SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

Werkzeuge: make, diff, patch, Versionsverwaltung

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

make

Problem:

- Projekte bestehen üblicherweise aus vielen Dateien
- Von Hand bauen ist mühsam und fehleranfällig
- Alles immer neu bauen dauert lange

Idee:

- Spezifiziere Abhängigkeiten zwischen Dateien (Datei B wird aus Datei A erzeugt)
  $\longrightarrow$ Dateiname: Makefile
- Abhängigkeiten bilden gerichteten azyklischen Graphen
- make liest die Spezifiaktion ein und prüft die Zeitstempel der Abhängigkeiten
- Datei wird neu erzeugt werden, wenn sie älter ist als eine ihrer Abhängigkeiten oder sie nicht existiert
- Aufruf: make
  $\longrightarrow$ Erstes Ziel in Makefile wird als Wurzel verwendet

Syntax: Einfache Regel

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
hello: hello.c hello.h
  cc -o hello hello.c
```

- Datei `hello` hängt von `hello.c` und `hello.h` ab
- Befehlszeile darunter muss mit einem Tabulator beginnen!
- Abhängigkeiten und Befehle trennbar

```
hello: hello.c
hello: hello.h

hello:
  cc -o hello hello.c
```

Mehrere Ziele automatisch bauen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Problem: Nur erstes Ziel wird automatisch beachtet
- Pseudoziel einfügen, das von anderen abhängt
- Kanonischer Name: `all`

```
all: hello bye

hello: hello.c hello.h
  cc -o hello hello.c

bye: bye.c
  cc -o bye bye.c
```

Automatische Variablen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
hello: hello.c hello.h
  cc -o hello hello.c
```

- Problem: Duplikation, fehleranfällig
- $@: Aktuelles Ziel
- $<: Erste Abhängigkeit

```
hello: hello.c hello.h
  cc -o $@ $<
```

Variablen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
hello: hello.c hello.h
  cc -o $@ $<
```

- Problem: Übersetzer in Befehl fest eingetragen
- Verwende Variablen
- Klammern sind wichtig, sonst ist nur erster Buchstabe nach $ der Variablenname
- CC wird normalerweise vom System bereits gesetzt, kann aber vom Benutzer überschrieben werden:

```
make CC=icc
```

```
hello: hello.c hello.h
  $(CC) -o $@ $<
```

Generische Regeln

```
all: hello bye

hello: hello.o
  $(CC) -o $@ $<

bye: bye.o
  $(CC) -o $@ $<

hello.o: hello.c hello.h
  $(CC) -c $<

bye.o: bye.c
  $(CC) -c $<
```

- Problem: Duplikation
- `.c.o`: `.o`-Dateien werden aus `.c` erzeugt
- `.o`: Datei ohne Endung wird aus `.o` erzeugt

```
all: hello bye

hello: hello.h

.c.o:
  $(CC) -o $@ $<

.o:
  $(CC) -c $<
```

diff

- Findet zeilenweise Änderungen in Textdateien
- Theoretischer Hintergrund: Längste gemeinsame Teilsequenz (longest common subsequence, LCS)
- NP-hart in Anzahl der zu vergleichenden Sequenzen
  $\longrightarrow$ Üblicherweise zwei, daher polynomiell lösbar

Beispiel: LCS

ABAB
ABBA

ABAB
ABBA

- Beispiel zeichenweise statt zeilenweise
- Longest common subsequence: ABA
- Nicht verwechseln mit longest common substring: AB oder BA

Beispiel

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
%cat alt
1
2
alt
3
4
%cat neu
1
2
neu
3
4
```

Standardformat

```
%diff alt neu
3c3
< alt
---
> neu
```

- Standardformat von diff
- <: Entfernte Zeilen
- >: Hinzugefügte Zeilen
- Problem: Kein Kontext, patch funktioniert nur mit exakt der alten Datei

Context Diff

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
%diff -c alt neu
*** alt
--- neu
***************
*** 1,5 ****
  1
  2
! alt
  3
  4
--- 1,5 ----
  1
  2
! neu
  3
  4
```

- Zeigt (bis zu) drei Zeilen Kontext
- Oben: alt, unten neu
- `!`: geänderte Zeilen
- Problem: Format recht langatmig (Kontext wird wiederholt)

Unified Diff

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
%diff -u alt neu
--- alt
+++ neu
@@ -1,5 +1,5 @@
 1
 2
-alt
+neu
 3
 4
```

- Heute gebräuchlichstes Format, ebenfalls (bis zu) drei Zeilen Kontext
- -: Entfernte Zeilen
- +: Hinzugefügte Zeilen

patch

- Umkehrung von diff: Wende Änderungen auf Datei an
- Datei muss nicht exakt mit Original übereinstimmen, z.B. verschobene Zeilen kann patch kompensieren
- Verwendung: `patch $DATEI < $DIFF`
  $\longrightarrow$ Wende `DIFF` auf `DATEI` an
- Dateiangabe optional, kann aus `DIFF` ausgelesen werden

diff/patch vs. Minus

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
b - a = c          diff      a    b  >  c

a + c = b          patch     a <  c -o  b

a - b = -c         diff      b    a  > _c

b + -c = a         patch     b < _c -o  a
                   patch -R  b <  c -o  a
```

Versionsverwaltungssysteme

... gibt es wie Sand am Meer:

- Source Code Control System (SCCS)
- Revision Control System (RCS)
- Concurrent Versioning System (CVS)
- Visual Source Safe (VSS)
- Perforce (P4)
- Bitkeeper
- Darcs
- Bazaar
- Arch
- Monotone
- Team Foundation Server (TFS)
- Subversion (SVN)
- SVK
- Git
- Mercurial (HG)
- ...

Warum Versionsverwaltung?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Probleme/Fragen:

- Was habe ich gestern gemacht?
- Warum wurde das gemacht?
- Wer hat das gemacht?
- Was wurde seit der letzten Veröffentlichung gemacht?
- Archivierung älterer Versionen
- Beherbergung mehrer Entwicklungszweige
- Zusammenarbeit von mehreren Entwicklern

Es gibt keinen Grund gegeben Versionsverwaltung:

- Plattenplatz ist billig: weniger als 10 Cent/GB
  $\longrightarrow$ 10 Cent für mehr als 250.000 vollgeschriebene Seiten (80x50)
- Versionsverwaltungssysteme sind hinreichend schnell und einfach zu bedienen
- Liefern direkt Mehrwert: Was habe ich gerade geändert? usw.

Aber...

- Versionsverwaltungssysteme können nicht die Kommunikation zwischen Entwicklern ersetzen
- Sie können nur die Zusammenarbeit erleichtern
- Soziale Probleme sind nicht technisch lösbar

Verwendungsbeispiel: Fehlersuche

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Revision 2000 zeigt ein Fehlersympton
- Revision 1000 zeigt Fehlersympton nicht
- Idee: Binäre Suche
- Teste Version in der Mitte
  - Falls Fehler: neue obere Schranke
  - Kein Fehler: neue untere Schranke
- Wiederholen bis Fehler exakt eingegrenzt ist
- Laufzeit: O(ld n), da Intervall in jedem Schritt halbiert wird
  $\rightarrow$ Hier also 10 Schritte ($2^{10} = 1024 > 2000 - 1000$)

Fallbeispiel: RCS

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Revision Control System
- Eines des ersten Systeme (SCCS ist älter)
- Verwaltet einzelne Dateien
- Nomenklatur/Befehle heutiger Systeme stammt zumeist von RCS-Befehlen: co (checkout), ci (commit)
- Speichert Autor, Zeitstempel, Commit-Nachricht
- Verwendet Rückwertsdeltas
  $\longrightarrow$ Neuste Version am häufigsten angefragt, muss nicht aus älteren rekonstruiert werden
- Weiterentwicklung: CVS (Concurrent Versioning System)
  - begann als Sammlung von Shellskripten für RCS
  - benutzt selbes Dateiformat
  - Netzwerkunterstützung: „Multiplayer-RCS“

Beispiel (leicht gekürzt)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
head     1.2;

1.2
date     2009.12.14.09.53.04;     author mallon;   state Exp;
branches;
next     1.1;
1.1
date     2009.12.14.09.52.05;     author mallon;   state Exp;
branches;
next     ;

1.2
log
@Reformat
@
text
@#include <stdio.h>

int main(void)
{
        printf("hello world\n");
}
@

1.1
log
@Initial revision
@
text
@d3 4
a6 1
int main(void) { printf("hello world\n"); }
@
```

Fallbeispiel: Git

Grundideen:

- Speichere Inhalte statt Unterschiede
  $\longrightarrow$ Unterschiede aus Inhalten berechenbar
- Versionen bilden gerichteten azyklischen Graphen
  $\longrightarrow$ Eine Version zeigt auf seine Vorgängerversion(en)
- Hierarchische Ordnung von Objekten, die als Hash ihres Inhalts abgelegt werden: Dateiinhalte, Verzeichnisse, Commits

Blobs

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Blobs (Binary Large Obects, Datien) werden gespeichert
- Name ist der Hash ihres Inhalts

```
#include <stdio.h>
int main(void) { printf("hello, world\n"); return 0; }
```

→ 4727e0131811ddce76dbb8321724d85b56405fe8 (laut git hash-object)

Trees

- Trees (Verzeichnisse) enthalten Listen von Blob-/Tree-Hashs mit Datei-/Verzeichnisnamen
- Name ist wiederrum Hash

```
100644 blob 4727e0131811ddce76dbb8321724d85b56405fe8 hello.c
```

$\longrightarrow$ 8d5bd2d4136f0a4f6a57e8a1b88f9ed59768278c

Commits

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Ein Commit besteht aus einem Tree, einem Committer, einem Autor, einer Liste von Elterncommits (leer bei erstem) und einer Nachricht
- Name ist wieder ein Hash

```
tree 8d5bd2d4136f0a4f6a57e8a1b88f9ed59768278c
author Christoph Mallon <mallon@cs.uni-saarland.de> 1262424493 +0100
committer Christoph Mallon <mallon@cs.uni-saarland.de> 1267002986 +0100

A hello world program.
```

$\longrightarrow$ da4b16f97e2db9d7b594f9410acfb03435990d3f

Zweige, symbolische Referenzen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Ein Zweig (Branch, eigtl. Ast) ist eine Datei, die den Hash eines Commits enthält
- Der Dateiname ist der Name des Zweigs
- Wird mit jedem Commit auf diesen Zweig auf den neuen Commit umgebogen

```
%cat .git/refs/heads/master
da4b16f97e2db9d7b594f9410acfb03435990d3f
```

- Eine symbolische Referenz zeigt auf einen Zweig
- HEAD: Aktueller Zweig

```
%cat .git/HEAD
ref: refs/heads/master
```