

Paketierung von Software für UCS

Thema:	Beschreibung der Erstellung von Softwarepaketen für und mit Univention Corporate Server
Datum:	15. Dezember 2009
Seitenzahl:	22
Versionsnummer:	4532
Autoren:	Univention GmbH feedback@univention.de

Inhaltsverzeichnis

1	Einführung	3
2	Voraussetzungen und Vorbereitung	3
3	Kontrolldateien	4
3.1	control	7
3.2	copyright	9
3.3	changelog	9
3.4	rules	10
3.5	preinst, prerm, postinst, postrm	15
3.6	conffiles, dirs	18
4	Erzeugen des Pakets	18
5	Besonderheiten bei UCS-Paketen	19
5.1	Eigene Skripte	21
5.2	Eigene Module	22
6	Weiterführende Informationen	22

1 Einführung

Univention Corporate Server basiert auf der Debian-Distribution. Diese verfügt mit dem Programm `dpkg` und dessen Frontend `apt` über ausgereifte Werkzeuge zur Paketverwaltung, die es erlauben, Pakete unter Berücksichtigung von Abhängigkeiten zu installieren und zu entfernen.

Diese Dokumentation soll helfen, Softwarepakete für UCS das selbst aus über 1000 Paketen besteht, zu erstellen, um beispielsweise eigene Software oder Konfigurationen effizienter einzubinden.

2 Voraussetzungen und Vorbereitung

Um Debian-Pakete zu bauen, müssen folgende Programme installiert sein:

- `dh-make`
- `devscripts`
- `build-essential`

Ist dies nicht der Fall, müssen die Pakete nachinstalliert werden (siehe UCS Handbuch unter "Software Nachinstallieren").

Es soll nun ein Paket erstellt werden, das nur aus einem Python-Skript `testdeb.py` besteht. Dieses Skript legt im Verzeichnis `/tmp` eine Datei Namens `testdeb-DATUM+UHRZEIT` an.

Zunächst muss ein Arbeitsverzeichnis mit dem Namen des zukünftigen Pakets und der Versionsnummer erstellt werden (`<Paketname>-<Version>`, alles in Kleinbuchstaben).

```
mkdir testdeb-0.1
cd testdeb-0.1
```

In diesem Verzeichnis wird die Datei `testdeb.py` angelegt, die das nachfolgende Python-Skript, das eigentlich zu installierende Programm, enthält:

```
#!/usr/bin/env python
# Beispielskript Univention-Paketerstellung

import time, os

file='/tmp/testdeb-%s' % time.strftime('%y%m%d%H%M', time.localtime())
f=open(file,'a')
f.close()
```

Für die Erstellung des Pakets wird das Unterverzeichnis `debian/` benötigt. Dieses Verzeichnis enthält einige wichtige Kontrolldateien, die im Folgenden genauer erklärt werden. Durch den Befehl `dh_make` wird dieses Verzeichnis erstellt und mit Vorlagen der Kontrolldateien befüllt.

```
dh_make -e max@muster.de --createorig
```

`dh_make` fragt nach dem Start um was für eine Art von Paket es sich handelt. Hier stehen vier Varianten zur Auswahl, die für spezielle Anpassungen in den Vorlagen für die Kontrolldateien sorgen:

Mit Binärpaket ist hier das erstellte Debian-Paket gemeint. Mit Quellpaket das Verzeichnis oder Archiv mit den Quellen zum Programm und den Debian Paketkonfigurationen (z.B. gibt es ein Quellpaket für Samba).

```
single binary ein Quellpaket, ein Binärpaket
multiple binary ein Quellpaket, mehrere Binärpakete
library Pakete für Programmbibliotheken
kernel module Paket für Kernelmodule
```

In dem hier aufgeführten Beispiel wird ein "single binary"-Paket erstellt (in dieser Dokumentation soll auch nur auf diesen Pakettyp eingegangen werden). Nach der Eingabe von "s" gefolgt von einem ENTER, erscheint folgende Ausgabe:

```
Type of package: single binary, multiple binary, library, or kernel module?
[s/m/l/k] s

Maintainer name : Max Muster
Email-Address   : max@muster.de
Date            : Mon, 21 Mar 2005 13:46:39 +0100
Package Name    : testdeb
Version         : 0.1
Type of Package : Single
Hit <enter> to confirm:
Currently there is no top level Makefile. This may require additional tuning.
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the testdeb Makefiles install into \${DESTDIR} and not in / .
```

Die von `dh_make` ermittelten Informationen (hier `testdeb` als Paketname, `0.1` als Paketversion, der Name des Maintainers u.s.w.) werden bei der Erstellung der Kontrolldateien berücksichtigt. Durch die Option `-createorig` wird eine nicht "debianisierte" Kopie des Quellverzeichnisses angelegt.

3 Kontrolldateien

Durch die Kontrolldateien im Verzeichnis `debian` werden die Eigenschaften des Pakets beschrieben. Zunächst ein kleiner Überblick über die möglichen Kontrolldateien bevor spezieller auf einige wichtige Dateien eingegangen wird.

Dateien mit der Endung `.ex` sind optional und werden bei der Paketerstellung nicht berücksichtigt. Durch Umbenennen, das Weglassen der Endung `.ex`, wird dafür gesorgt, dass die entsprechende Datei bei der Erstellung des Pakets verwendet wird. Mit `→programm`

wird auf das Programm verwiesen, das die entsprechende Kontrolldatei während der Paketerstellung weiterverarbeitet.

Datei	Funktion
README.Debian	Hinweis auf Unterschiede zur unpacketierten Software, z.B. spezielle Einstellungen, Compileroptionen u.s.w., wird nach <code>/usr/share/doc/<Paketname></code> kopiert. → <code>dh_installdocs</code>
changelog	Änderungshistorie des Pakets (siehe 3.2 .)
conffiles.ex	Eine Liste der im Paket verwendeten Konfigurationsdateien. Konfigurationsdateien werden von <code>dpkg</code> während der Installation gesondert behandelt und nicht automatisch überschrieben. In Problemfällen wird der Benutzer um eine Lösung des Konfliktes gefragt.
control	Enthält Informationen über ein Paket, wie z.B. den Paketnamen oder Paketabhängigkeiten. Diese Informationen können mittels <code>dpkg -info</code> abgefragt werden (siehe 3.1).
copyright	Enthält das Copyright des Paketerstellers und des Autors der Upstream-Version sowie weitere Lizenzinformationen (siehe 3.2).
cron.d.ex	Automatisch zu erstellende crontab-Einträge. → <code>dh_installcron</code>
dirs	Eine Liste von Verzeichnissen, die ggf. von <code>dpkg</code> erstellt werden müssen. → <code>dh_files</code>
docs	Eine Liste von Dateien, die nach <code>/usr/share/doc/<Paketname></code> kopiert werden. → <code>dh_installdocs</code>
emacsen-install.ex	Emacs-spezifische Kontrolldatei, die die Installation, automatische Übersetzung in Binärformate, automatisches Laden sowie das Entfernen von Emacs-Lisp-Dateien steuert. → <code>dh_installemacsen</code>
emacsen-remove.ex	siehe emacsen-install.ex
emacsen-startup.ex	siehe emacsen-install.ex
ex.doc-base.package	Wird benötigt, um HTML-Dokumentationen bei <code>doc-base</code> zu registrieren.
init.d.ex	Kann ein Startup-Skript enthalten, welches beim Starten und Herunterfahren des Rechners ausgeführt wird
manpage.1.ex	Beispielrumpf einer Manpage. → <code>dh_installman</code>
manpage.sgml.ex	Beispielrumpf einer Manpage (SGML-Format). → <code>dh_installman</code>
menu.ex	Ermöglicht Einträge in das Debian-Menü.
postinst.ex	Nach der Installation des Pakets auszuführendes Skript (siehe 3.5).
postrm.ex	Nach dem Entfernen des Pakets auszuführendes Skript (siehe 3.5).
preinst.ex	Vor der Installation des Pakets auszuführendes Skript (siehe 3.5).
prerm.ex	Vor dem Entfernen des Pakets auszuführendes Skript (siehe 3.5).
rules	Enthält das zentrale Regelwerk für <code>dpkg-buildpackage</code> zum Bauen des Pakets (siehe 3.4).

<code>watch.ex</code>	Überwacht den Server der Upstream-Version auf Neuerungen.
-----------------------	---

3.1 control

Die `control`-Datei enthält verschiedene Informationen, die `dpkg` benötigt, um das Paket bearbeiten zu können. Die von `dh_make` erstellte `control`-Datei sieht wie folgt aus:

```
Source: testdeb
Section: unknown
Priority: optional
Maintainer: Max Muster <max@muster.de>
Build-Depends: debhelper (>= 4.0.0)
Standards-Version: 3.6.1

Package: testdeb
Architecture: any
Depends: \${shlibs:Depends}, \${misc:Depends}
Description: <insert up to 60 chars description>
<insert long description, indented with spaces>
```

Der obere Block enthält Informationen für die Erstellung des Quell-Pakets.

`Source` Name des Quellpakets.

`Section` Dient der Kategorisierung des Pakets (z.B. durch grafische Paketmanager).

`Priority` Dient der Priorisierung des Pakets (z.B. durch grafische Paketmanager). Diese Einstellung hat keine Auswirkung unter UCS.

`Maintainer` Name und die Email-Adresse des Paketerstellers.

`Build-Depends` Namen von Paketen, die zur Übersetzung dieses Pakets zwingend benötigt werden.

`Standards-Version` Enthält die Version des Debian-Policy-Standards anhand der das paket das letzte mal aktualisiert wurde.

Der untere Teil beschreibt das Binärpaket (ein Quellpaket könnte auch mehrere Binärpakete erzeugen).

`Package` Name eines Binärpakets

`Architecture` Gibt die CPU-Architektur (z.B. amd64 oder i386) an, für die dieses Paket gebaut werden soll. Wird es bei `any` belassen, ersetzt `dpkg-gencontrol` die entsprechende Architektur. Handelt es sich bei dem Paket ausschließlich um Skripte oder Dokumentationen, die architekturunabhängig sind, kann `any` durch `all` ersetzt werden (das Paket kann dann auf jeder Architektur installiert werden).

`Description` Muss eine max. 60 Zeichen lange Kurzbeschreibung des Pakets enthalten. Darunter folgt, durch Leerzeichen am Zeilenanfang eingerückt, eine beliebig lange, ausführliche Beschreibung. Leerzeilen in der Beschreibung können mit einem Leerzeichen und einem Punkt erzeugt werden.

Für Pakete können verschiedene Arten von Abhängigkeiten zu anderen Paketen definiert werden.

Depends Abhängigkeit zur Laufzeit des Pakets. Hier können auch Variablen wie `$shlibs:Depends` oder `$python:Depends` stehen, die durch Paketnamen von Programmbibliotheken ersetzt werden, die beim Erstellen des Pakets als zwingend benötigt ermittelt werden (siehe `dh_shlibdeps` auf Seite 14).

Recommends Für Pakete, die häufig zusammen mit dem Ursprungspaket eingesetzt werden und mitinstalliert werden sollten (aber nicht müssen), z.B. Pakete mit Plugins.

Suggests Für Pakete die gut mit dem Ursprungspaket zusammenarbeiten, aber nicht benötigt werden, z.B. Pakete mit Dokumentationen.

Pre-Depends Die angegebenen Pakete müssen installiert und konfiguriert sein, bevor das Ursprungspaket installiert werden kann (in aller Regel nicht notwendig).

Conflicts Das Paket kann nicht zusammen mit den hier aufgeführten Paketen installiert werden.

Provides Das Paket tritt unter einem weiteren Namen auf.

Replaces Das Paket ersetzt die hier aufgeführten Pakete (z.B. wichtig bei der Umbenennung eines Pakets).

Die Angabe der Paketnamen in diesen Feldern erfolgt nacheinander, durch Kommata getrennt. Ein Paketname kann aber auch aus einer Liste alternativer Pakete bestehen, die dann durch das Pipe-Zeichen `|` getrennt werden müssen. Die Angabe eines Pakets kann auch mit einer exakten Versionsnummer erfolgen. Dafür stehen folgende Vergleichsoperatoren zur Verfügung:

Zeichen	Bedeutung
<<	niedriger
<=	niedriger oder gleich
=	gleich
>=	höher oder gleich
>>	höher

Diese werden, zusammen mit der gewünschten Versionsnummer, direkt hinter den Paketnamen in Klammern gefasst angegeben. Ein Beispiel:

```
Depends: libc6 (>= 2.3.2.ds1-4), exim4 | mail-transport-agent
Conflicts: libgg0, libggi1
Recommends: libncurses5 (>> 5.3)
Suggests: libgii0-target-x (= 1:0.8.5-2)
Replaces: vim-python (<< 6.0), vim-tcl (<= 6.0)
Provides: www-browser, news-reader
```

Die `control` Datei des Beispielpakets sieht folgendermaßen aus:

```
Source: testdeb
Section: univention
Priority: optional
Maintainer: Max Muster <max@muster.de>
```

```
Build-Depends: debhelper (>= 5.0.0)
Standards-Version: 3.6.3

Package: testdeb
Architecture: all
Depends: \${shlibs:Depends}, \${misc:Depends}, \${python:Depends}
Description: Ein Beispielpaket im Rahmen einer UCS-Dokumentation.
Dieses Paket soll die Struktur von Debian Paketen beschreiben.
Außerdem erläutert die Dokumentation:
.
+ Grundsätzliches zu Debian/Univention Paketen.
+ Installationsverlauf.
+ Aufbau eines Pakets.
+ Inhalt und Funktion der Kontroll Dateien.
.
For more information about UCS, refer to:
http://www.univention.de
```

Es wurde eine zusätzliche Abhängigkeit auf `python:Depends` hinzugefügt. Diese Variable wird während der Paketerstellung von dem Debhelper-Programm `dh_python` durch die Namen der benutzten Python-Pakete ersetzt. `dh_python` untersucht dabei den Inhalt des Pakets nach Python-Skripten und -Modulen und generiert aus diesen Informationen Abhängigkeiten für das Paket.

3.2 copyright

Die `copyright`-Datei, die sich unterhalb des `debian`-Verzeichnisses befindet, enthält sowohl Lizenzinformationen des Pakets als auch die des Original-Quellcodes. Dazu zählen auch die Kontaktadressen der Entwickler. Die von `dh_make` erstellte Version:

```
This package was debianized by Max Muster <max@muster.de> on
Mon, 21 Mar 2005 13:46:39 +0100.

It was downloaded from <fill in ftp site>

Copyright:
Upstream Author(s): <put author(s) name and email here>

License:
<Must follow here>
```

Die Datei kann nach Belieben angepasst werden, es gelten keine Syntax-Regeln.

3.3 changelog

Die `changelog`-Datei enthält eine Historie der Änderungen, die an diesem Paket vorgenommen wurden. Die von `dh_make` erzeugte Datei sollte ungefähr den folgenden Inhalt aufweisen:

```
testdeb (0.1-1) unstable; urgency=low

* Initial Release.

-- Max Muster <max@muster.de> Mon, 21 Mar 2005 13:46:39 +0100
```

Wird eine neue Paketversion gebaut, muss zuvor die Versionsnummer erhöht und ein Kommentar zu den erfolgten Änderungen in dieser Datei eingetragen werden. Am einfachsten ist dies mit dem Programm `debchange` bzw. dem Alias `dch` aus dem Paket `devscripts` zu erledigen, da es auch gleich die `changelog`-Datei in einem Editor öffnet und auf die Eingabe der Änderungen wartet. Vor dem Laden kann von `debchange` automatisch die Versionsnummer erhöht werden. Speichert man das Dokument und schließt den Texteditor wieder, werden die soeben angegebenen Informationen in der `changelog`-Datei gespeichert. Folgender Befehl, ausgeführt im Arbeitsverzeichnis, löst diese Aktion aus:

```
dch -i
```

Danach sollte die `changelog`-Datei so aussehen:

```
testdeb (0.1-2) unstable; urgency=low

* Nur eine Versionsnummer höher!

-- Max Muster <max@muster.de> Mon, 21 Mar 2005 17:55:47 +0100

testdeb (0.1-1) unstable; urgency=low

* Initial Release.

-- Max Muster <max@muster.de> Mon, 21 Mar 2005 13:46:39 +0100
```

Achtung:

Das Datums- und Zeitformat muss RFC822-konform sein. Die aktuelle Uhrzeit/Datum kann mit dem Befehl `date -R` im RFC822-Format ausgegeben werden.



3.4 rules

Die `rules`-Datei enthält die eigentlichen Befehlssequenzen, mit denen `dpkg-buildpackage` das Paket schließlich erstellt. Die Datei ist ein Makefile und enthält mehrere Regeln (Rules), die bestimmen, wie mit dem Quellcode verfahren werden soll. Generell ist eine Regel folgendermaßen aufgebaut:

```
ZIEL: VORAUSSETZUNG
      BEFEHL
      ...
```

Jede Regel besteht aus mindestens einem **ZIEL** (Target), das der Name einer Aktion (z.B. **build**: oder **install**:) oder ein Dateiname sein kann. Wildcards (z.B. `"*"`) in den Dateinamen sind ebenfalls erlaubt. Die **BEFEHL**-Zeile muß mit einem Tab beginnen und wird durch die Shell ausgeführt.

Diese Regel steuert das Verhalten von `make`. Sie gibt an, wann das Ziel veraltet ist und wie `make` es erneuern kann. Das Kriterium, nach dem `make` entscheidet, ob das Ziel veraltet ist, ist in der **VORAUSSETZUNG** beschrieben. Sie kann eine oder mehrere Dateien, ein anderes Ziel oder auch eine Datei innerhalb eines Archivs enthalten. Wie bei **ZIEL** sind auch hier Wildcards erlaubt.

Ein **ZIEL** ist veraltet, sobald es nicht existiert oder mindestens eine der Dateien der **VORAUSSETZUNG** neuer als das **ZIEL** ist. In einem solchen Fall wird mittels des unter **BEFEHL** angegebenen Kommandos das **ZIEL** neu erzeugt.

Zu beachten ist, dass `dh_make` lediglich ein Beispiel der `rules`-Datei erzeugt. Der Inhalt dieser Datei sollte immer überprüft werden. Die von `dh_make` generierte Datei sollte in etwa den folgenden Inhalt aufweisen:

```
#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# This file was originally written by Joey Hess and Craig Small.
# As a special exception, when this file is copied by dh-make into a
# dh-make output file, you may use that output file without restriction.
# This special exception was added by Craig Small in version 0.37
# of dh-make.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

CFLAGS = -Wall -g
ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
    CFLAGS += -O0
else
    CFLAGS += -O2
endif

configure: configure-stamp
configure-stamp:
    dh_testdir
    # Add here commands to configure the package.

    touch configure-stamp

build: build-stamp

build-stamp: configure-stamp
    dh_testdir
    # Add here commands to compile the package.
    $(MAKE)
    #docbook-to-man debian/testdeb.sgml > testdeb.1
    touch build-stamp

clean:
    dh_testdir
    dh_testroot
    rm -f build-stamp configure-stamp
    # Add here commands to clean up after the build process.
```


Die Regeln **configure** und **configure-stamp** sind für die Konfiguration des Pakets zuständig. Wenn das Quellpaket autoconf/automake verwendet, sind dort evtl. schon Schritte angegeben, die `dh_make` aus der `config`-Datei des Quellpakets generiert hat. **build** und **build-stamp** steuern den eigentlichen Erstellungsprozess des Pakets. Die Regel **clean** verwaltet die notwendigen Schritte zum Entfernen temporärer Dateien aus dem Paketbaum.

binary-indep erstellt architekturunabhängige Pakete. Ist das nicht der Fall, ist hier keine Änderung notwendig. **binary-arch** erzeugt dementsprechend architekturabhängige Paketbestandteile.

Da in diesem Paket in der `control`-Datei **Architecture:** auf **all** gesetzt wurde, sollten alle Anweisungen zum Erzeugen des Pakets in der Regel **binary-indep** platziert werden. **binary-arch** bleibt in diesem Fall leer.

Unterhalb der **binary-arch**-Regel finden sich eine Menge mit `dh_` beginnende Debhelper-Programme. In den meisten Fällen gibt der Name Aufschluss über ihre Funktion. Nähere Informationen finden Sie in den Manpages des jeweiligen Programms. Sie können sich diese mit dem Befehl `man` anzeigen lassen. Zum Beispiel:

```
man dh_testdir
```

Da eine Beschreibung aller Debhelper-Programme einen enormen Umfang annehmen würde, werden im Folgenden nur die wichtigsten Programme kurz aufgeführt:

Programmname	Beschreibung
<code>dh_testdir</code>	Überprüft, ob das aktuelle Arbeitsverzeichnis sich im obersten Verzeichnis des Quellcode-Verzeichnissesbaums befindet.
<code>dh_testroot</code>	Überprüft, ob mit <code>root</code> -Rechten gearbeitet wird (kann mit <code>fakeroot</code> simuliert werden).
<code>dh_installmanpages</code>	Installiert die Manpages. Es muss der Pfad zu den Manpages relativ zum obersten Verzeichnis angegeben werden.
<code>dh_installdocs</code>	Installiert <code>copyright</code> , <code>README.Debian</code> , <code>TODO.Debian</code> und Dateien aus <code>debian/paketname.docs</code> nach <code>/usr/share/doc/paketname</code> .
<code>dh_installdeb</code>	Installiert die Maintainer-Skripte, z.B <code>postrm</code> .
<code>dh_strip</code>	Entfernt nicht benötigte Debug-Symbole aus Bibliotheken und Programmen (wird nur bei Binärprogrammen benötigt).
<code>dh_gencontrol</code>	Verarbeitet die <code>control</code> -Datei.
<code>dh_md5sums</code>	Generiert MD5-Prüfsummen für jede Datei des Pakets.
<code>dh_fixperms</code>	Korrigiert Dateirechte auf Standardwerte.
<code>dh_builddeb</code>	Erzeugt das Debian-Paket.
<code>dh_installinfo</code>	Installiert Info-Seiten, ähnlich .
<code>dh_examples</code>	Kopiert Beispieldateien in das Paket.
<code>dh_shlibdeps</code>	Generiert Abhängigkeiten auf Bibliotheken und Binär-Dateien.

dh_python	Durchsucht Python-Skripte bzw. -Module und generiert aus gefundenen Informationen Abhängigkeiten für das Paket. Ersetzt die Variable <code>{python:Depends}</code> aus der <code>control</code> -Datei mit diesen Informationen (siehe 3.1.)
-----------	---

Die Änderungen an der `rules`-Datei beschränken sich im wesentlichen auf das Auskommentieren der **`$(MAKE)`** Aufrufe und der Anweisung zum Installieren des Python-Skripts

```
#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# This file was originally written by Joey Hess and Craig Small.
# As a special exception, when this file is copied by dh-make into a
# dh-make output file, you may use that output file without restriction.
# This special exception was added by Craig Small in version 0.37
# of dh-make.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

CFLAGS = -Wall -g
ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
    CFLAGS += -O0
else
    CFLAGS += -O2
endif

configure: configure-stamp
configure-stamp:
    dh_testdir
    # Add here commands to configure the package.

    touch configure-stamp

build: build-stamp

build-stamp: configure-stamp
    dh_testdir
    # Add here commands to compile the package.
#    $(MAKE)
#docbook-to-man debian/testdeb.sgml > testdeb.1
    touch build-stamp

clean:
    dh_testdir
    dh_testroot
    rm -f build-stamp configure-stamp
    # Add here commands to clean up after the build process.
#    -$(MAKE) clean
    dh_clean

install: build
```

```
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs
# Add here commands to install the package into debian/testdeb.
# $(MAKE) install DESTDIR=$(CURDIR)/debian/testdeb

install -m 0755 testdeb.py debian/testdeb/usr/bin/

# Build architecture-independent files here.
binary-indep: build install
dh_shlibdeps
dh_python
dh_md5sums
dh_gencontrol
dh_installdeb
dh_builddeb

# Build architecture-dependent files here.
binary-arch: build install

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure
```

Mit dem Befehl

```
install -m 0755 testdeb.py debian/testdeb/usr/bin/
```

wird die im Arbeitsverzeichnis befindliche Datei `testdeb.py` mit Standardzugriffsrechten nach `debian/testdeb/usr/bin/` kopiert.

3.5 preinst, prerm, postinst, postrm

Es existieren vier so genannte Maintainer-Skripte, die zu unterschiedlichen Zeitpunkten während der Paketinstallation bzw. -deinstallation aufgerufen werden:

`preinst` Vor der Installation
`prerm` Vor der Deinstallation
`postinst` Nach der Installation
`postrm` Nach der Deinstallation

Es handelt sich dabei Shell-Skripte, die von `dh_make` schon mit einem Grundgerüst gefüllt wurden. Um das im Paket befindliche Python-Skript nach erfolgreicher Installation zu starten, müssen Anpassungen in der `postinst`-Datei vorgenommen werden. Die von `dh_make` generierte Version sollte folgenden Inhalt haben:

```
#!/bin/sh
# postinst script for testdeb
#
# see: dh_installdeb(1)
```

```
set -e

# summary of how this script can be called:
# * <postinst> 'configure' <most-recently-configured-version>
# * <old-postinst> 'abort-upgrade' <new version>
# * <conflictor's-postinst> 'abort-remove' 'in-favour' <package>
#   <new-version>
# * <deconfigured's-postinst> 'abort-deconfigure' 'in-favour'
#   <failed-install-package> <version> 'removing'
#   <conflicting-package> <version>
# for details, see http://www.debian.org/doc/debian-policy/ or
# the debian-policy package
#

case "$1" in
    configure)

        ;;

        abort-upgrade|abort-remove|abort-deconfigure)

        ;;

        *)
            echo "postinst called with unknown argument \`$1'" >&2
            exit 1
        ;;
esac

# dh_installdeb will replace this with shell code automatically
# generated by other debhelper scripts.

#DEBHELPER#

exit 0
```

Grundsätzlich kann ein Aufruf, der nach einer erfolgreichen Installation ausgeführt werden soll, innerhalb des **configure**-Blocks erfolgen, da das Paket auf jeden Fall während der Installation konfiguriert wird. Weitere Blöcke mit Übergabeparametern wurden bereits von `dh_make` erzeugt und brauchen in diesem Fall nicht beachtet werden.

Der Eintrag **#DEBHELPER#** wird, wie in den darüber stehenden Kommentaren beschrieben, automatisch von den Debhelper-Programmen ersetzt. Er sollte nicht entfernt werden.

Im nachfolgenden Beispiel ist lediglich ein Aufruf zum Starten des Skripts `testdeb.py` einzufügen. Zusätzlich wird die Univention Configuration Registry-Variablen **update/server** auf <http://download.univention.de/> gesetzt. Das Fragezeichen nach der Variable weist Univention Configuration Registry an, die Variable nur zu setzen, wenn sie noch keinen Wert enthält. Ist bereits eine Version kleiner der "0.1-2" installiert, wird die Univention Configuration Registry-Variablen **timeserver1** auf time.fu-berlin.de gesetzt.

```
#!/bin/sh
```

```
# postinst script for testdeb
#
# see: dh_installdeb(1)

set -e

# summary of how this script can be called:
# * <postinst> 'configure' <most-recently-configured-version>
# * <old-postinst> 'abort-upgrade' <new version>
# * <conflictor's-postinst> 'abort-remove' 'in-favour' <package>
#   <new-version>
# * <deconfigured's-postinst> 'abort-deconfigure' 'in-favour'
#   <failed-install-package> <version> 'removing'
#   <conflicting-package> <version>
# for details, see http://www.debian.org/doc/debian-policy/ or
# the debian-policy package
#

case "$1" in
  configure)
    /usr/bin/testdeb.py
    univention-config-registry set \
      update/server?http://download.univention.de/
    if [ -n "$2" ] && dpkg --compare-versions "$2" lt 0.1-2; then
      univention-config-registry set timeserver1?time.fu-berlin.de
    fi
    ;;

  abort-upgrade|abort-remove|abort-deconfigure)
    ;;

  *)
    echo "postinst called with unknown argument \"\$1\"" >&2
    exit 1
    ;;
esac

# dh_installdeb will replace this with shell code automatically
# generated by other debhelper scripts.

#DEBHELPER#

exit 0
```

Die anderen Maintainer-Skripte verhalten sich analog zu `postinst`, werden aber an anderer Stelle von `dpkg` ausgeführt. Nicht benötigte Skripte können gefahrlos entfernt bzw. auskommentiert werden.

3.6 conffiles, dirs

Die `conffiles`-Datei enthält den absoluten Namen von Konfigurationsdateien, falls es solche gibt. Diese Dateien werden dann bei der Installation gesondert behandelt (nicht überschrieben).

```
#
# If you want to use this conffile, remove all comments and put files that
# you want dpkg to process here using their absolute pathnames.
# See the policy manual
#
# for example:
# /etc/testdeb/testdeb.conf
```

Die `dirs`-Datei enthält die von `dpkg` zu erstellende Verzeichnisse. Das sind alle Verzeichnisse, in die das Paket Dateien installieren soll. Die Verzeichnisse werden einfach untereinander aufgelistet (ohne den führenden Schrägstrich "/"). Üblicherweise ist diese Datei von `dh_make` mit folgendem Inhalt erzeugt worden.

```
usr/bin
usr/sbin
```

4 Erzeugen des Pakets

Bevor der eigentliche Erstellungsprozess des Pakets beginnen kann, sollte das Paketverzeichnis von nicht verwendeten Dateien "gesäubert" werden. Grundsätzlich können Dateien unterhalb von `debian/`, die auf `.ex` enden (`.ex` steht für `example`, d.h. Beispiel) und nicht benötigt werden, gelöscht werden. Dateien die die Endung `.ex` tragen, aber im Paket verwendet werden sollen, müssen spätestens jetzt umbenannt werden, so dass sie keine Endung `.ex` mehr aufweisen.

```
mv preinst.ex preinst
```

Sind alle nicht verwendeten Dateien entfernt, kann der Erstellungsprozess gestartet werden. Dazu muss folgender Befehl im Paket-Quellverzeichnis (`testdeb-0.1/`) ausgeführt werden:

```
dpkg-buildpackage -rfakeroot
```

Die Option `-rfakeroot` teilt `dpkg-buildpackage` mit, bei der Erstellung des Pakets eine simulierte Administratorumgebung zu benutzen, in der normale Benutzer Dateirechte manipulieren können (siehe `install` Anweisung in der `rules` Datei in 3.4).

Es sollte eine Ausgabe ähnlich der folgenden erscheinen:

```
dpkg-buildpackage: source package is testdeb
dpkg-buildpackage: source version is 0.1-1
dpkg-buildpackage: source maintainer is Max Muster <max@muster.de>
dpkg-buildpackage: host architecture is i386
```

```
fakeroot debian/rules clean
dh_testdir
dh_testroot
rm -f build-stamp configure-stamp
# Add here commands to clean up after the build process.
dh_clean
  dpkg-source -b testdeb-0.1
dpkg-source: building testdeb in testdeb_0.1.orig.tar.gz
dpkg-source: building testdeb in testdeb_0.1-1.diff.gz
dpkg-source: building testdeb in testdeb_0.1-1.dsc
  debian/rules build
dh_testdir
# Add here commands to configure the package.
touch configure-stamp
dh_testdir
# Add here commands to compile the package.
#docbook-to-man debian/testdeb.sgml > testdeb.1
touch build-stamp
  fakeroot debian/rules binary
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs
# Add here commands to install the package into debian/testdeb.
install -m 0755 testdeb.py debian/testdeb/usr/bin/
dh_shlibdeps
dh_python
dh_md5sums
dh_gencontrol
dpkg-gencontrol: warning: unknown substitution variable ${shlibs:Depends}
dpkg-gencontrol: warning: unknown substitution variable ${misc:Depends}
dh_installdeb
dh_builddeb
dpkg-deb: building package 'testdeb' in './testdeb_0.1-1_all.deb'.
  dpkg-genchanges
dpkg-genchanges: not including original source code in upload
dpkg-buildpackage: binary and diff upload (original source NOT included)
```

Das erstellte Paket liegt nun eine Verzeichnisebene höher. Es kann als Benutzer root mittels

```
dpkg -i ../testdeb_0.1-1_all.deb
```

installiert werden. Nach der Installation befindet sich in `/usr/bin` das Python-Skript `testdeb.py` und eine vom `postinst`-Skript erstellte Datei (`testdeb-DATUM+UHRZEIT`) unter `/tmp`.

5 Besonderheiten bei UCS-Paketen

Pakete, die für UCS erstellt werden, nutzen teilweise zusätzliche Mechanismen, die auf einem normalen Debian-System nicht existieren. Eine Erweiterung ist die Möglichkeit Uni-

vention Configuration Registry-Variablen zu registrieren und eigene Univention Configuration Registry-Templates mitzubringen.

Um diese Erweiterung zu nutzen, muss das Paket `univention-config-dev` zu den Build-Depends: hinzugefügt werden. Dadurch steht während des Bauvorgangs das Skript `univention-install-config-registry` zur Verfügung, das die Registrierung von Variablen und das Hinzufügen von Univention Configuration Registry-Templates automatisch durchführt. Anhand einer Konfigurationsdatei wird für das Skript vorgegeben, welche Operationen es durchführen soll. Diese Datei muss unterhalb des `debian/`-Verzeichnisses liegen und den Namen des Pakets mit der Endung `.univention-config-registry` tragen.

Der Inhalt der Datei kann aus mehreren Abschnitten bestehen, die wie folgt aufgebaut sind:

```
Type: <type>
[Multifile|File]: <filename>
[Subfile: <sub-filename>]
Variables: <variable1>
Variables: <variable2>
...
```

Im Feld `<type>` wird der Typ des Univention Configuration Registry-Templates eingetragen:

`file` In diesem Fall gibt es genau eine Template-Datei für eine Konfigurationsdatei

`multifile` Setzt sich eine Konfigurationsdatei aus mehreren Univention Configuration Registry-Templates zusammen, dann muss für diese Datei ein solcher Eintrag erstellt werden.

`subfile` Für jedes einzelne Univention Configuration Registry-Template, das Teil eines Multifile-Templates ist, muss ein solcher Eintrag erstellt werden.

Für die Typen `multifile` und `subfile` muss als folgender Eintrag `Multifile` verwendet werden, während dieser für den Typ `file` `File` heißen muss. In diesem Feld wird der Name der Konfigurationsdatei eingetragen, wobei der führende `/` wegfällt.

Sollte es sich um den Typ `subfile` handeln muss einschließlich noch ein Eintrag `Subfile` folgen in dem der Name des Univention Configuration Registry-Templates eingetragen wird. Auch hier fällt der führende `/` weg.

```
Type: file
File: etc/logrotate.conf
Variables: log/rotate/weeks

Type: multifile
Multifile: etc/hosts
Variables: interfaces/eth0/type
Variables: interfaces/eth.*/address
Variables: hostname
Variables: domainname

Type: subfile
Multifile: etc/hosts
```

```
Subfile: etc/hosts.d/00-base
Variables: interfaces/eth.*/address
Variables: hostname
Variables: domainname
```

Anschließend wird für jeden Block eine Liste von Univention Configuration Registry-Variablen eingetragen, bei deren Setzen die Konfigurationsdatei neu erzeugt werden soll. Es gibt die Möglichkeit Muster anzugeben, die eine Menge von Univention Configuration Registry-Variablen beschreiben. Im Beispiel wird das Muster `interfaces/eth.*/address` angegeben, mit dem die Variablen für alle Netz-Interface Adressen abgedeckt werden.

Anhand dieser Informationen kann das Skript `univention-install-config-registry` eine Registrierung durchführen. Damit es die genannten Univention Configuration Registry-Templates installieren und als Konfigurationsdateien markieren kann, müssen diese in einem bestimmten Verzeichnis liegen. Im Hauptverzeichnis des Quellpakets muss es dafür ein Verzeichnis `conffiles` geben. Unterhalb dieses Verzeichnisses wird die benötigte Verzeichnisstruktur für die einzelnen Konfigurationsdateien, wie sie später auch auf dem System installiert werden, erstellt. Für das obige Beispiele würde folgende Struktur genutzt werden:

```
conffiles/etc/logrotate.conf
conffiles/etc/hosts.d/00-base
```

Damit das Skript während des Bauvorgangs aufgerufen wird, muss es in der Datei `debian/rules` eingetragen werden:

```
...
install:
    ...
    univention-install-config-registry
...
```

Sind all diese Anpassungen durchgeführt, wird das Paket bei der Installation die Univention Configuration Registry-Templates installieren und registrieren.

5.1 Eigene Skripte

Univention Configuration Registry ist in der Lage auf Änderungen von Univention Configuration Registry-Variablen mit dem Aufruf von Skripten oder Modulen zu reagieren. Module erlauben es zusätzlich auf kontextspezifische Parameter zu reagieren.

Um ein Template für ein Skript in einem Paket einzubinden, muss es in der `.univention-config-registry` Datei aufgeführt werden.

```
Type: script
Script: script.py
Variables: script/variablen/*
```

Der Eintrag 'Type: script' gibt an das es sich bei der Template Datei um ein Skript handelt. Die Zeile 'Script: script.py' verweist auf das Skript mit dem Namen 'script.py', welches direkt im Unterordner `conffiles` des Pakets abgelegt werden muss. Durch das Setzen der Eigenschaft 'Variables:' auf 'script/variablen/*' wird angegeben, dass das Skript bei jeder Änderung einer Univention Configuration Registry-Variablen, welche auf das angegebene Muster passt, ausgeführt wird.

5.2 Eigene Module

Einem Modul werden beim Aufruf kontextspezifische Informationen mitgegeben. Dazu ist es nötig, dass das Modul in Python geschrieben ist und die Methode 'def handler(bc,changes)' implementiert. Dabei enthält die Variable 'bc' den Namen der Univention Configuration Registry-Variable und 'changes' den neuen Wert auf den die Variable gesetzt wurde.

Die Einbindung in das Paket erfolgt ähnlich wie bei einem Skript. Dazu wird ein Eintrag in der `.univention-config-registry` Datei angelegt, der folgendem Aufbau entspricht.

```
Type: module
Module: module.py
Variables: module/variable
```

Die Eigenschaft 'Type: module' gibt vor das es sich um ein Modul handelt welches im Ordner `conffiles` abgelegt ist. Durch den Eintrag 'Module: module.py' wird der Name des Moduls angegeben. Über den Wert 'module/variable' der Eigenschaft 'Variables' wird vorgegeben, für welche Univention Configuration Registry-Variablen das Module aufgerufen werden soll.

Auf dem Zielsystem werden die Skripte und Module in den Ordnern `/etc/univention/template/script` bzw. `/etc/univention/template/modules` abgelegt.

6 Weiterführende Informationen

- "Debian New Maintainers' Guide", Josip Rodin, Jaldhar Vyas, Will Lowe
<http://www.debian.de/doc/manuals/maint-guide/index.de.html>
- "Debian Policy Manual", Ian Jackson, Christian Schwarz
<http://www.debian.org/doc/debian-policy/>
- "Debian Developer's Reference", Andreas Barth, Adam Di Carlo, Raphaël Hertzog, Christian Schwarz, <http://www.debian.org/doc/developers-reference/>