

## Merkblatt für SVN, Funktionen und Kontrollstrukturen

**SVN** SVN steht für subversion und ist ein frei erhältlicher Standard für Versionskontrolle. Versionskontrolle ist notwendig, wenn mehrere Programmierer an einem System arbeiten und gleichzeitig Dateien editieren und verändern. Dann muss ein konsistenter Zustand des Gesamtsystems aufrechterhalten werden, wobei SVN behilflich sein kann. Das SVN verfügt über ein gemeinsames *Repository*, einem großen Pool, wo alle Daten liegen. Das Repository ist in Module aufgeteilt, die abgegrenzte Einheiten darstellen und sich gegenseitig nicht beeinflussen sollten. Ein Modul könnte beispielsweise den Quellcode für ein System beinhalten. Wenn nun ein Programmierer an dem System etwas verändern will, kann er sich die Dateien vom Repository auf seinen lokalen Rechner kopieren, er erstellt sich eine sogenannte *Working Copy*. In SVN-Sprache macht der Programmierer dann einen *checkout*. Seine kopierte Version kann er nun beliebig verändern; wenn er mit seinen Änderungen fertig ist (diese sollten den Code nach Möglichkeit immer lauffähig halten) kann er seine Version des Codes wieder in das Repository zurückschreiben. In SVN-Sprache sagt man, der Programmierer macht einen *commit*. Das SVN prüft jetzt, ob auch noch andere Programmierer dieselben Stellen im Code verändert haben und versucht einen konsistenten Zustand herzustellen. Falls dies nicht möglich ist, gibt es einen Konflikt und die Programmierer oder ein Versionsverwalter muss ihn manuell lösen.

Falls ein Modul schon ausgecheckt ist und wieder daran gearbeitet werden soll, sollte vorher ein *update* gemacht werden, um die Version des Moduls auf den aktuellsten Stand zu bringen.

Im folgenden werden alle relevanten SVN-Befehle kurz beschrieben. Im Linux-svn client können diese Befehle direkt nach dem executable `svn` geschrieben werden.

- `checkout <Modulname>`: Erzeugt eine working copy auf dem lokalen Rechner
- `update <Modulname | Dateiname | Ordner>`: Bringt die Datei/das Modul auf den aktuellen Stand wie er im Repository abgespeichert ist. Falls die eigene working copy aktueller als die Dateien im Repository sind, werden diese Änderungen nicht gelöscht.
- `add <Ordner | Datei>`: Fügt dem Repository eine Datei hinzu (nachher immer `svn commit` ausführen)
- `remove <Datei>`: Löscht eine Datei aus dem Repository (dafür darf sie in der working copy nicht mehr vorhanden sein, nachher immer `svn commit` ausführen)
- `commit <Datei | Ordner | Modul>`: Schreibt alle Änderungen von der lokalen working copy ins Repository.

Um eine SVN working copy auf einem lokalen Rechner zu speichern gibt es zwei Möglichkeiten. Man kann entweder das Kommandozeilentool und die oben aufgeführten Befehle oder eine graphische Oberfläche (s.u.) verwenden. Wenn man einen checkout von ausserhalb des TU Netzwerkes machen will, muss man den Pfad plus Netzwerk mit der `-d` Option angeben: Wenn man

beispielsweise das Modul `team1` von den Lehrstuhlrechner des Lehrstuhls 16 auschecken will, gibt man folgendes ein:

```
svn checkout  
svn+ssh://username@svnavab.informatik.tu-muenchen.de:/svn/progpraktss06/trunk/Team1
```

Hier muss natürlich statt `username` der jeweilige Benutzername und statt `team1` die jeweilige Teamnummer eingesetzt werden. Der String, der `svn` folgt, ist folgendermaßen aufgeteilt:

- `checkout` ist der `svn`-Befehl für einen checkout.
- `svn+ssh` steht dafür, dass eine *secure shell* zum repository-server aufgebaut werden soll.
- `username@svnavab.informatik.tu-muenchen.de` beschreibt den Benutzer, der eine *working copy* anlegen will und den remote host, auf dem das repository liegt
- `/svn/progpraktss06/trunk` beschreibt den Pfad auf dem remote host, wo sich das `svn` root-directory befindet, wo also alle module liegen.
- `Team1` ist der Name des Moduls/Verzeichnisses, das ausgecheckt werden soll.

Für diesen checkout muss auch eine Umgebungsvariable gesetzt sein, nämlich `SVN_EDITOR`. Um z.B. `vi` als Standardeditor zu setzen, muss ein `export SVN_EDITOR=vi` gesetzt werden. Um diesen export nicht immer wieder neu tätigen zu müssen, kann man unter Linux das `export` auch in die `.bashrc` Datei im Home-Verzeichnis schreiben, da diese beim Starten der Bash-Shell automatisch geladen wird.

Das Kommandozeilentool kann man unter Linux sowie unter Windows mit CygWin benutzen. In Windows gibt es allerdings ein nettes Tool mit Benutzeroberfläche, *TortoiseSVN*, das man sich kostenlos unter <http://tortoisesvn.tigris.org/> herunterladen kann. Von uns wird die Version 1.8 empfohlen, da erst ab dieser Version keine Konflikte mehr durch commits von Windows und Unix entstehen können.

Nach der Installation von TortoiseSVN kann man im Windows Explorer mit einem Rechtsklick das Menü *TortoiseSVN* erreichen. Unter *Settings* → *Network* stellt man als SSH-Client den von Tortoise mitgelieferten Client *TortoisePlink* ein. Nun erstellt man sich ein lokales Arbeitsverzeichnis und wählt mittels Rechtsklick, den Befehl *checkout*. Im erscheinenden Dialog muss das Verzeichnis des SVN Repositories (s.o.) angegeben werden. Mit den Befehlen *Update* und *Commit* kann man seine lokale Kopie mit dem Repository abgleichen. *Update* lädt die neueste Version vom Server und *Commit* die lokale Kopie auf den Server. Neu angelegte Dateien müssen zuvor mittels *Add* hinzugefügt und später mit *Commit* hochgeladen werden.

**Funktionen** Die Syntax von Funktionen kann fast eins-zu-eins von Javaprogrammen übernommen werden. Ein wichtiger Unterschied zu Java ist die Unterscheidung zwischen *Funktionsdeklaration* und *Funktionsdefinition*. Die Deklaration lässt den Compiler lediglich wissen, dass es eine Funktion mit bestimmtem Prototyp (Signatur) gibt. Eine Funktionsdefinition haucht der Funktion sozusagen Leben ein, d.h. sie beschreibt, welchen Algorithmus die Funktion bei ihrem Aufruf ausführen soll.

Eine Funktionsdeklaration einer Funktion `f` mit 3 Parametern (`x`, `y` und `z`) könnte folgendermaßen aussehen:

```
int f( float x, float y, float z );  
//Hier werden die Parameternamen explizit genannt  
  
int f( float, float, float );  
//Hier verzichtet man auf Parameternamen (reine Signatur)
```

Eine Funktionsdefinition dagegen muss immer den Parametern Namen geben, falls sie verwendet werden, was meistens sinnvoll ist. Sie hängt einen Rumpf an die Deklaration an. Eine Funktionsdefinition für die Funktion `f` könnte folgendermaßen aussehen:

```
int f( float x, float y, float z )  
{  
    x = y = z;  
}
```

Wieso trennt man nun Deklaration und Definition? Dies hat mehrere Gründe. Einer davon ist, dass man dadurch eine Schnittstelle für Programmierer anbieten kann, ohne dass man die wirkliche Implementierung Preis gibt. Ein anderer ist, dass bei jeder Definition Speicher vom Compiler allokiert wird, die obige Definition der Funktion `f` sagt dem Compiler, dass er `f` an dieser Stelle erzeugen soll, also Speicher dafür reservieren soll. Die Deklaration dagegen sagt dem Compiler nur, dass es solch eine Deklaration mit bestimmter Aufrufsignatur gibt, allerdings wird noch kein Speicher allokiert.

Aus diesen Gründen werden Funktionsdeklaration und -definition auch an verschiedenen Stellen im Code gestellt, meistens sogar in verschiedenen Dateien. C++ unterscheidet zwischen *header*-Dateien (mit Dateiondung `.h`) und *source*-Dateien (mit Dateiondung `.cpp`). In header-Dateien stehen u.a. die Funktionsdeklarationen, in source-Dateien die Funktionsdefinitionen.

Hierzu ist noch hinzuzufügen, dass diese Trennung von Deklaration und Definition nicht nur für Funktionen, sondern auch für Variablen gilt. Bei einer normalen Deklaration, z.B.

```
int i;
```

ist genügend Information vorhanden, um die Variable auch zu definieren, ihr also Speicherplatz zuzuweisen. Um dies zu vermeiden, muss ein gesondertes Schlüsselwort angegeben werden, das dem Compiler sagt, dass nur der Variablenname bekannt gemacht werden soll, nicht aber Speicher allokiert wird. Dieses Schlüsselwort ist `extern`:

```
extern int a; //Deklaration ohne Definition
```

```
int b; //Deklaration & Definition
```

**Kontrollstrukturen** Ebenso wie in Java kann der Programmfluss eines C++-Programms durch Kontrollstrukturen gesteuert werden. Es gibt auch hier bedingte Anweisungen, Wiederholungsanweisungen usw. Die Syntax ist hier dieselbe wie in Java, bzw. in Java dieselbe wie in C(++), um den Erfindern dieser Sprache Genüge zu tun.

- *if-else*

```
if( CONDITION )
{
    //mache irgendwas
}
else
{
    //mache etwas anderes
}
```

Ein Beispiel dafür wäre:

```
int a = 1;
int b = a+1;
if ( a < b )
{
    b = b - 1;
}
```

if-Statements können auch abgekürzt werden, mit einer etwas kryptischeren, aber dafür prägnanten Syntax:

```
( CONDITION ) ? /*mache irgendwas*/ : /*mache etwas anderes*/ ;
```

D.h. für obiges Beispiel könnte man ebenso schreiben:

```
b = ( a < b ) ? b-1 : b;
```

- *Schleifen*

```
for ( INIT; STOP; STEP )
{
    //mache etwas
}
```

```
while ( CONDITION )
{
    //mache etwas
}
```

```
do
{
    //mache etwas
} while ( CONDITION );
```

Ein Beispiel für einen Schleifendurchlauf wäre:

```
int i=1, c=10;
for (int j = c; j > 0; j--)
{
    i *= j; // bzw.: i = i*j;
}
```

- *switch-case* Manchmal will man verschiedene Fälle schnell abprüfen, die alle eine integer-Zahl als Repräsentanten haben<sup>1</sup>. Dafür verwendet man die switch-case-Anweisung:

```
switch ( INTEGER )
{
case 0:
    //mache etwas
case 1:
    //mache etwas anderes
//...
case n:
    //mache etwas tolles
default:
    //mache etwas ganz normales.
}
```

Im folgenden Beispiel wird gleich eine Datenstruktur eingeführt, die bei solchen Anweisungen immer gerne verwendet wird: *Enumerations*. Enumerations sind Aufzählungen, die jedem Wert dieser Aufzählung einen Integerwert zuweisen. Z.B. ist enum Wochentag { Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag }; solch eine Aufzählung. Der Compiler kennt nun die Wochentage und weist ihnen auch die Integerwerte 0-6 zu. Eine switch-case-Anweisung kann sich dieser Aufzählungen direkt bedienen:

```
enum Wochentag {
    Montag,
    Dienstag,
    Mittwoch,
    Donnerstag,
    Freitag,
    Samstag,
    Sonntag
};
//...
switch(Wochentag)
{
case Montag:
    cout << "Heute ist Montag." << endl;
    break;
case Dienstag:
    //...
}
```

In dem Beispiel wird auch gleich eine wichtige Fehlerquelle aufgezeigt: Will man die switch-case-Anweisung nach der Ausführung eines Falles verlassen, so muss man das Schlüsselwort `break` benutzen. Sonst werden die dahinterliegenden Fälle auch getestet, dies soll manchmal vermieden werden, insbesondere wenn man einen default-Fall hat.

---

<sup>1</sup>Eigentlich muss es nur eine ganze Zahl sein, nicht unbedingt eine integer-Zahl, d.h. es können auch alle anderen ganzzahligen Datentypen verwendet werden, z.B. `char` oder `unsigned short`