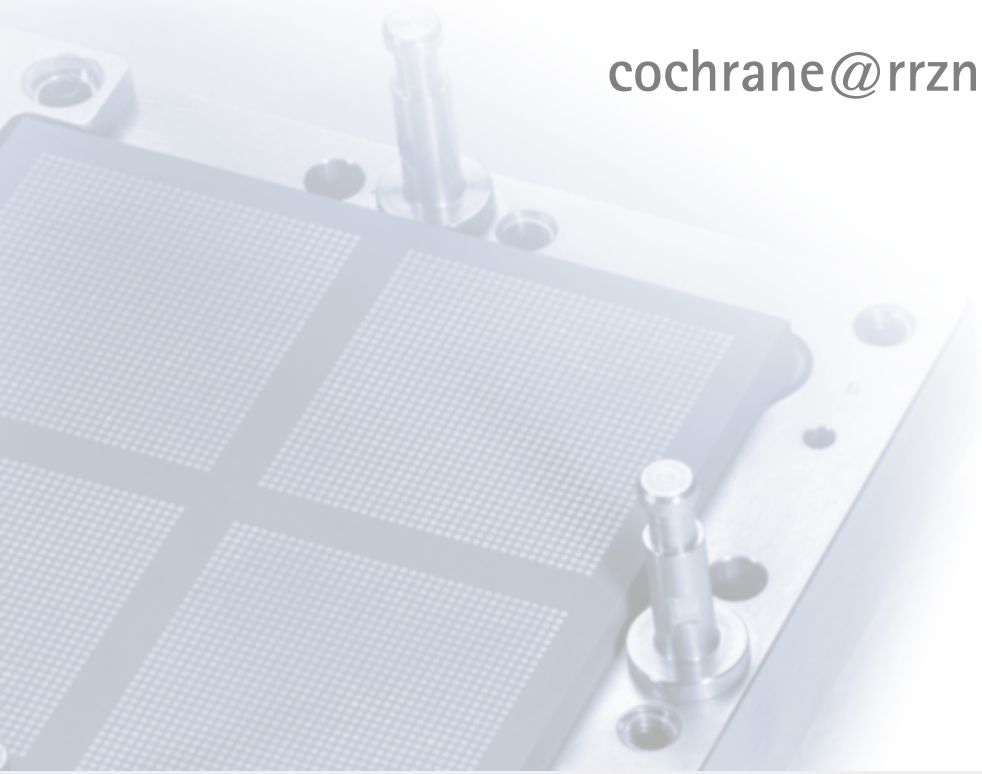
R|R|Z|N

Leibniz Universität Hannover

Versionskontrolle mit Subversion und Git

Dr. Paul Cochrane

cochrane@rrzn.uni-hannover.de



Einleitung

- Mein erstes Mal mit diesem Kurs
- »Versuchskaninchen«
- Feedback aller Art gut, meiner Meinung nach
- Linux Fokus; aber Konzepte sind überall gleich

Einleitung

- Mein erstes Mal mit diesem Kurs
- »Versuchskaninchen«
- Feedback aller Art gut, meiner Meinung nach
- Linux Fokus; aber Konzepte sind überall gleich

Kurs Ablauf

- Versionskontrolle im Allgemeinen
- Subversion
- Git

Ziele des Kurses

dass...

- mehr Leute Versionskontrolle benutzen
- Versionskontrolle besser benutzt wird
- man Grundkenntnisse in Subversion und Git bekommt
- man ein Einblick in die Möglichkeiten bekommt
- Erfahrung mit den Tools gesammelt wird

Was ist Versionskontrolle?

Verwalter/Beamter Organisiert und pflegt die Geschichte und verschiedene Versionen, Zweige und Abläufe eines Projekts.

Zeitmaschine Man kann alte Zustände von Dateien und gesamten Projekten »sehen« und wiederherstellen.

Quasi-Backup Wenn die Repository auf einem anderen Rechner (Server) gelagert wird, dann ist das eine Sicherung des Projekts.

Nutzung von Versionskontrolle

Wer benutzt Versionskontrolle?

Alle Leute, die auf ältere Versionen eines Dokuments zurückgreifen möchten. Oder Leute, die solche Situationen bereits erlebt haben:

- »Es wäre schön wenn ich die Version von vor 2 Stunden hätte...«
- »Ich hatte das vor 3 Tagen echt schön geschrieben. Wie ging das wieder?«
- »Oh nein! Ich habe die Datei gelöscht!«

In welchen Gebieten wird Versionskontrolle benutzt?

- Softwareentwicklung
- Text- und Dokumentbearbeitung
- Grafikbearbeitung/design
- Systemadministration

Nutzung von Versionskontrolle (fort.)

Was für Dateien sollten unter Versionskontrolle gehalten werden?

- Dateien, die geändert werden
- hauptsächlich Textdateien
  - Programmquelltext
  - Dokumentation
  - Dissertationen
- aber auch binär Dateien
  - Grafik-Dateien; `.png`, `.tiff`
  - Dokumente; `.pdf`, `.odt`

. . . und was nicht?

- automatisch erstellte Dateien, z.B.: `.o`, `.log`, `.pdf`
- »Backup«-Dateien, z.B.: `datei~`, `datei.bak`

Systeme für Versionskontrolle

Versionen in Dateinamen

Kommt dies vielleicht bekannt vor?

```
$ ls
datei.1  datei.20090803  datei.keep  datei.new  datei.old.2
datei.2  datei.alt       datei.neu   datei.old
```

Das ist besser als nichts, aber was ist zwischen den Versionen passiert? Welche ist eigentlich aktuell?

Automatische Versionsverwaltung

Es gibt auch mehrere Programme, die mit der Verwaltung und Organisierung von Dateien helfen können:

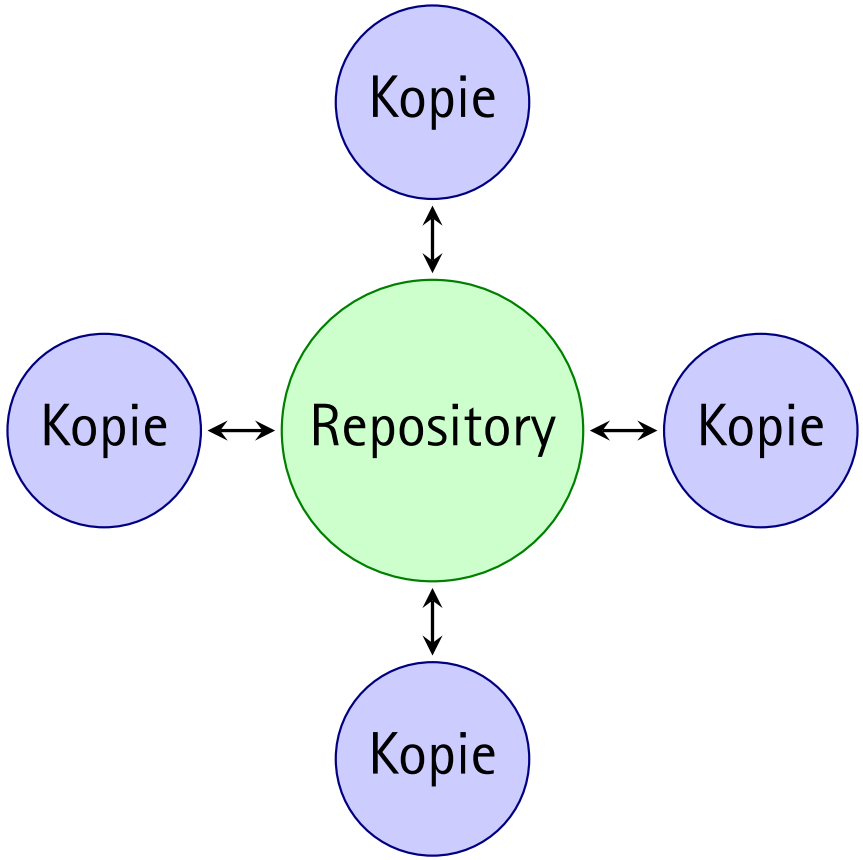
- SCCS, RCS, CVS, Subversion, Git, Mecurial, Arch, Darcs, SVK, ...

Editierungsmodelle

- lock-modify-unlock (RCS, auch CVS, Subversion)
  - Datei wird ausgecheckt und abgeschlossen; kann von nur einem Nutzer geändert werden
  - Datei wird modifiziert
  - Datei wird eingecheckt und aufgeschlossen
  - **Nachteil:** sehr inflexibel
  - **Vorteil:** mit Grafikdateien reduziert Arbeitsaufwand
- copy-modify-merge (CVS, Subversion, Git...)
  - Alle Dateien sind von der Repository kopiert und können ohne Hinderung modifiziert werden
  - Beim Einchecken werden Änderungen in Dateien automatisch mit anderen Änderungen zusammengeflossen
  - **Nachteil:** Konflikte zwischen Änderungen können auftauchen
  - **Vorteile:** sehr flexibel; mehrere Leute können gleichzeitig arbeiten

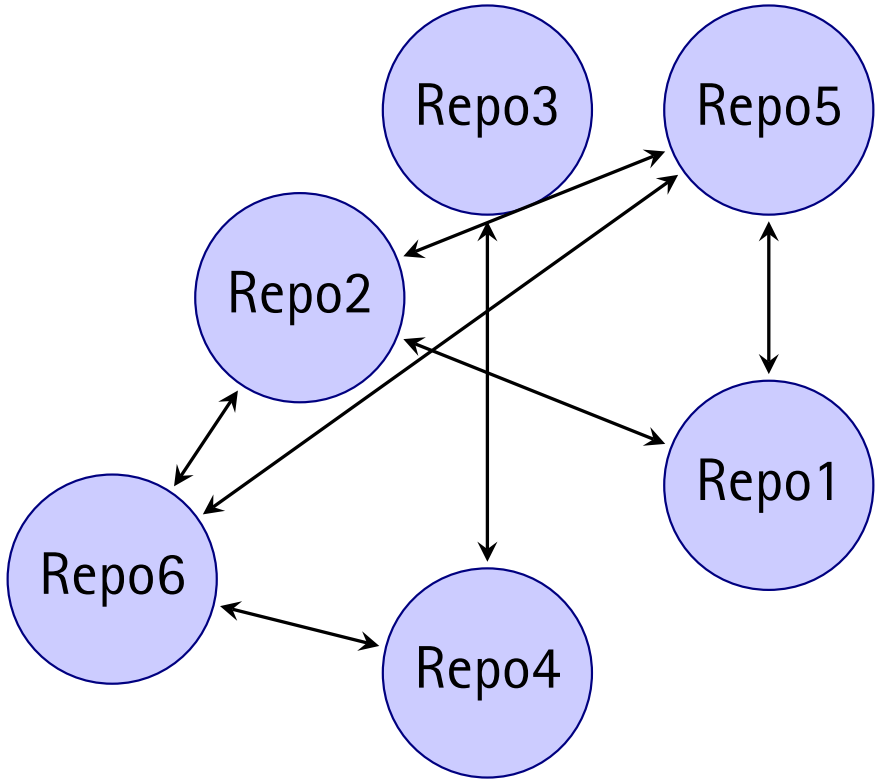
Repositorymodelle

- Centralised Repository Model (client-server)



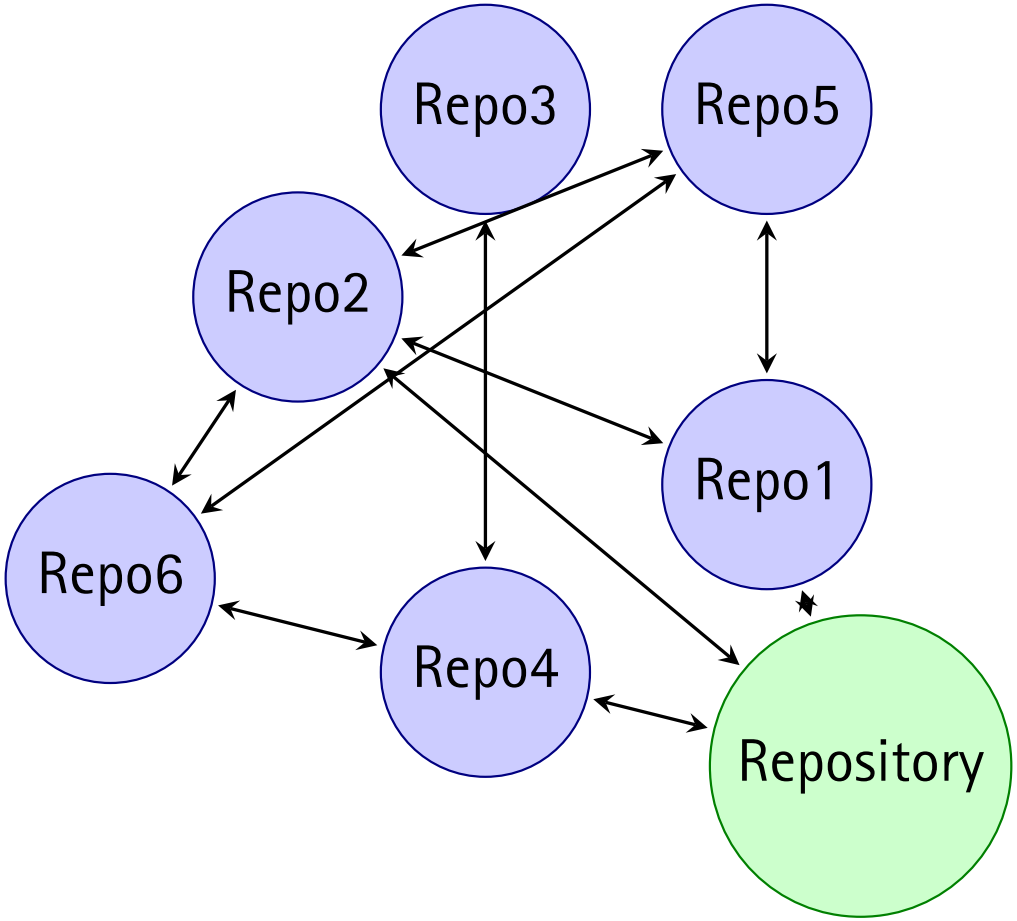
Repositorymodelle (fort.)

- Distributed Repository Model (pur); z.B. Linux Kernel



Repositorymodelle (fort.)

- Distributed Repository Model (mit zentralem Server); z.B. Github



Server Zugriff über SSH

- Alle Kursteilnehmer haben einen Account auf dem Kurs-Versionskontrolleserver (`vcs.rrzn.uni-hannover.de`).
- Benutzernamen haben diese Form: `vcskurs<x>` wo `<x>` ist eine Zahl
- Zugriff zum Server erfolgt über die Secure-Shell (`ssh`)

Zugriff testen und Passwort ändern:

```
$ ssh vcskurs<x>@vcs.rrzn.uni-hannover.de
$ passwd
$ exit
```

SSH-Passphrase

Eine Passphrase erleichtert den Zugriff auf einen entferten Server über die Secure-Shell. Mit der einmaligen Eingabe der Passphrase kann man sich mehrere Male auf dem Server (oder vielleicht mehrere Server) einloggen. Man muss das Passwort gar nicht eingeben! Aber: eine Passphrase ist viel länger als ein Passwort! Ein Beispiel:

```
Freude, schoene Goetterfunken.  Tochter aus Elysium.
```

Die Secure-Shell benutzt »Public-Key-Encryption«, also wir verteilen unsere öffentliche Schlüssel auf dem entferten Server und bauen eine verschlüsselte Verbindung auf. Über diese Verbindung werden unsere Dateien zum Server hin- und hergeschickt.

SSH-Passphrase (fort.)

Ja, aber warum? Ein sehr guter Weg Dateien übers Netz zu übertragen ist über `ssh`. Es ist, aber, komplett nervig das Passwort ständig eingeben zu müssen. Dies ist der Weg um eine verschlüsselte Verbindung zu haben und die Bedienung von Subversion (und Git gewissermaßen) zu erleichtern.

Private und öffentliche Schlüssel generieren

```
ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/rrzn1/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/rrzn1/.ssh/id_rsa.
Your public key has been saved in /home/rrzn1/.ssh/id_rsa.pub.
```

Public Key auf dem VCS-Server stellen

```
ssh-copy-id vcskurs15@vcs.rrzn.uni-hannover.de
```

SSH-Passphrase (fort.)

SSH-Agent initieren

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-PtRdMd5000/agent.5000; export SSH_AUTH_SOCK;
SSH_AGENT_PID=5001; export SSH_AGENT_PID;
echo Agent pid 5001;
```

Diese Ausgabe kopieren und an der Konsole einfügen

SSH-Passphrase eingeben

```
$ ssh-add
Enter passphrase for /home/rrzn1/.ssh/id_rsa:
Identity added: /home/cochrane/.ssh/id_rsa (/home/rrzn1/.ssh/id_rsa)
```

Sich auf dem Server einloggen zum Testen

```
$ ssh vcskurs15@vcs.rrzn.uni-hannover.de
```

Subversion–Geschichte

- SCCS–Source Code Control System
  - erstes Versionskontrollsystem
- RCS–Revision Control System
  - freie und mehr entwickelte alternative zu SCCS
  - lock-modify-unlock
  - Grundfunktionalität der Versionskontrolle
    - checkout, checkin, log, diff
  - Decentralised Repository Model
- CVS–Concurrent Versions System
  - benutzt RCS als »backend«
  - erlaubt Dateien, gleichzeitig modifiziert zu werden
  - viele RCS-Operationen in einem Zug
  - Client-Server-Modell
  - Centralised Repository Model
  - wurde Standard-Tool der (Open-Source) Softwareentwicklung
  - Zeit zeigte Design-Schwächen

Subversion–Geschichte (fort.)

- SVN–Subversion
  - »Unterversion« aber auch »Umsturz«
  - »a better CVS«; daher sehr ähnliche Struktur und »Gefühl«
  - Client-Server-Modell
  - Centralised Repository Model
  - korrigiert Design-Schwachstellen von CVS
  - bessere Unterstützung für binäre Dateien
  - reduzierte Kommunikation mit dem Server
  - Dateien können umbenannt werden; Geschichte kommt mit
  - »Undo« möglich; `svn revert`
  - Dateieigenschaften
    - Ausführbarkeit
    - binär/text
    - Schlüsselwörte: `Id`, `Author`, `Revision` ...
  - wurde zum Standard-Tool der (Open-Source) Softwareentwicklung

Subersion–Konzepte

- Zentralisiertes Repository-Modell
- Client-Server Modell

Arbeitskopien

- Auf dem Client checkt man eine Arbeitskopie aus
- Eine Arbeitskopie ist nur eine Kopie der Dateien im Repository
- Repository bleibt auf dem Server
- Man kann mehrere Arbeitskopien gleichzeitig ausgecheckt haben

Editoren

- Ein Editor wird gebraucht, um Nachrichten und Kommentare im System zu schreiben
- Die `EDITOR` Umgebungsvariable steuert, welcher benutzt wird.

```
$ export EDITOR=emacs
```

Subversion–Konzepte (fort.)

Verzeichnisstruktur

Subversion hat eine standardisierte Verzeichnisstruktur:

trunk Der Hauptzweig der Entwicklung. Häufig der Ort, wo am meisten gearbeitet wird.

branches Nebenzweige der Entwicklung. Häufig werden Ideen hier erarbeitet, um die Entwicklung auf dem Trunk/Hauptzweig nicht zu stören. Manchmal wird hier auch exclusiv gearbeitet und nur sehr stabil, sehr reife Ideen werden in den Trunk eingeflossen.

tags Software-Releases werden häufig »getagged« um einen menschenfreundlichen Namen zu geben. Software unter Wartung (maintenance releases) werden manchmal als Tag, manchmal als Branch weiterentwickelt.

Subversion–Hilfe!

Alle Subversion-Befehle sind an der Kommandozeile verfügbar. Man muss nur `svn help` benutzen:

```
svn help                 # zeigt alle Kommandos
svn help <subcommand>    # zeigt Hilfe fuer angegebenes Kommando
```

Zum Beispiel:

```
$ svn help commit
commit (ci): Send changes from your working copy to the repository.
usage: commit [PATH...]

  A log message must be provided, but it can be empty.  If it is not
  given by a --message or --file option, an editor will be started.
  If any targets are (or contain) locked items, those will be
  unlocked after a successful commit.

Valid options:
  -q [--quiet]             : print nothing, or only summary information
  -N [--non-recursive]     : obsolete; try --depth=files or
--depth=immediates

 ...
```

Schritte zum ersten Commit

- Subversion installieren

```
$ sudo aptitude install subversion
```

- ein existierendes Projekt importieren
  - `svn import`
  - passiert nur einmal für eine Repository
  - und/oder …
- ein existierendes Projekt auschecken
  - `svn checkout`
  - kann mehrere Male für eine Repository passieren
  - passiert nach einem »import«
  - kann auch passieren, wenn keine Dateien im Repository sind
  - um an einem Projekt teilzunehmen
  - um den Quelltext eines Projekts zu sehen
- Jetzt kann man anfangen zu arbeiten :-)

Ein neues Projekt importieren

Oft sind bereits Dateien für ein Projekt vorhanden. Daran hat man gearbeitet aber noch nicht unter Versionskontrolle. Dies könnte ein Softwareprojekt sein um »Hallo Welt« an der Konsole auszugeben.

Wir laden jetzt die Dateien im Projekt »hallo« herunter:

```
$ wget http://www.rrzn.uni-hannover.de/fileadmin/kurse/material/svn_git/hallo.tar.gz
$ tar -xvzf hallo.tar.gz
```

Hier ist die Liste von Dateien:

```
$ ls hallo/
hallo.c  Makefile
```

Auf dem Subversion-Server legt man eine neue Repository mit

```
$ svnadmin create /path/to/repo
```

an. Dies wird normalerweise vom Systemadministrator gemacht.

Ein neues Projekt importieren (fort.)

Um das `hallo` Projekt in Subversion zu importieren, benutzt man den `svn import` Befehl:

```
$ svn import -m "<Kommentar>" <Verzeichnis> <Repository URL>
```

```
$ svn import -m "Das hallo-Projekt importiert" hallo/ \
    svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/userprojs/vcskurs15/hallo
Adding         hallo/hallo.c
Adding         hallo/Makefile

Committed revision 1.
```

Das `hallo` Verzeichnis ist noch keine Arbeitskopie. Wir haben nur die Dateien in das Repository importiert. Um jetzt an dem Projekt zu arbeiten, muss man das Projekt aus dem Repository mit `svn checkout` auschecken.

```
$ svn checkout <Repository-URL>/<Projektname> <Arbeitskopiename>
```

Ein neues Projekt importieren (fort.)

Aber bevor wir das machen, sollten wir das alte `hallo` Verzeichnis zur Seite schieben:

```
$ mv hallo hallo_keep
```

Jetzt kann die Arbeitskopie ausgecheckt werden.

```
$ svn checkout \
  svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/userprojs/vcskurs15/hallo hallo
A    hallo/hallo.c
A    hallo/Makefile
Checked out revision 1.
```

Wir können jetzt überprüfen ob das geklappt hat. Das Standard-Linux-Befehl `diff` zeigt uns die Unterschiede zwischen dem importierten Verzeichnis und der Arbeitskopie.

```
$ diff hallo hallo_keep
Only in hallo: .svn
```

Ein neues Projekt importieren (fort.)

Das `.svn` Verzeichnis ist ein verstecktes Verzeichnis[1], wo Subversion Verwaltungsdateien (unter anderem) anlegt. Hier werden zum Beispiel alle Dateien in der Hauptversion des Repository gespeichert so dass Subversion nicht auf den Server (und daher das Netz) zugreifen muss um verschiedene Operationen machen zu können.

Aufräumen ist gut, daher löschen wir jetzt das alte Verzeichnis:

```
$ rm -r hallo_keep
```

Aufgabe–Ein neues Projekt importieren

- Die `hallo` Beispieldateien herunterladen
- Das Verzeichnis in Subversion importieren
- Das importierte Verzeichnis zur Seite schieben
- Eine Arbeitskopie auschecken
- Die Arbeitskopie mit dem alten Verzeichnis vergleichen (`diff`)
- Das alte Verzeichnis löschen

Lösung–Ein neues Projekt importieren

```
$ wget http://www.rrzn.uni-hannover.de/fileadmin/kurse/material/svn_git/hallo.tar.gz .
$ tar -xvzf hallo.tar.gz
$ svn import hallo/ svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/userprojs/vcskurs15/hallo \
    -m "Das hallo Projekt importiert"
$ mv hallo hallo_keep
$ svn checkout svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/userprojs/vcskurs15/hallo hallo
$ diff hallo hallo_keep
$ rm -r hallo_keep
```

Commit mal endlich!

- Die `hallo.c` Datei ändern
- Das Programm bauen

```
$ make
```

- Funktioniert das Programm noch?

```
./hallo
```

- Den Status der Arbeitskopie anschauen
  - `svn status` oder `svn st`

```
$ svn status
M       hallo.c
?       hallo
```

- Um die erzeugte `hallo` Datei kümmern wir uns später

Commit mal endlich! (fort.)

- Den Unterschied/ die Unterschiede der Datei anschauen
  - `svn diff` oder `svn di` oder `svn di <dateiname>`

```
$ svn diff
Index: hallo.c
===================================================================
--- hallo.c (revision 1)
+++ hallo.c (working copy)
@@ -1,6 +1,6 @@
 #include <stdio.h>

 int main(void) {
-    printf("Hallo, Welt!\n");
+    printf("Moin, moin!\n");
     return(0);
 }
```

  - Die Ausgabe ist in »unified diff« Format: `diff -u`
  - Ein »unified diff« zeigt nicht nur Änderungen, sondern auch (defaultmäßig) 3 Zeilen Kontextinformation

Commit mal endlich! (fort.)

- Funktioniert das Programm noch?

```
$ make
cc -o hallo hallo.c
$ ./hallo
Moin, moin!
```

- Ja! Wir können jetzt …
- Die Änderung in das Repository einchecken (committen)
  - `svn commit` oder `svn ci` oder `svn ci <dateiname>`
  - `ci` ist eine Abkürzung für »checkin«

```
$ svn commit -m "Hallo wurde zum Norddeutschland angepasst" hallo.c
Sending        hallo.c
Transmitting file data .
Committed revision 2.
```

- Unser erstes echtes Commit ist eingecheckt!

Commit-Konzepte

Commits sind die häufigsten und vielleicht die wichtigsten Operationen bei Versionskontrolle.

Wann sollte ich etwas einchecken/committen?

- Hängt vom Fall zu Fall auf …
- Wann eine kleine abgeschlossene Idee fertig ist. Z.B.: wann
  - eine neue Funktion geschrieben worden ist (auch nur ein Stub)
  - ein Bug beseitigt ist; nur den Code des Bugs einchecken
- Auch: »wann das sinnvoll ist«
- Kleine Commits (wenig Änderungen) sind einfacher für Menschen zu verstehen.
- Zusammenarbeit: Die Möglichkeit geben, Kritik zu äußern. Dadurch werden Probleme früher gefunden und beseitigt. Große Commits vermeiden, dass andere Leute überprüfen was gemacht worden ist.

Commit-Konzepte (fort.)

Wie oft soll ich committen?

- So häufig wie möglich
- Ähnlich wie »Release early, release often«
- Sobald die Änderungen, die zu einer bestimmten Idee gehören, abgeschlossen sind
- Habt keine Angst zu committen!

Commit-Messages

Bei jedem Commit wird eine Nachricht/Kommentar (»commit message«) geschrieben um das Commit zu beschreiben. Dies ist ziemlich wichtig, denn hier beschreibt man die Änderungen/ Ideen, die gerade bearbeitet wurden. Commit-Nachrichten sind auch Nachrichten an andere Leute. Dieser Mensch kannst in 6 Monaten/einem Jahr DU(!) sein.

Was sollte man in der Commit-Message schreiben?

- Eine knappe Beschreibung der Änderungen, die im Commit enthalten sind
- Die Beschreibung kann manchmal länger sein als die eingentliche Änderung!
- Die Nachricht sollte später noch erklären, worum es bei der Änderung ging
- Merken: Wenn Probleme auftreten, greift man auf diese Information zu

Commit-Messages wie `Added some code.` helfen keinem!

Arbeitsablauf

Vorbereitungen

- `svn import` (nur einmal)
- `svn checkout/co` (normalerweise einmal)

Hauptschritte

- `svn mkdir`
- `svn move/mv`
- `svn status/st`
- `svn diff/di`
- `svn add` (nur einmal pro Datei)
- `svn remove/rm`
- `svn log`
- `svn update/up`
- `svn commit/ci`

Hallo wieder

Lasst uns mit dem `hallo` Projekt ein bisschen mehr spielen. Aber diesmal wir arbeiten an dem Projekt zusammen.

Das gemeinsame Projekt auschecken: `svn checkout`/`svn co`

```
$ svn co svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo\
     hallo_gemeinsam
A    hallo_gemeinsam/hallo.c
A    hallo_gemeinsam/Makefile
Checked out revision 1.
```

Die Standard-Verzeichnisstruktur existiert in diesem Falle nicht; dies lege ich jetzt an:

```
$ svn mkdir trunk branches tags
A         trunk
A         branches
A         tags
```

Hallo wieder (fort.)

Den Status der Arbeitskopie anzeigen:

```
$ svn status
A       trunk
A       branches
A       tags
```

Dann kann ich die Dateien in den `trunk` Ordner schieben:

```
$ svn mv hallo.c trunk/
A         trunk/hallo.c
D         hallo.c
$ svn mv Makefile trunk/
A         trunk/Makefile
D         Makefile
```

Hallo wieder (fort.)

Der Status der Arbeitskopie ist jetzt interessanter geworden:

```
$ svn st
D       hallo.c
A       trunk
A  +    trunk/hallo.c
A  +    trunk/Makefile
A       branches
D       Makefile
A       tags
```

Merkmale:

- Verzeichnisse anzulegen ist ein Prozess in Subversion, nicht nur im Betriebssystem
- Ein »move« einer Datei ist zuerst am neuen Ort anlegen und am alten Ort löschen
- Die Geschichte einer Datei oder eines Verzeichnisses kommt mit der Datei zum neuen Ort. Dies war ein großer Schritt gegenüber CVS.

Hallo wieder (fort.)

Die Änderungen dann einchecken:

```
$ svn ci
Deleting       Makefile
Adding         branches
Deleting       hallo.c
Adding         tags
Adding         trunk
Adding         trunk/Makefile
Adding         trunk/hallo.c

Committed revision 2.
```

Hallo wieder (fort.)

Jetzt könnt ihr auf den aktuellsten Zustand kommen: `svn update`/`svn up`.

```
$ svn up
D    hallo.c
D    Makefile
A    trunk
A    trunk/hallo.c
A    trunk/Makefile
A    branches
A    tags
Updated to revision 2.
```

Wieder zeigt Subversion alles was er macht:

- Die `hallo.c` und `Makefile` Dateien werden vom aktuellen Verzeichnis gelöscht (`D`)
- Die `trunk`, `branches` und `tags` Verzeichnisse werden angelegt (`A`)
- Und die `hallo.c` und `Makefile` Dateien werden im `trunk` Verzeichnis angelegt (`A`).

Hallo wieder (fort.)

Wir wollen eine Liste von Sachen, die wir in das `hallo` Projekt einbauen wollen. Daher legen wir eine Datei namens `TODO` an:

```
$ cd trunk/
$ echo "# TODO-Liste fuer das hallo-Projekt" > TODO
$ svn status
?      TODO
```

Subversion kennt diese Datei noch nicht, deshalb ist in der ersten (linken) Spalte ein Fragezeichen.

Damit Subversion über die neue Datei Bescheid weiss, muss man die Datei hinzufügen:

```
$ svn add TODO
A         TODO
```

Hallo wieder (fort.)

Danach kann man die Änderung einchecken:

```
$ svn ci -m "Eine TODO-Liste hinzugefuegt" TODO
Adding         TODO
Transmitting file data .
Committed revision 3.
```

Wir können jetzt Ideen zu der TODO-Liste hinzufügen:

```
$ cat TODO
# TODO-Liste fuer das hallo-Projekt
 - hallo in anderen Sprachen
 - mehrmals "hallo" sagen
```

Den Status anschauen

```
$ svn st
M       TODO
```

Die Datei wurde modifiziert, deshalb steht ein `M` in der ersten Spalte.

Hallo wieder (fort.)

Einen Diff ansehen hilft, um sicherzustellen, dass das, was man committet, genau das ist, was man einchecken möchte.

```
$ svn di
Index: TODO
===================================================================
--- TODO    (revision 3)
+++ TODO    (working copy)
@@ -1 +1,3 @@
 # TODO-Liste fuer das hallo-Projekt
+ - hallo in anderen Sprachen
+ - mehrmals "hallo" sagen
```

Und wie gehabt, schieben wir diese Information zum Server hoch:

```
svn ci -m "Einige Ideen fuer Hallo Projekt aufgeschrieben"
Sending        trunk/TODO
Transmitting file data .
Committed revision 4.
```

Sehen, was wir gemacht haben

Um zu sehen, was in der Vergangenheit gemacht worden ist, benutzt man eine Kombination aus `svn log` (grobe Information) und `svn diff` (detaillierte Information).

```
$ svn log
------------------------------------------------------------------------
r3 | vcskurs15 | 2010-09-15 19:00:43 +0200 (Wed, 15 Sep 2010) | 1 line

Eine TODO-Liste hinzugefuegt
------------------------------------------------------------------------
r2 | vcskurs15 | 2010-09-15 18:24:12 +0200 (Wed, 15 Sep 2010) | 3 lines

Die Subversion Standardverzeichnisstruktur angelegt und die Dateien in den
Trunk verschoben

------------------------------------------------------------------------
```

Manchmal sieht man nicht alle Commits, die eingecheckt sind. Dies kommt, weil die Arbeitskopie noch nicht mit dem Server akualisiert ist. Ein `svn update` hilft.

Sehen, was wir gemacht haben (fort.)

```
$ svn log --verbose
------------------------------------------------------------------------
r5 | vcskurs15 | 2010-09-17 10:27:30 +0200 (Fri, 17 Sep 2010) | 2 lines
Changed paths:
   M /trunk/TODO
   M /trunk/hallo.c
Neue Sprachen hinzugefuegt
------------------------------------------------------------------------
r4 | vcskurs15 | 2010-09-15 19:22:59 +0200 (Wed, 15 Sep 2010) | 1 line
Changed paths:
   M /trunk/TODO
Einige Ideen fuer Hallo Projekt aufgeschrieben
------------------------------------------------------------------------
r3 | vcskurs15 | 2010-09-15 19:00:43 +0200 (Wed, 15 Sep 2010) | 1 line
Changed paths:
   A /trunk/TODO
Eine TODO-Liste hinzugefuegt
------------------------------------------------------------------------
r2 | vcskurs15 | 2010-09-15 18:24:12 +0200 (Wed, 15 Sep 2010) | 3 lines
Changed paths:
   D /Makefile
   A /branches
   D /hallo.c
   A /tags
   A /trunk
   A /trunk/Makefile (from /Makefile:1)
   A /trunk/hallo.c (from /hallo.c:1)
Die Subversion Standardverzeichnisstruktur angelegt und die Dateien in den
Trunk verschoben
------------------------------------------------------------------------
```

Sehen, was wir gemacht haben (fort.)

Unterschied zwischen Arbeitskopie und `HEAD`-Version.

```
$ svn diff [<dateiname>]
```

```
$ svn diff hallo.c
Index: hallo.c
===================================================================
--- hallo.c (revision 3)
+++ hallo.c (working copy)
@@ -1,6 +1,17 @@
+// hallo: ein Programm um "hallo!" zu sagen
 #include <stdio.h>

 int main(void) {
+    // Nord
     printf("Moin, moin!\n");
+    // Mittel
+    printf("Tach!\n");
+    printf("Halli hallo!\n");
+    // Sued
+    printf("Gruess Gott!\n");
+    // weiter Suedlich
+    printf("Gruetzi!\n");
+    // noch weiter Suedlich...
+    printf("Gudday!\n");
     return(0);
 }
```

Sehen, was wir gemacht haben (fort.)

Unterschied zwischen verschiedenen Versionen

```
$ svn diff -r M:N [<dateiname>]
```

```
$ svn di -r2:5 hallo.c
Index: hallo.c
===================================================================
--- hallo.c (revision 2)
+++ hallo.c (revision 5)
@@ -1,6 +1,17 @@
+// hallo: ein Programm um "hallo!" zu sagen
 #include <stdio.h>

 int main(void) {
+    // Nord
     printf("Moin, moin!\n");
+    // Mittel
+    printf("Tach!\n");
+    printf("Halli hallo!\n");
+    // Sued
+    printf("Gruess Gott!\n");
+    // weiter Suedlich
+    printf("Gruetzi!\n");
+    // noch weiter Suedlich...
+    printf("Gudday!\n");
     return(0);
 }
```

Sehen, was wir gemacht haben (fort.)

Unterschied einer Änderung (»change«)

```
$ svn diff -c N [<dateiname>]
```

```
$ svn di -c 4
Index: TODO
===================================================================
--- TODO    (revision 3)
+++ TODO    (revision 4)
@@ -1 +1,3 @@
 # TODO-Liste fuer das hallo-Projekt
+ - hallo in anderen Sprachen
+ - mehrmals "hallo" sagen
```

Wenn Probleme auftreten...

Es gibt viele Sachen, die tagtäglich passieren können, die nicht gewünscht sind.

- man löscht eine Datei versehentlich
- man hat viele Änderungen gemacht, die nicht mehr relevant sind
- man hat ein Commit gemacht, das fehlerhaft war

Mit Versionskontrolle kann man solche Probleme sehr leicht umgehen. In Subversion löst man solche Problem wie folgendes:

Wenn Probleme auftreten... (fort.)

Die versehentlich gelöschte Datei

Die Lösungen: `svn update` oder `svn revert`

Ein `update` holt die aktuelle Version der Datei von der `HEAD`-Revision.

```
$ rm hallo.c   # oops!
$ svn up       # dies geht mit 'svn revert' auch
Restored 'hallo.c'
At revision 6.
$ ls           # wieder da!
hallo.c  Makefile
```

Oder etwas schlimmer, man hat Subversion benutzt, um die Datei zu löschen:

```
$ svn rm hallo.c   # oops!
D         hallo.c
$ svn revert hallo.c
Restored 'hallo.c'
At revision 6.
$ ls               # wieder da!
hallo.c  Makefile
```

Wenn Probleme auftreten... (fort.)

Irrelevante Codeänderungen

Sagen wir mal, dass man etwas Ungewünschtes als Codeänderung macht.

```
$ svn di
Index: hallo.c
===================================================================
--- hallo.c (revision 6)
+++ hallo.c (working copy)
@@ -13,5 +13,7 @@
     printf("Gruetzi!\n");
     // noch weiter Suedlich...
     printf("Gudday!\n");
+    // noch weiter Suedlich
+    printf("Arf!  Arf!  Arf!\n");
     return(0);
 }
```

Man stellt danach fest, dass das vielleicht übertrieben war, und möchte das rückgängig machen. Die Lösung: `svn revert`

```
$ svn revert hallo.c
Reverted 'hallo.c'
```

Wenn Probleme auftreten... (fort.)

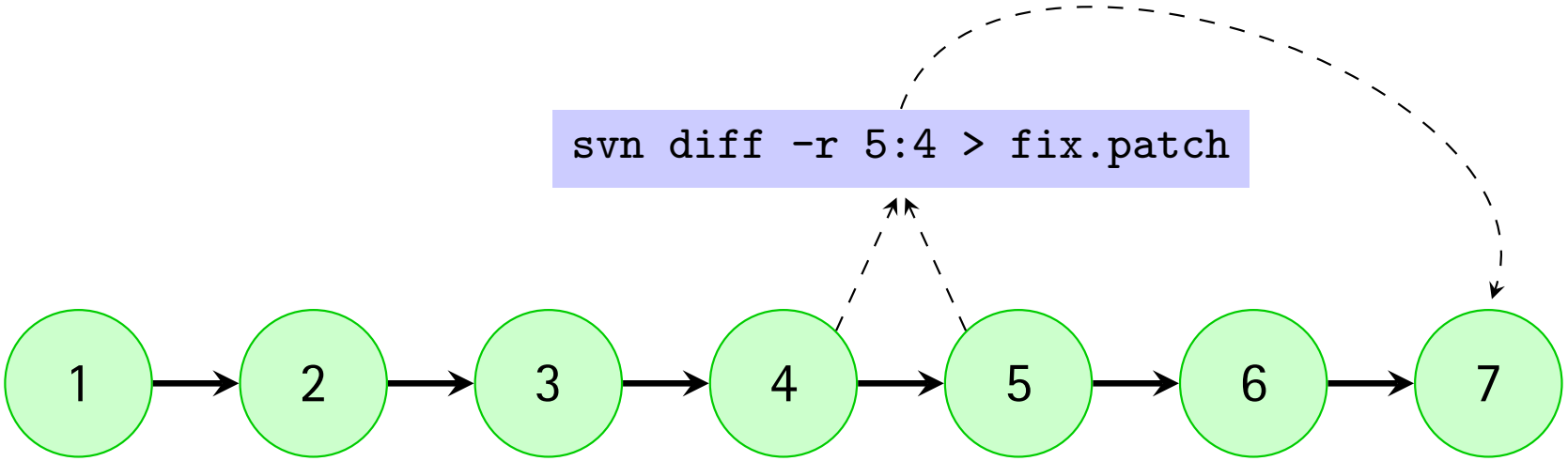
Das fehlerhafte Commit

Man stellt in Revision `<x>` fest, dass das Commit inkorrekt war, und möchte es rückgängig machen.

Die Lösung: man macht ein sogenanntes »rollback«. Es gibt mehrere Wege um ein »rollback« zu machen:

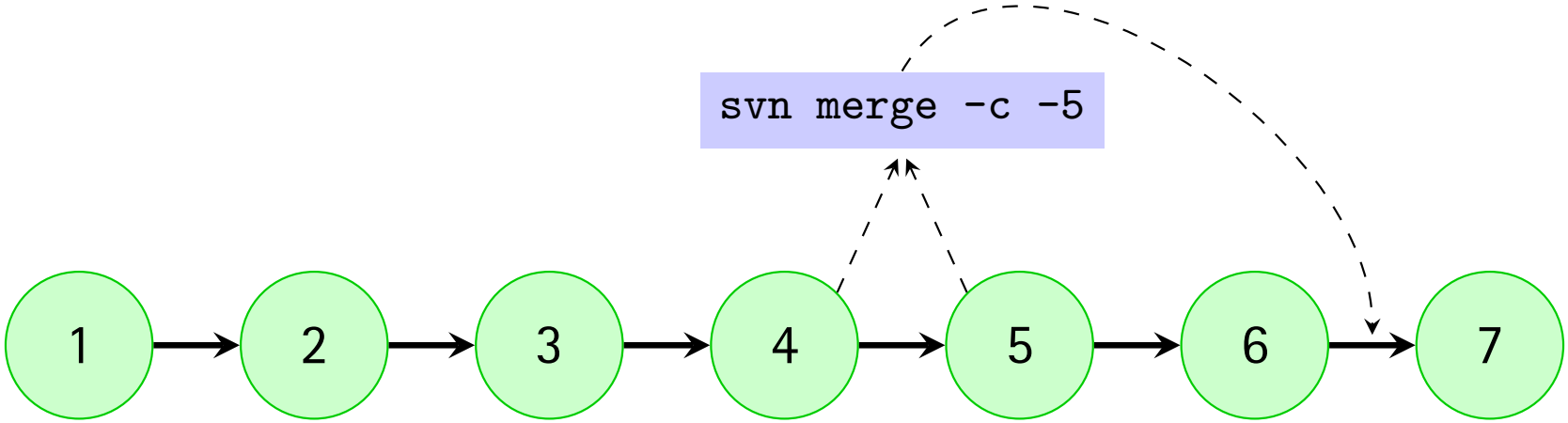
- `svn up -r N; cp datei datei.fixed; svn up; cp datei.fixed datei; svn ci`
- `svn di -r NEU:ALT > change.patch; patch -p0 < change.patch; svn ci`
- `svn merge -r NEU:ALT <URL>` oder `svn merge -c -NEU <URL>`; dann `svn ci`. Geht mit Subversion Version 1.5+. Siehe auch Branching und Merging später.

Rollback mit `svn diff`




```
$ svn diff -r 5:4 > fix.patch
$ patch -p0 < fix.patch
$ svn ci
Committed revision 7.
```

Rollback mit svn merge




```
$ svn merge -c -5
$ svn ci
Committed revision 7.
```

Konflikte

Wenn man mit anderen Leuten arbeitet, und/oder wenn man mit mehreren Arbeitskopien arbeitet, kann es vorkommen, dass man den selben Teil eines Dokuments wie jemand anders editiert und, dass die Änderungen nicht automatisch zusammenfließen können. Dies ist ein Nachteil des Simultan-Editierungs-Modells aber die Vorteile sind (für Textdateien) in der Mehrzahl. Um Konflikte zu beheben, braucht man Menschen und vielliecht auch Kommunikation zwischen Menschen. Subversion gibt auch Unterstützung um Konflikte zu beheben. Die geänderten Textstücke werden eindeutig markiert und der `svn resolved` Kommando steht zur Verfügung.

Wir versuchen dies mit dem `hallo` Projekt zu machen.

Wir haben zwei Arbeitskopien, `hallo_gemeinsam` und `hallo_gemeinsam2`.

Konflikte (fort.)

Sind die Arbeitskopien aktuell?

```
$ cd hallo_gemeinsam
$ svn up
U    trunk/hallo.c
U    trunk/TODO
Updated to revision 6.
$ cd ../hallo_gemeinsam2
$ svn up
At revision 6.
```

Wir ändern eine Zeile in `hallo.c` in `hallo_gemeinsam2`:

```
$ svn di hallo.c
Index: hallo.c
===================================================================
--- hallo.c (revision 6)
+++ hallo.c (working copy)
@@ -12,6 +12,6 @@
     // weiter Suedlich
     printf("Gruetzi!\n");
     // noch weiter Suedlich...
-    printf("Gudday!\n");
+    printf("Gudday, mate!\n");
     return(0);
 }
```

Konflikte (fort.)

Dann checken wir die Änderung ein:

```
$ svn ci -m "Ein Gruss jetzt mehr Neuseelaendisch" hallo.c
Sending        hallo.c
Transmitting file data .
Committed revision 7.
```

Jetzt ändern wir die selbe Zeile in `hallo_gemeinsam`:

```
$ svn di
Index: hallo.c
===================================================================
--- hallo.c (revision 6)
+++ hallo.c (working copy)
@@ -12,6 +12,6 @@
     // weiter Suedlich
     printf("Gruetzi!\n");
     // noch weiter Suedlich...
-    printf("Gudday!\n");
+    printf("G'day, mate!\n");
     return(0);
 }
```

Konflikte (fort.)

Aber, das geht doch wohl nicht! Die Änderung in der Arbeitskopie bezieht sich auf Revision 6 und die aktuellste Revision ist 7. *Und* dieselbe Zeile wurde geändert! Genau! Deshalb macht man ein `svn update` häufig während man arbeitet, insbesondere bevor man versucht etwas einzuchecken. Aber das hilft uns auch nicht in diesem Falle:

```
$ svn up
Conflict discovered in 'hallo.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

Was tun? Es gibt mehere Wege.

- die Datei jetzt editieren
- die Konflikte später beheben
- die Revision des anderen einfach so nehmen
- die Revision von sich selbst einfach so nehmen
- ein `diff` machen und dann entscheiden

Konflikte (fort.)

Zuerst mehr Hilfe abfragen:

```
$ svn up
Conflict discovered in 'hallo.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: h
  (p)  postpone     - mark the conflict to be resolved later
  (df) diff-full    - show all changes made to merged file
  (e)  edit         - change merged file in an editor
  (r)  resolved     - accept merged version of file
  (mf) mine-full    - accept my version of entire file (ignore their changes)
  (tf) theirs-full - accept their version of entire file (lose my changes)
  (l)  launch       - launch external tool to resolve conflict
  (h)  help         - show this list

Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

Konflikte (fort.)

`diff-full`: zeit ein »unified diff«

```
--- .svn/text-base/hallo.c.svn-base Sat Sep 18 12:33:44 2010
+++ .svn/tmp/tempfile.tmp   Sat Sep 18 12:52:40 2010
@@ -12,6 +12,10 @@
     // weiter Suedlich
     printf("Gruetzi!\n");
     // noch weiter Suedlich...
-    printf("Gudday!\n");
+<<<<<<< .mine
+    printf("G'day, mate!\n");
+=======
+    printf("Gudday, mate!\n");
+>>>>>>> .r7
     return(0);
 }
```

`edit`: zeigt quasi das gleiche, aber in einem Editor. Man kann danach ab sofort die `r` (»resolved«) Option wählen um das Konflikt zu beheben.

`mine-full`: nur meine Änderungen sind für die Datei gültig

`theirs-full`: nur die Änderungen des Anderen sind gültig; meine Änderungen werden weggeworfen

Konflikte (fort.)

postpone

```
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (h) help for more options: p
C    hallo.c
$ svn st
?      hallo
?      hallo.c.r6
?      hallo.c.mine
?      hallo.c.r7
C      hallo.c
```

Wie sieht `hallo.c` jetzt aus?

```
$ cat hallo.c
// hallo: ein Programm um "hallo!" zu sagen

......

    printf("Gruetzi!\n");
    // noch weiter Suedlich...
<<<<<<< .mine
    printf("G'day, mate!\n");
=======
    printf("Gudday, mate!\n");
>>>>>>> .r7
    return(0);
}
```

Konflikte (fort.)

Die Datei zeigt uns was »meine« Änderung ist, und was aktuelle in der Repository ist (in diesem Falle Revision 7) und markiert die Unterschiede mit `<<<<<<< .mine`, `=======` und `>>>>>>> .r7`.

Man sollte auch merken, dass das Programm nicht mehr kompiliert.

```
$ make
cc -o hallo hallo.c
hallo.c: In function 'main':
hallo.c:15: error: expected expression before '<<' token
hallo.c:17: error: expected expression before '==' token
hallo.c:19: error: expected expression before '>>' token
make: *** [hallo] Error 1
```

Das ist gut, weil wir nicht wollen, dass diese Änderungen eingecheckt werden, während das Programm noch nicht funktioniert.

Konflikte (fort.)

Wir editieren die Datei und müssen entscheiden welche Änderungen genommen werden sollen. Dies bedeutet auch, dass man vielleicht mit dem anderen Meschen reden muss :-). Es kommt vor, dass die Änderungen, die vorgenommen werden sollen, nicht eindeutig sind. In diesem Falle wollen wir aber die neuseeländische Version und nicht die australische Version.

Aber die Datei kann immer noch nicht eingecheckt werden.

```
$ svn ci
svn: Commit failed (details follow):
svn: Aborting commit: '/home/cochrane/hallo_gemeinsam/trunk/hallo.c' remains in conflict
```

Wir müssen Subversion sagen, dass das Konflikt jetzt behoben ist: `svn resolved <dateiname>`.

```
$ svn resolved hallo.c
Resolved conflicted state of 'hallo.c'
```

Da ist jetzt nichts einzuchecken, weil die Arbeitskopieversion die gleiche ist wie im Repository.

Aufgabe–Konflikte erzeugen und beheben

- Alle checken eine `hallo` Arbeitskopie aus
- Alle editieren und checken wieder ein
- Konflikte erzeugen und beheben
- `svn status`
- `svn diff`
- `svn commit`
- `svn resolved`

Eigenschaften

- Wie ignoriert man unnötige Dateien?
- Meine Kollegen benutzen Windows, wie passen unsere Textdateien zusammen?
- Mein Skript ist ausführbar, wie gebe ich Subversion Bescheid?
- Wie organisiere ich binäre Dateien, die ich einchecken möchte?
- Ich möchte, dass meine Quelltextdateien automatisch Revision- und Autorinformationen beinhalten. Wie geht das?

Alle diese Fragen sind durch Subversion-Eigenschaften (»`svn properties`«) beantwortet. Subversion kann nicht nur die Änderungen zu Dateien verwalten, sondern auch Eigenschaften von Dateien und Verzeichnissen.

Eigenschaften–Dateien ignorieren

Automatisch erzeugte Dateien sollten normalerweise nicht unter Versionskontrolle gehalten werden. Zum Beispiel:

- Objektdateien, Bibliotheken und Programme: `*.o`, `*.so`, `hallo` usw.
- LaTeXs automatisch erzeugte Dateien: `*.log`, `*.aux`, `*.toc`, `*.dvi`
- Backupdateien von Editoren: `*~`

Wir erweitern das `hallo` Projekt: beim Übersetzen wird eine Objektdatei `*.o` erzeugt.

```
$ ls
hallo  hallo.c  hallo.c~  hallo.o  Makefile  Makefile~  TODO
```

`svn status` zeigt uns die Dateien, die Subversion nicht kennt

```
$ svn st
?      hallo
```

Eigenschaften–Dateien ignorieren (fort.)

Subversion erkennt automatisch, dass die `*~` und `*.o` Dateien ignoriert werden können (sie sind standard automatisch erzeugten Dateien) und zeigt diese nicht an. Aber das `hallo` Programm wird immer noch angezeigt mit einem Fragezeichen in der ersten Spalte, um uns zu sagen, dass die Datei Subversion nicht bekannt ist.

Um Dateien in einem Verzeichnis zu ignorieren, müssen wir die `svn:ignore` Eigenschaft des Verzeichnisses entprechend ändern. Der Befehl um eine Eigenschaft zu editieren ist:

`svn propedit <eigenschaft> <verzeichnis>`

```
$ svn propedit svn:ignore .
Set new value for property 'svn:ignore' on '.'
```

Eigenschaften–Dateien ignorieren (fort.)

Es könnte sein, dass die Repository inzwischen geändert wurde, und deshalb müssen wir ein `svn update` machen. Ansonsten, sieht man den folgenden Fehler:

```
$ svn ci -m "Automatisch erzeugte Dateien werden ignoriert"
Sending        trunk
svn: Commit failed (details follow):
svn: Directory '/trunk' is out of date
```

Wie gesagt, Subversion verwaltet auch die Eigenschaften von Dateien und Verzeichnissen. Das heisst, dass wir auch an Verzeichnissen ein `svn diff` machen können:

```
$ svn di -c 9

Property changes on: .
___________________________________________________________________
Added: svn:ignore
   + hallo
*.o
*~
```

Eigenschaften–Dateien ignorieren (fort.)

Das ist eine standard `svn diff` Ausgabe, mit `+` wo Sachen hinzugefügt sind und `-` wo Sachen gelöscht sind.

Man sieht auch, dass man auch Wildcards benutzen kann in der `svn:ignore` Eigenschaft.

Wichtig

Immer `svn propedit` benutzen wenn Dateien ignoriert werden sollen.

Eigenschaften–Windows und Unix arbeiten zusammen

- Zeilen umbrüche in Textdateien auf Windows enden mit CRLF
- Zeilen umbrüche in Textdateien auf Unix enden mit CR

Auf Linux haben Windows-Textdateien ein `^M`-Zeichen am Ende jeder Zeile. Auf Windows, Linux-Textdateien haben gar keine Umbrüche! Die `svn:eol-style`-Subversion-Eigenschaft sorgt für Frieden zwischen den Welten. `svn propset svn:eol-style <style> <datei>`.

`svn:eol-style native` Für CRLF auf Windows und nur CR auf Unix.

`svn:eol-style CRLF` Wenn nur CRLF gewünscht ist.

`svn:eol-style CR` Wenn nur CR gewünscht ist.

`svn:eol-style LF` Wenn nur LF gewünscht ist.

```
$ svn ps svn:eol-style native hallo.c
property 'svn:eol-style' set on 'hallo.c'
```

Eigenschaften–Windows und Unix arbeiten zusammen (fort.)

Diese Änderung natürlich ändert die Datei; nicht den Inhalt, sondern eine Eigenschaft. Deshalb zeigt `svn status`, dass die Datei modifiziert wurde durch ein `M` in der *zweiten* Spalte.

```
$ svn st
 M      hallo.c
```

Natürlich müssen solche Änderungen auch eingecheckt werden

```
$ svn ci -m "EOL-Style auf native gesetzt" hallo.c
Sending        hallo.c

Committed revision 10.
```

Eigenschaften–ausführbare Dateien

Perl-, Python-, Shell- (usw.) Skripte sind häufig direkt ausführbar. Dies kann man in Unix mit dem `chmod`-Befehl setzen. Subversion aber sieht diese Dateien als nur Textdateien. Man möchte natürlich dass wo auch immer eine Arbeitskopie ausgecheckt wird, sie genauso funktioniert wie alle anderen. Deshalb setzt man die `svn:executable` Eigenschaft auf solchen Dateien.

`svn propset svn:executable <on/off> <dateiname>`

```
$ svn ps svn:executable on hallo.pl
property 'svn:executable' set on 'hallo.pl'
$ ls -l hallo.pl
-rwxr-xr-x 1 cochrane cochrane 65 2010-09-18 16:31 hallo.pl
$ svn ci -m "Ausfuehrbar Bit gesetzt" hallo.pl
Sending        hallo.pl

Committed revision 12.
$  ./hallo.pl
Moin!
```

Eigenschaften–binäre Dateien

Meistens merkt Subversion richtig ob eine Datei Text oder binäre Daten enthällt. Manchmal aber nicht. Zum Beispiel bei PDF-Dokumenten. In diesem Fall setzt man selber die `svn:mime-type` Eingeschaft.

`svn propset svn:mime-type <MIME-Beschreibung>`

```
$ svn add hallo.png hallo.svg
A  (bin)  hallo.png
A         hallo.svg
A         hallo.pdf
```

Eine PDF-Datei sollte z.B. unter Versionskontrolle sein (sie wird irgendwann geändert und die Geschichte der Datei ist uns wichtig), aber Subversion hatte nicht gemerkt, dass die Datei binäre Daten enthält.

Eigenschaften–binäre Dateien

Wir setzen also die `svn:mime-type` Eigenschaft:

```
$ svn ps svn:mime-type application/octet-stream hallo.pdf
property 'svn:mime-type' set on 'hallo.pdf'
$ svn ci
Adding  (bin)  trunk/hallo.pdf
Adding  (bin)  trunk/hallo.png
Adding         trunk/hallo.svg
Transmitting file data ...
Committed revision 13.
```

Jetzt weiß Subversion Bescheid und die Datei wird korrekt eingecheckt.

Eigenschaften–automatische Schlüsselwörte

Es ist gängig in manchen Softwareentwicklungsprojekten, dass man in jeder Datei den Autor, die Revision und das Datum der letzte Änderung schreibt. Dies kann man in Subversion durch bestimmte Schlüsselwörter und die `svn:keywords` Eigenschaft ermöglichen.

Wir fügen die folgenden Zeilen in `hallo.c` ein:

```
// $Id$
// $Author$
// $Revision$
// $Date$
```

Dies ändert eigentlich nichts von Subversion aus[2]. Man muss die Schlüsselwörte mit der `svn:keywords` Eigenschaft aktivieren.

```
$ svn ps svn:keywords "Author Revision Date" hallo.c
property 'svn:keywords' set on 'hallo.c'
```

Eigenschaften–automatische Schlüsselwörte (fort.)

Wir haben beide Inhalt und Eigenschaften geändert, so wir haben ein `M` in der ersten zwei Spalten in `svn status`:

```
$ svn st
MM     hallo.c
```

Ein `svn diff` zeigt auch was geändert wurde

```
$ svn di
Index: hallo.c
===================================================================
--- hallo.c (revision 10)
+++ hallo.c (working copy)
@@ -1,4 +1,8 @@
 // hallo: ein Programm um "hallo!" zu sagen
+// $Id$
+// $Author$
+// $Revision$
+// $Date$
 #include <stdio.h>

 int main(void) {

Property changes on: hallo.c
___________________________________________________________________
Added: svn:keywords
   + Author Revision Date
```

Eigenschaften–automatische Schlüsselwörte (fort.)

Wo sind diese magischen automatischen Schlüsselwörter? Die tauchen nur auf nachdem man die Änderung eingecheckt hat.

```
$ svn ci
Sending        trunk/hallo.c
Transmitting file data .
Committed revision 14.
$ cat hallo.c
// hallo: ein Programm um "hallo!" zu sagen
// $Id$
// $Author: vcskurs15 $
// $Revision: 14 $
// $Date: 2010-09-18 17:26:56 +0200 (Sat, 18 Sep 2010) $
```

Wo ist das Schlüsselwort für `Id`? Das wurde nicht gesetzt und daher nicht expandiert. (Ehrlich gesagt ist das `Id`-Keyword in den meisten Fällen alles was nötig ist, weil es eine kompakte Version der anderen Schlüsselwörter enthält.

Eigenschaften–automatische Schlüsselwörte (fort.)

Die unnötigen Zeilen löschen und die `svn:keywords`-Eigenschaft umsetzen:

```
$ vim hallo.c
$ svn ps svn:keywords "Id" hallo.c
property 'svn:keywords' set on 'hallo.c'
$ svn di
Index: hallo.c
===================================================================
--- hallo.c (revision 14)
+++ hallo.c (working copy)
@@ -1,8 +1,5 @@
 // hallo: ein Programm um "hallo!" zu sagen
 // $Id$
-// $Author$
-// $Revision$
-// $Date$
 #include <stdio.h>

 int main(void) {
Property changes on: hallo.c
___________________________________________________________________
Modified: svn:keywords
   - Author Revision Date
   + Id
```

Die alten Eigenschaften sind gelöscht und nur `Id` wurde hinzugefügt und man sieht das in dem `svn diff`. Cool!

Eigenschaften–automatische Schlüsselwörte (fort.)

Und die Änderung wird eingecheckt und das `Id`-Schlüsselwort entsprechend expandiert:

```
$ svn ci -m "Nur Id-Keyword ist noetig" hallo.c
Sending        hallo.c
Transmitting file data .
Committed revision 15.
$ cat hallo.c
// hallo: ein Programm um "hallo!" zu sagen
// $Id: hallo.c 15 2010-09-18 15:31:47Z vcskurs15 $
```

Eigenschaften–Sonstiges

Man kann mehr als nur Eigenschaften editieren und setzen. Man kann die auch löschen, holen und auflisten.

propedit Eigenschaften editieren
propset Eigenschaften setzen/einstellen
propdel Eine bestimmte Eigenschaft löschen
propget Eine bestimmte Eigenschaft holen
proplist Alle Eigenschaften einer Datei auflisten

```
$ svn propget svn:mime-type hallo.png
application/octet-stream
```

```
$ svn proplist hallo.c
Properties on 'hallo.c':
  svn:keywords
  svn:mergeinfo
  svn:eol-style
$ svn propget svn:eol-style hallo.c
native
```

Branches, Tags und Merging

Eine der mächtigsten Fähigkeiten eines Versionskontrollsystems ist die Möglichkeit mehrere parallele Entwicklungsrichtungen zu haben und diese »unter einem Hut« zu halten.

Beispiele:

Feature-Branches: für die Entwicklung neuer Fähigkeiten

- stören die anderen Entwicklungen nicht
- man sieht die Änderungen, die passieren und kann Rückmeldung geben
- normalerweise haben sie eine kurze Lebensdauer

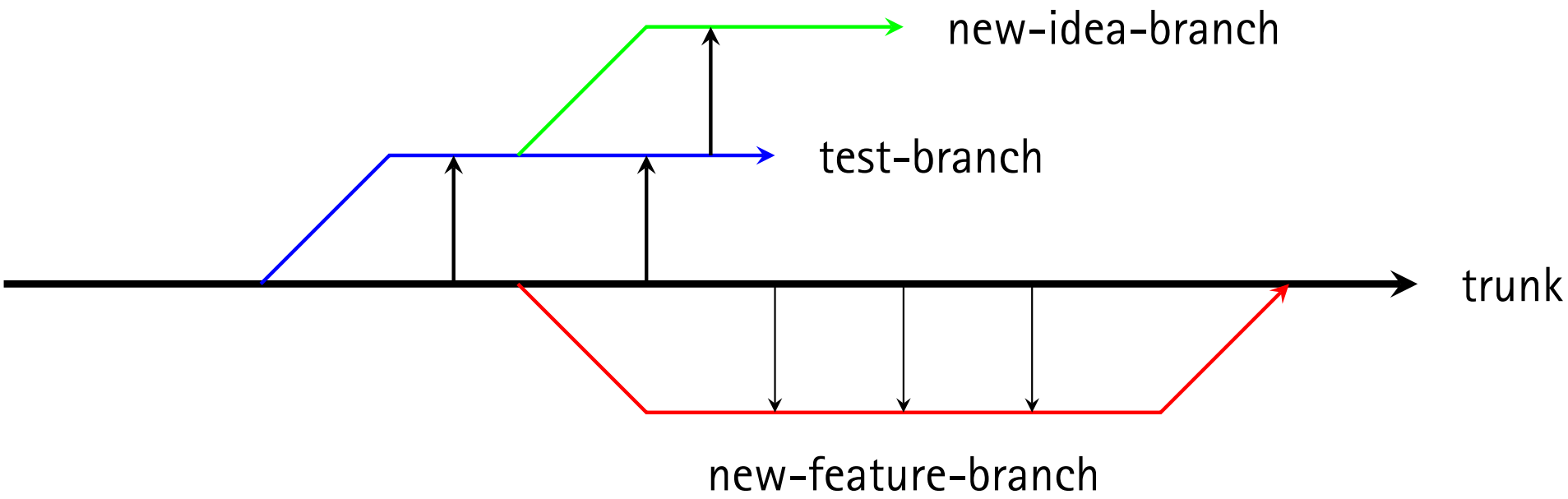
Maintenance-Branches: langfristige Wartung einer Software

- Bugs (gefunden im Wartungsbranch) können in Trunk einfließen
- Bugs (gefunden im Trunk) können in den Wartungsbranch einpflegt werden
- Nicht so statisch wie ein Tag (menschlich gesehen)

Branches, Tags und Merging (fort.)

Wie man Branches benutzt ist mehr oder weniger eine Frage der Policy des Projektes. Zum Beispiel:

- Branches sind "stable"; Trunk nur für Entwicklung
- Trunk ist "stable"; Branches nur für Entwicklung



Branches, Tags und Merging (fort.)

Ein neuer Branch wird in Subversion durch eine Kopier-Operation erzeugt:

`svn copy <quell-url> <ziel-url> -m «kommentar>"`

Dies passiert nur serverseitig; man muss den Branch danach auschecken durch entweder `svn checkout` oder `svn update`.

```
$ svn copy \
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/trunk \
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/branches/test-branch

Committed revision 16.
```

`svn checkout`

```
$ svn co
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/branches/test-branch
hallo_test-branch
A    hallo_test-branch/hallo.c
A    hallo_test-branch/hallo.png
A    hallo_test-branch/TODO
A    hallo_test-branch/hallo.pdf
A    hallo_test-branch/hallo.pl
A    hallo_test-branch/Makefile
A    hallo_test-branch/hallo.svg
 U   hallo_test-branch
Checked out revision 16.
```

Branches, Tags und Merging (fort.)

svn update

```
$ svn up
A    branches/test-branch
A    branches/test-branch/hallo.c
A    branches/test-branch/hallo.png
A    branches/test-branch/TODO
A    branches/test-branch/hallo.pdf
A    branches/test-branch/hallo.pl
A    branches/test-branch/Makefile
A    branches/test-branch/hallo.svg
Updated to revision 16.
```

Branches, Tags und Merging (fort.)

Tags

Tags sind quasi nur ein menschliches Kontrukt. Die Operation in Subversion ist identisch, der Unterschied ist in wie man als Mensch mit Tags und Branches umgeht. Ein Branch wird häufig gesehen als etwas, dass eine begrenzte Lebensdauer hat, weil die Entwicklung irgendwann wieder in den Trunk einfließen wird (und der Branch endet dort) oder die Entwicklung zeigt sich als unnötig und endet. Ein Tag wiederum wird häufig für "Releases" von Software benutzt und existiert daher dauerhaft.

»Tagging a release«

```
$ svn copy\
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/trunk\
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/tags/hallo_version_1.0\
-m "Tagging version 1.0"

Committed revision 17.
```

Branches, Tags und Merging (fort.)

Merging

Wenn die Hauptentwicklung im Trunk passiert und Branches sind »Feature-Branches«, ist es eine gute Idee, dass wenn man an einem Branch arbeitet, die neuesten Änderungen vom Trunk periodisch in den Branch hineinfließen (»merging«). Dies verhindert, dass die Branch-Entwicklung zu weit weg wandert von der Trunk-Entwicklung und es reduziert die Schwierigkeiten, die auftauchen könnten, wenn man die Branch-Entwicklung wieder in den Trunk einfließen möchte.

Seit Subversion 1.5, errinnert sich Subversion welche Merges vom Trunk bereits passiert sind, und nur die neuesten Änderungen werden vom Trunk geholt. Dies ist auch wichtig wenn man einen Branch in den Trunk einfließen lassen möchte: nicht alle vorherigen Merges sollen wieder auf den Trunk eingespielt werden!

Branches, Tags und Merging (fort.)

Als ein Beispiel von Merging können wir den `hallo_test-branch` benutzen, um das C-Programm in Perl nachzubauen.

```
$ svn di
Index: hallo.pl
===================================================
--- hallo.pl    (revision 16)
+++ hallo.pl    (working copy)
@@ -3,4 +3,14 @@
 use strict;
 use warnings;

-print "Moin!\n";
+# Nord
+print "Moin, moin!\n";
+# Mittel
+print "Tach!\n";
+print "Halli hallo!\n";
+# Sued
+print "Gruess Gott!\n";
+# weiter Suedlich
+print "Gruetzi!\n";
+# noch weiter Suedlich...
+print "Gudday, mate!\n";
```

```
$ ./hallo.pl
Moin, moin!
Tach!
Halli hallo!
Gruess Gott!
Gruetzi!
Gudday, mate!
$ svn ci -m "Perl macht C nach"
Sending        hallo.pl
Transmitting file data .
Committed revision 18.
```

Branches, Tags und Merging (fort.)

Die Entwicklung am Trunk geht weiter...

```
$ svn log
------------------------------------------------------------------------
r21 | vcskurs15 | 2010-09-18 20:49:34 +0200 (Sat, 18 Sep 2010) | 2 lines

hallo.py jetzt ausfuehrbar

------------------------------------------------------------------------
r20 | vcskurs15 | 2010-09-18 20:48:30 +0200 (Sat, 18 Sep 2010) | 2 lines

Makefile nicht mehr noetig

------------------------------------------------------------------------
r19 | vcskurs15 | 2010-09-18 20:47:48 +0200 (Sat, 18 Sep 2010) | 2 lines

C ist doof.  Entwicklung geht weiter in Python
```

Wir ziehen die Änderungen in den Branch...

```
$ svn merge \
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/trunk
--- Merging r16 through r21 into '.':
A    hallo.py
D    hallo.c
D    Makefile
```

Branches, Tags und Merging (fort.)

und arbeiten weiter.

```
$ svn st
 M      .
D       hallo.c
A  +    hallo.py
M       hallo.pl
D       Makefile
$ svn ci -m "Flexibler?"
Sending        .
Deleting       Makefile
Deleting       hallo.c
Sending        hallo.pl
Adding         hallo.py
Transmitting file data .
Committed revision 22.
$ svn ci -m "Doch etwas flexibler"
Sending        hallo.pl
Transmitting file data .
Committed revision 23.
```

Branches, Tags und Merging (fort.)

Wir sind bereit, zurück in den Trunk zu mergen, und machen ein letztes Merge vom Trunk

```
$ svn merge\
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/trunk
--- Merging r22 through r24 into '.':
U    hallo.py
$ svn up   # wir muessen aktuell sein...
At revision 24.
$ svn st
 M     .
M      hallo.py
$ svn ci -m "Neueste Aenderungen vom Trunk geholt"
Sending        .
Sending        hallo.py
Transmitting file data .
Committed revision 25.
```

Wir können jetzt unseren Branch mit dem Trunk zusammenführen

```
$ cd hallo_gemeinsam/trunk
$ svn up   # wir muessen aktuell sein...
$ svn merge --reintegrate
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/branches/test-branch
--- Merging differences between repository URLs into '.':
U    hallo.pl
 U   .
```

Branches, Tags und Merging (fort.)

Commit noch einmal

```
$ svn st
 M      .
M       hallo.pl
$ svn ci -m "merged test-branch mit trunk"
Sending        trunk
Sending        trunk/hallo.pl
Transmitting file data .
Committed revision 26.
```

Wir brauchen den Test-Branch nicht mehr, so wir können den ruhig löschen (die Geschichte des Branches bleibt immer noch in der Repository).

```
$ svn delete
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo/branches/test-branch
-m "test-branch loeschen"

Committed revision 27.
```

Aufgabe–Entwicklung mit Branches

- Das `hallo` Projekt auschecken
- Einen neuen Zweig anfangen
- Themen:
  - zufällige Grüße
  - Menüsystem
  - Grüße in mehreren Sprachen und aus mehreren Regionen
  - Überrascht mich!

Versionskontrolle in der Systemadministration

Obwohl Versionskontrolle häufig in der Softwareentwicklung benutzt wird, bedeutet es längst nicht, dass man sie nicht auch in anderen Gebieten verwenden kann.

Ein Gebiet, wo Versionskontrolle sehr wichtig ist, ist die Systemadministration. Man möchte die Konfiguration(sdateien) von vielen Programmen in Ordnung halten. Versionskontrolle gibt zusätzliche Unterstützung.

Aufgabe–Ein bestehendes leeres Projekt auschecken

- Beispiel-Konfigurationsdateien herunterladen und auspacken

```
$ wget http://www.rrzn.uni-hannover.de/fileadmin/kurse/material/svn_git/apach
```

- Ins Verzeichnis wechseln
- Arbeitskopie auschecken (Tipp: man kann eine leere Arbeitskopie ins aktuelle Verzeichnis auschecken ohne die bestehenden Dateien dadurch zu stören)
- `svn status`
- `svn add`
- `svn commit`

Lösung–Ein bestehendes leeres Projekt auschecken

```
$ svn co
svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/userprojs/vcskurs15/apache2 .
$ svn status
?      sites-enabled
?      magic
?      mods-available
?      envvars
?      sites-available
?      apache2.conf
?      httpd.conf
?      conf.d
?      mods-enabled
?      ports.conf
$ svn add apache2.conf httpd.conf ports.conf
A         apache2.conf
A         httpd.conf
A         ports.conf
$ svn ci
Adding         apache2.conf
Adding         httpd.conf
Adding         ports.conf
Transmitting file data ...
Committed revision 1.
```

Weitere Subversion-Fähigkeiten und Themen

- Herausfinden, wer welche Zeile geändert hat und in welche Revision: `svn blame <dateiname>`; auch `praise`, `annotate`
- Ein Projekt exportieren ohne die `.svn` Verzeichnisse zu haben; z.B. wenn man eine Release macht: `svn export`
- Dateilocking: `svn lock`, `svn unlock`
- Emails verschicken nach jedem Commit: Commit-Hooks; passiert serverseitig, wird vom Systemadministrator in der `post-commit` Datei eingestellt
- Andere Repositories von anderen Projekten automatisch auschecken: `svn:external` Eigenschaft
- Wenn man mehrere Dateien geändert hat, die zu unterschiedlichen Ideen gehören, kann man diese Dateien quasi beschriften mit dem Namen der Ideen: `svn changelist <changelist-name> <dateien...>`

Ressourcen

- Das Subversion-Buch
  - O'Reilly Media
  - http://svnbook.red-bean.com/
- Subversion Webseite: http://subversion.tigris.org/
- TortoiseSVN; Subversion für Windows: http://tortoisesvn.tigris.org/
- Wikipedia-Eintrag: http://en.wikipedia.org/wiki/Apache_Subversion/

GIT

Git–Geschichte

- Kommt aus der Linux-Kernel-Entwicklung
- Früher durften alle Linux-Kernel-Entwickler das propriatäre Versionskontrollsystem “BitKeeper” kostenlos benutzen
- Bitkeeper wurde im Jahr 2005 (weiter) eingeschränkt, so dass es nicht mehr so frei war
- Linus Torvalds war damit unglücklich und schrieb sein eigenes System

Ziele der Git-Entwicklung

- Verteilte Entwicklung
- Scalierbar bis auf tausende von Entwicklern
- Schnell und effizient
- Integrität und Vertraulichkeit aufrecherhalten
- Erzwungene Verantwortung
- Unveränderbare Objekte
- Atomische Operationen
- Entwicklung mit Branches unterstützen und fördern
- Komplette Repositories
- Sauber Entwurf
- Frei wie Freiheit

Konzepte

- Grob gesagt, ein versioniertes Dateisystem
- Das Versionskontrollsystem ist darauf entwickelt
- Die Arbeitskopie ist ein Repository(!)
- 130+ Befehle
- "porcelain" und "plumbing" Befehle
- Verteiltes Repositorymodell kann aber auch zentral benutzt werden (z.B. Github)
- Sehr schnell!

Wie Subversion, müssen wir Git erst installieren:

```
$ aptitude install git-core git-doc
```

Git auf Windows

- Das Cygwin-Paket: http://www.cygwin.com
- msysGit: http://code.google.com/p/msysgit

SHA1-Hashes

- SHA1-Hash spezifiziert *den gesamten Repository-Zustand* am Zeitpunkt des Commits
- Im normalen Fall ist nur das erste Stückchen nötig um ein Commit zu spezifizieren. Z.B. `f011c01`

Blobs, Trees und Commits

blob »Binary large objects«; quasi nur Dateien

tree Bäume von Dateien

commit Versionierter Zustand eines Baumes

tags Menschenfreundliche Namen eines Commits

Hilfe von Git

`git` oder `git -help`

```
$ git --help
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [-p|--paginate|--no-pager] [--bare]
[--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
   add         Add file contents to the index
   bisect      Find the change that introduced a bug by binary search
   branch      List, create, or delete branches
   checkout    Checkout a branch or paths to the working tree
   clone       Clone a repository into a new directory
   commit      Record changes to the repository
   diff        Show changes between commits, commit and working tree, etc
   fetch       Download objects and refs from another repository
   grep        Print lines matching a pattern
   init        Create an empty git repository or reinitialize an existing one
   log         Show commit logs
   merge       Join two or more development histories together
   mv          Move or rename a file, a directory, or a symlink
   pull        Fetch from and merge with another repository or a local branch
   push        Update remote refs along with associated objects
   rebase      Forward-port local commits to the updated upstream head
   reset       Reset current HEAD to the specified state
   rm          Remove files from the working tree and from the index
   show        Show various types of objects
   status      Show the working tree status
   tag         Create, list, delete or verify a tag object signed with GPG

See 'git help COMMAND' for more information on a specific command.
```

Hilfe von Git

Unter Unix sind alle Befehle durch die `man`-Pages verfügbar. Man benutzt aber dabei einen Bindestricht zwischen `git` und dem Kommando. Zum Beispiel:

- `man git-init`
- `man git-add`
- `man git-commit`

und so weiter...

Schnell an die Arbeit–Ein neues Projekt importieren

Das `hallo`-Projekt wieder erneut herunterladen und auspacken

```
$ wget http://www.rrzn.uni-hannover.de/..../hallo.tar.gz
$ tar -xvzf hallo.tar.gz
```

Eine neue Repository anlegen: `git init`

```
$ cd hallo
$ git init
Initialized empty Git repository in /home/cochrane/hallo/.git/
```

Das war's!

Wir brauchen keinen Server und keine Netzverbindung. Man könnte schön entspannt in einem Cafe sitzen, Kaffee schlürfen und arbeiten!

Schnell an die Arbeit–Ein neues Projekt importieren (fort.)

Wir haben noch nichts unserem Repository, aber das macht nichts aus. Wir können jetzt sehen, wie der Zustand der Arbeitskopie aussieht

`git status`

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Makefile
#       hallo.c
nothing added to commit but untracked files present (use "git add" to track)
```

Git gibt uns einen Tipp, was zunächst zu tun ist.

Schnell an die Arbeit–Ein neues Projekt importieren (fort.)

Um Dateien ins Repository zu bringen, muss man sie zuerst hinzufügen (genauso wie bei Subversion):

`git add <dateiname>` oder `git add .`

```
$ git add .
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: Makefile
#       new file: hallo.c
#
```

Die Dateien sind jetzt zum Index hinzugefügt, aber noch nicht im Repository. Man sagt, dass die Dateien »staged« sind. Dies ist ein nützliches Konzept, weil wir vielleicht manche Dateien »unstagen« möchten. Git sagt uns auch, wie man das macht.

Schnell an die Arbeit–Ein neues Projekt importieren (fort.)

In diesem Falle wollen wir ein Commit machen.

`git commit [<dateiname> ...]` oder `git commit -a`

```
$ git commit -a -m "Erstes Import des Hallo-Projektes"
Created initial commit f011c01: Erstes Import des Hallo-Projektes
 2 files changed, 16 insertions(+), 0 deletions(-)
 create mode 100644 Makefile
 create mode 100644 hallo.c
```

Merkmale:

- Erste Zeile der Commit-Nachricht wird gezeigt
- Keine Revisionnummern; Commits spezifiziert durch SHA1-Hashes. Ein Abschnitt des SHA1-Hashes wird gezeigt: `f011c01`. Dies dient als Referenz zum Commit.
- Dateimodi werden auch angezeigt. 644 bedeutet:
  - lesbar und schreibbar für den Besitzer der Datei (6)
  - lesbar für die Gruppe des Besitzers (4)
  - lesbar für alle andere Nutzer (4)

Konfiguration

`git config <eigenschaft> <wert>`

Lokal für diese Repository

```
$ git config user.name "VCS Kurs 15"
$ git config user.email "fake@gmx.de"
```

Global für alle Repositories und in `$HOME/.gitconfig` gespeichert

```
$ git config --global user.name "Paul Cochrane"
$ git config --global user.email "cochrane@rrzn.uni-hannover.de"
```

`git config --list`

Im `hallo` Arbeitskopie:

```
$ git config --list
user.email=cochrane@rrzn.uni-hannover.de
user.name=Paul Cochrane
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.name=VCS Kurs 15
user.email=fake@gmx.de
```

Direkt im HOME-Verzeichnis:

```
$ git config --list
user.email=cochrane@rrzn.uni-hannover.de
user.name=Paul Cochrane
```

Arbeitsablauf

Sehr ähnlich wie bei Subversion

Vorbereitungen:

- `git init` (nur einmal)
- `git checkout`
- `git config`

Hauptschritte:

- `git add` (nur einmal pro Datei)
- `git mv`
- `git status`
- `git diff`
- `git rm`
- `git log`
- `git show`
- `git commit`

Sehen, was wir gemacht haben

Wir fügen wieder andere Sprachen zur `hallo.c` hinzu, bauen das Programm und testen, ob es noch funktioniert. Was wurde geändert? `git diff`

```
$ git diff
diff --git a/hallo.c b/hallo.c
index 008ac22..df60c79 100644
--- a/hallo.c
+++ b/hallo.c
@@ -2,6 +2,9 @@
 #include <stdio.h>

 int main(void) {
-    printf("Hallo, Welt!\n");
+    // nord
+    printf("Moin, moin!\n");
+    // sued
+    printf("Gudday, mate!\n");
     return(0);
 }
```

Der Unterschied ist zwischen dem Repository und der Arbeitskopie. Mehr spezifisch: zwischen dem `master`-Branch und der Arbeitskopie. Im Allgemeinen: zwischen dem aktuellen ausgecheckten Branch und der Arbeitskopie. Git ist etwas allgemeiner als Subversion mit Branches.

Sehen, was wir gemacht haben (fort.)

Wir checken die Änderungen ein

```
$ git commit -a
Created commit 0d964bb: Eine suedliche Variante hinzugefuegt
 1 files changed, 4 insertions(+), 1 deletions(-)
```

- Das SHA1-Hash ist anders als im `diff` weil das Commit den Zustand der Repository geändert hat und so ein neues Hash erzeugt hat.

`git log`

```
$ git log
commit 0d964bbeceec6d461e8d98a1d6e844300d66d30e
Author: VCS Kurs 15 <fake@gmx.de>
Date:   Sun Sep 19 14:19:00 2010 +0200

    Eine suedliche Variante hinzugefuegt

commit f011c01e07c19299df6e9db2c757e317c285a6b9
Author: Paul Cochrane <cochrane@nb206ptc.rrzn.uni-hannover.de>
Date:   Sun Sep 19 12:19:42 2010 +0200

    Erstes Import des Hallo-Projektes
```

Sehen, was wir gemacht haben (fort.)

`git diff <hash1> <hash2> [<dateiname>]`

```
$ git diff \
    0d964bbeceec6d461e8d98a1d6e844300d66d30e\
    f011c01e07c19299df6e9db2c757e317c285a6b9
diff --git a/hallo.c b/hallo.c
index df60c79..008ac22 100644
--- a/hallo.c
+++ b/hallo.c
@@ -2,9 +2,6 @@
 #include <stdio.h>

 int main(void) {
-    // nord
-    printf("Moin, moin!\n");
-    // sued
-    printf("Gudday, mate!\n");
+    printf("Hallo, Welt!\n");
     return(0);
 }
```

```
$ git diff 0d964bb f011c01
diff --git a/hallo.c b/hallo.c
index df60c79..008ac22 100644
--- a/hallo.c
+++ b/hallo.c
@@ -2,9 +2,6 @@
 #include <stdio.h>

 int main(void) {
-    // nord
-    printf("Moin, moin!\n");
-    // sued
-    printf("Gudday, mate!\n");
+    printf("Hallo, Welt!\n");
     return(0);
 }
```

Zeigen, was wir haben

git show [<blob>|<tree>|<commit>|<tag>]

```
$ git show hallo.c
commit 0d964bbeceec6d461e8d98a1d6e844300d66d30e
Author: VCS Kurs 15 <fake@gmx.de>
Date:   Sun Sep 19 14:19:00 2010 +0200

    Eine suedliche Variante hinzugefuegt

diff --git a/hallo.c b/hallo.c
index 008ac22..df60c79 100644
--- a/hallo.c
+++ b/hallo.c
@@ -2,6 +2,9 @@
 #include <stdio.h>

 int main(void) {
-    printf("Hallo, Welt!\n");
+    // nord
+    printf("Moin, moin!\n");
+    // sued
+    printf("Gudday, mate!\n");
     return(0);
 }
```

```
$ git show master
commit 0d964bbeceec6d461e8d98a1d6e844300d66d30e
Author: VCS Kurs 15 <fake@gmx.de>
Date:   Sun Sep 19 14:19:00 2010 +0200

    Eine suedliche Variante hinzugefuegt

diff --git a/hallo.c b/hallo.c
index 008ac22..df60c79 100644
--- a/hallo.c
+++ b/hallo.c
@@ -2,6 +2,9 @@
 #include <stdio.h>

 int main(void) {
-    printf("Hallo, Welt!\n");
+    // nord
+    printf("Moin, moin!\n");
+    // sued
+    printf("Gudday, mate!\n");
     return(0);
 }
```

Zeigen, was wir haben (fort.)

git show [<blob>|<tree>|<commit>|<tag>]

```
$ git show master^
commit f011c01e07c19299df6e9db2c757e317c285a6b9
Author: Paul Cochrane <cochrane@nb206ptc.rrzn.uni-hann
Date:   Sun Sep 19 12:19:42 2010 +0200

    Erstes Import des Hallo-Projektes

diff --git a/Makefile b/Makefile
new file mode 100644
index 0000000..0e42f68
--- /dev/null
+++ b/Makefile
@@ -0,0 +1,9 @@
+.PHONY: clean
+
+default: hallo
+
+hallo: hallo.c
+       $(CC) -o hallo $<
+
+clean:
+       rm -f hallo *.o
diff --git a/hallo.c b/hallo.c
new file mode 100644
index 0000000..008ac22
--- /dev/null
+++ b/hallo.c
@@ -0,0 +1,7 @@
+// hallo: ein Programm um "hallo!" zu sagen
+#include <stdio.h>
+
+int main(void) {
+    printf("Hallo, Welt!\n");
+    return(0);
+}
```

Dateien ignorieren

Ähnlich wie Subversion mit der `svn:ignore`-Eigenschaft und noch ähnlicher wie CVS mit der `.cvsignore`-Datei. In Git braucht man nur eine `.gitignore`-Datei mit den Einträgen für die Dateien, die man ignorieren möchte. Wie bei Subversion, sind Wildcards wie `*.o` auch möglich.

```
$ make
cc -o hallo hallo.c
$ ls
hallo  hallo.c  Makefile
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       hallo
#       hallo.c~
nothing added to commit but untracked files present (use "git add" to track)
$ vim .gitignore
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

Dateien ignorieren

Die `.gitignore`-Datei hält man auch unter Versionskontrolle. Daher fügen wir sie jetzt hinzu und machen einen Commit.

```
$ git add .gitignore
$ git commit .gitignore
Created commit 38d2e3c: Das hallo Programm und *~ Datein werden jetzt
ignoriert
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
```

Zeigen, was wir für Branches haben

`git show-branch`

- zeigt die verfügbaren Branches
- zeigt durch ein Sternchen * welcher Branch aktuell ist

```
$ git show-branch --list
* [master] Das hallo Programm und *~ Datein werden jetzt ignoriert
```

```
git show-branch --more=35
[master] Das hallo Programm und *~ Datein werden jetzt ignoriert
[master^] Eine suedliche Variante hinzugefuegt
[master~2] Erstes Import des Hallo-Projektes
```

Zeigen, was wir für Branches haben (fort.)

`git branch` zeigt verfügbare Branches und welcher Branch aktuell ist

```
$ git branch
* master
```

`git branch -v` zeigt auch den Commit-Hash und die Commit-Nachricht des letzten Commits

```
$ git branch -v
* master 38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
```

Ein neuer Branch

`git branch <branchname> [<anfangscommit>]`

```
$ git branch hallo-perl
$ git branch -v
  hallo-perl 38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
* master     38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
$ git checkout hallo-perl
Switched to branch "hallo-perl"
$ git branch -v
* hallo-perl 38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
  master     38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
```

Dies hätte man auch mit `git checkout -b hallo-perl` machen können.

```
$ vim hallo.pl
$ chmod +x hallo.pl
$  ./hallo.pl
Moin, moin!
Gudday, mate!
$ git add hallo.pl
$ git commit -m "Das hallo Programm in Perl uerbersetzt" hallo.pl
Created commit 8e92aec: Das hallo Programm in Perl uerbersetzt
 1 files changed, 8 insertions(+), 0 deletions(-)
 create mode 100755 hallo.pl
```

Ein neuer Branch (fort.)

```
$ git branch -v
* hallo-perl 8e92aec Das hallo Programm in Perl uerbersetzt
  master     38d2e3c Das hallo Programm und *~ Datein werden jetzt ignoriert
$ git show-branch --more=10
* [hallo-perl] Das hallo Programm in Perl uerbersetzt
 ! [master] Das hallo Programm und *~ Datein werden jetzt ignoriert
--
*  [hallo-perl] Das hallo Programm in Perl uerbersetzt
*+ [master] Das hallo Programm und *~ Datein werden jetzt ignoriert
*+ [master^] Eine suedliche Variante hinzugefuegt
*+ [master~2] Erstes Import des Hallo-Projektes
```

Grafische Darstellung der Branches mit `gitk`

```
$ aptitude install gitk
$ gitk
```

Ein neuer Branch (fort.)

Weiter an `hallo.pl` arbeiten...

```
$ git diff hallo-perl^ hallo-perl
diff --git a/hallo.pl b/hallo.pl
index 411c241..9abe587 100755
--- a/hallo.pl
+++ b/hallo.pl
@@ -4,5 +4,11 @@
 use warnings;
 use strict;

-print "Moin, moin!\n";
-print "Gudday, mate!\n";
+my %greetings = (
+    "nord" => "Moin, moin!",
+    "sued" => "Gudday, mate!",
+);
+
+for my $greeting ( keys $greetings ) {
+    print $greeting, "\n";
+}
```

Ein neuer Branch (fort.)

Wie sehen die Branches jetzt aus?

```
$ git show-branch --more=10
* [hallo-perl] Die hallo-Perl-Version ist jetzt flexibler
 ! [master] Das hallo Programm und *~ Datein werden jetzt ignoriert
--
*  [hallo-perl] Die hallo-Perl-Version ist jetzt flexibler
*  [hallo-perl^] Das hallo Programm in Perl uerbersetzt
*+ [master] Das hallo Programm und *~ Datein werden jetzt ignoriert
*+ [master^] Eine suedliche Variante hinzugefuegt
*+ [master~2] Erstes Import des Hallo-Projektes
```

Und einfach, weil wir das können (Yes!, we can):

```
$ git diff master^ hallo-perl
....
$ git diff hallo-perl^ master~2
....
```

Ein bisschen mehr am Master-Branch arbeiten...

Merge zurück in den Master-Branch

Unsere Arbeit am neuen Zweig ist zu Ende. Wir möchten die Änderungen wieder in den Master-Branch (Trunk) einfließen lassen.

Zum Master-Branch wechseln

```
$ git checkout master
Switched to branch "master"
```

Überprüfen, ob alles sauber ist (ansonsten können Probleme leichter auftreten, wenn das Merge durchzuführt wird)

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Merge mal endlich!

```
$ git merge hallo-perl
Updating 38d2e3c..4738673
Fast forward
 hallo.pl |   14 ++++++++++++++
 1 files changed, 14 insertions(+), 0 deletions(-)
 create mode 100755 hallo.pl
```

Ups!

Weiter am Master-Branch arbeiten

```
$ git commit -a
Created commit 96e56ce: Noch eine Sprache hinzugefuegt
 1 files changed, 2 insertions(+), 0 deletions(-)
cochrane@nb206ptc:~/hallo$ vim hallo.c
cochrane@nb206ptc:~/hallo$ make
cc -o hallo hallo.c
cochrane@nb206ptc:~/hallo$ ./hallo
Moin, moin!
Gruess Gott!
Gudday, mate!
Arf, arf!
cochrane@nb206ptc:~/hallo$ git commit -a
Created commit b3fc5cb: Die Sprachen werden noch suedlicher
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Ups! (fort.)

Jetzt alles wieder in Perl nachmachen und Fehler finden

```
cochrane@nb206ptc:~/hallo$ git checkout hallo-perl
Switched to branch "hallo-perl"
cochrane@nb206ptc:~/hallo$ vim hallo.pl
cochrane@nb206ptc:~/hallo$ git commit -a
Created commit 55686f4: Weitere Entwicklungen
 1 files changed, 1 insertions(+), 0 deletions(-)
cochrane@nb206ptc:~/hallo$ vim hallo.pl
cochrane@nb206ptc:~/hallo$ vim hallo.pl
cochrane@nb206ptc:~/hallo$ git commit -a
Created commit 869aa90: Noch eine Sprache hinzugefuegt
 1 files changed, 2 insertions(+), 1 deletions(-)
cochrane@nb206ptc:~/hallo$ ./hallo.pl
Global symbol "$greetings" requires explicit package name at ./hallo.pl line
14.
Type of arg 1 to keys must be hash (not scalar dereference) at ./hallo.pl
line 14, near "$greetings ) "
Execution of ./hallo.pl aborted due to compilation errors.
cochrane@nb206ptc:~/hallo$ vim hallo.pl
$ ./hallo.pl
Gudday, mate!
Quark, quark!
Arf, arf!
Moin, moin!
```

Fehler korrigiert!

Ups! (fort.)

Schnell die funktionierende Version einchecken...

```
cochrane@nb206ptc:~/hallo$ git diff
diff --git a/hallo.pl b/hallo.pl
index ed0b3a5..3f08db0 100755
--- a/hallo.pl
+++ b/hallo.pl
@@ -11,6 +11,6 @@ my %greetings = (
     "suedlicher" => "Arf, arf!",
 );

-for my $greeting ( keys %greetings ) {
-    print $greeting, "\n";
+for my $location ( keys %greetings ) {
+    print $greetings{$location}, "\n";
 }
cochrane@nb206ptc:~/hallo$ git commit -a
Created commit a71053f: Gravierende Bugs korrigiert
 1 files changed, 2 insertions(+), 2 deletions(-)
```

Ups! (fort.)

Erneuter Merge mit Master-Branch

```
cochrane@nb206ptc:~/hallo$ git checkout master
Switched to branch "master"
cochrane@nb206ptc:~/hallo$ git merge hallo-perl
Merge made by recursive.
 hallo.pl |    6 ++++--
 1 files changed, 4 insertions(+), 2 deletions(-)
```

`git log --graph` ist cool

```
cochrane@nb206ptc:~/hallo$ git log --graph --pretty=oneline --abbrev-commit
*   3067f40... Merge branch 'hallo-perl'
|\
| * a71053f... Gravierende Bugs korrigiert
| * 869aa90... Noch eine Sprache hinzugefuegt
| * 55686f4... Weitere Entwicklungen
* | b3fc5cb... Die Sprachen werden noch suedlicher
* | 96e56ce... Noch eine Sprache hinzugefuegt
|/
* 4738673... Die hallo-Perl-Version ist jetzt flexibler
* 8e92aec... Das hallo Programm in Perl uerbersetzt
* 38d2e3c... Das hallo Programm und *~ Datein werden jetzt ignoriert
* 0d964bb... Eine suedliche Variante hinzugefuegt
* f011c01... Erstes Import des Hallo-Projektes
```

`gitk` auch

```
$ gitk
```

Aufgabe–Mit einem Git-Repository arbeiten

- Ein bestehendes Projekt holen
- Ein Git-Repository anlegen
- Dateien zum Repository hinzufügen und einchecken
- Dateien ändern und committen
- Branches machen, zu anderen Branches wechseln
- Sachen ändern und committen
- Die Branch-Änderungen in den Master-Branch einfließen
- Branches löschen
- `git log`, `git status`, `git diff` benutzen

Mit anderen (auch online) arbeiten

`git clone` Ein bestehendes Repository »klonen« oder »nachbilden« in ein neues Verzeichnis, und den aktuellen Branch holen

`git pull` Zieht Änderungen von einem anderen Repository ins lokale Repository

`git push` Schiebt Änderungen von einem lokalen Repository hoch zu einem anderen

`git format-patch` Generiert eine Patch-Datei, die man danach als Mail-Anhang verschicken kann

Git in der Systemadministration

Als Administrator eines Linux-Rechners möchte man wissen was in den verschiedenen Konfigurationen geändert worden ist, wie Dateien in der Vergangenheit aussahen (»als das damals funktioniert hatte...«) und vielleicht auch eine Sicherung aller Konfigurationsdateien an einem anderen Ort haben, um die Konfiguration wieder zurückzuspielen. Zum Beispiel, das `/etc/`-Verzeichnis.

```
$ cd /etc
$ git init
# /etc/shadow und /etc/shadow- sollten nicht gesichert werden
$ echo "shadow" >> .gitignore
$ echo "shadow-" >> .gitignore
$ git add .
$ git commit -a
```

Jetzt ist die Konfiguration gespeichert, dokumentiert und »unter Kontrolle«.

Siehe auch das `etckeeper`-Programm. »store /etc in git, mercurial or bzr«.

Aufgabe–Die Apache-Konfiguration sichern

Die Aufgabe vom Subversion-Teil nachmachen, aber diesmal mit Git und ohne den Server.

Weiteres

`git stash`

- Ähnlich wie das »changelist«-Konzept in Subversion.
- Man ist mittendrin ein Thema zu erarbeiten
- Etwas muss schnell gemacht werden
- `git stash` um aktuellen Zustand zu speichern (explizit: `git stash save`
- Arbeitskopie ist jetzt »sauber«
- Man kann am neuen Thema arbeiten und die Änderungen einchecken
- Mit `git stash pop` erstellt man den alten Zustand wieder
- Man kann weiter am alten Thema arbeiten
- `git stash` funktioniert wie ein Stack (push/pop-Operationen)

Weiteres (fort.)

`git stash list` Alle »gestashte« Zustände auflisten

`git stash show` Einen Zustand zeigen

`git stash apply` Einen Zustand auf die aktuelle Arbeitskopie daraufspielen

`git stash pop` Den »obersten« Zustand auf die aktuelle Arbeitskopie daraufspielen

Git mit Subversion-Repositories

Git spielt mit anderer Versionskontrollsystemen auch gut zusammen. Darunter CVS, Arch, und glücklicherweise Subversion. Dafür braucht man das `git-svn` Paket:

```
$ aptitude install git-svn
```

Die wichtigsten Kommandos sind:

git-svn clone Eine Subversion-Repository auschecken und in ein Git-Repository lokal umwandeln

git-svn fetch Unbekannte Revisionen vom Subversion-Repository holen

git-svn rebase Unbekannte Revisionen vom Subversion-Repository in die lokale Git-Repository holen, einspielen und das lokale Git-Repository zum End-Commit »vorspulen«.

git-svn log Ein Subversion-Log erzeugen

git-svn dcommit Alle lokalen Git-Repository-Commits einzeln und hintereinander ins Subversion-Repository aufspielen

Eine Subversion-Repository »clonen«

git-svn clone <svn-repository-url>

```
$ git-svn clone\
    svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo
Initialized empty Git repository in /home/cochrane/Kurs/hallo/.git/
    A   hallo.c
    A   Makefile
r1 = b423079e7a66c29b0890efec929a192f420ddc63 (git-svn)
    D   hallo.c
    D   Makefile
    A   trunk/hallo.c
    A   trunk/Makefile
W: +empty_dir: branches
W: +empty_dir: tags

....

r51 = 04468d313715388661ddc6b76b03d21d3b779fcc (git-svn)
Checked out HEAD:
  svn+ssh://vcskurs15@vcs.rrzn.uni-hannover.de/svnroot/projects/hallo r51
```

Arbeiten...

Jetzt arbeitet man mit Git wie gewohnt.

- `git branch`
- `git checkout -b <branchname>`
- `git commit`

Wenn man die Arbeit von anderen, die mit Subversion (oder `git-svn`) gearbeitet hatten, bekommen möchte, benutzt man: `git-svn rebase`

```
$ git-svn rebase
    M   trunk/hallo.py
r52 = d76aa8a9eae3f25e14b17c6cacea9be80da23b76 (git-svn)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/git-svn.
```

Arbeiten. . . (fort.)

Man committet auch im lokalen Git-Repository (man kann jetzt im Cafe oder in einer Kneipe sitzen; man muss nicht online sein!).

Wir verbessern und erweitern `hallo.py` und machen Commits.

```
$ git commit -a
Created commit 4a7a298: Code wurde etwas verstaendlicher: die Schleife ging
ueber Regionen nicht
 1 files changed, 2 insertions(+), 2 deletions(-)
$ git commit -a
Created commit d8b5899: Die Region wird jetzt erwaehnt wenn begruesst wird
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Arbeiten. . . (fort.)

Sobald wir wieder Netz haben und zufrieden mit der Entwicklung sind, können wir unsere Änderungen zum Server hochladen mit `git-svn dcommit`. Dieses Kommando spielt alle Commits einzeln und hintereinander hoch zum Server. Ein `git-svn rebase` vorher ist eine gute Idee, weil andere in der Zwischenzeit auch etwas zum Server eingecheckt haben könnten.

```
$ git-svn rebase
Current branch master is up to date.
$ git-svn dcommit
Committing to svn+ssh://vcs.rrzn.uni-hannover.de/svnroot/projects/hallo ...
    M   trunk/hallo.py
Committed r53
    M   trunk/hallo.py
r53 = dd95db3b2f51612e042fca61431fc368f8b94006 (git-svn)
No changes between current HEAD and refs/remotes/git-svn
Resetting to the latest refs/remotes/git-svn
trunk/hallo.py: needs update
    M   trunk/hallo.py
Committed r54
    M   trunk/hallo.py
r54 = 75b7b7ebb20c4782dc9aff06bf3a6e54c1556249 (git-svn)
No changes between current HEAD and refs/remotes/git-svn
Resetting to the latest refs/remotes/git-svn
```

Ressourcen

- O'Reilly Git-Buch
- Scott Chacons Railsconf Git Vortrag:
  http://www.gitcasts.com/git-talk
- Wikipedia-Eintrag:
  http://en.wikipedia.org/wiki/Git_(software)
- `git-svn` Tutorial:
  http://trac.parrot.org/parrot/wiki/git-svn-tutorial

Danke!

Vielen Dank für Eure Aufmerksamkeit!

:-)