hilfe:

```
> svnadmin create /path/repository  
> ls /path/repository  
conf/  dav/  db/  format  hooks/  locks/  README.txt  
>
```

Nun steht unter `file:///path/repository` ein leeres Subversion-Repository bereit.



74 Worte zur Repository-Url

Eine Repository-Url kann verschieden aussehen:

- `file://` ein lokales Repository.
- `http://` ein Repository auf einem Webserver.
- `https://` ... via SSL.
- `svn://` Zugang zu einem `svnserve` Server.
- `svn+ssh://` Zugang zu einem Repository durch einen SSH-Tunnel.

Keine Sorge!

- Man braucht die Url im täglichen Arbeitsablauf **nie**.
- Nur beim `checkout` und `import` ist sie unumgänglich.*



import

- Wir versuchen uns an einem neuen, leeren Repository.
- Wir importieren unsere existierende Struktur (Dateien und Verzeichnisse) mit `svn import`.

```
> svnadmin create /path/repository
> mkdir example; cd example
> mkdir trunk branches tags
> svn import -m "Hello Svn!" file:///path/repository
Adding          trunk
Adding          branches
Adding          tags

Committed revision 1.
>
```



list

- Wir wollen uns ein Repository ansehen.
- Dazu dient `svn list` bzw. kurz `svn ls`.

```
> svn list file:///path/repository  
branches/  
tags/  
trunk/
```

- Ist ein Projekt neu und leer, bekommt man nichts weiter von `svn list` angezeigt.
- Will man detailliert Dateien ansehen, arbeitet `svn cat` wie man es erwartet.



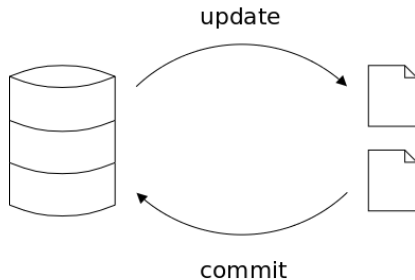
checkout

- Wir wollen eine lokale Kopie eines Repositories erstellen (um damit zu arbeiten).
- Dazu dient `svn checkout` bzw. kurz `svn co`.

```
> svn checkout file:///tmp/testrepo/ example
A   example/trunk
A   example/branches
A   example/tags
Checked out revision 1.
>
```



Täglicher Arbeitsablauf



update

- Die erste Handlung eines Arbeitstages: Update!
- Vor einem Commit: Update!
- `svn update` oder kurz `svn up`.

```
> svn update  
At revision 1.
```

Im Repository hat sich nichts geändert, wir sind up-to-date.

```
> svn update  
A    main.cpp  
U    readme.txt  
Updated to revision 3.
```

Hier wurde die Datei `readme.txt` geändert **U** und die Datei `main.cpp` neu hinzugefügt **A**.



add

- Mit `svn add` fügt man neue Dateien in der lokalen Kopie dem Repository hinzu.

```
> vim helloworld.cpp
> svn add helloworld.cpp
A      helloworld.cpp
>
```

- Binäre Dateien erkennt Subversion in der Regel von selbst.

```
> gimp svn_commands.png
> svn add svn_commands.png
A (bin)  svn_commands.png
>
```



commit (1)

Wir übertragen eine lokale Änderung ins Repository mit

`svn commit` oder kurz `svn ci`.

Wenn wir keinen Kommentar zu unserem commit angegeben haben, öffnet sich unser Lieblingseditor:

```
--This line, and those below, will be ignored--  
M      trunk/readme.txt  
A      trunk/matrix.h
```

Hier können wir den Kommentar ergänzen. Zur Unterstützung listet svn die betroffenen Dateien auf. Hier wurde die `readme.txt` geändert **M** und die `matrix.h` hinzugefügt **A**.



commit (2)

```
> svn commit -m "Ein menschenlesbarer Kommentar."  
Adding          trunk/matrix.h  
Sending         trunk/readme.txt  
Transmitting file data ...  
Committed revision 4.
```

- Wenn alles passt, wird der `commit` ausgeführt und die Daten übertragen.



status

- Was habe ich zuletzt gemacht?

```
> svn status
?      run
A      matrix.cpp
C      readme.txt
D      bar.cpp
M      matrix.h
```

- **?** Datei ist nicht versionskontrolliert
- **A** Datei ist zum Hinzufügen vorgemerkt
- **C** Datei hat Konflikte durch ein Update
- **D** Datei ist zum Löschen vorgemerkt
- **M** Der Inhalt von bar.c hat lokale Änderungen

-v erzählt mehr, **-q** weniger, **-u** kontaktiert das Repository.



diff

- Was habe ich zuletzt wirklich geändert?
- `svn diff` zeigt den Unterschied zur letzten Revision.

```
> svn diff
Index: helloworld.cpp
=====
--- helloworld.cpp (revision 1)
+++ helloworld.cpp (working copy)
@@ -1,8 +1,12 @@
 #include <iostream>
 using namespace std;
+void hello() {
+ cout << "Hello World" << endl;
+}
 int main()
 {
-cout << "Hello World" << endl;
+hello();
 }
```

Mit `svn diff -r N` kann man mit Revision N vergleichen.



Hinter den Kulissen



Parameter

- Wird kein Parameter angegeben, nimmt `svn` immer das aktuelle Verzeichnis `.` an.
- `svn diff` zeigt rekursiv alle Änderungen an.
- `svn diff matrix.h matrix.cpp` zeigt nur die Änderungen dieser beiden Dateien an.
- `svn add core` fügt das komplette Verzeichnis `core` dem Repository hinzu, mit allen Daten und Unterverzeichnissen.
- `svn diff -N core` fügt nur das Verzeichnis `core` hinzu.

`-N` ist sehr wichtig, da `svn` normalerweise alles rekursiv macht.



Das Verzeichnis .svn

In jedem Verzeichnis der lokalen Kopie liegt das Verzeichnis `.svn/`

```
> ls .svn
all-wcprops  format      props/      tmp/
entries      prop-base/  text-base/
>
```

- Das Verzeichnis enthält die kompletten Daten der aktuellen Revision.
- Vorteil: Für die Operationen `svn status`, `svn diff` und `svn revert` muss nicht das Repository bemüht werden, die Informationen liegen lokal vor.



Im Allgemeinen gilt: Pfoten weg.



Properties

- Dateien und Verzeichnisse können Eigenschaften (Properties) haben.
- `svn propset` Setzt eine Property, mit `-F` aus einer Datei.
- `svn propedit` Öffnet die Property im Editor.
- `svn propget` Zeigt eine Property an.
- `svn proplist` Zeigt existierende Properties an, mit `-v` auch was drin steht.
- `svn propdel` Löscht eine Property.

Auch hier muss ein `svn commit` ausgeführt werden.



Properties (2)

```
> svn propset copyright '(c) 2009 example.com' main.cpp
property 'copyright' set on 'main.cpp'
> svn propget copyright main.cpp
(c) 2009 example.com
> svn proplist main.cpp
Properties on 'main.cpp':
  copyright
> svn status
M      main.cpp
> svn diff
Property changes on: main.cpp
-----
Name: copyright
+ (c) 2009 example.com
> svn commit -m "copyright set for main.cpp" main.cpp
Sending          main.cpp
Committed revision 5.
>
```



Properties (3) svn:ignore

Lästig:

```
> svn status
?      20091110_1014.log
?      20091110_0837.log
M      main.cpp
```

Lösung:

```
> svn propset svn:ignore "*log" .
property 'svn:ignore' set on '.'
> svn status
M      main.cpp
```

global-ignores setzt man in der `~/subversion/config` für Dateien, die nie in ein Repository gehören: `.*~ *.o *.swp` usw.



Keyword Substitution

Aus

```
// header.h  
// $Date$  
// $Revision$
```

mach

```
// header.h  
// $Date: 2009-11-12 16:14:59 +0100 (Thu, 12 Nov 2009) $  
// $Revision: 42 $
```

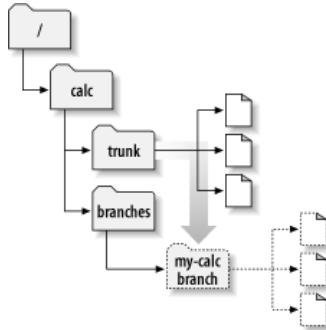
- \$Date\$
- \$Revision\$
- \$Author\$
- \$HeadURL\$
- \$Id\$

Keyword Substitution schaltet man mit der Property `svn:keywords` ein:

```
svn propset svn:keywords \ "Date Revision" header.h
```



Rocket Science



Konflikte (1)

Alice ändert

```
// hello.cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello Subversion!";
}
```

Bob ändert

```
// hello.cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello Svn!";
}
```

- Alice checkt ein.
- Bob will einchecken und erhält die Meldung:

```
svn: Commit failed (details follow): File 'hello.cpp' is out of date
```

- Bob führt `svn update` aus ...

```
Conflict discovered in 'hello.cpp'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```



Konflikte (2)

```
Conflict discovered in 'hello.cpp'.  
Select: (p) postpone, (df) diff-full, (e) edit,  
(h) help for more options:
```

- (p) postpone Datei im Konfliktzustand lassen.
- (df) diff-full Unterschiede anzeigen.
- (e) edit Datei im Konfliktzustand editieren.

```
#include <iostream>  
using namespace std;  
int main()  
{  
<<<<<<< .mine  
    cout << "Hello Svn!";  
=====  
    cout << "Hello Subversion!";  
>>>>>>> .r2  
}
```



Konflikte (3)

```
> ls
hello.cpp  hello.cpp.mine  hello.cpp.r1  hello.cpp.r2

> svn commit -m "changed hello-string"
svn: Commit failed (details follow):
svn: Aborting commit: '/home/bob/svn-work/hello.cpp'
      remains in conflict
```

- Den Konflikt lösen mit `svn resolve`
 - `working` Die aktuelle Version, Konflikt wurde manuell gelöst.
 - `base` Zurück zur Version vor dem Konflikt.
 - `mine-full` Meine Version annehmen.
 - `theirs-full` Die Version des Updates annehmen.

```
> svn resolve --accept working hello.cpp
```

Verwirft man seine Änderungen mit `svn revert`, löst dies auch den Konflikt.



Branch

Wenn man am Projekt aufwendig experimentieren möchte, legt man eine Verzweigung (Branch) an.

```
> svn copy file:///path/repository/trunk \  
           file:///path/repository/branches/my-branch \  
           -m "Creating a private branch."
```

Danach dann den Branch auschecken:

```
> svn checkout \  
           file:///path/repository/branches/my-branch
```

Wie gewohnt weiter arbeiten.



Merge

Einen Branch will man mit dem Stamm (Trunk) des Projekts aktuell halten. Dazu dient `svn merge`.

```
> cd my-branch
> svn merge \
    file:///path/repositor/trunk
U hello.cpp
```

`svn merge` schreibt immer auch die Property `svn:mergeinfo`. Diese beinhaltet Infos darüber, was bereits in den Branch gemerged wurde. `svn propget svn:mergeinfo`.

Tipp: Mit der Option `--dry-run` kann man den merge testen. Mit `svn merge --reintegrate` kann man einen Branch zurück in den Stamm (Trunk) mergen.



log

- Wir können uns die Kommentare zu den Revisionen mit `svn log` ansehen:

```
> svn log hello.cpp
-----
r3 | alice | 2009-11-12 15:33:55 (Do, 12 Nov 2009) | 2 lines
Hello Subversion!
-----
r2 | bob | 2009-11-12 15:27:12 (Do, 12 Nov 2009) | 2 lines
Hello World
-----
```



blame

```
> svn blame hello.cpp
2      bob #include <iostream>
2      bob
2      bob using namespace std;
2      bob
2      bob int main()
2      bob {
3      alice      cout << "Hello Subversion!" << endl;
2      bob }
2      bob
```

Wem `svn blame` nicht gefällt, kann seine Kollegen mit `svn praise` wertschätzen.



That's all, folks!

Vielen Dank!

Mehr Infos gibt es hier:

- `svn help`
- <http://svnbook.red-bean.com/>



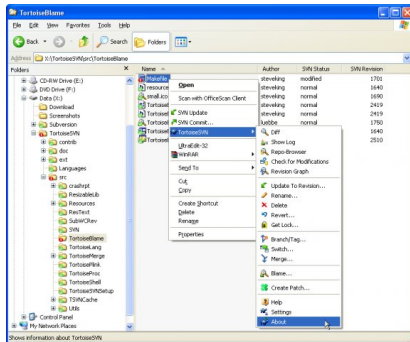
Anhang

Subversion in „bunt“



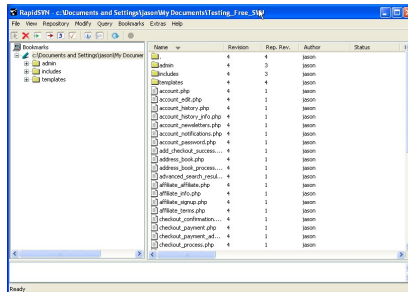
TortoiseSVN

- Plugin für den Windows Explorer
- <http://tortoisesvn.tigris.org/>



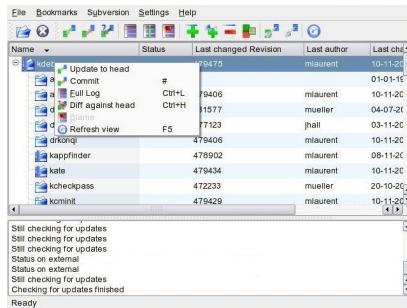
RapidSVN

- GUI für Subversion (gibts für Linux, Windows und Mac OS)
- <http://rapidsvn.tigris.org/>



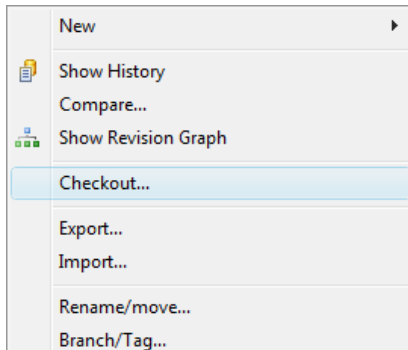
KDEsvn

- GUI für Subversion unter Linux, integrierbar in den Konqueror (KDE)
- <http://kdesvn.alwins-world.de/>



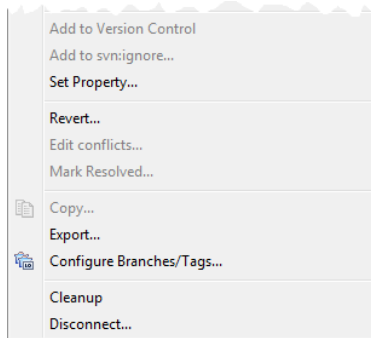
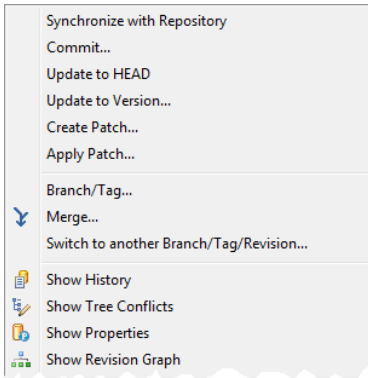
Subclipse (1)

- Plugin für Eclipse
- <http://subclipse.tigris.org/>



Subclipse (2)

- Hier finden sich alle* Befehle, die wir in svn kennen:



Subclipse (3)

- Diffs in Subclipse

```
Workspace file: AbstractJhlClientAdapter.java
}
notificationHandler.logCommandL
notificationHandler.setBaseDir(
    svnClient.lock(files, comment,
    for (int i = 0; i < files.length
        notificationHandler.notifyL
    )
} catch (ClientException e) {
    notificationHandler.logExceptio
// throw new SVNClientException(
}
}

Repository file: AbstractJhlClientAdapter.java
    svnClient.lock(files, comment
    // Workaround problem that ]
    boolean errors = false;
    ISVNStatus status[] = this.g
    for (int i = 0; i < status.l
        if (status[i].getLockOw
            errors = true;
            notificationHandler.
        } else {
            notificationHandler.
        }
    }
    if (errors)
        throw new SVNClientExcep
    } catch (ClientException e) {
```



Netbeans

- <http://www.netbeans.org> (auch für C++, PHP u.a.)

