Mitschrift für die Vorlesung

Open-Source-Programmierwerkzeuge

von
Norman Ulbrich

(Stand SS09, Version 2.1)

Mitgeschrieben und zusammengestellt von:
Fabian Müller (fabian.mueller@mni.fh-giessen.de)

Inhaltsverzeichnis

1 Themen

1.1 Versionsverwaltung

- Was ist Versionsverwaltung?
- Wie benutzt man Versionsverwaltung?
- Wie funktioniert Versionsverwaltung?

1.2 Debugging

- Was ist Debugging?
- Welche Arten von Debugging gibt es?
- Verwenden eines Debuggers

1.3 Tools und ihre Kombinationen

- Welche Programme gibt es?
- Kombination von Programmen
- Skriptprogrammierung

1.4 Wünsche und Anregungen

- Wie funktioniert „make“
- VI / VIM

Leistungsnachweis: Klausur

2 Literatur

Open-Source-Programmierwerkzeuge Zeller, Krinke, dpunkt-Verlag http://sun-book.red-bean.com	*Expert C Programming – Deep C Secrets* van der Linden, Prentice Hall
Why Programs fail – A guide to systematic Debugging Zeller, dpunkt-Verlag	

3 Versionsverwaltung

3.1 Warum Versionsverwaltung

Das Benutzen von Versionsverwaltung erlaubt

- Einen „sauberen“ Stamm zu generieren.
- Änderungen seit z.B. dem letzten Release zu verfolgen/ermitteln.
- Zu prüfen, wie eine Lösung früher umgesetzt wurde.
- Festzustellen, wann eine Änderung gemacht wurde.
- Zu ermitteln, welcher Entwickler eine Änderung gemacht hat.
- Einen alten Stand zu generieren um Fehlerberichte eines Kunden nachzuvollziehen.
- Verschiedene Entwicklungsstände zusammen zu führen.

3.2 Vergleich mit Kopien von Verzeichnissen

Kopien mit Datum oder Versionsstand

- Gleiche Dateien werden jedesmal erneut abgelegt => Platzverschwendung.
- Vergleich möglich aber keine Historie.
- Für gemeinsame Entwicklung ungeeignet

3.3 Archivieren mit Hardlinks

- Vergleiche möglich aber keine Historie.
- Fürs gemeinsame Entwickeln ungeeignet

3.4 Beispiele für Versionsverwaltung

- „Kopien Archivieren“
- SCCS (Source Code Control System)
- RCS (Revision Control System)
- CVS (Concurrent Version Control)
- Subversion (svn)
- Bitkeeper
- Git
- Bazaar (bzr)
- Microsoft Visual Source Safe
- PVCS
- Mercurial (hg)

3.5 Terminologie

Das **Repository** ist der Ort, an dem das Versionsverwaltungssystem die versionierten Daten und alle nötigen Metainformationen ablegt. Man könnte es als „Lagerhaus“ für alle Versionen bezeichnen.

Im **Arbeitsverzeichnis** liegt eine Versionskopie der Daten im Repository. Jeder Entwickler hat sein eigenes Arbeitsverzeichnis. Hier finden Entwicklungen und Tests statt und hier werden Änderungen ins Repository übertragen.

Daten aus dem Repository ins Arbeitsverzeichnis zu holen bezeichnet man als **Checkout**, manchmal als **Get**, sie im Repository zu hinterlegen **Commit** oder **Checkin**. Neuere Versionen aus dem Repository holt man sich per **Update**.

Verschiedene Arten eines Standes im Repository bezeichnet man als **Zweig** oder **Branch**.

Wenn zwei Entwickler dieselbe Datei an derselben Stelle ändern und einchecken bekommt der zweite einen **Konflikt**.

Übungen: http://www.fh-giessen.de/~hg10597

Um einen bestimmten Entwicklungsstand zu markieren versieht man ihn mit einem **Tag**. Hierbei handelt es sich um einen Namen, den man den Dateiversionen gibt, z.B. „beta-0.2“, „release-1.0“, „C7* CCU* MR2959* 2009-03-18* NoUlbr added several OBD channels“ oder „beta-4-00-45-IWwinmon“.

Verschiedene Entwicklungszweige werden durch einen **Merge** zusammengefasst. Solange keine Konflikte vorliegen, kann die Versionsverwaltung bei Textdateien den Großteil der Arbeit automatisch erledigen. Für Binärdaten geht das in der Regel nicht.

Wird ein Stand aus dem Repository benötigt, der keine Administrationsdaten der Versionsverwaltung enthält, so ist manchmal von einem **Export** die Rede.

Um Konflikte (gerade bei Binärdateien) zu vermeiden, kann man einen **Lock** auf eine Datei setzen um sie gegen Einchekchen anderer zu schützen. Je nach Versionsverwaltung können andere Benutzer einen **Lock** entfernen oder nicht. In jedem Fall ist Kommunikation zwischen den Entwicklern angebracht.

3.6 Verwaltungsinformationen

In der Regel merkt sich die Versionsverwaltung, welche Dateien man in welcher Version ausgecheckt hat. Dafür legt sie im Arbeitsverzeichnis ein Unterverzeichnis an, in dem sie derartige Verwaltungsinformationen ablegt, z.B. wo das Repository liegt, Zugangsdaten, welche Dateien in welcher Reihenfolge ausgecheckt wurden und möglicherweise weitere Informationen. Diese Verzeichnisse heißen z.B. „CVS“ oder „.svn“; es gibt sie meist in jeder Hierarchiestufe erneut. Bei Subversion ist auch der BASE-Stand, also die Version, die man ausgecheckt hat, in diesen Informationen enthalten. Das ermöglicht es, die Änderungen seitdem nachzuvollziehen, ohne auf das Repository zuzugreifen,

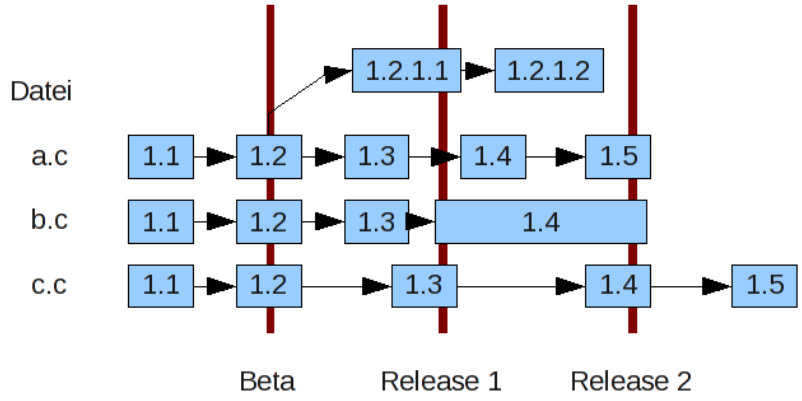
Das ist nützlich, wenn das Repository nicht auf dem lokalen Rechner, und der Zugriff nicht möglich, langsam oder kostspielig ist.

3.7 Versionsnummern

Je nach Versionsverwaltung sehen Versionsnummern unterschiedlich aus. Meist hat eine Datei eine Version wie: „1.13“, eine andere Datei kann die Version „1.2“ haben eine dritte „3.15“. Bei einfachen Branches entstehen Versionsnummern wie „1.10.2.3“ oder „1.2.3.4“; bei mehrfacher Branches werden weitere Ziffern angehängt und es entsteht z.B. „1.10.2.2.2.1“. Bei Subversion hat nicht jede Datei ihre eigene Versionsnummern sondern der komplette Stand im Repository. Wenn von zehn Dateien nur zwei geändert und eingecheckt werden, bekommen trotzdem alle Dateien im Repository eine neue Version. Bei acht davon ist der Inhalt der gleiche wie vor dem Einchecken. Die Versionsnummer ist eine einfache Zahl wie „42“ oder „3305“.

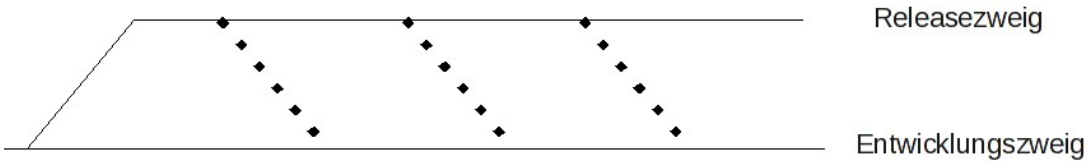
4 Branches

4.1 Wie kann man sich Branches vorstellen? Ein Beispiel von CVS:

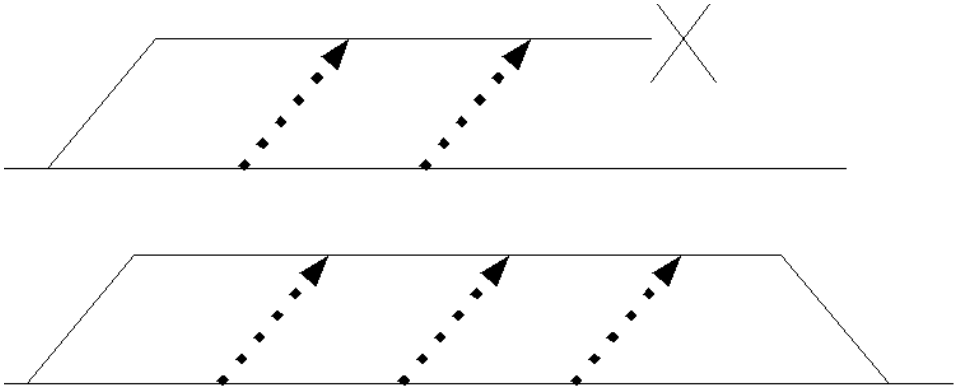


Die Verwendung von Branches kann aus verschiedenen Gründen sinnvoll sein:

Bevor eine Version veröffentlicht wird ist möglicherweise eine Testphase nötig. In dieser Zeit werden keine weiteren Features hinzugefügt sondern nur Fehler behoben. Wenn dieser Teil nicht von den bzw. nicht von allen Entwicklern, sondern von einer Testabteilung übernommen wird, können nicht beide Parteien auf demselben Sand arbeiten – entweder nur Korrekturen oder nur Erweiterungen. Deswegen wird ein Branch erzeugt; der aktuelle Stand wird innerhalb eines Repositorys kopiert und speziell markiert abgelegt. Die Tests und Fehlerkorrekturen werden in diesem Zweig erledigt währenddessen dürfen im Hauptzweig neue Features entwickelt und eingecheckt werden. Die Korrekturen aus dem Releasezweig fließen ebenfalls ein. Möglicherweise werden aus dem Releasezweig Patches für die Software erstellt.



Wenn ein Feature größere Änderungen erfordert, kann es sinnvoller sein, einen Branch dafür zu erstellen. Dort können Änderungen problemlos durchgeführt und eingecheckt werden, ohne die Stabilität des Hauptzweiges zu beeinträchtigen. Damit die Abweichungen nicht zu groß werden, sollten regelmäßig die Änderungen im Hauptzweig in den Entwicklungszweig übernommen werden, damit keine Konflikte entstehen bzw. die Konflikte leichter zu beheben sind. Stellt sich im Laufe der Entwicklung heraus, dass das Feature nicht oder nicht auf die vorgesehene Weise umzusetzen ist, kann man den Entwicklerzweig einfach verlassen. Hat man Erfolg, dann werden die Änderungen in den Hauptzweig übernommen und der Entwicklerzweig gelöscht (damit man nicht versucht, die Änderungen nochmal zu integrieren).



Branches sind manchmal auch sinnvoll um eine Datei zu erstellen, z.B. für eine speziell angepasste Produktversion. Nicht immer läßt sich so ein Unterschied auf andere geeignete Weise im Buildprozess unterbringen; ein Branch kann einfacher sein.

4.2 Wie Änderungen gespeichert werden (können)

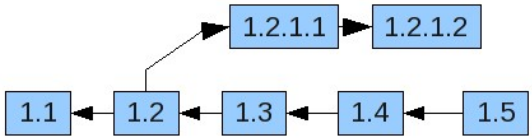
In CVS werden Änderungen als „Patches“ zwischen den Versionen gespeichert; die neuste Version wird im Klartext gespeichert, ältere werden bei Bedarf aus Patches erstellt. Diese Vorgehensweise verkleinert in der Regel den zur Archivierung benötigten Speicherplatz, erlaubt aber dennoch schnellen Zugriff auf die neueren Stände (der viel öfter passiert als der Zugriff auf die älteren Stände). Da bei CVS das Repository im Dateisystem einsehbar ist, kann man theoretisch Änderungen an einer Datei nachvollziehen, ohne CVS danach zu fragen. Als Bsp.: Auszüge aus einer solchen Datei: (hw.c,v):

```
head     1.3,
access;
smbols
    initial 1.1.1.1 cvstest 1.1.1.1;
locks, strict;
```

```
comment @ * @;

1.3
date 2009.04.15.21.07.56; author hg10597 sateExp;
branches;
next     1,2;

desc
@@

...

1.3
log
@@changed printf to puts
@
text
@#include <include stdio.h>
int main(int argc, char *args[]) {
     if (argc >1) {
          fprintf(stderr, „no arguments allowed\n“);
     }
     return 1;
     puts(„hello world“);
     return 0;
}
@

1.2
log
@added check for arguments
@
text
```

```
@d8 1
a8 1
     printf(„hello world!\n“);
@

1.1
log
@Initial revision
@
text
@d4 4
@
```

4.3 Wie kann man sich die Historie von Branches bei CVS vorstellen?



4.4 Welche Dateien werden archiviert

Im allgemeinen werden nicht wirklich alle Daten eines Projektes versioniert. Class- oder Objektdateien oder mittels JavaDoc generierte Dokumentation lassen sich beispielsweise leicht aus den Quelltexten erzeugen und müssen (bzw. sollten) nicht im Repository hinterlegt werden. Der Grund dafür ist nicht etwa Speicherplatz oder Zugriffszeit zum Ein- und Auschecken, sondern die Gefahr, dass Informationen veralten können.

Wenn „relevante“ Änderungen am Quelltext vorgenommen werden, die entsprechenden Dateien aber nicht neu generiert oder nicht neu eingecheckt werden, ist der Stand im Repository inkonsistent.

Im allgemeinen gilt: nur Sachen einchecken, die nicht generiert werden können. Manchmal gibt es aber Ausnahmen: Wenn die Generierung teuer oder umständlich ist, sie z.B. lange dauert oder eine teure Software gebraucht wird, die deswegen nur einmal auf einem bestimmten Computer installiert ist, kann es sinnvoll sein, auch die so erzeugten Dateien einzuchecken, damit alle Entwickler darauf zugreifen können. Idealerweise wird die Generierung automatisiert, damit es keine oder nur kurze Inkonsistenzen im Repository gibt; das Generieren kann sogar direkt beim Einchecken der betroffenen Quelltextdateien passieren.

4.5 Properties / Ignorieren von Dateien

Bei jeder Statusabfrage werden auch Dateien angezeigt, über die die Versionsverwaltung nichts weiß. Damit diese Meldungen nicht den Blick aufs Wesentliche verdecken, kann man sie bei der Anzeige ausblenden lassen.

```
svn st
? .class
? .class
! wichtig.java
? .class
M .java
```

Je nach Versionsverwaltung ist das Vorgehen dafür unterschiedlich. Bei CVS wird der Name der Datei oder ein Muster (wie z.B. *swp; *.class) in die Datei „.cvsignore“ eingetragen. In Subversion gibt es sogenannte „Properties“, die solche und andere Einstellungen erlauben. Die reservierten Namen beginnen mit „svn:“. Die Property zum ignorieren vom Dateien heißt „svn:ignore“.

4.6 Grafische Programme für Versionsverwaltung

Bei vielen Entwicklungsumgebungen gibt es heute die Möglichkei ein Versionsverwaltungssystem zu nutzen. Darüber hinaus existieren verschiedene Programme, die einen grafischen Zugriff auf das Repository ermöglichen.

Über spezielle Erweiterungen lassen sich auch Dateimanager wie der Windows-Explorer so nachrüsten, dass versionierte Dateien und Verzeichnisse speziell markiert angezeigt werden und Checkout und -In durch Klick möglich wird. Meist passiert das intern über den Aufruf der entsprechenden Kommandozeilenprogramme.

4.7 Eine Auswahl von Programmen

4.7.1 Standalone-Clients

- Wincvs (Windows)
- KDEsvn (KDE)
- RapidSVN (Linux, Windows)
- gitk, git-gui (Tcl/Tk-Application)

4.7.2 Desktop-Clients

- Ksvn (KDE)
- TortoiseCVS (Windows)
- TortoiseSVN (Windows)

4.7.3 Plugin-Clients

- Subclipse (Eclipse)
- JGit / EGit (Eclipse)
- VisualSVN (Visual Studio, nutzt TortoiseSVN)

4.8 Vergleich Subversion – Git

Git ist ein „verteiltes“ System. Jeder Benutzer hat eine lokale Kopie des gesamten Repositories. Die meisten Operationen laufen lokal ab und sind daher schnell; nur eine Synchronisation des lokalen Repositories mit einem nicht lokalen, benötigt Netzwerkzugriff. Dagegen kann beim Zugriff auf ein zentrales Subversion-Repository das Netzwerk einen Flaschenhals bilden, besonders, wenn viele Operationen zwischen verschiedenen Ständen ausgeführt wird. Ein Checkout zwecks einfachem Holen von Dateien ist aber günstiger zu haben, weil keine Historie kopiert wird. Um dieses Problem zu umgehen, kann man Git-Repositories klein halten. Statt selbst alle Informationen und abhängige Teile zu enthalten gibt es Verweise auf andere Repositories (sog. Sub-Module).

Umgekehrt liegt in einem Subversion-Repository alles, was zu einem Projekt gehört. Damit ist auch

klar, was alles für ein Backup gesichert werden muss. Durch die verteilte Natur von Git ist das dort nicht der Fall. Für eine komplette Sicherung müssten möglicherweise viele andere Repositories herunter geladen werden, nur um sicher zu gehen, dass es auch von ihnen ein Backup gibt. Aufgrund der verteilten Natur gibt es keine besonderen Zugriffsrechte auf ein Repository. Jeder entscheidet selbst, von wem er welche Änderungen übernimmt. Eine Ausnahme ist ein mögliches, öffentliches Repository, bei dem man verschiedene Benutzer einrichten kann. Bei Subversion werden Branches im öffentlichen Repository angelegt. Jeder mit Leserechten auf das Repository kann also den Branch sehen. Da Git immer lokal arbeitet, sind auch die Branches erstmal lokal; niemand anderes bekommt mit, wenn ein Branch angelegt wird, was auch die Namensgebung des Branches deutlich vereinfachen kann (Man kann triviale Namen wie „test“ oder „feature.xy.test“ verwenden, was in einem öffentlichen Repository weniger leicht möglich ist).

Aktuelle Versionen von Subversion untertützen das automatisierte Mergen recht gut. Bei früheren Versionen bereits musste man zumeist genau aufpassen, welche Versionen bereits gemerged wurden. Git ist hier aber weiter entwickelt, denn Merges werden häufiger verwendet. Z.B. Wenn eine neue Version von jemand anderen ins eigene Repository eingepflegt wird. Bei Merges in Subversion erscheint außerdem immer die Person, die den Merge durchführt, als der Autor, der Änderungen – auch wenn jemand anderes die Änderungen im Branch gemacht hat. In solch einem Fall ist diese Information falsch.

Abgesehen von zusätzlichen Checkoutinformationen und mehrfachen Repository ist auch die Speicherung effizienter. Das Mozilla-Repository braucht in Subversion etwa 12GB, in Git fallen 420MB an (Zum Vergleich mit CVS ist das Repository etwa 3GB groß). Allerdings fällt die Größe pro Repository an; die Gesamtgröße bei vielen Git-Anwendungen werden wahrscheinlich deutlich höher sein als bei einem Subversion-Repository mit vielen Checkouts. Da Subversion recht verbreitet und älter ist, gibt es dafür mehr Programme und Plugins für IDEs und grafische Schnittstellen.

In Subversion ist es möglich nur einen Teil des Repositories auszuchecken, z.B. wenn man nur ein Verzeichnis oder ein Unterprojekt haben möchte. Bei Git bekommt man immer den kompletten Stand im Repository (das man sich aber auch erst besorgen muss). Es ist aber möglich solche „Teile“ im Voraus in einzelnen Repositories zu pflegen und als externes Repository einzubinden.

Die Versionsnummern bei Subversion sind natürliche Zahlen, und daher relativ einfach. Selbst beim Mozillaprojekt sind die Repository-Versionsnummern nur sechs Stellen lang. Git versioniert mit SHA-1-Hashes, die Revisionsnummern sind daher 40Bytes lang. Bei der Angabe einer Version reicht aber ein eindeutiger Teil der Revisionsnummer aus.

Da Git SHA-1-Hashes zur Versionierung verwendet, kann man leicht prüfen, ob die Daten im Repository denen entsprechen, die eingecheckt wurden. Wenn der Hash-Wert der neuesten Version mit dem beim Einchecken übereinstimmt, stimmt das gesamte Repository überein. Das gilt auch dann, wenn man z. B. Durch einen Plattencrash das eigene Repository verloren hat und sich eines aus einem Backup oder von woanders holt; gleicher Hashwert; gleicher Inhalt.

Einen Vergleich von Versionsverwaltungssystemen gibt es unter
http://en.wikipedia.org/wiki/Comparison_of_revision_control_software.

5 Debugging

Geschichte: Bereits Thomas Edison hat den Begriff „Bug“ verwendet und schrieb 1878, dass „Bugs - kleine Fehler und Schwierigkeiten“ bei der Entwickelung seiner Erfindungen auftraten.

Der erste „tatsächliche“ Bug war eine Motte, die 1947 in den Relais von Harvard Mark II ums Leben kam. Sie wurde mit Tesafilm ins Logbuch und mit dem Eintrag „First actual case of bug being found“ versehen.

5.1 Was ist Debugging?

Unter Debugging versteht man im Allgemeinen das Suchen und Beheben von Fehlern in Programmen. Die Vorgehensweisen dabei können sehr unterschiedlich sein und hängen stark von der Art des Programms ab. Fehler in Betriebssystemen oder Microcode sind oft schwerer zu Untersuchen und erfordern andere Methoden als Fehler in Applikationen.

5.2 Welche Arten von Debugging gibt es?

Quellcode lesen / Nachdenken

Bei selbstentwickelten oder quelloffenem Code kann man durch lesen des Quellcodes und einem „Schreibtischtest“ nachvollziehen, was ein Programm gerade macht, und welchen internen Zustand es hat.

In manchen Fällen ist das beinahe die einzige Möglichkeit, Fehlerursachen zu finden, z.B. wenn die entsprechende Zielhardware nicht verfügbar ist, oder Bootloader, die ins RAM geladen werden und dann dort laufen.

Unabhängig von der eigentlichen Fehlersuche ist das Lesen von Quellcode im Sinne eines gemeinsamen Code-Reviews ein sinnvolles Mittel um möglichst schon im Vorfeld Fehler und Schwachpunkte in einem Programm zu finden; sei es im Entwurf oder im Programmierstil. Als stärkster Verfechter von Reviews gilt wohl Extreme Programming, bei dem immer zwei Entwickler an einem Computer sitzen und gemeinsam arbeiten.

Die Effizienz einer solchen Vorgehensweise ist oft erstaunlich hoch.

5.3 Debugausgaben

Um den Zustand eines Programms verfolgen zu können, kann man verschiedene Dinge ausgeben, z.B. die Namen von aufgerufenen Funktionen oder die Werte von Variablen. Ein Nachteil hierbei ist, die benötigte Zeit zum Programmieren der Ausgabe und Übersetzen des Programmes. Möglicherweise müssen die Aufrufe auch wieder entfernt werden, wenn man den Fehler gefunden hat. Speziell Funktionen oder Makros erlauben die Ausgabe zu deaktivieren, ohne die

entsprechende Anweisungen zu entfernen.

```
#ifdef DEBUG
     printf(...);
#endif

#ifdef DEBUG
     #define dbgout(x)x

#else
     #define dbgout(x)
#endif

dbgout(...);
```

```
#ifdef DEBUG
     log(...) {
          if(logging) {
               ...
          }
     }

#endif
     log(...);
#ifndef DEBUG
     #define log(...)
#endif
```

Auch das Schreiben einer Logdatei kann sehr nützlich bei der Fehlersuche sein, besonders, wenn man diese Funktion auch in der Releaseversion aktivieren kann. Auf diese Weise kann man – je nach Genauigkeit der Ausgabe – ziemlich genau verfolgen, was passiert ist. Verschiedene Prioritäten der Meldungen können dabei helfen (z.B. „Debug“, „Info“, „Warnung“, „Fehler“).

Debugausgaben wie Portumschaltungen o.Ä. Bei hardwarenahen Programmen gibt es vielleicht keine Möglichkeit eine Meldung auszugeben; oftgibt es nichtmal eine Anzeige per Text. In solchen Fällen kann es sinnvoll sein, den Zustand des Programms über LEDs oder Ports auszugeben, die per Oszilloskop überprüft werden müssen.

5.4 Verwendung eines Debuggers

Wenn möglich ist das Verwenden eines Debuggers meist die einfachste und komfortabelste Lösung. Im Idealfall kann man das Programm an jeder beliebigen Stelle anhalten, den Programm- und Maschinencode und die Aufrufreihenfolge sehen und schrittweise durchgehen, die Werte von Variablen, Ausdrücken, Registern und Speicher anzeigen und ändern lassen, und vielleicht sogar den Code zur Laufzeit anpassen.

5.5 Welche Arten von Fehlern gibt es?

Bei syntaktischen Fehlern entspricht die Eingabe nicht der Grammatik der Sprache. Beispiele sind ein fehlendes Komma oder Semikolon, Zeichenketten, die über das Zeilenende hinausgehen, und fehlende oder überzählige Klammern. Syntaxfehler werden vom Compiler oder Interpreter entdeckt.

Beisiele:

`if (a == b)` => runde Klammern fehlen.

`while ( i > 0 ) {` => schließende runde Klammer fehlt.

`j = 5;` => Semikolon fehlt.

Bei semantischen Fehlern entspricht die Eingabe der Grammatik. Dennoch ist sie nicht gültig. Manche Fehler kann der Compiler oder Interpreter finden. Andere Konstruktionen sind erlaubt,

haben aber eine andere Bedeutung als beabsichtigt. Gerade bei Sprachen wie C oder ähnlicher kompakter Syntax entstehen leicht solche Fehler durch falsche Prioritäten.

Beispiele für ungültigen Code:

```
a = srand(0);
b = exit(1);
```

Einer Variablen wird das Ergebnis einer Funktion zugewiesen, die kein Ergebnis liefert.

`int i, *j = &i;` => Einer Int-Variablen wird ein Pointer zugewiesen.

In Ansi-C ist so etwas verboten (solange kein Typecast verwendet wird).

Beispiele für gültigen Code.

`if (a == b)` => Zuweisung statt Vergleich (zweites '=' fehlt)

`if (( c = fgetc() ) != EOF )` => c bekommt das Ergebnis des Vergleichs zugewiesen, nicht das gelesene Zeichen Zeichen (zusätzliche Klammerung fehlt)

`if ( a && b )` => bitweise Vergleich von booleschen Ausdrücken (zweites '&' fehlt)

`c1 = ( bcd & 0x0F ) >> 4; c2 = bcd & 0x0F;`

=>Shiftoperation hat eine höhere Priorität als '&' (Klammerung fehlt).

Manchmal sind die verwendeten Funktionen fehlerfrei, das Ergebnis ist aber trotzdem nicht das Gewünschte. Das kann passieren, wenn man einen falschen Algorithmus verwendet. Ein Beispiel ist ein instabiler Sortiere-Algorithmus statt eines stabilen.

Durch „vergessene“ Anweisungen oder Prüfungen entstehen Fehler, die zwar schnell auffallen (falls der Fehler überhaupt eintritt!), aber nicht immer leicht zu finden sind, weil der Code „richtig aussieht“.

Beispiele:

```
int *p= (int *) malloc (100 * sizeof(p[0]));
if ( p == NULL ) ...  // Test auf NULL fehlt
*p=5;

int *p = new int[100];    // vom Compiler abhängig.
*p = 5;

FILE *f = fopen(„ttt“, „rb“);
if ( f == NULL ) ... => Test auf NULL-Zeiger fehlt
fread(buf, sizeof(buf[0]), sizeof(buf) / sizeof (buf[0]), f);

$$= newNametype($1.line, symbol($1.val));  // Zuweisung des Ergebnisses fehlt
```

Wenn nicht alle möglichen Ergebnisse einer Operation abgefragt und behandelt werden, bleibt ein Fehler manchmal unentdeckt – bis er später doch auffällt.

Beispiele:

```
FILE *f = fopen("ttt", "rb");
if (( f == NULL) && ( errno == EACCES )) { ... };
BESSER:
if ( f == NULL ) {
     if ( errno == EACCES ) { ... }
     ...
} // Test auf andere Fehler vergessen.
switch ( pid = fork() ) {
     case -1: perror("");exit(1);
     case  0: break;
     default: break;
     ...
} // Fehlerfall "-1" nicht behandelt.
```

Manchmal schwer zu finden sind zu kleine Variablen für die benötigten Werte. Einige Firmen haben deswegen Vorschriften zur Benennung von Variablen, damit man Ihnen die Größe direkt ansieht; z.B. u16Length oder i8Temparatur.

Beispiele:

~~char~~ `int c = fgetc(f); // fgetc liefert int zurück(wg. EOF)`

`for ( int i = 0; i < 100000; ++i )` // int hat einen Wertebereich von -32767 bis 32767; mehr ist nicht garantiert.

Nicht nur in C/C++ muß man bei den Arraygrenzen aufpassen. Man kann leicht versuchen, vor Anfang oder hinter dem Ende zuzugreifen. In Pascal kann man die Grenzen sogar selbst definieren, z. B. „`Array[1..20] of integer`“ oder „`array[15..17] of byte`“.

Beispiele:

`for ( i = 0; i <= arraysize; ++i ) { ... };` // Element arraysize gehört nicht mehr zum Array.

Oder: for `( i = 0; i < sizeof(array)/sizeof(array[0]); ++i )`

`arr[c]++;` // Je nach Wert von c kann das ein Zugriff vor das Array sein. Abhängig von Compiler und Optionen sind Characters vorzeichenbehaftet oder vorzeichenlos.

Eine besonders tückische Fehlerart sind Fehler im Compiler. Der eigene Quelltext ist korrekt, das erzeugte Programm aber nicht. So etwas ist schwer nachzuweisen, wenn es gelingt, muß man trotzdem (erstmal) um den Fehler herum programmieren. // Beispiellos

Wenn falsche Spezifikationen oder Anforderungen vorliegen, ist man als Entwickler oft chancenlos. Wenn man unsinnige Anforderungen entdeckt oder eine Idee für eine bessere Vorgehensweise hat, sollte man nachfragen bzw. sie vorschlagen. Je nach Erfahrung mit dem Themengebiet ist das aber nicht immer möglich. Zumindest braucht man in so einem Fall keinen Fehler zu suchen, sondern kann auf die Anforderungen verweisen.

5.6 Wie äußern sich Fehler?

Im Idealfall tritt der Fehler beim ersten Test auf, und das Programm stürzt direkt ab. Das ist gut, weil man sofort bemerkt, dass etwas nicht stimmt. Außerdem wird bei einem Absturz - je nach Betriebssystem – oft ein Coredump erzeugt oder der Start eines Debuggers angeboten. In beiden Fällen kommt man leicht an die Stelle, an der sich das Problem äußert. Meist findet man die Ursache im Quelltext kurz davor. Falsche Ergebnisse fallen weniger leicht auf als ein Absturz. Durch intensives Testen kann man solche Fehler meist finden, was aber oft einen gewissen Aufwand erfordert. Je nach Anwendung kann für die Tests eine bestimmte Umgebung nötig sein, um realistische Bedingungen zu schaffen. Wenn die Software unter Laborbedingungen funktioniert, im Feld jedoch nicht, sind die Tests nutzlos bis schädlich (weil sie falsche Sicherheit geben).

Schließlich gibt es den Fall, dass das Programm längst ausgeliefert ist und ein Kunde den Fehler bemerkt. Einige Fehler sind harmlos bis lästig, aber man kann mit ihnen leben; manche kann man umgehen, indem man Programmeinstellungen ändert oder seine Arbeitsweise geringfügig anpasst. Andere Fehler sind kritisch: Abstürze, Datenverlust oder ein angreifbares System können die Folge sein.

5.7 Welche Schwierigkeiten gibt es beim Debuggen?

Was die Programmierung erschwert, erschwert meist auch das Debugging. Zu den Schwierigkeiten gehören:

- interne Abhängigkeiten von Funktionen und Datenstrukturen (Initialisierung, Konsistenz)
  Verschiedene Variablen müssen „passende“ Werte enthalten, damit ein konsistenter Zustand entsteht. Ein Beispiel hierfür ist die doppelt verkettete Liste: Wenn die Verkettung an einer Stelle falsch ist, fällt das nicht unbedingt sofort auf – aber irgendwann bestimmt.

  Die Funktion strtok speichert ihren Zustand und darf deshalb nur von einem Aufrufer genutzt werden, bis die entsprechenden Aufrufe alle abgeschlossen sind.

- Externe Abhängigkeiten, z.B. bezüglich Netzwerkverbindungen, Dateien, Plattenplatz oder Plattenfehlern
  Allgemein kann man Fehlerbehandlung nur schlecht testen, wenn der Fehler nicht auftritt. Manche Fehler muss man umständlich provozieren, um ihre Behandlung testen zu können, z.B. Speicher- oder Plattenplatzmangel, Netzwerkstörungen, fehlschlagende Prozesserzeugung oder nicht vorhandene Programme.
  Bei anderen Fehlern ist das de facto nicht möglich, z.B. ungünstige Zeitpunkte für Signale, Events oder Interaktionen mit anderen Programmen.

- Rekursionen
  Manche Formen der Rekursion wie die Fakultätsberechnung oder Operation mit einfach verketteten Listen sind einigermaßen leicht zu debuggen. Andere können deutlich schwieriger sein. Z.B. das balancieren von Bäumen. In solchen Fällen hilft oft eine grafische Darstellung mit Stift und Papier bzw. allgemein von aktuellen Werten.

- Mehrere Threads
  Fehler in Programmen mit mehreren Threads lassen sich oft schwerer finden, weil es meist in irgendeiner Form Abhängigkeiten zwischen den Threads gibt. Manchmal kann man Threads deaktivieren, um nur noch einen Pfad im Programmablauf zu haben. Das ist jedoch nicht immer möglich, manchmal behebt dies den Fehler, weil er nur bei mehreren Threads auftritt. Bei Raceconditions muß man vielleicht „raten“, wo der Fehler liegt, entsprechenden Debugcode schreiben um das Problem einzukreisen, und einen Weg finden das Fehlverhalten zu reproduzieren.
- Mehrere Prozesse
  Bei mehreren Prozessen läßt sich das Programm möglicherweise gar nicht mehr debuggen. Sollte man alle beteiligten bzw. benötigten Programme im Quellcode haben, kommt man vielleicht mit mehreren parallelen Debuggingläufen weiter. Spätestens wenn eine Form von Echtzeit hinzukommt, geht das aber nicht mehr. In so einem Fall kann eine Logdatei (oder ggf. mehrere) weiterhelfen.
- Signale und Events
  Da solche Nachrichten asynchron zum Programmablauf behandelt werden, ist es manchmal schwer, die entsprechenden Fehler zu reproduzieren, wenn man nicht weiß, in welchem Zustand das Programm war, als es die Nachricht bekommen hat und der Fehler passiert ist. Manchmal bemerkt man ihn auch erst viel später und kann ihn nicht der Nachrichtenbehandlung zuordnen. Ein Beispiel für solche Probleme ist eine Signalbehandlungsroutine, die direkt oder indirekt malloc aufruft, um Speicher anzufordern. (Hier ist zu bedenken, dass auch viele Bibliotheksfunktionen malloc aufrufen, z.B. printf). Wenn das Signal kommt, während gerade ein malloc-Aufruf aus dem Programm seine internen Datenstrukturen ändert, wird ein Absturz vielleicht erst passieren, wenn später wieder jemand Speicher anfordert oder freigibt.
- Callbacks
  Callbacks sind eine Form von Nachrichen, die bei grafischen Programmen verbreitet sind. Abhängig davon, welche Nachrichten Sie verarbeiten, kann auch hier das Debugging schwierig sein. Callbacks zum Zeichnen eines Fensters kann man nicht verfolgen, wenn der Debugger dabei in den Vordergrund kommt und das Programmfenster überdeckt. Wenn man dann den Programmlauf fortsetzt, dann wechselt der Fokus wieder und die Callbackfunktion zum Zeichnen wird sofort wieder aufgerufen.
- Timing und Echtzeitverhalten
  Wenn Echtzeit eine Rolle spielt, ist die Nutzung von Debuggern in der Regel nicht möglich, weil das Schrittweise abarbeiten (des Menschen) um etliche Größenordnungen langsamer ist als ein „normaler“ Programmlauf. Echtzeitverhalten gibt es z.B. bei der Kommunikation zwischen Geräten oder bei sicherheitskritischen Anwendungen.
- Heisenbugs: Programme, die sich anders verhalten, wenn man sie beobachtet
  Unterschiede zwischen Release und Testversion können entstehen, wenn Debug-Code Seiteneffekte hat.
  z.B.:
  ```
  #ifdef DEBUG
   if ( ... = ... )
  #endif
  ```

- verteilte Programmlogik
  Wenn das betroffene Programm eine Client/Server-Anwendung ist, ist möglicherweise das Debugging auf mehreren Rechnern nötig. (vielleicht tritt das Problem nicht auf, wenn Client und Server auf derselben Machine laufen.) Hier kann eine Analyse der ausgetauschten Daten sinnvoll sein, um zu prüfen, welche Seite den (ersten) Fehler verursacht. Falls alle übertragenen Daten korrekt sind, ist der Fehler auf der Seite zu suchen, auf der er sich äußert.

5.8 Defensives Programmieren

Fehler sollten so früh wie möglich entdeckt und behoben werden. Viele Probleme kann der Compiler finden. Aus diesem Grund sollte (zumindest für den eigenen Code) immer die höchste Warnstufe eingeschaltet sein. Weiterhin müssen die Warnungen auch beachtet werden! In vielen Fällen weist eine Warnung auf einen Fehler hin. Selbst wenn das nicht der Fall ist, sollte der Quelltext so verändert werden, damit die Warnung nicht mehr auftritt. Andernfalls riskiert man, zwischen unwichtigen Warnungen die wichtigen zu übersehen. Manchmal müssen einige Warnungen für Teile des Quelltextes abgeschaltet werden, weil der Code richtig ist, aber nicht (mit vertretbarem Aufwand) so geändert werden kann, dass die Warnungen nicht mehr auftreten. Die dafür nötigen Anweisungen lassen sich in entsprechende Include-Dateien verlagern, um den Quelltext selbst portabel zu halten.

```
#pragma warning(4711, off)
...
#pragma warning(4711, default)


hallo.c, 42: warning 4711
...
#include „warn4711off.h“
...
#include „warn4711default.h“
```

Oft gibt es eine Compileroption, um Warnungen als Fehler ausgeben zu lassen, sodass man sie nicht ignorieren kann. In den früheren Zeiten von C hat der Compiler quasi alles gemacht, was man ihm gesagt hat. Da das Typsystem so gut wie alle Zuweisungen erlaubte (`int *p = 42;`) und selbst die Anzahl der benötigten und übergebenen Funktionsargumente nicht geprüft wurde, konnten sehr leicht Fehler entstehen. So ein Verhalten war beschlossen worden, um den Compiler kleiner und schneller machen zu können. Für die Fehlersuche im Quelltext gab es das Programm „lint“, das sogar genauer prüft als moderne Compiler. Weil es ein eigenständiges Programm ist, ist es leider möglich, seine Existenz zu ignorieren und es nicht zu benutzen. Mit Zusicherungen (engl. asserts) kann man auf einfache Weise bestimmte Prüfungen durchführen. Wenn die Bedingung nicht erfüllt ist, wird eine entsprechende Meldung ausgegeben, die die Bedingung, den Namen und die Zeile, der Quelltextdatei enthält, und das Programm wird sofort beendet (wobei meist ein Coredump geschrieben wird). Abgesehen von der Meldung, wo das Problem aufgetreten ist, besteht der Vorteil von Zusicherungen darin, dass man sie „ausschalten“ kann, indem man das Makro NDEBUG definiert; dann werden sie durch Code wie „`((void) 0)`“ ersetzt, der vom Compiler entfernt

wird. Allerdings heißt das auch, dass man keine Prüfungen damit machen darf, die in der Releaseversion relevant sind; Fehlerbehandlung muß auch dort stattfinden.

Sinnvolle Begriffe für Zusicherungen sind Funktionen, die einen Zeiger auf ein Objekt als Parameter haben oder die Wurzel einer Zahl berechnen sollen. Nach der Anweisung „`assert (p!=NULL);`“ bzw. „`assert(x>=0);`“ kann man sicher sein, dass kein Nullzeiger bzw. kein negative Zahl übergeben wurde. Wenn das doch der Fall ist und die Zusicherung fehlschlägt, ist der Aufrufer schuld, weil er ungültige Werte übergeben hat, und er wird deutlich darauf hingewiesen.

5.9 Vorgehensweise bei der Fehlersuche

Wenn einem ein unerwartetes Verhalten eines Programms auffällt, aber das Programm verhält sich korrekt, sollte man dem nachgehen. Um festzustellen ob man einen Fehler entdeckt, hat, muss man sich klarmachen, welches Programmverhalten denn gewünscht ist.

Wenn man erkennt, dass es sich um einen Fehler handelt, ist die weitere Vorgehensweise oft abhängig vom Projekt. Generell sollte man die Version feststellen, in der man den Fehler beobachtet hat, und ihn reproduzieren. (Fehler, die nicht reproduzierbar sind bzw. zu sein scheinen, sind die schlimmsten, weil man von einer Korrektur, falls eine möglich ist, auch nach umfangreichen Tests nie wissen kann, ob sie den Fehler wirklich behoben hat.) Anschließend prüft man in neueren Programmversionen falls vorhanden, ob dort das Problem behoben ist. Falls nicht, wird der Fehler – je nach Vorschrift – dokumentiert, indem z.B. ein Eintrag in einer Fehlerdatenbank erstellt wird. Bei kleineren oder Privaten Projekten kann eine (hinreichend ausführliche) Notiz ausreichen, wenn man das Problem nicht sofort löst. Je nach Schwere des Fehlers kann man ihn vielleicht zurückstellen und später beheben. Normalerweise ist es jedoch sinnvoll, das sofort zu tun, da es auch die Planungssicherheit erhöht. Man kann schätzen, wie lange es dauert, ein bestimmtes Feature zu implementieren, aber niemand kann sagen, wie lange es dauert, einen Fehler zu finden.

Wenn man den Fehler reproduzieren kann, empfiehlt es sich, die entscheidenden Schritte dazu zu ermitteln.

Wenn man weiß, unter welchen Umständen genau der Fehler auftritt, hat man oft schon eine Idee, wo das Problem denn liegen könnte. Obwohl man dabei leicht ins „Spielen“ verfällt, ist es nicht effektiv, endlos herum zu probieren, bis man etwas Brauchbares gefunden hat. Nach einer vorher festgelegten Zeit (z.B. zehn Minuten, es sollte nicht zu lange sein) ist eine strukturierte Vorgehensweise besser. - Nein, ausprobieren alleine ist nicht strukturiert.

Man sträubt sich leicht gegen den formalen Ansatz, weil man denkt, anders ginge es schneller, und die zusätzliche Arbeit könne man sich sparen. Das ähnelt dem Irrglauben, ein nicht funktionierendes Stück Software unbedingt korrigieren zu müssen, weil es viel zu lange dauern würde, es neu zu schreiben.

Ab einem gewissen Komplexitätsgrad (und der ist kleiner, als man denkt) ist es sinnvoll, Ein- und Ausgaben und Verhalten des Programms in schriftlicher Form zu dokumentieren. Es erleichtert das Finden der für Fehler wesentlichen Schritte und erschwert, dass man versehentlich den selben Test mehrfach macht, aber einen anderen vergisst. Hierzu ein Auszug aus dem Buch „Why programs fail“: In „Zen and the Art of Motorcycle Maintenence“, Robert M. Pirsig writes about the virtue of a logbook in cycle maintenance:

> „Everything gets written down, formally, so that you know all the time where you are, where you you've been, where you're going, and where you want to get. In scientific work and electronics technology this is neccassary because otherwise the problems get so cmplex

> you get lost in them and confused and forget what you know and what you don't know and have to give up.“

And beware – this quote applies to motorcycle maintanence. Real programs are typically much more complex than motorcycles. For a motorcycle maintainer it would probably be amazing that people would debug programs without keeping logbooks.

Zu einer guten Fehlerbeschreibung gehören neben der Programmversion und äußeren Gegebenheiten (System, Einstellungen, Erweiterungen, usw) die gemachten Eingaben, die erwarteten Ausgaben und die tatsächlichen Ausgaben. Mit diesen Informationen stellt man eine Vermutung auf, wo das Problem liegt. Dabei wird man zu Beginn oft „raten“. Entsprechend der Vermutung sagt man vorher, wie sich das Programm bei anderen Eingaben verhält, oder welchen inneren Zustand es hat. Durch entsprechende Tests oder mit der Hilfe des Debuggers prüft ma, ob die Vorhersage zutrifft oder nicht. Dementsprechend wird die Vermutung bestätigt oder Verworfen und durch eine andere oder genauere Vermutung ersetzt. Diese Vorgehensweise wiederholt man, bis man eine „zuverlässige“ Vermutung für den Fehlergrund hat, die in der Regel auch die korrekte Korrektur beinhaltet (z.B. bedeutet „die Funktion bekommt eine um eins zu große Arraygröße übergeben“, dass das Verringern des Argumentes um eins die Größe korrigiert: „bubblesort(arr, argc);“ wird geändert in „bubblesort(arr, argc – 1);“). Man wendet diese Korrektur nun an, stellt eine neue Vermutung auf (meist die, dass der Fehler behoben ist), und wiederholt den zuvor fehlgeschlagenen Test. Falls der Fehler behoben ist, ist man fertig; falls nicht, muss man entscheiden, ob die „Korrektur“ zurückgenommen werden muß. „Falsche“ Korrekturen bedeuten neue Fehler. Manchmal bekommt man aber auch eine Korrektur für einen Fehler, den man gar nicht gesucht hat, den man aber trotzdem beheben sollte.

Wie immer sind umfangreiche Tests sinnvoll, um sicherzustellen, dass der Fehler behoben und kein neuer eingebaut wurde. Wenn automatisierte Tests existieren, sollten neue Tests für den untersuchten Fall geschrieben werden, damit sich der Fehler nicht unentdeckt wieder einschleicht. Schließlich werden die korrigierte Version und die neuen Testfälle in die Versionsverwaltung eingecheckt. Wenn es zu dem behobenen Fehler einen Eintrag in einer Fehlerdatenbank gibt, ist auch hier entsprechend zu verfahren; meist wird das das Schließen des Eintrags bedeuten.

5.10 Testen

Wenn möglich sollten alle Tests auf Knopfdruck vollautomatisch ablaufen. Auf diese Weise kann man „mal eben“ prüfen, ob eine Fehlerkorrektur oder Funktionserweiterung neue Fehler in das Programm eingebaut hat. Leider ist das für die meisten Programme nicht so einfach, z.B. weil grafische Programme nur schwer automatisiert werden können, oder weil ein Benutzer sich in einigen Punkten eben doch anders verhält, als eine Simulation.

Es genügt nicht, wenn nur korrekte Eingaben auf richtige Ergebnisse getestet werden. Ungültige Eingaben müssen korrekt zurückgewiesen werden, und auch für solche Fälle sind Tests nötig.

Nach Änderungen an einem Programmabschnitt müssen mindestens die Tests durchgeführt werden, die den entsprechenden Abschnitt testen. Ideal wäre ein kompletter Test, was aber in der Regel aus Zeitgründen nur bei automatisierten Tests möglich ist. Im Idealfall werden die Tests geschrieben, bevor der Code dazu geschrieben wird. Alternativ können Code- und Testautor/Tester verschiedene Personen sein. Andernfalls ist man leicht vom Code beeinflusst, oder hat eine eingeschränkte Sichtweise und testet deswegen zu wenig. Leider ist das meist nicht so einfach, unter anderem weil die Anforderungen während der Implementierung noch geändert oder präzisiert werden.

Wenn ein Fehler entdeckt wurde, für den es noch keinen Test gibt, wird einer entwickelt. Auf diese Weise kann man sicher stellen, dass der Fehler auch in der nächsten Version nicht enthalten ist.

6 Shellscripts

6.1 Was ist ein Shellscript?

Prinzipiell sind Shellskripts eine Folge von Shellbefehlen, die man auch interaktiv eingeben könnte. Im einfachsten Fall ersetzt man also eine Reihe von Befehlen durch den Aufruf des Skripts.

6.2 Warum verwendet man Shellskripts?

Vorteile:

- relativ leicht zu schreiben, zu verstehen und zu ändern
- Programmierung auf hoher, abstrakter Ebene
- Einigermaßen portabel (abhängig von der verwendeten Shell und den aufgerufenen Programmen)

Nachteile:

- weniger effizient als compilierter Code (langsamer)
- dauernd neue Prozesse
- Abhängigkeiten zu vorhandenen Programmen und Programmvarianten

6.3 Entwickeln von Shellskripten

„Shell“ ist im Prinzip eine Programmiersprache, bei der die meisten Anweisungen aus Aufrufen von Programmen bestehen. Die zahllosen existierenden Shells unterscheiden sich u.A. durch ihre Syntax, ihre Fähigkeiten und Namen von eingebauten Kommandos. Ähnlich der Assemblersprache wird auch hier die Programmierung umso leichter, je mehr Programme man kennt.

Im Rahmen dieser Vorlesung sind mit Shellskripts Skripts für die Bourne-Again-Shell (bash) gemeint, die Standard-Shell bei den meisten Linux-Distributionen und MacOS X ist.

6.4 Ausführen von Skripts

Prinzipiell gibt es zwei Möglichkeiten, Skripts auszuführen. Zum einen kann man den entsprechenden Interpreter starten und ihm den Skriptnamen und ggf. weitere Argumente übergeben. Zum anderen kann man die Datei ausführbar machen und den Namen des Interpreters und ggf. nötige Argumente in die Shebangzeile schreiben, z.B.

```
#!/bin/bash, #!/opt/local/bin -F oder #!/usr/bin/expect --
```

6.5 Umleitungen

Die Ein-, Aus- und Fehlerausgaben lassen sich auf verschiedene Weise umleiten. Am leichtesten zu verstehen sind <, > und 2>, die die Streams auf Dateien umleiten: Eingaben werden aus der angegebenen Datei gelesen (statt von der Standardeingabe, meist die Tastatur), Ausgaben werden in

die angegebene Datei geschrieben (statt auf die Standard- oder Standardfehlerausgabe, meist das Terminal). Mit der Anweisung 2>&1 wird die Standardfehlerausgabe auf die Standardausgabe umgeleitet, was beim Protokollieren oder Betrachten von Ausgaben (z.B. mit less) hilfreich ist. Umgekehrt kann mit >&2 die Standardausgabe auf die Fehlerausgabe umgelenkt werden, was z.B. beim Ausgeben von Fehlermeldungen mit echo sinnvoll ist. Wenn es mehr als eine Umleitung für ein Kommando gibt, ist die Reihenfolge der Umleitung wichtig.

`Cmd >t 2>&1` // Standard-Output in Datei t umgeleitet, danach Error auf stdout

`Cmd 2>&1 >t` // stderr nach stdout(Konsole) danach stdout in Datei t

Bei der ersten Anweisung landen beide Ausgaben in der Datei; bei der zweiten nur die "normale" Ausgabe, während die Fehler "normal" angezeigt werden.

Eine sehr mächtige Umleitung bietet | : Die Ausgabe eines Kommandos wird zur Eingabe des nächsten. Dieses Konzept lässt sich nicht vollständig mit Umleitungen in und aus Dateien simulieren. Abgesehen vom Plattenplatz (z.B. bei Kommandos wie „dd if=/dev/hda1 | gzip -9 > dasi.gz“) ist der gleichzeitige Ablauf der Programme auch deswegen entscheidend, weil die Ausgaben des ersten Programms vielleicht gar nicht mehr gebraucht werden. Ein einfaches Beispiel ist head als zweites Programm, ein noch besseres ist yes als erstes.

6.6 Hilfe!

Das help-Kommando bietet Hilfe zu eingebauten Kommandos und Schlüsselworten. In vielen Fällen findet man damit schneller die gewünschten Informationen als in der Manpage, weil man nicht danach suchen muss. Allerdings sind die Hilfetexte manchmal weniger ausführlich.

6.7 Kontrollstrukturen

Mit einem Semikolon lassen sich Kommandos nacheinander ausführen als währen sie durch einen Zeilenwechsel getrennt. `Bash -c „a;b“ | c;d`

Für Verzweigungen gibt es die Schlüsselworte if und case, für Wiederholungen for und while. Wie in C werden Kommandos nach && bzw || nur ausgeführt, wenn der Befehl davor erfolgreich bzw. erfolglos war.

Für Kommandofolgen kann man Funktionen definieren.

```
alias ls=“ls –color=auto“
```

6.8 Exitcode von Programmen

Programme teilen über ihren Exitcode mit, ob sie erfolgreich waren: Null heißt „Erfolg“, ein anderer Wert heißt „Fehler“. Daraus ergibt sich gegenüber den meisten Sprachen eine umgekehrte Logik für Bedingungen, die man aber meist gar nicht bemerkt: Wenn der Befehl erfolgreich war, ist die Bedingung erfüllt, sonst nicht. Mit „!“ kann man dieses Ergebnis logisch umdrehen.

6.9 Befehle für Bedingungen

Da jedes Programm einen Exitcode zurück gibt kann man auch jedes als Bedingung verwenden. Besonders häufig ist hier das Kommando „test“, mit dem man Dateien auf Eigenschaften und Zeichenketten auf Gleichheit testen kann. Aus Gründen der Lesbarkeit eignet sich auch das

Synonym [, das als letztes Argument] bekommen muss. Ob eine Datei existiert, kann man mit „test -f File“ prüfen, ob es ein Verzeichnis ist, mit „[-d Dir]“. Wie üblich bedeutet der Exitcode null die Antwort „ja“.

6.10 VARIABLEN

Die Shell unterscheidet nicht zwischen "normalen" Variablen und Umgebungsvariablen. Man kann Werte mit = zuweisen und mit vorangestelltem $ auslesen. Die Anweisung "FILE = input.txt" setzt die (Umgebungs-)Variable FILE auf den Wert "input.txt". Mit "echo $FILE" kann man sich den Wert anzeigen lassen.

6.11 PARAMETER

Auch Parameter werden in der Shell in Umgebungsvariablen abgelegt. Sie sind nummeriert und beginnen mit 0 (dem Skriptnamen); dieses Konzept gleicht argv in C. Den Wert des ersten Parameters bekommt man also mit $1.

6.12 SPEZIELLE Variablen

Die Anzahl der übergebenen Argumente erhält man mit $#, alle Argumente (in doppelten Anführungszeichen) mit "$@".

```
"$@" ->   "$1" "$2" "$3"
"$*" -> "$1c$2c$3"  c = blank
```

Auf den Exitcode des letzten Befehls greift man mit $? zu, auf die aktuell eingestellten Optionen mit $-. Es gibt noch weitere solcher Variablen, aber sie werden seltener benötigt; bei Bedarf kann man sie in der bash-Manpage nachschlagen.

6.13 EINSTELLUNGSSACHE

Abgesehen von der Möglichkeit über Umgebungsvariablen Optionen einzustellen (PS1, IFS, ...), ist das auch mit dem set-Kommando möglich. Für Entwicklung und Debugging von Shellskripts sind besonders folgende Optionen hilfreich:

set -e: Wenn ein Kommando fehlschlägt, wird die Shell beendet. Das gilt aber nicht für nicht erfüllte Bedingungen: if [-f file]; then mv file file.old; fi wird einfach nichts tun, wenn file nicht existiert (oder keine Datei ist). Beispiel:

```
$ set -e; for f in *; do test -f "$f" && gzip "$f"; done
```

set -u: Nicht existierende Umgebungsvariablen bei der Expansion bedeuten einen Fehler. Beispiel:

```
$ set -u; unset X; echo $X
bash: X: unbound variable
```

set -v: Eingelesene Zeilen werden ausgegeben. Beispiel:

```
$ set -v; ls -d *
ls -d *
"..."
```

set -x: Kommandos werden ausgegeben, bevor sie ausgeführt werden. Dies unterscheidet sich dahingehend von der Option -v, dass Alias-, Variablen- und Expansionen von Dateinamen

bereits stattgefunden haben. Man erfährt also, welches Kommando "wirklich" ausgeführt wird. Beispiel:

```
set -x; ls $HOME
     ls --color=auto     /home/hg1057

$ type ls
     ls is aliased to "ls --color=auto"
$ type -a ls
     ....
     ls is /usr/bin/ls
     ls is /bin/ls
$ which ls
```

6.14 IF

Die einfache Bedingung führt den Test aus und verzweigt entsprechend des Ergebnisses. Weil if mit fi beendet werden muss, gibt es das Folgekommando elif; bei Bedingungen im Elsezweig wären sonst mehrere fis am ende nötig. Beispiele:

```
if [ -z "$DIR" ]; then DIR=. fi

if [ ! -e "$DIR" ] ; then
        mkdir "$DIR"
elif [ ! -d "$DIR" ] ; then
        echo "'$DIR' is not a directory" >&2
        exit 1
else
        echo "'$DIR' already exists"
fi
```

6.15 FOR

Mit for wird eine Anweisungsfolge nacheinander auf die angegebenen Werte angewandt. Häufig macht man soetwas für Dateien oder eine Folge von Zahlen (die oft mit Dateinamen korelliert). Beispiele:

```
for f in *.wav; do
 lame "$f" "${f%.wav}.mp3" && rm "$f"
done
     ->erzeugt Dateien mit Endung ".mp3" aus Dateien mit Endung
".wav" und
     ->löscht die Ursprungsdatei

for f in *.wav; do echo "$f"; done
     ->listet alle Dateien mit Endung ".wav" auf

for i in {20..1}; do
 let j = $i + 1
 mv -i file$i.txt file$j.txt
```

```
done

for f in *.tex; do
 sed -i-e 's/Don Knuth/Donald Knuth/g' "$f"
done
```

6.16 WHILE

Die While-Schleife wiederholt den Anweisungsblock, bis die Bedingung nicht mehr erfüllt ist. Damit sind auch manche Zählweisen möglich. Beispiele:

```
while ! mount /mnt/usb/ ; do
 sleep 1
done

i = 1; j = 1; while [ $i -lt 100 ]; do
 if [ -f a$i.txt ]; then
  mv -i a$i.txt b `printf "%02i" $j`.txt
  let j=$i+1
 fi
 let i=$i+1
done
```

6.17 CASE

Mittels case wird zwischen verschiedenen Alternativen ausgewählt. Verwendungszwecke sind unter Anderem Auswertung von Exitcodes und Parametern. Beispiel (Auszug aus bashbug):

```
while [ $# -gt 0 ]; do
  case "$1" in
    --help)    shift; do_help=y ;;
    --version) shift; do_version=y ;;
    --)        shift; break;;
    -*)        echo "bashbug; ${1}: invalid option" >&2
               echo "$USAGE" >&2
               exit 2;;
     *)        break;;
  esac
done
```

6.18 Lesen von Eingaben

Mit dem Kommando read werden Zeilen von der Standardeingbe gelesen und in Umgebungsvariablen gespeichert. Dabei werden Worte an den Wortgrenzen getrennt (Details zu Wortgrenzen unter IFS in der bash-Manpage) und nacheinander in die angegebenen Variablen geschrieben; wenn es mehr Worte als Variablen gibt, landet der Rest der Zeile in der letzten Variablen. Durch Umleitung der Eingabe kann man aus einer Datei lesen. Beispiele:

// gibt eine Datei zeilenweise aus und fasst Whitespaces zusammen

```
cat file | while read LINE; do echo $LINE; done
```

// Gibt jedes Wort einer Datei in einer eigenen Zeile aus

```
while read LINE ; do for WORD in $LINE ; do echo $WORD ; done ;
done < file
```

// Erzeugt eine tabellarische Ansicht und leitet die Ausgabe in eine Datei um

```
echo „Hammer 10.00
Schraube 0.05
Bohrmaschine 100.00“ | while read WARE PREIS; do printf „%-15s
%6.2f\n“ $WARE $PREIS; done > Preisliste.txt
```

=>

```
Hammer          10.00
.....            0.05
.....          100.00
```

6.19 Schreiben von Ausgaben

Zum Schreiben auf die Standardausgabe oder in Dateien kann man echo und printf verwenden und ggf. die Ausgabe umleiten.

7 make

7.1 Was ist make?

make ist ein Programm, das Schritte zum Erzeugen eines Projektes ausführt. Es wird meist bei Programmierprojekten eingesetzt, ist aber nicht darauf beschränkt. make bekommt in der Konfigurationsdatei, dem Makefile, gesagt, welche Dateien (Ziele, z.B. Objektdateien) von welchen (z.B. Quelltextdateien) abhängen (aus ihnen erzeugt werden). make liest dann die Zeitstempel der Ziele und Voraussetzungen aus dem Dateisystem. Wenn ein Ziel älter ist als seine Voraussetzung, werden die entsprechenden Regeln zur Erzeugung ausgeführt.

Gegenüber einem Skript, das alle Compileraufrufe für ein Projekt durchführt, hat make den Vorteil, dass nur die nötigen Schritte durchgeführt werden (also die, bei denen eine Voraussetzung neuer ist, als das Ziel). Das ist besonders wichtig, wenn es häufige Änderungen gibt, z.B. während einer Debuggingphase, während der man dauernd neu übersetzt und testet. Wenn das Übersetzen aller Dateien z.B. eine halbe Stunde dauert, kommt man mit der Fehlerkorrektur nicht so recht weiter. Das übersetzen von nur ein, zwei geänderten Dateien und das Linken des Projektes sind aber vielleicht in einer halben Minute passiert.

7.2 Aufbau von Makefiles

Makefiles enthalten die Abhängigkeiten der Ziele von Voraussetzungen und die Regeln um die Ziele zu erstellen.

Der Aufbau dabei ist:

```
Ziel : Voraussetzung
<Tabulator>Kommandos
```

Eine solche Konstruktion könnte z.B. sein:

```
hello : hello.c
     gcc -o hello hello.c
```

Auch mehrere Voraussetzungen sind möglich:

```
prog : a.o b.o
     gcc -o prog a.o b.o
a.o : a.c
     gcc -c a.c
b.o : b.c
     gcc -c b.c
```

Dabei werden zuerst die Voraussetzungen eines Ziels erstellt (falls nötig). Dazu wird beim Aufruf „make“ ein Abhängikkeitsbaum aufgebaut, der anschließend abgearbeitet wird.

Beispiel zu oben:

1.) Bisher existieren nur die Dateien, a.c und b.c. Es wird „make“ aufgerufen (ohne Argumente). Da kein Ziel angegeben ist, nimmt make das erste, das im Makefile vorkommt, also prog. prog hängt von a.o und b.o ab. Beide Dateien gibt es nicht, aber es gibt Regeln, um sie zu erstellen. Ziele, die nicht existieren, sind immer „älter“ als ihre Voraussetzungen. Um a.o aus a.c zu erzeugen, wird „gcc -c a.c“ ausgeführt, für b.o „gcc -c b.c“. Jetzt existieren a.o und b.o, die Voraussetzungen für prog. Der Aufruf „gcc -o prog a.o b.o“ erzeugt prog aus den Objektdateien.

2.) Es wird erneut „make“ aufgerufen. prog hängt von a.o und b.o die ihrerseits von a.c bzw. b.c abhängen. a.c und b.c sind nicht neuer als a.o bzw. b.o, make braucht nichts dafür zu tun, a.o und b.o sind ihrerseits nicht neuer als prog. Es gibt also nichts zu tun; prog ist aktuell.

3.) a.c wird um eine Funktion erweitert; anschließend wird „make“ aufgerufen. prog hängt ab von a.o und b.o, die ihrerseits von a.c bzw. b.c abhängen. b.c ist älter als b.o; hier gibt es nichts zu tun. Beim speichern a.c wurde jedoch der Zeitstempel geändert: a.c ist neuer als a.o. Also wird „gcc -c a.c“ auferufen um a.o zu aktualisieren. Nun muß auch das Ziel prog neu erzeug werden, denn seine Voraussetzungen sind nun neuer; es wird also „gcc -o prog a.o b.o“ aufgerufen.

Die Kommando-Zeilen müssen mit einem Tabulator beginnen; andernfalls wird make die Anwendung nicht verstehen (und eine möglicherweise nichtssagende Fehlermeldung ausgeben). Für jede dieser Zeilen wird eine (neue) Sub-Shell ausgeführt, die die Anweisung abarbeitet.

7.3 Abhängigkeiten

Objektdateien hängen meist nicht nur von ihrem Quelltext ab. Meist werden auch Include-Dateien verwendet; wenn sich dort etwas ändert, müssen alle Quelltexte, die die Inlcude-Dateien direkt oder indirekt verwenden, ebenfalls übersetzt werden (Wenn z.B. Deklarationen und Definitionen einer Funktion geändert werden, müssen auch alle Aufrufe angepasst werden. Passiert das nicht, muss beim Übersetzen des Aufruf einen Fehler ausgeben. Würden die aufrufenden Quelltexte nicht neu übersetzt, wäre „merkwürdiges Programmverhalten“ die Folge).

Header-Dateien in den Abhängigkeiten aufzuzählen ist schon nach kurzer Zeit nicht mehr vernünftig wartbar. Wenn eine Headerdatei eine andere benutzt, müßten beide aufgezählt werden;

wenn sich „irgendwo“ etwas ändert, müßte man alle Abhängigkeiten von Headerndateien aktualisieren. Aus diesem Grund macht man das nicht selber, sondern man fragt den Compiler, welche Datei von welcher abhängt. Dazu dient die Option -MM (bzw. -M, wenn include-Dateien des Systems auch aufgeführt werden sollen); die entstehende Ausgabe leitet man in eine Datei um, die man im Makefile verwendet.

Beispiel für die entsprechende Makeregeln:

```
depend.mak : $(wildcard *.c)
     gcc -MM *.c > depend.mak
-include depend.mak
```

7.4 Abbruch bei Fehlern

Wenn beim Erzeugen eines Ziels ein Fehler passiert, bricht make ab. Die folgenden Befehle sind in der Regel nicht sinnvoll, weil sie Dateien oder Bedingungen voraussetzen, die wegen des Fehlers vermutlich nicht existieren bzw. erfüllt sind.

7.5 Pseudoziele

In vielen Fällen sind die Ziele einer Regel keine Dateien, sondern werden dazu verwendet um bestimmte Aktionen auszuführen, wenn beim Aufruf das entsprechenden Ziel angegeben wird. Ein bekanntes Ziel dieser Art ist „clean“: Wenn „make clean“ aufgerufen wird, werden. z.B. alle Objekdateien gelöscht. Die Regel dafür könnte so aussehen:

```
clean:
     rm -f *.o
```

Häufig verwendet wird auch das Ziel dist-clean, das außerdem die Abhängigkeiten und das „Endziel“ löscht um für die Veröffentlichung des Archivs aufzuräumen.

```
dist-clean: clean
     rm -f prog depend.mak
```

Da make ohne Angaben von Zielen das erste Ziel aus dem Makefile verwendet, wird dort häufig „all“ definiert, das alle sinnvollen Ziele als Voraussetzung, aber keine weiteren Regeln hat:

```
all: prog
```

Weil eine vorhandene Datei mit dem Namen eines Pseudoziels das Ausführen der entsprechenden Regeln verhindern würde, gibt es den Konfigurationseintrag „.PHONY“ (künstlich, unecht), mit dem die Regeln für Pseudoziele auch in einem solchen Fall ausgeführt werden. „make clean“ würde also auch dann aufräumen, wenn eine Datei mit dem Namen „clean“ existiert. Beispiel für die Verwendung:

.PHONY: all tests clean dist-clean

7.6 Vordefinierte Regeln

In GNU make gibt es bereits etliche eingebaute Regeln um Ziele aus Voraussetzungen zu erstellen. Wenn die Datei „hello.c“ existiert, wird make mit der Anweisung make hello die folgende Regel ausführen:

%: %.c

#commands to execute (builtin)

$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

„LINK.c“ hat dabei den Wert „$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $ (TARGET_ARCH)“ es wird also der C-Compiler mit entsprechenden Flags aufgerufen. Das Prozentzeichen hat ungefähr die Bedeutung des Sterns in der Shell; allerdings wird auf beiden Seiten des Doppelpunkts die gleiche Zeichenkette dafür eingesetzt. „hello“ hängt also von „hello.c“, „foo“ von „foo.c“. Auch für Objektdateien gibt es entsprechende Regeln:

%.o: %.c

#commands to execute (built-in):

$(COMPILE.c) $(OUTPUT_OPTION) $C

Nach dem gleichen Prinzip existieren Regeln für Ausgaben des Präprozessors und für C++, Assembler-, Pascal- und Fortrandateien.

Man kann sich mit „make -np“ alle Regeln (inklusive derer des entsprechenden Makefiles) anzeigen lassen.

7.7 Variablen

Make unterstützt Variablen, deren Wert wie in der Shell mit $ ermittelt wird. Wenn der Variablenname länger als ein Zeichen ist, muss er in runde Klammern geschrieben werden, z.B. : „$(CC)“, „$(CXXFLAGS)“ und „$(SHELL)“. Wenn ein Dollarzeichen an die Shell weitergegeben werden soll, muss es verdoppelt werden, andernfalls würde make die Variable expandieren (was meist die leere Zeichenkette ergäbe). Beispiel:

printshell:

```
    @echo „the shell's SHELL variable is '$$SHELL', make's SHELL variable is '$(SHELL)'“
```

Das @ am Zeilenanfang bedeutet, dass das Kommando nicht von make ausgegeben werden soll, bevor es ausgeführt wird.

Variablen können auf verschiedene Arten gesetzt werden. Eine davon sieht so aus wie in der Shell:

CFLAGS= -Wall -pedantic -g

Bei den dabei entstehenden „rekursiv expandierten“ Variablen werden evtl. bei der Definition benannte Variablen erst dann expandiert, wenn sie benötigt werden, nicht schon bei der Zuweisung.

Alternativ ist eine Zuweisung wie in Pascal möglich:

CFLAGS := -Wall -pedantic -g

Bei diesen Variablen wird die rechte Seite bei der Zuweisung expandiert. Das ist ein Unterschied, wenn dabei Variablen benutzt werden, die erst später definiert werden oder später einen anderen Wert bekommen, z.B. wenn der Wert die Ausgabe des Programmes „date“ ist.

DATE = `date` // aktuelles Datum

DATE := `date` // immer gleich: Datum bei der Zuweisung

Mit dem Zuweisungsoperator += können Werte an Variablen angehängt werden. Bei ?= erfolgt die Zuweisung nur, wenn die Variable noch nicht existiert.

7.8 Variablen für einzelne Regeln

Gerade bei der Ersetzung mit Prozentzeichen braucht man eine Möglichkeit, sich auf die expandierten Werte zu beziehen. Hierfür gibt es u.A. die folgenden Variablen, die je nach Ziel und Voraussetzung einen entsprechenden Wert annehmen:

$@: das Ziel

$<: die erste Voraussetzung

$^: alle Voraussetzungen, durch Leerzeichen getrennt

$?: alle Voraussetzungen, die neuer sind als das Ziel, durch Leerzeichen getrennt.

Beispiel für die Benutzung:

```
$(BIN): $(OBJS)
     $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
%.o: %.c
     $(CC) $(CFLAGS) -o $@ -c $<
```

7.9 Variablen für Programme

Ähnlich, wie bei den Zielen gibt es auch für Kommandos vordefinierte Variablen. Der Vorteil ihrer Benutzung ist, dass man den Defaultwert überschreiben kann, um z.B. eine Konfiguration für ein bestimmtes System anzupassen. Einige Beispiele für Variablen und ihren Wert:

```
AR = ar
CC = cc
CPP = $(CC) -E
CXX = g++
LINT = lint
RM = rm -f
```

Nach diesem Prinzip gibt es auch Flags, die den Programmen mitgegeben werden, und die man ebenso überschreiben kann. Die Flags passend zu den angegebenen Programmen sind:

```
ARFLAGS = rv
CFLAGS = <leer>
CPPFLAGS = <leer>
CXXFLAGS = <leer>
LDFLAGS = <leer>
LINTFLAGS = <leer>
```

7.10 Probleme mit make

Da make mit Zeitstempel arbeitet, kann es Probleme gebenm wenn sich an diesen etwas unvorhergesehenes ändert. So etwas kann passieren , wenn Dateien über mehrere Zeitzonen kopiert werden. Eine Voraussetzung kann neuer sein, als ihr Ziel, aber aufgrund einer Zeitverschiebung ist ihr Zeitstempel „älter“. Manchmal ist die Auflösung von Zeitstempeln ein Problem, weil sie „Sekundenbasiert“ ist.

7.11 Neuerzeugung des Projekts

Um ein Projekt neu zu erstellen, kann man mit „clean“ oder „dist-clean“ aufräumen und dann make erneut starten. Wenn man nur ein, zwei Dateien übersetzen will, kann man mit touch ihren Zeitstempel aktualisieren. Ein anschließender make-Aufruf wird die entsprechenden Ziele dann neu erstellen.

7.12 Alternativen zu make

- Cons
- ant
- rake