

Bash Style Guide und Kodierungsrichtlinie

Mit ergänzenden Hinweisen zum Testen von Skripten

Fritz Mehner

Fachhochschule Südwestfalen, Iserlohn

mehner@fh-swf.de

Inhaltsverzeichnis

1	Zeilenlänge	2	6	Dateien	5
2	Einrückung	2	7	Kommandozeilenschalter	6
3	Kommentare	2	8	Verwendung von Shell-Funktionen	6
3.1	Kopfkommentare in Dateien	2	9	SUID/SGID-Skripte	7
3.2	Zeilenendkommentare	2	10	Testen	8
3.3	Abschnittskommentare	3	10.1	Syntaxprüfung	8
3.4	Funktionskommentare	3	10.2	Testumfang	8
3.5	Komentierungsstil	3	10.3	Verwendung von <code>echo</code>	8
4	Variablen und Konstanten	3	10.4	Testen mit Hilfe von <i>bash</i> -Optionen	8
4.1	Verwendung von Variablen	3	10.5	Testen mit Hilfe von <code>trap</code>	9
4.2	Verwendung von Konstanten	4	10.6	Der Debugger <code>bashdb</code>	9
5	Erfolgskontrollen	4	11	Weitere Informationsquellen	9
5.1	Kommandozeilenparameter	4	12	Kurzfassung	12
5.2	Variablen, Befehle und Funktionen	4			
5.3	Ablaufprotokoll, Abschlußmeldung	5			

Skriptprogrammierung steht gelegentlich in dem Ansehen, lediglich schnelle, schwer verständliche „Wegwerflösungen“ zu erzeugen, die keinen besonderen Qualitätsanforderungen genügen müssen. Dabei wird gerne übersehen, daß es viele Bereiche gibt, in denen langlebige Skripte die Regel sind: Systemverwaltung, Start und Konfigurierung von Betriebssystemen, Software-Installationen, Automatisierung von Benutzeraufgaben und anderes mehr. Alle diese Lösungen müssen selbstverständlich gewartet, erweitert und dokumentiert werden.

Bei der Skriptprogrammierung sind deshalb grundsätzlich die selben Anforderungen zu erfüllen, wie bei der Programmierung in einer Produktionssprache (Zweckerfüllung, Richtigkeit der Lösung, Erfüllung der Vorgaben, Robustheit, Wartbarkeit) und deshalb sind auch die gleichen Maßstäbe anzulegen. Ein Programm ist dann wartbar, wenn Aufbau und Wirkungsweise von jemandem, der das Programm nicht selbst geschrieben hat, zügig erfaßt werden können und damit erfolgreiche Änderungen in angemessener Bearbeitungszeit möglich sind. In welchem Maße diese Anforderung erfüllt wird, hängt wesentlich vom verwendeten Programmierstil ab. Die hier behandelten Vorgaben dienen hauptsächlich dazu, gut verständlichen und damit wartbaren Code zu erzeugen und sie sind deshalb zu beachten.

1 Zeilenlänge

Die Gesamtlänge einer Zeile (einschließlich Kommentar) sollte **88 Zeichen** nicht überschreiten. Damit kann das Suchen in Querrichtung vermieden werden und die Datei bleibt bei üblichen Papierbreiten ohne abgeschnittene oder umbrochene Zeilenenden abdruckbar. Anweisungen sind deshalb gegebenenfalls zu zerlegen, Texte sind geeignet zu umbrechen.

2 Einrückung

Die Einrückung von Programmkonstruktionen muß grundsätzlich mit der **logischen Schachtelungstiefe** übereinstimmen. Die Einrücktiefe einer Stufe stimmt üblicherweise mit der gewählten Tabulatorsprungweite des Editors überein; man wählt meist 2, 4 oder 8.

3 Kommentare

3.1 Kopfkomentare in Dateien

Jeder Datei wird ein Kopfkomentar vorangestellt, der die wichtigsten Angaben zu dieser Datei enthält. Die bevorzugte Form zeigt das folgende Beispiel:

```
#!/bin/bash
#=====
#
#     FILE:  stale-links.sh
#
#     USAGE:  stale-links.sh [-d] [-l] [-oD logfile] [-h] [starting directories]
#
#     DESCRIPTION:  List and/or delete all stale links in directory trees.
#                   The default starting directory is the current directory.
#                   Don't descend directories on other filesystems.
#
#     OPTIONS:  see function 'usage' below
# REQUIREMENTS:  ---
#     BUGS:  ---
#     NOTES:  ---
#     AUTHOR:  Dr.-Ing. Fritz Mehner (Mn), mehner@fh-swf.de
#     COMPANY:  FH Südwestfalen, Iserlohn
#     VERSION:  1.3
#     CREATED:  12.05.2002 - 12:36:50
#     REVISION:  20.09.2004
#=====
```

Gegebenenfalls sind weitere Informationen zu ergänzen (zum Beispiel Copyright-Vermerk, Projektzuordnung).

3.2 Zeilenendkommentare

Aufeinanderfolgende Zeilenendkommentare beginnen in derselben Spalte. Nach dem Kommentareleitungszeichen # folgt immer ein Leerzeichen um das Erfassen des Wortanfanges zu erleichtern.

```
found=0           # count links found
deleted=0        # count links deleted
```

3.3 Abschnittskommentare

Bilden mehrere Zeilen einen Abschnitt mit zusammengehörigen Anweisungen, dann ist ein solcher Bereich mit einem Abschnittskommentar zu versehen.

```
#-----
# delete links, if demanded write logfile
#-----
if [ "$action" == 'd' ] ; then
    rm --force "$file" && deleted=$((deleted+1))
    echo "removed link : $file"
    [ "$logfile" != "" ] && echo $file >> $logfile
fi
```

Die Einrücktiefe des Kommentars entspricht dabei der Schachtelungstiefe der nachfolgenden Anweisung.

3.4 Funktionskommentare

Jede Funktion wird durch einen Kopfkomentar beschrieben. Dieser enthält den Funktionsnamen, eine Kurzbeschreibung und die Beschreibung der Parameter (falls welche vorhanden sind). Der Name des Autors und das Erstellungsdatum sollten bei nachträglichen Ergänzungen hinzugefügt werden.

```
==== FUNCTION =====
# NAME: usage
# DESCRIPTION: Display usage information for this script.
# PARAMETER 1: ---
# CREATED: 04.11.2004 23:08:05 CET / AUTHOR: Dr.-Ing. Fritz Mehner
=====
```

3.5 Kommentierungsstil

Für den Umfang und den Stil der Kommentare gilt allgemein:

Kurz, treffend und hinreichend genau.

Umfangreiche Beschreibungen gehören in die externe Dokumentation. Nur in Ausnahmefällen wird der Aufbau oder der verwendete Kunstgriff zusätzlich beschrieben. Für Anweisungen gilt:

Der Kommentar beschreibt den *Zweck* der Anweisung.

Nur in Ausnahmefällen wird der Aufbau oder der verwendete Kunstgriff zusätzlich beschrieben. Der folgende Kommentar ist zum Beispiel nicht besonders hilfreich, da er nur das wiederholt, was am Zeilenanfang ohnehin steht:

```
[ "$logfile" != "" ] && $(> $logfile) # variable $logfile empty ?
```

Der nachstehende Kommentar sagt hingegen ganz kurz, was hier beabsichtigt ist:

```
[ "$logfile" != "" ] && $(> $logfile) # empty an existing logfile
```

4 Variablen und Konstanten

4.1 Verwendung von Variablen

Für Variablen sind grundsätzlich sinntragende, selbstdokumentierende Namen zu verwenden (wie zum Beispiel eingabedatei). Bei Namen werden die ersten 31 Zeichen unterschieden. Lange Namen werden

durch Unterstriche gegliedert um die Lesbarkeit zu verbessern. Wenn ein Name nicht selbstdokumentierend ist, dann ist die Bedeutung und Verwendung beim ersten Auftreten durch einen Kommentar zu beschreiben.

4.2 Verwendung von Konstanten

Grundsätzlich gilt für alle Programmiersprachen: **Das Einstreuen von Konstanten in den Programmtext ist zu unterlassen!** Besonders zahlenwertige Konstanten haben über ihren Wert hinaus keine weitere, unmittelbare Bedeutung. Die Bedeutung des Wertes wird erst durch den Zusammenhang mit dem umgebenden Text deutlich. Bei Wertänderungen mehrfach auftretender Konstanten ist damit eine automatische Ersetzung im Editor nicht möglich, weil derselbe Wert in unterschiedlichen Bedeutungen gebraucht worden sein könnte. Derartige Programmtexte sind also schwer wartbar. Für den Umgang mit Konstanten, zu denen natürlich auch konstante Texte (wie zum Beispiel Dateinamen) zählen, gelten folgende Empfehlungen:

Globale Konstanten und Texte. Globale Konstanten und Texte (zum Beispiel Dateinamen, Ersatzwerte für Aufrufparameter und ähnliches) werden bei geringer Anzahl in einem eigenen Abschnitt am Anfang des Skripts zusammengefaßt und einzeln kommentiert.

```
startdirs=${@:-.}           # default start directory: current directory
action=${action:-l}       # default action is list
```

Umfangreiche Einzeltexte. Umfangreiche Einzeltexte (zum Beispiel Beschreibungen, Hilfe zu Aufrufoptionen) können als **here-Dokumente** geschrieben werden.

```
cat <<- EOT
List and/or delete all stale links in directory trees.

usage : $0 [-d] [-oD logfile] [-l] [-h] [starting directories]

-d   delete stale links
-l   list stale links (default)
-o   write stale links found into logfile
-D   delete stale links listed in logfile
-h   display this message
EOT
```

5 Erfolgskontrollen

5.1 Kommandozeilenparameter

Wenn eine Mindestanzahl oder eine festgelegte Anzahl von Kommandozeilenparametern vorhanden sein muß, dann ist diese Anzahl zu überprüfen. Im Fehlerfall wird das Skript mit einer Fehlermeldung beziehungsweise mit einem Hinweis auf die notwendigen Aufrufparameter beendet.

Die übergebenen Parameterwerte sind ebenfalls auf Gültigkeit zu überprüfen. Wird zum Beispiel der Name einer Eingabedatei übergeben, dann ist vor dem Lesezugriff zu überprüfen, ob diese Datei vorhanden und lesbar ist (zum Beispiel mit dem Test [-r "\$eingabedatei"]).

5.2 Variablen, Befehle und Funktionen

Variablen müssen vor dem Gebrauch natürlich einen sinnvoll Anfangswert erhalten haben. Dieser Sachverhalt ist zu überprüfen:

```
[ -e "$1" ] && expand --tabs=$number $1 > $1.expand
```

Diese Zeile prüft, ob die Datei vorhanden ist, deren Name im Parameter **\$1** übergeben wurde. Da die Bewertung logischer Ausdrücke abgebrochen wird, sobald das Ergebnis feststeht (die sogenannte „short circuit evaluation“), unterbleibt die Weiterverarbeitung, wenn die vorangestellte Bedingung falsch ist. Der Rückgabewert des jeweils letzten Befehls wird in der Variablen **\$?** abgelegt und kann damit zur weiteren Ablaufsteuerung verwendet werden:

```
mkdir "$neuesverzeichnis" 2> /dev/null
if [ $? -ne 0 ] ; then
    ...
fi
```

Wenn in diesem Beispiel kein Verzeichnis angelegt werden konnte, dann ist der Rückgabewert von `mkdir` ungleich Null. Die Variable **\$?** wird auch dazu verwendet, den Rückgabewert einer Funktion zu überprüfen.

5.3 Ablaufprotokoll und Abschlußmeldung

Skripte, die interaktiv verwendet werden, sollten eine Abschlußmeldung ausgeben, die eine Erfolgsanzeige enthält und die zur Plausibilitätskontrolle verwendet werden kann, zum Beispiel

```
mn4:~/bin # ./stale-links -o stale-links.log /opt

... searching stale links ...
1. stale link: '/opt/dir link 23'
2. stale link: '/opt/file link 71'
3. stale link: '/opt/file link 7'

stale links found : 3
stale links deleted : 0
logfile: 'stale-links.log'
```

Umfangreiche Ausgabe können in Log-Dateien geschrieben werden. Das erleichtert gegebenenfalls auch die Fehlersuche.

6 Dateien

Dateinamen Für Dateinamen sind sinnvolle Grundnamen zu wählen. Die Dateiendungen sollen, wenn möglich, auf den Inhalt hinweisen (`.dat`, `.log`, `.lst`, `.tmp` und so weiter).

Temporäre Dateien Temporäre Dateien zum Ablegen von umfangreichen Zwischenergebnissen werden üblicherweise im zentralen Verzeichnis `/tmp` erzeugt und nach Gebrauch natürlich wieder beseitigt. Zur Erzeugung zufälliger Namen kann `mktemp` verwendet werden (siehe man 1 `mktemp`):

```
#-----
# Cleanup temporary file in case of keyboard interrupt or termination signal.
#-----
cleanup_temp {
    [ -e $tmpfile ] && rm --force $tmpfile
    exit 0
}

trap cleanup_temp SIGHUP SIGINT SIGPIPE SIGTERM

tmpfile=$(mktemp) || { echo "$0: creation of temporary file failed!"; exit 1; }

# ... use tmpfile ...
rm --force $tmpfile
```

Die Funktion `cleanup_temp` wird aufgerufen, wenn eines der vier genannten Signale von der **trap**-Anweisung abgefangen wird. Die Funktion löscht dann die temporäre Datei. Eine temporäre Datei bleibt bei einem Abbruch mit `SIGKILL` erhalten, da dieses Signal nicht abgefangen werden kann.

Sicherungskopien Müssen von Dateien mehrere ältere Kopien zurückgehalten werden, dann empfiehlt sich die Verwendung des Datums als Bestandteil der Dateinamen der Kopien:

```
timestamp=$(date +"%Y%m%d-%M%S")           # generate timestamp : YYYYMMDD-mmss
mv logfile logfile.$timestamp
```

Die Datei `logfile` wird nun zum Beispiel in `logfile.20041210-3116` umbenannt. Die Datums- und Uhrzeitbestandteile sind in umgekehrter Reihenfolge angeordnet. Die so benannten Dateien erscheinen in Verzeichnislisten nach dem Alter sortiert.

Zwischenergebnisse Zwischenergebnisse die in Dateien geschrieben werden, können durch die Verwendung von `tee` gleichzeitig auch auf die Standardausgabe ausgegeben werden. Damit können sie zur Ablaufkontrolle oder zum Testen des Skripts dienen:

```
echo $output_string | tee --append $TMPFILE
```

7 Kommandozeilenschalter

Aufruf externer Programme In einem Skript sind bei den Aufrufen von Systemprogrammen grundsätzlich **die langen Formen der Kommandozeilenschalter** zu verwenden, wenn diese zur Verfügung stehen. Die langen Namen sind meist selbstdokumentierend und erleichtern damit das Lesen und Verstehen eines Skripts. In der folgenden `useradd`-Anweisung werden die Langformen der Schalter `-c`, `-p` und `-m` verwendet:

```
useradd --comment      $full_name          \
        --password     $encrypted_password \
        --create-home  \
        $loginname
```

Mit Hilfe der Umbrüche (Zeichen `\` am Zeilenende) wird die Entstehung einer überlangen Zeile vermieden. Die tabellenartige Ausrichtung erhöht die Lesbarkeit.

Kommandozeilenschalter eigener Skripte Für die Bezeichnung eigener Schalter (Kurzform) sind naheliegende oder gebräuchliche Buchstaben zu wählen (zum Beispiel `-f`, mit Argument, für die Angabe einer Datei (file), oder `-d`, mit oder ohne Argument, zur Steuerung des Umfangs von Testausgaben (debug)). Vorschläge für Langformen finden sich in den **GNU Coding Standards**¹.

8 Verwendung von Shell-Funktionen

Wenn möglich und sinnvoll, sollten Shell-Funktionen anstatt externer Programme verwendet werden. Jeder Aufruf von `sed`, `awk`, `cut` und so weiter erzeugt einen eigenen Prozeß. Das kann in Schleifen zu einer erheblichen Verlängerung der Laufzeit führen. Das nachfolgende Beispiel verwendet Kommandosubstitutionen zur Ermittlung des Datei- und des Pfadnamens:

¹<http://www.gnu.org/prep/standards.html>

```

for pathname in $(find $search -type f -name "*" -print)
do
    basename=${pathname##*/}           # replaces basename(1)
    dirname=${pathname%/*}           # replaces dirname(1)
    ...
done

```

Für die Mustersuche in Zeichenketten steht unter anderem der Vergleichsoperator `=~` zur Verfügung.

```

metacharacter='[~&|] '
if [[ "$pathname" =~ $metacharacter ]]
then
    # treat metacharacter
fi

```

Die Verwendung von Regulären Ausdrücken (POSIX, `regex(7)`) ist möglich.²

9 SUID/SGID-Skripte

Shell-Skripte sind außer von Benutzereingaben auch von Umgebungsvariablen, von Initialisierungsdateien und den verwendeten Systemprogrammen abhängig. Zum Schreiben sicherer Programme sind Shell-Skriptsprachen nicht gut geeignet, da alle genannten Einrichtungen zu Angriffen auf das System verwendet werden können oder selbst Sicherheitslücken enthalten können. Soll ein Skript trotzdem mit SUID/SGID-Rechten laufen, sind eine Reihe von Maßnahmen zu treffen [Whe03, GSS03]. Ohne Anspruch auf Vollständigkeit hier die wichtigsten:

- Das Skript in einem Verzeichnis ablegen, in dem es nicht unberechtigt geändert werden kann.
- Überprüfen, ob die Umgebungsvariable `BASH_ENV` leer ist.
- `umask` auf den Wert `077` setzen.
- Die Umgebungsvariablen `PATH` und `IFS` auf sichere Werte setzen.
- In ein sicheres Arbeitsverzeichnis wechseln. Den Wechsel überprüfen.
- Systemprogramme und Datendateien mit absoluten Pfadnamen verwenden.
- Alle Rückgabewerte von Systemprogrammen überprüfen.
- Die Parameterliste, wenn möglich, mit `--` abschließen (Ende der Parameterliste).
- Kommandozeilenparameter nur in Anführungszeichen verwenden (zum Beispiel `"$1"`).
- Kommandozeilenparameter auf unerlaubte Zeichen hin untersuchen.
- Vom Aufrufer übergebene Pfade überprüfen (relativ/absolut).
- Vor dem Öffnen einer neuen Datei die Option `noclobber` setzen, um das unbeabsichtigte Überschreiben einer existierenden Datei zu vermeiden.
- Temporäre Dateien in einem sicheren Verzeichnis anlegen. `mktemp` verwenden (Abschnitt 6).

²Die meisten Bash-Shells sind dazu mit den Optionen `cond-command` und `cond-regex` konfiguriert.

10 Testen

10.1 Syntaxprüfung

Wird ein Skript mit der **bash**-Aufrufoption **-n** ausgeführt, dann werden die Skriptbefehle zwar gelesen, aber nicht ausgeführt:

```
bash -n remove_ps.sh
```

Ein derartiger Aufruf kann zur Syntaxprüfung verwendet werden. Allerdings werden nur grobe Fehler gefunden. Ein verstümmeltes Schlüsselwort (**cho** statt **echo**) wird zum Beispiel nicht beanstandet, da es auch der Name eines Programmes oder einer Funktion sein könnte.

10.2 Testumfang

In der Entwicklungsphase ist es unbedingt geboten, sich eine Testumgebung mit Beispieldateien oder Beispieldaten von überschaubarem Umfang zusammenzustellen (zum Beispiel in einem dafür eingerichteten Verzeichnisbaum). Dadurch wird die Ablaufgeschwindigkeit der Skripten erhöht und die Gefahr gemindert, unbeabsichtigte Veränderungen an wichtigen Daten vorzunehmen.

10.3 Verwendung von `echo`

Verändernde Befehle, wie zum Beispiel das Löschen oder Umbenennen von Dateien, sollten unbedingt zu Testzwecken mit Hilfe von **echo** zunächst als Zeichenkette ausgegeben und überprüft werden. Das ist besonders dann geraten, wenn wildcards oder rekursive Verzeichnisdurchmusterungen verwendet werden. Die Anweisungen

```
liste=$(find . -name "*.sh" )
for datei in $liste
do
  rm $datei
done
```

löschen ohne weiteres alle Dateien mit der Endung `.sh` im gesamten Verzeichnisbaum. Wird der Löschbefehl zunächst in eine **echo**-Anweisung gesetzt, dann werden die Löschanweisungen so ausgegeben, wie sie ohne **echo** ausgeführt werden würden:

```
echo "rm $datei"
```

Nach erfolgter Kontrolle kann das **echo** entfernt werden.

10.4 Testen mit Hilfe von **bash**-Optionen

Kommandozeilenoption	set -o Option	Wirkung
-n	noexec	Befehle werden nicht ausgeführt; nur Syntaxprüfung (s. 10.1)
-v	verbose	Gibt die Zeilen eines Skripts vor der Ausführung aus.
-x	xtrace	Gibt die Zeilen eines Skripts nach den Ersetzungen aus.

Tabelle 1: Optionen, die die Fehlersuche unterstützen

Wenn die Zeilen

```
TMPFILE=$( mktmp /tmp/example.XXXXXXXXXX ) || exit 1
echo "program output" >> $TMPFILE
rm --force $TMPFILE
```

mit den *bash*-Optionen `-xv` mittels

```
bash -xv ./tempfile.sh
```

ausgeführt werden, dann wird die folgende Ausgabe erzeugt:

```
TMPFILE=$( mktemp /tmp/example.XXXXXXXXXX ) || exit 1
mktemp /tmp/example.XXXXXXXXXX
++ mktemp /tmp/example.XXXXXXXXXX
+ TMPFILE=/tmp/example.AVkuGd6796
echo "program output" >> $TMPFILE
+ echo 'program output'
rm --force $TMPFILE
+ rm --force /tmp/example.AVkuGd6796
```

Die mit `+` beginnenden Zeilen werden von der `-x`-Option erzeugt. Die Anzahl der Pluszeichen gibt die Ersetzungstiefe wieder. Diese Optionen können in einem Skript auch nur für einen Abschnitt gesetzt und wieder aufgehoben werden:

```
set -o xtrace          # --- xtrace ein ---
for datei in $liste
do
    rm $datei
done
set +o xtrace         # --- xtrace aus ---
```

10.5 Testen mit Hilfe von `trap`

Pseudosignal	Auslöser
DEBUG	Die Shell hat eine Anweisung ausgeführt.
EXIT	Die Shell beendet das Skript.

Tabelle 2: Pseudosignale

Die *bash*-Shell stellt zwei Pseudosignale bereit (Tabelle 2), auf deren Auftreten durch eigene Signalbehandlungen reagiert werden kann. Die folgende Abbildung 1 zeigt die Verwendung der beiden Pseudosignale zusammen mit **trap**-Anweisungen. Die Abbildung 2 zeigt die von diesem Abschnitt erzeugte Ausgabe.

10.6 Der Debugger `bashdb`

Der Debugger `bashdb`³ arbeitet mit der *bash* ab Version 3.0 und kann einfach aus dem Quellpaket installiert werden. Er kann auch mit dem graphischen Debugger-Front-End `ddd`⁴ zusammenarbeiten.

11 Weitere Informationsquellen

Die wichtigsten und verbindlichen Informationsquellen sind die Handbücher der lokal verwendeten Versionen der Shell und der Systemprogramme.

Über Shell-Programmierung wird gelegentlich in Fachzeitschriften berichtet, außerdem gibt es eine Reihe von Lehrbüchern zu diesem Thema. Einen guten Einstieg in Fragen der Sicherheit bei der Systemprogrammierung bieten [GSS03, Whe03]. Über neue Entwicklungen und aktuelle Sicherheitsfragen informieren plattformsspezifische Internetseiten.

³<http://bashdb.sourceforge.net>

⁴<http://www.gnu.org/software/ddd/>

Abbildung 1: Beispiel zur Verwendung von Pseudosignalen und **trap**

```

1  #===  FUNCTION  =====
2  #      NAME:  dbgtrap
3  #  DESCRIPTION:  Testhilfe: Überwachung der Variablen 'akt_verzeichnis'
4  #=====
5  dbgtrap ()
6  {
7      echo "akt_verzeichnis = \"\$akt_verzeichnis\""
8  }  # -----  end of function dbgtrap  -----
9
10 #-----
11 #  traps
12 #-----
13 trap dbgtrap DEBUG
14
15 trap 'echo "Bei exit : akt_verzeichnis = \"\$akt_verzeichnis\""' EXIT
16
17 #-----
18 #  Überwacher Bereich ...
19 #-----
20 akt_verzeichnis=$(pwd)
21 cd ..
22 akt_verzeichnis=$(pwd)
23 cd $HOME

```

```

akt_verzeichnis = ""
akt_verzeichnis = "/home/mehner"
akt_verzeichnis = "/home/mehner"
akt_verzeichnis = "/home"
akt_verzeichnis = "/home"
akt_verzeichnis = "/home"
Bei exit : akt_verzeichnis = "/home"

```

Abbildung 2: Ausgabe des Skripts in Liste 1

Literatur

- [Bur04] BURTCH, Ken O.: *Linux Shell Scripting with Bash (Developer's Library)*. Sams, 2004. – ISBN 0672326426
- [Coo09] COOPER, Mendel: *Advanced Bash-Scripting Guide*. <http://www.tldp.org/LDP/abs/html/>, 2009. – Comprehensive tutorial with many examples, available in several formats. Well suited for additional online help and as reference work.
- [FSF09] FSF: *Bash Reference Manual*. Free Software Foundation : <http://www.gnu.org>, 2 2009. – Bash shell, version 4.0. The official manual.
- [GSS03] GARFINKEL, Simson ; SPAFFORD, Gene ; SCHWARTZ, Alan: *Practical Unix & Internet Security (3rd Edition)*. O'Reilly Media, 2003. – ISBN 0596003234
- [NR05] NEWHAM, Cameron ; ROSENBLATT, Bill: *Learning the bash Shell (3rd Edition)*. O'Reilly Media, 2005. – ISBN 0596009658. – Textbook; covers the features of Bash Version 3.0.
- [Whe03] WHEELER, David A.: *Secure Programming for Linux and Unix HOWTO*. March 2003. – Version v3.010

12 Kurzfassung

Nr.	Programmierstil	Abschnitt
1	Zeilenlänge maximal 88 Zeichen.	1
2	Einrückung stimmt mit der Schachtelungstiefe überein.	2
3	Jede Datei hat einen Kopfkomentar.	3.1
4	Aufeinanderfolgende Zeilenendkommentare beginnen in derselben Spalte.	3.2
5	Zusammengehörige Anweisungen mit einem Abschnittskomentar versehen.	3.3
6	Eine Funktion hat immer einen Funktionskommentar.	3.4
7	Kommentare sind kurz, treffend und hinreichend genau.	3.5
8	Ein Kommentar beschreibt den Zweck einer Anweisung.	3.5

Nr.	Programmiervorschriften	Abschnitt
1	Für Variablen sind sinntragende, selbstdokumentierende Namen zu verwenden.	4.1
2	Das Einstreuen von Konstanten in den Programmtext ist zu vermeiden.	4.2
3	Die Anzahl und die Gültigkeit von Kommandozeilenparametern ist zu überprüfen.	5.1
4	Die Rückgabewert von Programmen und Skripten überprüfen (Variable \$?).	5.2
5	Interaktive Skripte geben eine Abschlußmeldung mit summarischen Angaben aus.	5.3
6	Dateinamen sind sinnvoll zu wählen; Dateiendungen verweisen den Inhalt.	6
7	Temporäre Dateien sollten mit <code>mktemp</code> erzeugt werden.	6
8	Für Kommandozeilenschalter ist in Skripten die Langform zu verwenden.	7
9	Wenn möglich, Shell-Funktionen anstatt externer Programme verwenden.	8

Nr.	SUID/SGID-Skripte	Abschnitt
1	Das Skript in einem Verzeichnis ablegen, in dem es nicht unberechtigt geändert werden kann.	9
2	Überprüfen, ob die Umgebungsvariable <code>BASH_ENV</code> leer ist.	9
3	<code>umask</code> auf den Wert <code>077</code> setzen.	9
4	Die Umgebungsvariablen <code>PATH</code> und <code>IFS</code> auf sichere Werte setzen.	9
5	In ein sicheres Arbeitsverzeichnis wechseln. Den Wechsel überprüfen.	9
6	Systemprogramme und Datendateien mit absoluten Pfadnamen verwenden.	9
7	Alle Rückgabewerte von Systemprogrammen überprüfen.	9
8	Die Parameterliste, wenn möglich, mit <code>--</code> abschließen (Ende der Parameterliste).	9
9	Kommandozeilenparameter nur in Anführungszeichen verwenden.	9
10	Kommandozeilenparameter auf unerlaubte Zeichen hin untersuchen.	9
11	Vom Aufrufer übergebene Pfade überprüfen (relativ/absolut).	9
12	Vor dem Öffnen einer neuen Datei die Option <code>noclobber</code> setzen, um das unbeabsichtigte Überschreiben einer existierenden Datei zu vermeiden.	9
13	Temporäre Dateien in einem sicheren Verzeichnis anlegen.	9

Nr.	Testen	Abschnitt
1	Mit kleinen Testdatenumfängen und mit <i>Kopien</i> der Originaldaten testen.	10.2
2	Kritische Befehle zum Testen zunächst mit echo ausgeben und kontrollieren.	10.3
3	Syntaxprüfung: bash -Kommandozeilenschalters -n .	10.4
4	Skriptzeilen vor der Ausführung ausgeben: bash -Kommandozeilenschalters -v .	10.4
5	Skriptzeilen nach den Ersetzungen ausgeben: bash -Kommandozeilenschalters -x .	10.4
6	Verfolgung einzelner Variablen ist mit trap möglich.	10.5
