

# *Versionsverwaltung mit RCS und CVS*

Alexander von Gernler

<grunk@adminigilde.org>

2. Erlanger LinuxTage

22. November 2003



# Motivation

---

## Warum Versionsverwaltung?

- Gegeben:
  - Größere Softwareprojekte
  - Langer Entwicklungszeitraum
  - Viele (evtl. wechselnde) Mitarbeiter
- Gesucht: Wie packen wir es an?

# Motivation

## Warum Versionsverwaltung?

- Gegeben:
  - Größere Softwareprojekte
  - Langer Entwicklungszeitraum
  - Viele (evtl. wechselnde) Mitarbeiter
- Gesucht: Wie packen wir es an?

## Ein erster Versuch (aus der Realität):

```
C:\PROJEKT\DELPHI> dir
DIALOG.PAS      DIALOG.PAS.BAK  DIALOG.OLD
DIALOG2.PAS    DIALOG_NEU.PAS DIALOG_ALEX.PAS
OLD\           VERSION2\
```

# Motivation

## Warum Versionsverwaltung?

- Gegeben:
  - Größere Softwareprojekte
  - Langer Entwicklungszeitraum
  - Viele (evtl. wechselnde) Mitarbeiter
- Gesucht: Wie packen wir es an?

## Ein erster Versuch (aus der Realität):

```
C:\PROJEKT\DELPHI> dir
DIALOG.PAS      DIALOG.PAS.BAK  DIALOG.OLD
DIALOG2.PAS    DIALOG_NEU.PAS  DIALOG_ALEX.PAS
OLD\           VERSION2\
```

⇒ **Total für die Tonne!**

## Was wir wollen

### **Anforderungen an eine Quellcodeverwaltung:**

- Unabhängig von der verwendeten Sprache
- Entwicklung des Codes über die Zeit nachvollziehbar
- Gleichzeitiges Arbeiten mehrerer Entwickler an einer Datei
- Änderungen gekennzeichnet durch Verursacher und Kommentar
- Arbeit im Team auch ohne örtliche Nähe möglich (Internet)

## Was wir wollen

### **Anforderungen an eine Quellcodeverwaltung:**

- Unabhängig von der verwendeten Sprache
- Entwicklung des Codes über die Zeit nachvollziehbar
- Gleichzeitiges Arbeiten mehrerer Entwickler an einer Datei
- Änderungen gekennzeichnet durch Verursacher und Kommentar
- Arbeit im Team auch ohne örtliche Nähe möglich (Internet)

### **Nicht nur für Megaprojekte**

Auch Ein-Mann-Softwareschmieden sollten derart systematisch und organisiert vorgehen! Dies ist keine Utopie – wie man es macht, wird in den folgenden Folien gezeigt.

## Verfügbare Tools (unvollständige Liste)

---

### Freie Software

- GNU CSSC
- RCS und CVS
- SCM
- Aegis
- OpenCM
- subversion

### Proprietäre Lösungen

- SCCS
- Rational ClearCase
- Perforce
- BitKeeper

Die Auswahl ist groß, aber die Freie Software Szene beschränkt sich meistens (noch?) auf wenige Lösungen, hauptsächlich CVS und seit neuestem BitKeeper.

- CVS ist *die* Versionsverwaltung, trotz aller offenen Wünsche
- Der Linux-Kern wird seit einiger Zeit mit BitKeeper verwaltet

# Übersicht über RCS

---

## Revision Control System

- Eingeführt 1981 von WALTER F. TICHY
- Wird ab 1989 von anderen Leuten gewartet.



# Übersicht über RCS

---

## Revision Control System

- Eingeführt 1981 von WALTER F. TICHY
- Wird ab 1989 von anderen Leuten gewartet.
- Sieht pro verwalteter Datei eine ,v-Datei mit der Historie vor
- Historie enthält den aktuellen Stand der Datei und die Deltas zu früheren Versionen



# Übersicht über RCS

## Revision Control System

- Eingeführt 1981 von WALTER F. TICHY
- Wird ab 1989 von anderen Leuten gewartet.
- Sieht pro verwalteter Datei eine ,v-Datei mit der Historie vor
- Historie enthält den aktuellen Stand der Datei und die Deltas zu früheren Versionen
- Beispiel: 

```
grunk@vario:~/projekt1% ls
beispiel.c      beispiel.c,v
```



# Übersicht über RCS

## Revision Control System

- Eingeführt 1981 von WALTER F. TICHY
- Wird ab 1989 von anderen Leuten gewartet.
- Sieht pro verwalteter Datei eine ,v-Datei mit der Historie vor
- Historie enthält den aktuellen Stand der Datei und die Deltas zu früheren Versionen
- Beispiel: 

```
grunk@vario:~/projekt1% ls
beispiel.c      beispiel.c,v
```
- Codeänderungen werden meist exklusiv vorgenommen
- Aus- und Einchecken mit Setzen/Aufheben des Locks durch 

```
co -l beispiel.c bzw. ci -u beispiel.c
```



# RCS Schritt für Schritt

## 1. Anlegen der ersten Quellcodedatei:

```
grunk@vario:~/projekt2% vim main.c
```

```
#include <stdio.h>

int main (int argc, char** argv) {
    printf("Wer das liest, ist doof!\n");
}
```

## 2. Erstes Einchecken der Datei:

```
grunk@vario:~/projekt2% ci -u main.c
```

```
main.c,v <-- main.c
```

```
enter description, terminated with single '.' or end of file:
```

```
NOTE: This is NOT the log message!
```

```
>> File containing my main program
```

```
>> .
```

```
initial revision: 1.1
```

```
done
```

## Dateien ändern

3. Oha, Fehler aufgetaucht. Also Datei auschecken und sperren:

```
grunk@vario:~/projekt2% co -l main.c
main.c,v --> main.c
revision 1.1 (locked)
done
```

4. Jetzt Änderung vornehmen: Argument von `printf` soll jetzt "Hallo Welt!\n" heißen.

```
grunk@vario:~/projekt2% vim main.c
```

```
#include <stdio.h>
int main (int argc, char** argv) {
    printf("Hallo Welt!\n");
}
```

## Veränderungen prüfen

### 5. Schauen, welche Änderungen am File passiert sind:

```
grunk@vario:~/projekt2% rcsdiff main.c
=====
RCS file: main.c,v
retrieving revision 1.1
diff -r1.1 main.c
4c4
<  printf("Wer das liest, ist doof!\n");
--
>  printf("Hallo Welt!\n");
```

Mit dem Schalter `-u` bekommt man das sehr gebräuchliche *unified diff* Format, das kontextbezogene Diffs liefert.

## Änderungen einchecken

---

6. Neue Version der Datei in die Versionsverwaltung geben und Sperre aufheben:

```
grunk@vario:~/projekt2% ci -u main.c
main.c,v <-- main.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> changing the printf message to a less offending one
>> .
done
```

## Änderungen einchecken

6. Neue Version der Datei in die Versionsverwaltung geben und Sperre aufheben:

```
grunk@vario:~/projekt2% ci -u main.c
main.c,v <-- main.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> changing the printf message to a less offending one
>> .
done
```

Für Kommentare gilt:

- wirklich aussagekräftige Sätze wählen
- in der Projektsprache bleiben (meist Englisch)
- Netiquette, 72 Zeichen pro Zeile etc.
- Loginname nicht extra erwähnen, geschieht durch RCS

# Vereinfachung und Übersicht

---

## **Viele ,v-Dateien im Arbeitsverzeichnis**

- Machen das Arbeiten im Verzeichnis unübersichtlich
- Werden vom Anwender nie direkt benutzt
- Könnten also versteckt werden

## Vereinfachung und Übersicht

---

### **Viele ,v-Dateien im Arbeitsverzeichnis**

- Machen das Arbeiten im Verzeichnis unübersichtlich
- Werden vom Anwender nie direkt benutzt
- Könnten also versteckt werden

### **Die Lösung**

- Wir verschieben alle ,v-Dateien in ein Unterverzeichnis
- Das Unterverzeichnis heißt standardmäßig RCS
- *Keine* Umstellung nötig, ci und co finden sich automatisch zurecht.

# Vereinfachung und Übersicht

---

## Viele ,v-Dateien im Arbeitsverzeichnis

- Machen das Arbeiten im Verzeichnis unübersichtlich
- Werden vom Anwender nie direkt benutzt
- Könnten also versteckt werden

## Die Lösung

- Wir verschieben alle ,v-Dateien in ein Unterverzeichnis
- Das Unterverzeichnis heißt standardmäßig RCS
- *Keine* Umstellung nötig, ci und co finden sich automatisch zurecht.
- So sieht das dann aus:

```
grunk@vario:~/projekt3% ls
RCS/      Makefile  config.h  main.c
```

# RCS Tags

## Möglichkeit, Strings von RCS einfügen zu lassen

- Sehr nützlich, wenn in der Datei Bezug auf die aktuelle Version genommen werden soll (Versions-Strings)
- RCS Tags beginnen und enden mit dem Dollar-Zeichen (\$)
- **Es gibt:** `$Author$, $Date$, $Header$, $Id$, $Locker$, $Log$, $Name$, $RCSfile$, $Revision$, $Source$ und $State$`
- Ein Tag wird bei Checkin und Checkout automatisch von RCS aktualisiert.
- Aus `$Id$` wird in meinen Folien (CVS-verwaltet) etwa:

```
$Id: cvs-rcs.tex,v 1.19 2003/11/24 11:01:28 sialgern Exp $
```

# Pro/Contra RCS

---

## Vorteile

- Komplette Versionskontrolle mit Kommentaren und Benutzernamen
- Kein gegenseitiges Überschreiben von Änderungen

## Nachteile

- Exklusive Lockstrategie: Nur eine Änderung zu einem Zeitpunkt
- Zugriff auf lokales Dateisystem unbedingt erforderlich, also nur beschränkt netzwerkfähig
- Gut für einzelne Dateien, unpraktisch für ganze Bäume
- Keine projektglobalen Snapshots möglich

## Also noch eins drauf: CVS

---

- 1989 programmiert BRIAN BERLINER CVS, das bisher nur als lose Sammlung von Shell-Skripten existiert hat.
- Baut auf den Mechanismen von RCS auf



## Also noch eins drauf: CVS

---

- 1989 programmiert BRIAN BERLINER CVS, das bisher nur als lose Sammlung von Shell-Skripten existiert hat.
- Baut auf den Mechanismen von RCS auf
- Verwendet lokale Arbeitskopien für Entwickler statt exklusiven Locks
  - Mehrbenutzerfähig (mehrere Entwickler an einer Datei)
  - Netzwerkfähig (kein direkter Dateisystemzugriff mehr nötig), verwendet `ssh` (früher `rsh`)
  - Kann einfache Konflikte beim Check-In selbst lösen
- Kann mit ganzen Projektbäumen auf einmal umgehen
- Erlaubt das Festhalten des Projektzustands zu einem bestimmten Zeitpunkt mit Hilfe von Versions-Tags



# Struktur von CVS

---

## Trennung von Arbeitskopie und Repository

- Arbeitskopie lokal auf den Rechnern der Entwickler, Operationen auf Kopie und Repository mit dem Befehl `cv`
- Repository (, `v`-Dateien) auf einem zentralen Server, Operationen durch einen `cv` `server` Prozess

# Struktur von CVS

---

## Trennung von Arbeitskopie und Repository

- Arbeitskopie lokal auf den Rechnern der Entwickler, Operationen auf Kopie und Repository mit dem Befehl `cv`
- Repository (, `v`-Dateien) auf einem zentralen Server, Operationen durch einen `cv` `server` Prozess

## Verzeichnisstruktur

- Repository auf dem Server

```
grunk@server: /cvs% ls
CVSROOT/      vortraege     www
```

# Struktur von CVS

## Trennung von Arbeitskopie und Repository

- Arbeitskopie lokal auf den Rechnern der Entwickler, Operationen auf Kopie und Repository mit dem Befehl `cv`
- Repository (, `v`-Dateien) auf einem zentralen Server, Operationen durch einen `cv` `server` Prozess

## Verzeichnisstruktur

- Repository auf dem Server

```
grunk@server:/cvs% ls
CVSROOT/      vortraege      www
```

- Arbeitskopie auf dem Entwicklungsrechner

```
grunk@vario:~/proj/vortraege/erlanger_linuxtage_2003% ls
CVS/          Makefile        PPRfyma.sty      Tichy.eps
CVS.eps       MangaRamblo.eps Rock.eps          cvs-rcs.tex
```

# CVS Schritt für Schritt

---

## 1. Repository anlegen

```
grunk@server:/cvs% cvs -d /cvs init
```

```
grunk@server:/cvs% mkdir vortraege
```

# CVS Schritt für Schritt

## 1. Repository anlegen

```
grunk@server:/cvs% cvs -d /cvs init
grunk@server:/cvs% mkdir vortraege
```

## 2. (noch leere) Arbeitskopie aus dem Repository auschecken

```
grunk@vario:~/proj% cvs -d server:/cvs checkout vortraege
grunk@vario:~/proj% cd vortraege && ls
CVS/
grunk@vario:~/proj/vortraege% ls CVS
Entries      Repository Root
```

- Checkout über eine Remote-Shell Verbindung (ssh)
- CVS/ beinhaltet interne Verwaltungsdaten, uninteressant für Benutzer
- Jetzt kann gearbeitet werden!

## CVS Schritt für Schritt (2)

### 3. Dateien erstellen und zum Repository hinzufügen

```
grunk@vario:~/proj/vortraege% vim Makefile
...
grunk@vario:~/proj/vortraege% cvs add Makefile
cvs add: scheduling file 'Makefile' for addition
cvs add: use 'cvs commit' to add this file permanently
```

### 4. Änderungen permanent machen

```
grunk@vario:~/proj/vortraege% cvs commit
cvs commit: examining .
```

```
CVS: -----
CVS: Enter Log. Lines beginning with 'CVS:' are removed automatic
CVS:
CVS: Committing in .
CVS:
CVS: Added Files:
CVS:  Makefile
CVS: -----
```

## CVS Schritt für Schritt (3)

```
RCS file: /cvs/vortraege/Makefile,v
done
Checking in Makefile;
/cvs/vortraege/Makefile,v <-- Makefile
initial revision: 1.1
done
```

5. Entwicklerkollegen arbeiten inzwischen am Code

6. Arbeitskopie updaten

```
grunk@vario:~/proj/vortraege% cvs update
P Makefile
U cvs-rcs.tex
```

7. Ändern, einchecken, und so weiter...

## Konfliktbehandlung

---

- Zwei Entwickler checken `main.c` aus, Version 1.4
- Entwickler A verändert `main.c` und checkt Änderungen ein
- `main.c` nun in Version 1.5 auf dem Server, aber in 1.4 auf Workstation von Entwickler B

## Konfliktbehandlung

---

- Zwei Entwickler checken `main.c` aus, Version 1.4
- Entwickler A verändert `main.c` und checkt Änderungen ein
- `main.c` nun in Version 1.5 auf dem Server, aber in 1.4 auf Workstation von Entwickler B

### **Konflikt automatisch auflösbar, wenn**

- Änderungen an verschiedenen Stellen in der Datei
- Änderungen nicht ineinander verschachtelt

## Konfliktbehandlung

---

- Zwei Entwickler checken `main.c` aus, Version 1.4
- Entwickler A verändert `main.c` und checkt Änderungen ein
- `main.c` nun in Version 1.5 auf dem Server, aber in 1.4 auf Workstation von Entwickler B

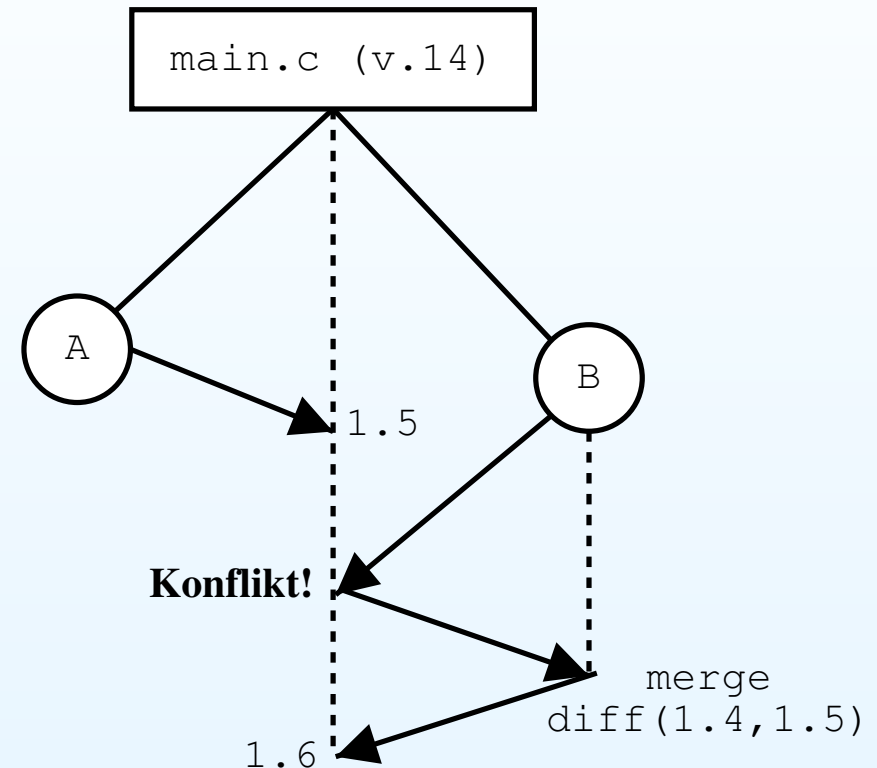
### Konflikt automatisch auflösbar, wenn

- Änderungen an verschiedenen Stellen in der Datei
- Änderungen nicht ineinander verschachtelt
- Dann kann Entwickler B auf `main.c` (v1.5) updaten, ohne dass seine Änderungen zerstört werden
- Danach committet er `main.c` (v1.6) als Ergebnis seiner vorherigen Änderungen an v1.4, durchgeführt an v1.5

## Konfliktbehandlung (2)

### Bei nicht auflösbarem Konflikt

- muß Entwickler B per Hand die Änderungen von 1.4 auf 1.5 einpflegen
- erst nach dem Entfernen der von CVS eingefügten Konfliktzeilen kann die Datei committed werden



**Generell: Es kann *nur* von der aktuellsten Version aus committed werden**

## Nachteile von CVS

---

- Keine atomaren Änderungen mehrerer Dateien
- Keine Verwaltung von Softlinks, Umweg über Makefile
- Pro Datei ein Aufruf von RCS nötig, also langsam
- Ineffizienter Umgang mit Binärdateien
- Umbenennen von Verzeichnissen nicht versionsverwaltet

## Noch Fragen?



- Folien erstellt mit  $\text{\LaTeX}$ , `prosper`, `make` und `CVS` unter `OpenBSD 3.4 / i386`
- Quellcode der Folien auf Anfrage
- Fragen werden auch per Mail gerne beantwortet:  
<grunk@admingilde.org>
- Mails: 72 Zeichen pro Zeile, keine HTML-Mails