Debian-Leitfaden für Neue Paketbetreuer

Josip Rodin, Osamu Aoki, Helge Kreutzmann, Tobias Quathamer, Erik Schanze und Eduard Bloch

Diesem Dokument liegen die Beispiele der folgenden zwei Dokumente zu Grunde:

- Making a Debian Package (AKA the Debmake Manual), Copyright © 1997 Jaldhar Vyas.
- The New-Maintainer's Debian Packaging Howto, Copyright © 1997 Will Lowe.
  The rewrite of this tutorial document with updated contents and more practical examples is available as "Guide for Debian Maintainers". Please use this new tutorial as the primary tutorial document.

MITWIRKENDE

	TITEL : Debian-Leitfaden für Neue Paketbetreuer	

AKTION	*NAME*	*DATUM*	*UNTERSCHRIFT*
VERFASST DURCH	Josip Rodin, Osamu Aoki, Helge Kreutzmann, Tobias Quathamer, Erik Schanze und Eduard Bloch	21. Juli 2022	
Deutsche Übersetzung		21. Juli 2022	
Deutsche Übersetzung		21. Juli 2022	
Deutsche Übersetzung		21. Juli 2022	
Deutsche Übersetzung		21. Juli 2022	

VERSIONSGESCHICHTE

NUMMER	DATUM	BESCHREIBUNG	NAME

Inhaltsverzeichnis

Kapitel 1

Einstieg, aber richtig!

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Dieses Dokument versucht, einem typischen Debian-Benutzer und zukünftigen Entwickler in einer verständlichen Sprache die Technik der Paketerstellung für Debian beizubringen, begleitet von funktionierenden Beispielen. Ein altes lateinisches Sprichwort lautet: *Longum iter est per praecepta, breve et efficax per exempla!* (Es ist ein langer Weg mit Regeln, aber ein kurzer und effizienter mit Beispielen!)

Dieses Dokument wird für die Debian-Veröffentlichung `Buster` bereitgestellt, da sie viele Übersetzungen enthält. Sie wird aber in den nachfolgenden Veröffentlichungen entfernt, da die Inhalte zunehmend veralten. 1

Eines der Dinge, die Debian zu einer hervorragenden Distribution machen, ist das Paket-System. Obwohl massenhaft Software im Debian-Format vorhanden ist, muss man manchmal auch Software installieren, die nicht in diesem Format vorliegt. Sie fragen sich vermutlich, wie man eigene Pakete erstellt und vielleicht meinen Sie, es sei eine sehr komplizierte Aufgabe. Nun, wenn Sie ein absoluter Linux-Neuling sind, dann ist es wirklich schwierig, aber als Anfänger würden Sie dieses Dokument jetzt nicht lesen. :-) Sie sollten schon ein wenig Kenntnisse über die Unix-Programmierung mitbringen, aber Sie brauchen ganz sicher kein Guru zu sein. 2

Eines ist wohl sicher: um Debian-Pakete richtig zu bauen und zu warten, brauchen Sie viel Zeit. Schätzen Sie das nicht falsch ein; damit unser System funktioniert, muss der Betreuer sowohl technisch kompetent sein als auch fleißig und sorgfältig arbeiten.

Falls Sie Hilfe beim Erstellen des Pakets brauchen, lesen Sie bitte Abschnitt 1.4.

Neuere Versionen dieses Dokuments sollten immer online über http://www.debian.org/doc/maint-guide/ und in dem Paket `maint-guid` zu finden sein. Die Übersetzungen sind in Paketen wie beispielsweise `maint-guide-es` verfügbar. Bitte beachten Sie, dass diese Dokumentation etwas veraltet sein kann.

Da dies eine Anleitung ist, wird bei wichtigen Themen jeder Schritt im Detail erklärt. Teile davon mögen Ihnen irrelevant vorkommen. Haben Sie Geduld! Einige seltene Fälle werden übersprungen und stattdessen werden nur Verweise geliefert, um dieses Dokument nicht zu kompliziert werden zu lassen.

1.1 Soziale Dynamik von Debian

Es folgen einige Beobachtungen über Debians soziale Dynamik. Die Darstellung ist mit der Hoffnung verbunden, dass es Sie für Ihre Arbeit mit Debian vorbereiten wird.

1In diesem Dokument wird davon ausgegangen, dass Sie ein `Jessie`-System oder ein neueres System verwenden. Wenn Sie diesen Text auf einem älteren System (auch einem älteren Ubuntu-System o.ä.) lesen, müssen Sie mindestens die zurückportierten Pakete `dpkg` und `debhelper` installieren.

2Sie können aus der Debian-Referenz (http://www.debian.org/doc/manuals/debian-reference/) den grundlegenden Umgang mit einem Debian-System lernen. Sie enthält auch einige Hinweise, um etwas über Unix-Programmierung zu lernen.

- Wir sind alle Freiwillige.
  - Sie können anderen nicht vorgeben, was getan werden soll.
  - Sie sollten sich selbst für Ihre Tätigkeit motivieren.
- Freundliche Zusammenarbeit ist die Triebfeder.
  - Ihr Beitrag sollte andere nicht zu sehr belasten.
  - Ihr Beitrag ist nur wertvoll, wenn andere ihn würdigen.
- Debian ist keine Schulklasse, in der Sie automatisch vom Lehrer Aufmerksamkeit bekommen.
  - Sie sollten in der Lage sein, viele Sachen selbst zu lernen.
  - Aufmerksamkeit von anderen Freiwilligen ist eine sehr knappe Ressource.
- Debian verbessert sich ständig.
  - Es wird von Ihnen erwartet, hochqualitative Pakete zu erstellen.
  - Sie sollten sich an Änderungen selbst anpassen.

Es gibt mehrere Arten von Personen, die innerhalb von Debian mit verschiedenen Rollen zusammenarbeiten:

- **Ursprünglicher Autor (»upstream author«)**: Die Person, die das ursprüngliche Programm geschrieben hat.
- **Betreuer des Originalprogramms (»upstream maintainer«)**: Die Person, die das Programm zurzeit betreut.
- **Betreuer (»maintainer«)**: Die Person, die ein Debian-Paket des Programms erstellt oder betreut.
- **Sponsor**: Eine Person, die Betreuern hilft, Pakete in das offizielle Debian-Paketarchiv hochzuladen (nachdem sie den Inhalt überprüft hat).
- **Mentor**: Eine Person, die neuen Betreuern beim Paketieren usw. hilft.
- **Debian-Entwickler (»Debian Developer«, DD)**: Ein Mitglied des Debian-Projekts mit unbeschränkten Rechten, Pakete in das offizielle Debian-Paketarchiv hochzuladen.
- **Debian-Betreuer (»Debian Maintainer«, DM)**: Eine Person, die beschränkte Rechte hat, Pakete in das offizielle Debian-Paketarchiv hochzuladen.

Bitte beachten Sie, dass Sie nicht über Nacht offizieller **Debian-Entwickler** (»Debian Developer«, DD) werden können, weil es dafür mehr als nur technische Fähigkeiten braucht. Bitte lassen Sie sich davon nicht entmutigen. Wenn Ihr Paket für andere nützlich ist, können Sie es entweder als **Betreuer** über einen **Sponsor** oder als **Debian-Betreuer** trotzdem hochladen.

Bitte beachten Sie, dass Sie kein neues Paket erstellen müssen, um offizieller Debian-Entwickler zu werden. Auch Beiträge zu existierenden Paketen können ein Weg sein, offizieller Debian-Entwickler zu werden. Es gibt viele Pakete, die auf einen guten Betreuer warten (siehe Abschnitt 2.2).

Da in diesem Dokument nur auf technische Aspekte der Paketierung fokussiert wird, lesen Sie bitte die folgenden Texte, um zu erfahren, wie Debian funktioniert und wie Sie daran mitarbeiten können:

- Debian: 17 years of Free Software, ”do-ocracy”, and democracy (http://upsilon.cc/~zack/talks/2011/20110321-taipei.pdf) (einführenden Folien)
- Wie können Sie Debian helfen? (http://www.debian.org/intro/help) (offiziell)
- Die Debian GNU/Linux-FAQ, Kapitel 13 - »Zum Debian-Projekt beitragen« (https://www.debian.org/doc/manuals/debian-faq/-contributing.de.html) (semi-offiziell)
- Debian Wiki, HelpDebian (http://wiki.debian.org/HelpDebian) (ergänzend)
- Debian New Member site (https://nm.debian.org/) (offiziell)
- Debian Mentors FAQ (http://wiki.debian.org/DebianMentorsFaq) (ergänzend)

1.2 Programme, die zum Entwickeln notwendig sind

Bevor Sie loslegen können, müssen Sie sicherstellen, dass einige zusätzliche Pakete richtig installiert sind, die für die Entwicklung benötigt werden. Beachten Sie, dass die Liste keine Pakete enthält, die als `essential` oder `required` markiert sind - wir gehen davon aus, dass Sie diese schon installiert haben.

Die folgenden Pakete sind in der Standardinstallation von Debian enthalten, also werden Sie sie vermutlich schon haben (und zusätzliche Pakete, von denen diese abhängen). Sie sollten diese dennoch mit `aptitude show` *`Paket`* oder `dpkg -s` *`Paket`* überprüfen.

Das wichtigste Paket, dass auf Ihrem Entwicklungssystem installiert werden sollte, ist das Paket `build-essential`. Wenn Sie es installieren, wird es andere Pakete *hinterherziehen (»pull in«)*, die für eine grundlegende Build-Umgebung notwendig sind.

Für manche Arten von Paketen ist das alles, was Sie benötigen. Es gibt allerdings noch eine weitere Sammlung von Paketen, die - obwohl sie nicht essenziell für jede Paketerstellung sind - sinnvoll installiert werden können oder sogar von Ihrem Paket benötigt werden:

- `autoconf`, `automake` und `autotools-dev` - Viele neuere Programme benutzen configure-Skripte und `Makefile`-Dateien, die mit Hilfe dieser Programme vorproduziert wurden (siehe »`info autoconf`«, »`info automake`«). Das Paket `autotools-dev` enthält aktuelle Versionen von bestimmten auto-Dateien sowie Informationen über die beste Art und Weise, diese Dateien zu verwenden.
- `dh-make` und `debhelper` - `dh-make` wird benötigt, um ein Gerüst unseres Beispielpakets zu erstellen. Es verwendet einige der `debhelper`-Werkzeuge für die Paketerstellung. Sie sind nicht zwingend erforderlich, um Pakete zu erstellen, aber für neue Betreuer *sehr* empfohlen. Sie vereinfachen den Einstieg in den ganzen Prozess sehr, ebenso die spätere Kontrolle (siehe dh_make(8), debhelper(1), `/usr/share/doc/debhelper/README`). 3

  Das neue `debmake` kann als Alternative zum Standard `dh-make` verwendet werden. Es enthält mehr Funktionalitäten und außerdem HTML-Dokumentation mit ausführlichen Paketierungsbeispielen (im Paket `debmake-doc`).
- `devscripts` - Dieses Paket enthält einige nützliche Skripte, die für die Betreuer hilfreich sein können, aber nicht zum Bauen der Pakete benötigt werden. Die von diesem Paket empfohlenen und vorgeschlagenen Pakete sind ebenfalls einen Blick wert (siehe `/usr/share/doc/devscripts/README.gz`).
- `fakeroot` - Dieses Hilfsprogramm ermöglicht Ihnen, die Identität von »root« vorzutäuschen, was für einige Teile des Build-Prozesses benötigt wird (siehe fakeroot(1)).
- `file` - Dieses nützliche Programm kann den Typ einer Datei feststellen (siehe file(1)).
- `gfortran` - Der GNU-Fortran-95-Compiler wird benötigt, wenn Ihr Programm in Fortran geschrieben ist (siehe gfortran(1)).
- `git` - Dieses Paket stellt ein beliebtes Versionskontrollsystem zur Verfügung, das dafür entworfen wurde, bei sehr großen Projekten schnell und effizient zu arbeiten. Es wird in vielen hoch angesehenen Open-Source-Projekten eingesetzt, beispielsweise beim Linux-Kernel (siehe git(1), Git-Handbuch (`/usr/share/doc/git-doc/index.html`)).
- `gnupg` - Ein Werkzeug, mit dem Sie Pakete digital *unterschreiben* können. Dies ist besonders wichtig, wenn Sie Pakete an andere Leute verteilen wollen und das werden Sie sicher, wenn Ihre Arbeit in die Debian-Distribution aufgenommen wird (siehe gpg(1)).
- `gpc` - Der GNU-Pascal-Compiler wird benötigt, wenn Ihr Programm in Pascal geschrieben ist. Erwähnenswert ist an dieser Stelle der »Free Pascal Compiler« `fp-compiler`, der sich dafür ebenfalls gut eignet (siehe gpc(1), ppc386(1)).
- `lintian` - Das ist Debians Paket-Prüfer, der Sie nach der Paketerstellung über häufige Fehler informiert und die gefundenen Fehler erklärt (siehe lintian(1), Lintians Benutzerhandbuch (https://lintian.debian.org/manual/index.html)).
- `patch` - Ein sehr nützliches Programm, das eine Datei mit einer Auflistung der Unterschiede im Dateiinhalt (erstellt mit dem Programm **diff**) auf die ursprüngliche Datei anwendet und daraus die neue Version erzeugt (siehe patch(1)).
- `patchutils` - Dieses Paket enthält einige Hilfsprogramme, um mit Patches zu arbeiten, beispielsweise die Befehle **lsdiff**, **interdiff** und **filterdiff**.

3Es gibt auch einige spezialisierte aber ähnliche Pakete wie `dh-make-perl`, `dh-make-php` usw.

- `pbuilder` - Dieses Paket enthält Programme, um eine **chroot**-Umgebung aufzubauen und zu betreuen. Beim Bauen eines Debian-Pakets in dieser **chroot**-Umgebung wird geprüft, ob die Build-Abhängigkeiten stimmen, wodurch FTBFS-Fehler (»Fails To Build From Source«, kann nicht aus den Quellen gebaut werden) verhindert werden (siehe pbuilder(8) und pdebuild(1)).
- `perl` - Perl ist eine der am meisten gebrauchten interpretierten Skriptsprachen auf heutigen Unix-ähnlichen Systemen, oft bezeichnet als »Unix' Schweizer Offizierskettensäge« (siehe perl(1)).
- `python` - Python ist eine weitere der am meisten gebrauchten interpretierten Skriptsprachen auf Debian-Systemen, die bemerkenswerte Stärke mit einer sehr klaren Syntax kombiniert (siehe python(1)).
- `quilt` - Dieses Paket hilft Ihnen dabei, eine große Anzahl von Patches zu verwalten, indem es die Änderungen verfolgt, die jeder Patch vornimmt. Patches können angewandt, entfernt und erneuert werden sowie vieles mehr. (Lesen Sie quilt(1) und `/usr/share/doc/quilt/quilt.pdf.gz`).
- `xutils-dev` - Einige Programme, üblicherweise die für X11 erstellten, benutzen diese Programme zum Erzeugen von `Makefile`-Dateien aus einer Gruppe von Makro-Funktionen (siehe imake(1), xmkmf(1)).

Die kurzen Erklärungen oben dienen nur dazu, Ihnen eine Einführung in die verschiedenen Pakete zu geben. Bevor Sie weitermachen, lesen Sie bitte die Dokumentation jedes relevanten Programms, auch die, die als Abhängigkeiten installiert wurden wie **make**, zumindest was die normale Arbeitsweise angeht. Das mag Ihnen am Anfang überflüssig vorkommen, aber schon bald werden Sie *sehr* froh darüber sein, sich schon vorher informiert zu haben. Falls Sie später gezielte Fragen haben, wird empfohlen, die oben erwähnten Dokumente erneut zu lesen.

1.3 Dokumentation, die zum Entwickeln gebraucht wird

Es folgen *sehr wichtige* Dokumente, die Sie neben diesem Dokument auch lesen sollten:

- `debian-policy` - Das Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) (Debian-Richtlinien-Handbuch) beinhaltet Beschreibungen der Struktur und des Inhalts des Debian-Archivs, mehrere Besonderheiten des Betriebssystemdesigns, den »Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) « (der beschreibt, wo jede Datei und jedes Verzeichnis sein sollte) usw. Das Wichtigste für Sie ist, dass es die Anforderungen beschreibt, die ein Paket erfüllen muss, um in die Distribution aufgenommen zu werden (lesen Sie die lokalen Kopien `/usr/share/doc/debian-policy/policy.pdf.gz` und `/usr/share/doc/debian-policy/fhs/fhs-3.0.pdf.gz`).
- `developers-reference` - Die Debian-Entwicklerreferenz (http://www.debian.org/doc/devel-manuals#devref) beschreibt alle Dinge, die nicht speziell die technischen Details der Paketerstellung betreffen, beispielsweise die Struktur des Archivs, wie man Pakete umbenennt, aufgibt, adoptiert, NMUs durchführt, Fehler verwaltet, gute Pakete erstellt, wie und wo man ins Archiv hochlädt usw. (lesen Sie die lokale Kopie `/usr/share/doc/developers-reference/developers-reference.pdf`).

Hier einige *wichtige* Dokumente, die Sie zusätzlich zu diesem Dokument auch lesen sollten:

- Das Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) ist eine sehr gute Einführung für das GNU-Build-System, das als GNU Autotools bekannt ist. Die wichtigsten Komponenten sind Autoconf, Automake, Libtool und Gettext.
- `gnu-standards` - Dieses Paket enthält zwei Teile der Dokumentation des GNU-Projekts: die GNU Coding Standards (http://www.gnu.org/prep/standards/html_node/index.html) und Informationen für Betreuer von GNU-Software (http://www.gnu.org/prep/maintain/html_node/index.html) . Obwohl Debian nicht verlangt, dass diese befolgt werden, sind sie trotzdem hilfreich als Richtlinie und sinnvoll (lesen Sie die lokalen Kopien in `/usr/share/doc/gnu-standards/standards.pdf.gz` und `/usr/share/doc/gnu-standards/maintain.pdf.gz`).

Falls dieses Dokument einem der oben erwähnten Dokumente widerspricht, dann haben diese Recht. Bitte schreiben Sie einen Fehlerbericht zu dem Paket `maint-guide`.

Das Folgende ist eine alternative Anleitung, die Sie zusammen mit diesem Dokument auch lesen können:

- Debian-Paketier-Anleitung (http://www.debian.org/doc/packaging-manuals/packaging-tutorial/packaging-tutorial)

1.4 Wo man Hilfe bekommen kann

Bevor Sie sich entschließen, Ihre Frage an einer öffentlichen Stelle zu stellen, lesen Sie bitte diese gute Dokumentation:

- Dateien in `/usr/share/doc/`*`Paket`* für alle einschlägigen Pakete
- Inhalte von **`man`** *`Befehl`* für alle einschlägigen Pakete
- Inhalte von **`info`** *`Befehl`* für alle einschlägigen Pakete
- Inhalte des Mailinglistenarchivs von debian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/)
- Inhalte des Mailinglistenarchivs von debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)

Sie können Websuchmaschinen effizienter benutzen, indem Sie bei dem Suchausdruck Angaben wie `site:lists.debian.org` verwenden, um den Suchbereich einzuschränken.

Das Erstellen eines kleinen Testpakets ist ein guter Weg, um die Details der Paketerstellung zu lernen. Das Untersuchen von existierenden, gut betreuten Paketen ist die beste Art, zu lernen, wie andere Leute Pakete machen.

Falls Sie immer noch Fragen über das Paketieren haben, für die Sie keine Antworten in der verfügbaren Dokumentation und den Web-Ressourcen finden konnten, können Sie diese interaktiv stellen:

- debian-mentors@lists.debian.org-Mailingliste (http://lists.debian.org/debian-mentors/) (eine Mailingliste für Anfänger),
- debian-devel@lists.debian.org-Mailingliste (http://lists.debian.org/debian-devel/) (eine Mailingliste für Experten),
- IRC (http://www.debian.org/support#irc) wie `#debian-mentors`,
- Teams konzentrieren sich auf bestimmte Paketgruppen (vollständige Liste unter https://wiki.debian.org/Teams (https://wiki.debian.org/Teams));
- Sprachspezifische Mailinglisten wie debian-devel-{french,italian,portuguese,spanish}@lists.debian.org oder debian-devel@debian.or. (Vollständige Listen unter https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) und https://lists.debian.org/users.html (https://lists.debian.org/users.html)).

Die erfahreneren Debian-Entwickler werden Ihnen gerne helfen, falls Sie nach den verlangten Recherchen Ihre Frage vernünftig formulieren.

Wenn Sie einen Fehlerbericht erhalten (ja, tatsächliche Fehlerberichte!), ist es Zeit für Sie, tiefer in die Debian-Fehlerdatenbank (http://www.debian.org/Bugs/) einzusteigen und die dort vorhandene Dokumentation zu lesen, damit Sie mit den Berichten effizient umgehen können. Ich empfehle dringend, die Debian-Entwicklerreferenz, 5.8. »Fehlerbehandlung« (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling) zu lesen.

Selbst wenn alles gut funktioniert hat, ist es jetzt an der Zeit, mit dem Beten anzufangen. Warum? Weil in wenigen Stunden (oder Tagen) Benutzer überall auf der Welt Ihr Paket verwenden werden, und wenn Sie einen kritischen Fehler gemacht haben, werden Sie von unzähligen verärgerten Debian-Benutzern mit E-Mails überschüttet ⋯war nur ein Scherz. :-)

Entspannen Sie sich und stellen Sie sich auf Fehlerberichte ein, denn es ist noch viel mehr Arbeit zu erledigen, bevor Ihr Paket vollständig im Einklang mit den Debian-Richtlinien sowie dessen bewährten Verfahren ist (nochmals: lesen Sie die *wirkliche Dokumentation* für Details). Viel Glück!

Kapitel 2

Erste Schritte

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Lassen Sie uns anfangen, indem Sie Ihr eigenes Paket erstellen (oder, noch besser, ein vorhandenes übernehmen).

2.1 Arbeitsschritte beim Bau von Debian-Paketen

Falls Sie ein Debian-Paket mit einem Programm von (anderen) Originalautoren erstellen, ist ein Teil der typischen Arbeitsschritte des Debian-Paketbaus die Erstellung mehrerer speziell benannter Dateien für jeden Schritt, wie folgt:

- Besorgen Sie sich eine Kopie der Software der Originalautoren, normalerweise in einem komprimierten Tar-Format.
  - *Paket-Version*`.tar.gz`
- Fügen Sie Debian-spezifische Paketanpassungen zu dem Programm der Originalautoren im Verzeichnis `debian` hinzu und erstellen Sie ein nicht natives Quellpaket (d.h. die Menge an Eingabedateien, die zum Bau des Debian-Pakets verwandt wird) im Format `3.0 (quilt)`.
  - *Paket_Version*`.orig.tar.gz`
  - *Paket_Version-Revision*`.debian.tar.gz`[1]
  - *Paket_Version-Revision*`.dsc`
- Erstellen Sie aus den Debian-Quellpaketen die Debian-Binärpakete, das sind normale installierbare Paketdateien im `.deb`-(oder in dem vom Debian-Installer verwandten `.udeb`-)Format.
  - *Paket_Version-Revision_Arch*`.deb`

Bitte beachten Sie, dass das Zeichen, das *`Paket`* und *`Version`* trennt, von `-` (Bindestrich) im Tarball-Namen zu _ (Unterstrich) im Debian-Paketdateinamen geändert wurde.

Im obigen Dateinamen wird der *`Paket`*-Teil durch den **Paketnamen**, der *`Version`*-Teil durch die **Version der Originalautoren**, der *`Revision`*-Teil durch die **Debian-Revision** und der *`arch`*-Teil durch die **Paketarchitektur**, wie im »Debian Policy Manual« festgelegt, ersetzt. [2]

Jeder Schritt dieser Übersicht wird in späteren Abschnitten mit detaillierten Beispielen erklärt.

1Für die Debian-Quellpakete im älteren `1.0`-Format wird stattdessen *`Paket_Version-Revision`*`.diff.gz` benutzt.

2Lesen Sie 5.6.1 ”Source” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source) , 5.6.7 ”Package” (http://www.debian.org/doc/-debian-policy/ch-controlfields.html#s-f-Package) und 5.6.12 ”Version” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version) . Die **Paketarchitektur** folgt dem Debian Policy Manual, Kapitel 5.6.8 »Architecture« (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) und wird während des Paketbauprozesses automatisch zugewiesen.

2.2 Ihr Programm auswählen

Sie haben sich wahrscheinlich schon ein Paket ausgesucht, das Sie erstellen wollen. Zuerst müssen Sie überprüfen, ob das Paket bereits in der Distribution existiert, indem Sie Folgendes benutzen:

- den Befehl **aptitude**,
- die Webseite Debian-Pakete (http://www.debian.org/distrib/packages) ,
- die Webseite Debian Package Tracker (https://tracker.debian.org/) .

Wenn es das Paket schon gibt, na, dann installieren Sie es! :-) Falls es **verwaist (»orphaned«)** wurde (wenn als Betreuer »Debian QA Group (http://qa.debian.org/) « eingetragen ist), dann können Sie es übernehmen, wenn es noch verfügbar ist. Sie können auch ein Paket adoptieren, dessen Betreuer einen »Request for Adoption« (**RFA**) geschrieben hat. 3

Es gibt mehrere Ressourcen zur Paketeigentümerschaft:

- den Befehl **wnpp-alert** aus dem Paket `devscripts`;
- arbeit-bedürfende und voraussichtliche Pakete (http://www.debian.org/devel/wnpp/) ;
- Debian-Fehlerdatenbank-Protokolle: Fehler im Pseudo-Paket `wnpp` in `unstable` (http://bugs.debian.org/wnpp) ;
- Debian-Pakete, die Aufmerksamkeit benötigen (http://wnpp.debian.net/) ;
- Durchsuchen Sie basierend auf Debtags die `wnpp`-Fehler (http://wnpp-by-tags.debian.net/) .

Als wichtige Randbemerkung sei darauf hingewiesen, dass Debian bereits für fast alle Arten von Programmen Pakete enthält und die Anzahl der Pakete im Debian-Archiv wesentlich größer ist als die der Mitwirkenden mit Berechtigung zum Hochladen. Daher werden Beiträge zu Paketen, die bereits im Archiv enthalten sind, von anderen Entwicklern wesentlich mehr gewürdigt (und haben bessere Chancen, gesponsert zu werden) [4]. Sie können auf verschiedene Arten beitragen:

- Pakete übernehmen, die verwaist wurden, aber aktiv benutzt werden;
- Mitglied in einem Paketierungs-Team (http://wiki.debian.org/Teams) werden;
- Fehler von sehr beliebten Paketen sortieren und bewerten;
- Vorbereiten von QA- oder NMU-Uploads (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload) .

Wenn Sie ein Paket übernehmen möchten, laden Sie sich das Quell-Paket herunter (z. B. mit »`apt-get source` *`Paketname`*«) und nehmen Sie es unter die Lupe. Leider enthält dieses Dokument keine umfassende Anleitung zum Übernehmen von Paketen. Der Vorteil ist, dass schon jemand das Paket für Sie vorbereitet hat und Sie keine Schwierigkeiten haben sollten, herauszufinden, wie das Paket funktioniert. Doch lesen Sie weiter, denn viele der folgenden Ratschläge werden auch für Sie nützlich sein.

Falls das Paket neu ist und Sie es gern in Debian integrieren möchten, gehen Sie wie folgt vor:

- Zuerst sollten Sie sicher sein, dass das Programm funktioniert und es bereits einige Zeit ausprobiert haben, damit Sie die Nützlichkeit bestätigen können.
- Überprüfen Sie auf der Site »Arbeit-bedürfende und voraussichtliche Pakete (http://www.debian.org/devel/wnpp/) «, dass niemand bereits an diesem Paket arbeitet. Falls noch niemand daran arbeitet, schreiben Sie mit **reportbug** einen ITP-Fehlerbericht (»Intent To Package«; Absicht, das Paket zu erstellen) an das `wnpp`-Pseudopaket. Wenn schon jemand an dem Paket arbeitet, nehmen Sie mit ihm Verbindung auf, falls nötig. Andernfalls finden Sie bestimmt ein anderes interessantes Paket, das von niemandem betreut wird.
- Die Software **muss eine Lizenz haben**.

3 Lesen Sie die Debian-Entwicklerreferenz 5.9.5 »Adoption eines Pakets« (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting) .

4 Trotzdem gibt es natürlich immer neue Programme, die es wert sind, für Debian paketiert zu werden.

  - Für den Bereich `main` verlangen die Debian-Richtlinien, dass es **die Debian-Richtlinien für Freie Software komplett erfüllt** (DFSG (http://www.debian.org/social_contract#guidelines)) und *kein Paket außerhalb von `main` benötigt*, um zu kompilieren oder ausgeführt zu werden. Dies ist der erwünschte Fall.
  - Für den Bereich `contrib` muss es zu den DFSG konform sein, darf aber ein Paket außerhalb von `main` für die Kompilierung oder Ausführung erfordern.
  - Für den Bereich `non-free` darf es gegen Punkte der DFSG verstoßen, es **muss aber verteilbar sein**.
  - Sind Sie sich unsicher, in welchen Bereich das Paket gehört, schicken Sie den Lizenztext an debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) und bitten um Rat.
- Das Programm sollte **keine** Sicherheitsprobleme und Betreuungssorgen zum Debian-System hinzufügen.
  - Das Programm sollte gut dokumentiert und der Quellcode verständlich (d.h. nicht verschleiert) sein.
  - Sie sollten den oder die Autoren des Programms kontaktieren und sicherstellen, dass sie mit dem Paketieren einverstanden und Debian wohlgesonnen sind. Es ist wichtig, dass die Autoren auch später im Fall von Problemen über das Programm befragt werden können. Versuchen Sie also nicht, aufgegebene Programme zu paketieren.
  - Das Programm sollte sicherlich **nicht** als »setuid root« laufen, oder noch besser, es sollte für die Ausführung überhaupt keine setuid- oder setgid-Rechte brauchen.
  - Das Programm sollte kein Daemon sein oder in ein `*/sbin`-Verzeichnis installiert werden und auch keinen Port als Root öffnen.

Natürlich ist der zuletzt aufgeführt Punkte eher eine Sicherheitsmaßnahme und sollte Sie vor tobenden Benutzern schützen, falls Ihr setuid-Daemon irgendetwas Schlimmes anstellt ⋯wenn Sie mehr Erfahrungen im Erstellen von Paketen gesammelt haben, können Sie auch solche Software paketieren.

Als neuer Betreuer wird Ihnen empfohlen, Erfahrung im Paketieren einfacher Pakete zu sammeln und Ihnen abgeraten, komplizierte Pakete zu erstellen.

- Einfache Pakete
  - einfaches Binärpaket, arch=all (Sammlung von Daten, wie Hintergrundgraphiken)
  - einfaches Binärpaket, arch=all (in interpretierten Sprachen wie POSIX-Shell geschriebene Programme)
- Mittel-komplexe Pakete
  - einfaches Binärpaket, arch=any (ELF-Binärprogramme, kompiliert aus Sprachen wie C und C++)
  - mehrere Binärpakete, arch=any+all (Pakete für ELF-Binärprogramme + Dokumentation)
  - Quelle der Originalautoren ist weder im Format `tar.gz` noch im Format `tar.bz2`
  - die Quellen der Originalautoren enthalten Inhalte, die nicht verteilt werden dürfen
- Hochkomplexe Pakete
  - Interpretermodulpakete, die von anderen Paketen verwandt werden
  - generische ELF-Bibliothekspakete, die von anderen Paketen verwandt werden
  - mehrere Binärpakete, darunter ein ELF-Bibliothekspaket
  - Quellpakete mit mehreren Quellen von Originalautoren
  - Kernel-Modul-Pakete
  - Kernel-Patch-Pakete
  - jedes Paket mit nicht trivialen Betreuerskripten

Paketieren von hochkomplexen Paketen ist nicht zu schwer, erfordert aber ein bisschen mehr Wissen. Sie sollten spezielle Hilfestellungen für jede komplexe Funktionalität erbitten. Beispielsweise haben einige Sprachen ihre eigenen Unter-Richtliniendokumente:

- Perl-Richtlinien (http://www.debian.org/doc/packaging-manuals/perl-policy/)

- Python-Richtlinien (http://www.debian.org/doc/packaging-manuals/python-policy/)
- Java-Richtlinien (http://www.debian.org/doc/packaging-manuals/java-policy/)

Es gibt einen alten lateinischen Spruch: *fabricando fit faber* (Übung macht den Meister). Es wird *nachdrücklich* empfohlen, zu üben und mit allen Schritten der Debian-Paketierung mit einfachen Paketen zu spielen, während diese Anleitung gelesen wird. Ein trivialer Tarball der Originalautoren ist `hello-sh-1.0.tar.gz`, der einen guten Startpunkt darstellt. Er wird wie folgt erstellt:[5]

```
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo "Hello!"
EOF
$ chmod 755 hello
$ cd ..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

2.3 Besorgen Sie sich das Programm und probieren Sie es aus

Als Erstes müssen Sie die Originalquellen des Programms finden und herunterladen. Wahrscheinlich haben Sie bereits die Quellcode-Datei von der Homepage des Autors. Quellen freier Unix-Programme sind üblicherweise im Format **tar**+**gzip** mit der Erweiterung `.tar.gz`, im Format **tar**+**bzip2** mit der Erweiterung `.tar.bz2` oder im Format **tar**+**xz** mit der Erweiterung `.tar.xz`. Sie enthalten üblicherweise ein Verzeichnis, das *`Paket-Version`* heißt, sowie darin alle Quellcode-Dateien.

Falls die neueste Version der Quellen über Versionskontrollsysteme (VCS) wie Git, Subversion oder CVS verfügbar ist, müssen Sie sie mit »`git clone`«, »`svn co`« oder »`cvs co`« herunterladen und dann selbst einen Tarball im Format **tar**+**gzip** erstellen, indem Sie die Option »`--exclude-vcs`« verwenden.

Kommt der Quellcode in einem anderen Archivtyp daher (beispielsweise wenn der Dateiname auf `.Z` oder `.zip` endet [6]), sollten Sie ihn auch mit den geeigneten Werkzeugen entpacken und neu packen.

Falls die Quellen Ihres Programms Inhalte enthalten, die nicht den DFSG genügen, sollten Sie sie auch entpacken, um solche Inhalte zu entfernen, und mit einer geänderten Version der Originalautoren, die `dfsg` enhält, neu packen.

Als Beispiel verwende ich hier ein Programm namens **gentoo**, einen GTK+-Dateimanager. [7]

Erstellen Sie ein Unterverzeichnis in Ihrem Home-Verzeichnis namens `debian` oder `deb` oder irgendetwas, das Sie passend finden (beispielsweise wäre in diesem Fall einfach `~/gentoo` völlig in Ordnung). Kopieren Sie das heruntergeladene Archiv dorthin und entpacken Sie es (mit »`tar xzf gentoo-0.9.12.tar.gz`«). Vergewissern Sie sich, dass es keine Warnmeldungen beim Entpacken gab, nicht mal so genannte *irrelevante*, da die Entpackwerkzeuge anderer Leute diese Anomalien ignorieren oder auch nicht ignorieren könnten und sie daher beim Entpacken Probleme bekommen könnten. Ihre Shell-Befehlszeile könnte dann wie folgt aussehen:

```
$ mkdir ~/gentoo ; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvzf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
```

Jetzt haben Sie ein neues Unterverzeichnis namens `gentoo-0.9.12`. Wechseln Sie dorthin und lesen Sie die mitgelieferte Dokumentation *aufmerksam* durch. Meistens gibt es Dateien mit den Namen `README*`, `INSTALL*`, `*.lsm` oder `*.html`.

[5] Machen Sie sich keine Sorge um das fehlende `Makefile`. Sie können den Befehl **hello** einfach mittels **debhelper** wie in Abschnitt 5.11 oder durch Veränderung der Quellen der Originalautoren installieren, indem Sie ein neues `Makefile` mit dem Ziel `install` wie in Kapitel 3 hinzufügen.

[6] Sie können das Archivformat herausfinden, indem Sie den Befehl **file** verwenden, wenn die Dateierweiterung nicht ausreicht.

[7] Das Programm ist bereits paketiert worden. Die aktuelle Version (http://packages.qa.debian.org/g/gentoo.html) verwendet die Autotools als Baustruktur und unterscheidet sich signifikant von den folgenden Beispielen, die auf Version 0.9.12 basieren.

Sie müssen eine Anleitung finden, wie man das Programm kompiliert und installiert (meistens wird von einer Installation in das Verzeichnis `/usr/local/bin` ausgegangen, aber das werden Sie nicht machen. Mehr dazu später in Abschnitt 3.3).

Sie sollten das Paketieren mit einem komplett aufgeräumten (»pristine«, makellosen) Quellcode-Verzeichnis anfangen oder die Quellen einfach frisch entpacken.

2.4 Einfache Bausysteme

Einfache Programme enthalten normalerweise eine `Makefile`-Datei und können rein durch den Aufruf von »`make`« kompiliert werden. [8] Einige von ihnen unterstützen »`make check`«, wodurch die mitgelieferten Selbsttests gestartet werden. Die Installation in das Zielverzeichnis wird üblicherweise mittels »`make install`« durchgeführt.

Versuchen Sie nun, das Programm zu kompilieren und auszuführen. Stellen Sie sicher, dass es einwandfrei funktioniert und nichts anderes während der Installation oder der Ausführung beschädigt.

Meistens können Sie außerdem »`make clean`« (oder besser »`make distclean`«) ausführen, um im Build-Verzeichnis aufzuräumen. Manchmal gibt es sogar ein »`make uninstall`«, womit alle installierten Dateien gelöscht werden können.

2.5 Beliebte portable Bausysteme

Viele freie Software-Programme sind in den Sprachen C oder C++ geschrieben. Die meisten von ihnen verwenden Autotools oder CMake, um auf verschiedene Plattformen portierbar zu sein. Diese Bauwerkzeuge müssen zuerst benutzt werden, um das `Makefile` und andere benötigte Quelldateien zu erzeugen. Anschließend werden solche Programme mit dem üblichen »`make; make install`« gebaut.

Autotools ist das GNU-Buildsystem, das aus Autoconf, Automake, Libtool und gettext besteht. Sie erkennen solche Quellen an den Dateien `configure.ac`, `Makefile.am` und `Makefile.in`. [9]

Der erste Schritt im Arbeitsablauf der Autotools ist üblicherweise, dass der ursprüngliche Autor »`autoreconf -i -f`« im Quellenverzeichnis aufruft und die Quellen dann mit den erzeugten Dateien verteilt.

```
configure.ac-----+-> autoreconf -+-> configure
Makefile.am -----+        |      +-> Makefile.in
src/Makefile.am -+        |      +-> src/Makefile.in
                          |      +-> config.h.in
                      automake
                      aclocal
                      aclocal.m4
                      autoheader
```

Das Bearbeiten der Dateien `configure.ac` und `Makefile.am` erfordert etwas Wissen über **autoconf** und **automake**. Siehe »`info autoconf`« und »`info automake`«.

Der zweite Schritt im Arbeitsablauf der Autotools ist üblicherweise, dass der Benutzer diese verteilten Quellen erhält und »`./configur && make`« in den Quellen aufruft, um das Programm zu einem ausführbaren ***`Binärdatei`***-Befehl zu kompilieren.

```
Makefile.in -----+               +-> Makefile -----+-> make -> binary
src/Makefile.in -+-> ./configure -+-> src/Makefile -+
config.h.in -----+               +-> config.h -----+
                 |
  config.status -+
  config.guess --+
```

[8] Viele moderne Programme kommen mit einem Skript `configure`, das bei der Ausführung ein für Ihr System angepasstes `Makefile` erstellt.

[9] Autotools ist zu umfangreich, um in dieser Anleitung berücksichtigt zu werden. Dieser Abschnitt dient nur der Bereitstellung von Schlüsselwörtern und Referenzen. Falls Sie sie benötigen, stellen Sie sicher, dass Sie das Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) und die lokale Kopie der `/usr/share/doc/autotools-dev/README.Debian.gz` lesen.

Sie können in der Datei `Makefile` viele Dinge ändern, beispielsweise können Sie den voreingestellten Installationsort für Dateien ändern, idem Sie die Befehlszeilenoption »**./configure --prefix=/usr**« benutzen.

Obwohl es nicht erforderlich ist, kann die Aktualisierung der Datei `configure` und anderer Dateien mittels »`autoreconf -i -f`« die Kompatibilität der Quellen verbessern. 10

CMake ist ein alternatives Build-System. Sie erkennen solche Quellen an der Datei `CMakeLists.txt`.

2.6 Name und Version des Pakets

Falls die Quellen der Originalautoren als `gentoo-0.9.12.tar.gz` existieren, können Sie `gentoo` als (Quell-)**Paketnamen** und `0.9.12` als die **Versionsnummer der Originalautoren** verwenden. Diese werden in der Datei `debian/changelog` verwandt, die auch später in Abschnitt 4.3 beschrieben wird.

Obwohl dieser einfache Ansatz meistens funktioniert, könnte es sein, dass Sie den **Paketnamen** und die **Versionsnummer der Originalautoren** durch Umbenennen der Originalquellen anpassen müssen, damit diese den Debian-Richtlinien und bestehenden Konventionen genügen.

Sie müssen den **Paketnamen** so wählen, dass er nur aus Kleinbuchstaben (`a-z`), Ziffern (`0-9`), Plus- (`+`) und Minus- (`-`) Zeichen und Punkten (`.`) besteht. Er muss mindestens zwei Zeichen lang sein, mit einem alphanumerischen Zeichen beginnen und darf mit keinem existierenden Paket übereinstimmen. Es empfiehlt sich, die Länge unter 31 Zeichen zu halten. 11

Falls die Originalautoren einen generischen Ausdruck wie `test-suite` für den Namen verwenden, ist es eine gute Idee, ihn umzubenennen, um den Inhalt explizit zu identifizieren und den Namensraum sauber zu halten. 12

Sie sollten die **Versionsnummer der Originalautoren** so wählen, dass sie nur aus alphanumerischen Zeichen (`0-9A-Za-z`), Pluszeichen (`+`), Tilden (`~`) und Satzpunkten (`.`) besteht. Sie muss mit einer Ziffer (`0-9`) beginnen. 13 Es ist eine gute Idee, die Länge falls möglich innerhalb von 8 Zeichen zu halten. 14

Falls die Originalautoren kein normales Versionierungsschema wie `2.30.32` sondern eine Art von Datum wie `11Apr29`, einen zufälligen Kodenamen oder einen VCS-Hashwert als Teil der Version verwenden, stellen Sie sicher, dass sie das von der **Version der Originalautoren** entfernen. Diese Informationen können in der Datei `debian/changelog` festgehalten werden. Falls Sie eine Versionsnummer erfinden müssen, verwenden Sie das Format `YYYYMMDD`, wie `20110429`, als Versionsnummer der Originalautoren. Das stellt sicher, dass **dpkg** neuere Versionen korrekt als Upgrades interpretiert. Falls Sie in der Zukunft reibungslose Übergänge zu dem normalen Versionsschemata wie `0.1` sicherstellen müssen, verwenden Sie das Format `0~YYMMDD`, wie `0~110429`, als Versionsnummer der Originalautoren.

Versionsnummern 15 können mit dpkg(1) wie folgt verglichen werden:

```
$ dpkg --compare-versions Ver1 Op Ver2
```

Die Versionsvergleichsregeln können wie folgt zusammengefasst werden:

- Zeichenketten werden von Anfang bis Ende verglichen.
- Buchstaben sind größer als Ziffern.
- Zahlen werden als Ganzzahlen verglichen.
- Buchstaben werden in ASCII-Sortierreihenfolge verglichen.

10Sie können dies automatisieren, indem Sie das Paket `dh-autoreconf` installieren. Lesen Sie Abschnitt 4.4.3.

11Die Standardfeldlänge für den Paketnamen beträgt bei **aptitude** 30. Für mehr als 90% der Pakete ist die Länge des Namens weniger als 24 Zeichen.

12Falls Sie Debian-Entwicklerreferenz 5.1. »Neue Pakete« (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage) folgen, wird der ITP-Prozess normalerweise solche Dinge abfangen.

13Diese strengere Regel sollte Ihnen helfen, verwirrende Namen zu vermeiden.

14Die Standardfeldlänge für die Version beträgt bei **aptitude** 10. Die Debian-Revision mit einleitendem Bindestrich verwendet typischerweise 2 Zeichen. Für mehr als 80% der Pakete ist die Versionsnummer der Originalautoren weniger als 8 Zeichen lang und die Debian-Revision weniger als 2 Zeichen. Für mehr als 90% der Pakete ist die Versionsnummer der Originalautoren weniger als 10 Zeichen lang und die Debian-Revision weniger als 3 Zeichen.

15Versionsnummern können die **Versionsnummer der Originalautoren** (*`Version`*), die **Debian-Revision** (*`Revision`*) oder die **Version** (*`Version-Revision`*) sein. Lesen Sie Abschnitt 8.1, um zu erfahren, wie die **Debian-Revision** erhöht wird.

- Es gibt besondere Regeln für Punkt (`.`), Plus- (`+`) und Tilde- (`~`) Zeichen, wie folgt:
  `0.0` < `0.5` < `0.10` < `0.99` < `1` < `1.0~rc1` < `1.0` < `1.0+b1` < `1.0+nmu1` < `1.1` < `2.0`

Ein schwieriger Fall tritt auf, wenn die Originalautoren `gentoo-0.9.12-ReleaseCandidate-99.tar.gz` als Vorabveröffentlichung von `gentoo-0.9.12.tar.gz` veröffentlichen. Sie müssen sicherstellen, dass das Upgrade korrekt funktioniert, indem Sie die Quellen der Originalautoren in `gentoo-0.9.12~rc99.tar.gz` umbenennen.

2.7 Einrichten von dh_make

Richten Sie die Umgebungsvariablen `$DEBEMAIL` und `$DEBFULLNAME` in der Shell ein, damit verschiedene Debian-Wartungswerkzeug
Ihre E-Mail-Adresse und Ihren Namen für Pakete verwenden können. [16]

```
$ cat >>~/.bashrc <<EOF
DEBEMAIL="Ihre.E-Mail.Adresse@example.org"
DEBFULLNAME="Vorname Nachname"
export DEBEMAIL DEBFULLNAME
EOF
$ . ~/.bashrc
```

2.8 Das erste nicht native Debian-Paket

Normale Debian-Pakete sind nicht native Debian-Pakete, die aus Quellen von Originalautoren erzeugt werden. Falls Sie ein nicht natives Debian-Paket aus den Quellen der Originalautoren `gentoo-0.9.12.tar.gz` erstellen wollen, können sie ein erstes nicht natives Debian-Paket dafür mittels des Befehl **dh_make** wie folgt erstellen:

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

Natürlich ersetzen Sie den Dateinamen mit dem Namen Ihres ursprünglichen Quellcode-Archivs. [17] Siehe dh_make(8) für Details.

Sie sollten irgendeine Ausgabe sehen, die Sie fragt, welche Art Paket Sie erstellen wollen. Gentoo ist ein »single binary package«—es wird nur ein Binärpaket, d.h. eine `.deb`-Datei erstellt, also wählen Sie die erste Option (mit der »`s`«-Taste), überprüfen nochmal die Informationen auf dem Bildschirm und bestätigen mit »*`EINGABE`*«. [18]

Dieser Aufruf von **dh_make** erstellt eine Kopie des ursprünglichen Tarballs als `gentoo_0.9.12.orig.tar.gz` im übergeordneten Verzeichnis, um später die Erstellung eines nicht nativen Debian-Quellpakets mit dem Namen `debian.tar.gz` zu ermöglichen:

```
$ cd ~/gentoo ; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo_0.9.12.orig.tar.gz
```

[16] Der folgende Text setzt voraus, dass Sie Bash als Login-Shell benutzen. Falls Sie eine andere Login-Shell wie beispielsweise die Z-Shell verwenden, benutzen Sie die entsprechenden Konfigurationsdateien statt `~/.bashrc`.

[17] Wenn die originalen Quellen das Verzeichnis `debian` und seinen Inhalt enthalten, rufen Sie den Befehl **dh_make** mit der zusätzlichen Option `--addmissing` auf. Das neue Quellformat `3.0 (quilt)` ist robust genug, selbst mit solchen Paketen umzugehen. Sie müssen wahrscheinlich die von der Originalversion bereitgestellten Inhalte für Ihr Debian-Paket aktualisieren.

[18] Es gibt hier mehrere Auswahlmöglichkeiten: »`s`« für »Single binary package« (einzelnes Binärpaket), »`i`« für »Arch-Independent package« (Architekturunabhängiges Paket), »`m`« für »Multiple binary packages« (mehrere Binärpakete), »`l`« für »Library package« (Bibliothekspaket), »`k`« für »Kernel module package« (Kernel-Modul-Paket), »`n`« für »Kernel patch package« (Kernel-Patch-Paket) und »`b`« für »`cdbs`«-Pakete. Dieses Dokument konzentriert sich auf den Befehl **dh** (aus dem Paket `debhelper`), um ein einzelnes Binärpaket zu erstellen, aber zeigt auch auf, wie es für architekturunabhängige Pakete oder mehrere Binärpakete verwandt werden kann. Das Paket `cdbs` bietet eine alternative Paketierungs-Skriptinfrastruktur zum Befehl **dh** und wird in diesem Dokument nicht behandelt.

Bitte beachten Sie zwei entscheidende Merkmale in dem Dateinamen `gentoo_0.9.12.orig.tar.gz`:

- Paketname und Version sind durch das Zeichen »_« (Unterstrich) getrennt.
- Die Zeichenkette `.orig` wurde vor dem `.tar.gz` eingefügt.

Beachten Sie außerdem, dass viele Schablonendateien im Quellverzeichnis im Unterverzeichnis `debian` erstellt werden. Diese werden in Kapitel 4 und Kapitel 5 erklärt. Weiterhin sollte Ihnen klar sein, dass das Paketieren kein vollautomatischer Prozess sein kann. Sie müssen die ursprünglichen Quellen für Debian verändern (siehe Kapitel 3). Danach müssen Sie die geeigneten Methoden für den Bau des Debian-Paketes verwenden (Kapitel 6), sie testen (Kapitel 7) und hochladen (Kapitel 9). Alle diese Schritte werden erläutert.

Wenn Sie versehentlich einige der Schablonendateien gelöscht haben, während Sie sie bearbeitet haben, können Sie diese wiederherstellen, indem Sie erneut **dh_make** mit der Option `--addmissing` in einem Debian-Quellverzeichnis aufrufen.

Das Aktualisieren eines existierenden Pakets kann kompliziert werden, weil es eventuell ältere Techniken verwendet. Während Sie die Grundlagen lernen, bleiben Sie bei der Erstellung eines neuen Pakets; weitere Erklärungen werden in Kapitel 8 gegeben.

Bitte beachten Sie, dass die Quellen kein in Abschnitt 2.4 und Abschnitt 2.5 beschriebenes Bausystem enthalten müssen. Es könnte eine reine Sammlung von graphischen Daten usw. sein. Die Installation der Dateien könnte rein mit `debhelper`-Konfigurationsdateien wie `debian/install` (siehe Abschnitt 5.11) erfolgen.

Kapitel 3

Den Quellcode verändern

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Bitte beachten Sie, dass an dieser Stelle nicht auf *alle* Details eingegangen wird, wie die ursprünglichen Quellen korrigiert werden können, aber hier sind ein paar grundlegende Schritte und Probleme, auf die man häufig stößt.

3.1 Einrichten von quilt

Das Programm **quilt** bietet eine grundlegende Methode, um Änderungen an den ursprünglichen Quellen für das Debian-Paket aufzuzeichnen. Es ist sinnvoll, eine leicht geänderte Voreinstellung zu verwenden, daher sollte ein Alias **dquilt** für die Debian-Paketierung erstellt werden, indem folgendes zur `~/.bashrc` hinzugefügt wird; dabei stellt die zweite Zeile die gleiche Shell-Vervollständigungsfunktionalität wie der Befehl **quilt** für den Befehl **dquilt** bereit:

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
. /usr/share/bash-completion/completions/quilt
complete -F _quilt_completion -o filenames dquilt
```

Dann erstellen Sie die `~/.quiltrc-dpkg` wie folgt:

```
d=. ; while [ ! -d $d/debian -a $(readlink -e $d) != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # falls in Debian-Paketbaum mit ungesetztem $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35: ↩
        diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

Siehe quilt(1) und `/usr/share/doc/quilt/quilt.pdf.gz` für eine Anleitung, wie **quilt** benutzt wird.

3.2 Fehler in den ursprünglichen Quellen korrigieren

Nehmen wir an, Sie finden den folgenden Fehler in dem ursprünglichen `Makefile`, wo statt »`install: gentoo`« besser »`install: gentoo-target`« stehen sollte.

```
install: gentoo
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Wir beheben dies und speichern es mit dem Befehl **quilt** wie folgt als `fix-gentoo-target.patch`: 1

```
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

Sie können die Datei `Makefile` wie folgt ändern:

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Jetzt muss **dquilt** noch mitgeteilt werden, dass der Patch erzeugt werden soll, so dass die Datei `debian/patches/fix-gentoo-tar` `patch` erstellt wird. Außerdem sollten Sie eine Beschreibung gemäß DEP-3: Patch Tagging Guidelines (http://dep.debian.net/-deps/dep3/) hinzufügen:

```
$ dquilt refresh
$ dquilt header -e
... Patch beschreiben
```

3.3 Installation von Dateien in ihr Zielverzeichnis

Die meisten Programme Dritter installieren sich in die Verzeichnishierachie `/usr/local`. Unter Debian ist diese für die private Benutzung durch den Systemadministrator reserviert, daher dürfen Pakete Verzeichnisse wie `/usr/local/bin` nicht verwenden, sondern sollten stattdessen Systemverzeichnisse wie `/usr/bin` verwenden und dem Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) (FHS) folgen.

Normalerweise wird make(1) benutzt, um das Programm automatisch zu bauen. Der Aufruf von »`make install`« installiert das Programm dann direkt in das Ziel (gemäß dem `install`-Ziel des `Makefile`s). Damit Debian vorab erstellte installierbare Pakete bereitstellen kann, verändert es das Bausystem, um Programme in ein Dateisystem-Abbild, welches unter einem temporären Verzeichnis erstellt wurde, anstatt in das tatsächliche Ziel zu installieren.

Diese beiden Unterschiede zwischen einerseits der normalen Programminstallation und andererseits dem Paketieren für Debian können vom Paket `Debhelper` transparent adressiert werden. Es benutzt dazu die Befehle **dh_auto_configure** und **dh_auto_install**, sofern die folgenden Bedingungen erfüllt sind:

- Das `Makefile` muss den GNU-Konventionen folgen und die Variable `$(DESTDIR)` unterstützen. 2
- Die Quelle muss dem »Filesystem Hierarchy Standard« (FHS) folgen.

Programme, die GNU **autoconf** einsetzen, folgen automatisch den GNU-Konventionen, so dass sie trivial zu paketieren sind. Aufgrund dieser Tatsache und weiterer Heuristik wird geschätzt, dass das Paket `debhelper` für ungefähr 90% aller Pakete richtig funktionieren wird, ohne dass irgendwelche tiefgreifenden Änderungen an deren Build-Systemen vorgenommen werden müssen. Daher ist das Paketieren nicht so kompliziert, wie es zunächst aussieht.

Falls Sie Änderungen an dem `Makefile` vornehmen müssen, sollten Sie vorsichtig sein, dass die Variable `$(DESTDIR)` unterstützt wird. Obwohl sie standardmäßig nicht gesetzt ist, wird die Variable `$(DESTDIR)` jedem Dateipfad vorangestellt, der für die Programminstallation verwendet wird. Das Skript für das Paketieren setzt `$(DESTDIR)` auf das temporäre Verzeichnis.

1Das Verzeichnis `debian/patches` sollte inzwischen existieren, wenn Sie **dh_make** wie vorher beschrieben ausgeführt haben. In diesem Beispiel wird es sicherheitshalber erstellt, nur für den Fall, dass Sie ein existierendes Paket aktualisieren.

2Lesen Sie GNU Coding Standards: 7.2.4 DESTDIR: Support for Staged Installs (http://www.gnu.org/prep/standards/html_node/DESTDIR.html#DESTDIR) .

Bei einem Quellpaket, das ein einzelnes Binärpaket erstellt, wird das temporäre Verzeichnis, das vom Befehl **dh_auto_install** benutzt wird, auf `debian/Paket` gesetzt. 3 Alles, was im temporären Verzeichnis enthalten ist, wird auf dem System eines Benutzers installiert, wenn dieser Ihr Paket installiert. Der einzige Unterschied ist, dass **dpkg** die Dateien relativ zum Wurzelverzeichnis statt zu Ihrem Arbeitsverzeichnis installieren wird.

Vergessen Sie nicht, dass Ihr Programm zwar unter `debian/Paket` installiert wird, es sich aber trotzdem korrekt verhalten muss, wenn es aus dem `.deb`-Paket unter dem Wurzelverzeichnis installiert wird. Daher dürfen durch das Build-System keine fest eingestellten Zeichenketten wie `/home/ich/deb/Paket-Version/usr/share/Paket` für Dateien im Paket festgeschrieben werden.

Dies ist der relevante Abschnitt aus dem `Makefile`[4] von `gentoo`:

```
# Where to put executable commands on 'make install'?
BIN     = /usr/local/bin
# Where to put icons on 'make install'?
ICONS   = /usr/local/share/gentoo
```

Sie sehen, dass die Dateien unter `/usr/local` installiert werden sollen. Wie oben erklärt ist diese Verzeichnis-Hierarchie unter Debian für lokale Benutzung reserviert, ändern Sie daher diese Pfade wie folgt:

```
# Where to put executable commands on 'make install'?
BIN     = $(DESTDIR)/usr/bin
# Where to put icons on 'make install'?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

Der genaue Ort, der für Programme, Icons, Dokumentation usw. verwandt werden sollte, ist im »Filesystem Hierarchy Standard« (FHS) spezifiziert. Sie sollten ihn durchblättern und die für Ihr Paket relevanten Abschnitte lesen.

Sie sollten also ausführbare Befehle in `/usr/bin/` statt `/usr/local/bin/` installieren, die Handbuchseiten in `/usr/share/man/man1/` statt `/usr/local/man/man1/` usw. Beachten Sie, dass im `Makefile` von `gentoo` keine Handbuchseite auftaucht. Da die Debian-Richtlinien aber verlangen, dass jedes Programm eine hat, schreiben Sie später eine und installieren sie unter `/usr/share/man/man1/`.

Manche Programme nutzen die `Makefile`-Variablen nicht, um solche Pfade zu definieren. Das bedeutet, dass Sie wahrscheinlich die C-Quelltexte direkt bearbeiten müssen, damit diese dann die richtigen Pfade benutzen. Aber wo soll man suchen, und wonach eigentlich genau? Sie können dies herausfinden, indem Sie folgendes eingeben:

```
$ grep -nr --include='*.[c|h]' -e 'usr/local/lib' .
```

Grep durchsucht das Quellverzeichnis rekursiv und gibt Ihnen den Dateinamen und die Zeilennummer für alle Treffer aus.

Sie müssen diese Dateien nun bearbeiten und in den entsprechenden Zeilen `usr/local/lib` durch `usr/lib` ersetzen. Dies kann wie folgt automatisch erledigt werden:

```
$ sed -i -e 's#usr/local/lib#usr/lib#g' \
       $(find . -type f -name '*.[c|h]')
```

Falls Sie stattdessen jede Ersetzung bestätigen möchten, kann dies wie folgt interaktiv durchgeführt werden:

```
$ vim '+argdo %s#usr/local/lib#usr/lib#gce|update' +q \
       $(find . -type f -name '*.[c|h]')
```

Als nächstes sollten Sie das »install«-Ziel suchen (suchen Sie nach der Zeile, die mit `install:` beginnt, das funktioniert üblicherweise) und alle Verweise auf Verzeichnisse umbenennen, die nicht denen entsprechen, die Sie am Anfang des `Makefile`s definiert haben.

Ursprünglich sah das `install`-Ziel von `gentoo` wie folgt aus:

3Bei einem Quellpaket, das mehrere Binärpakete erstellt, verwendet der Befehl **dh_auto_install** das temporäre Verzeichnis `debian/tmp`. Der Befehl **dh_install** teilt dann den Inhalt von `debian/tmp` mit Hilfe von `debian/Paket-1.install` und `debian/Paket-2.install` in die temporären Verzeichnisse `debian/Paket-1` und `debian/Paket-2` auf, um daraus die Binärpakete `Paket-1_*.deb` und `Paket-2_*.deb` zu erstellen.

4Dies ist nur ein Beispiel, wie ein `Makefile` aussehen sollte. Falls die `Makefile`-Datei durch den Befehl **./configure** erstellt wird, ist die richtige Vorgehensweise, um diese Art `Makefile` zu korrigieren, **./configure** durch den Befehl **dh_auto_configure** aufrufen zu lassen. Dabei kommen die voreingestellten Optionen zum Tragen, einschließlich der Option `--prefix=/usr`.

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Wir beheben diesen Fehler der Originalautoren und speichern es mit dem Befehl **dquilt** als `debian/patches/install.patch`.

```
$ dquilt new install.patch
$ dquilt add Makefile
```

Wir ändern dies für das Debian-Paket im Editor wie folgt:

```
install: gentoo-target
        install -d $(BIN) $(ICONS) $(DESTDIR)/etc
        install ./gentoo $(BIN)
        install -m644 icons/* $(ICONS)
        install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

Sie haben bemerkt, dass jetzt der Befehl »`install -d`« vor den anderen Befehlen in dieser Regel steht. Das ursprüngliche `Makefile` hatte das nicht, weil normalerweise `/usr/local/bin/` und andere Verzeichnisse schon auf dem System vorhanden sind, wenn Sie »`make install`« aufrufen. Weil wir aber in unseren frisch erstellten Verzeichnisbaum installieren, müssen wir jedes einzelne dieser Verzeichnisse anlegen.

Wir können am Ende der Regel noch weitere Dinge einfügen, wie beispielsweise die Installation zusätzlicher Dokumentation, die die Originalautoren manchmal weglassen:

```
        install -d $(DESTDIR)/usr/share/doc/gentoo/html
        cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Überprüfen Sie alles sorgfältig und falls es in Ordnung ist, bitten Sie **dquilt**, den Patch zu erstellen und die Datei `debian/patches/install.patch` neu zu erstellen. Fügen Sie außerdem noch eine Beschreibung hinzu:

```
$ dquilt refresh
$ dquilt header -e
... Patch beschreiben
```

Jetzt haben Sie eine Reihenfolge von Patches.

1. Die Fehlerkorrektur der originalen Quellen: `debian/patches/fix-gentoo-target.patch`
2. Die Debian-spezifische Änderung für's Paketieren: `debian/patches/install.patch`

Wann immer Sie Änderungen machen, die nicht spezifisch für das Debian-Paket sind (so wie `debian/patches/fix-gentoo-targ` `patch`), sollten Sie diese dem Betreuer des Originalprogramms zukommen lassen, damit er sie in der nächsten Programmversion verwenden kann und sie für andere auch nützlich sind. Denken Sie auch daran, Ihre Patches nicht spezifisch für Debian oder Linux (oder sogar Unix!) zu gestalten, sondern sie portierbar zu erstellen. Dadurch können Ihre Änderungen wesentlich leichter übernommen werden.

Beachten Sie, dass Sie dem Originalautor die `debian/*`-Dateien nicht schicken müssen.

3.4 Unterschiedliche Bibliotheken

Es gibt ein anderes typisches Problem: Bibliotheken variieren oft von Plattform zu Plattform. Beispielsweise kann ein `Makefile` eine Referenz auf eine Bibliothek enthalten, die auf Debian-Systemen nicht existiert. In diesem Fall müssen wir sie auf eine Bibliothek ändern, die unter Debian existiert und den gleichen Zweck erfüllt.

Nehmen wir an, eine Datei in dem `Makefile` (oder `Makefile.in`) Ihres Programms lautet wie folgt:

```
LIBS = -lfoo -lbar
```

Falls Ihr Programm nicht übersetzt, da die Bibliothek `foo` nicht existiert und auf Debian-Systemen ihr Äquivalent von der Bibliothek `foo2` bereitgestellt wird, können Sie dieses Bauproblem mittels `debian/patches/foo2.patch` beheben, indem Sie `foo` in `foo2` ändern: [5]

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e 's/-lfoo/-lfoo2/g' Makefile
$ dquilt refresh
$ dquilt header -e
b''…b'' Patch beschreiben
```

[5] Falls es API-Änderungen von der Bibliothek `foo` zu der Bibliothek `foo2` gibt, müssen die Quellen entsprechend angepasst werden, um zur neuen API zu passen.

Kapitel 4

Benötigte Dateien im Verzeichnis `debian`

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Im Quellverzeichnis des Programms gibt es ein neues Unterverzeichnis, das `debian` heißt. In diesem Verzeichnis sind einige Dateien, die wir ändern müssen, um die Eigenschaften des Pakets anzupassen. Die wichtigsten davon sind `control`, `changelog`, `copyright` und `rules`, die für jedes Paket benötigt werden. 1

4.1 `control`

Diese Datei enthält verschiedene Werte, die **dpkg**, **dselect**, **apt-get**, **apt-cache**, **aptitude** und andere Paketverwaltungswerkzeuge verwenden, um das Paket zu verwalten. Sie ist im Debian Policy Manual, Kapitel 5 »Control files and their fields« (http://www.debian.org/doc/debian-policy/ch-controlfields.html) definiert.

Dies ist die Datei `control`, die **dh_make** für uns erstellt hat:

```
 1 Source: gentoo
 2 Section: unknown
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10)
 6 Standards-Version: 4.0.0
 7 Homepage: <URL der Originalautoren einfügen, falls relevant>
 8
 9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: < bis zu 60 Zeichen Beschreibung einfügen>
13  <lange Beschreibung einfügen, mit Leerzeichen eingerückt>
```

(Die Zeilennummerierung habe ich hinzugefügt.)

Die Zeilen 1-7 sind die Steuerinformationen für das Quellcode-Paket. Zeilen 9-13 sind die Steuerinformationen für das Binärpaket.

Zeile 1 ist der Name des Quellcode-Pakets.

Zeile 2 bestimmt den Bereich (Section) der Distribution, in die das Quellcode-Paket gehört.

1In diesem Kapitel werden zur Vereinfachung Dateien im Verzeichnis `debian` ohne das einleitende `debian/` referenziert, wann immer die Bedeutung eindeutig ist.

Sie haben bestimmt schon gemerkt, dass das Debian-Archiv in mehrere Bereiche aufgeteilt ist: `main` (freie Software), `non-free` (nicht wirklich freie Software) und `contrib` (freie Software, die von non-free-Software abhängt). Jeder davon ist in Abschnitte eingeteilt, die die Pakete in grobe Kategorien sortieren. Dementsprechend gibt es `admin` für Programme für Administratoren, `devel` für Programmierwerkzeuge, `doc` für Dokumentation, `libs` für Programmbibliotheken, `mail` für E-Mail-Leseprogramme und -Daemons, `net` für Netzwerk-Anwendungen und Daemons, `x11` für X11-Programme, die nirgendwo anders unterkommen, und viele mehr. 2

Ändern Sie den Bereich also in x11 (das Präfix `main/` wird impliziert, also können wir es weglassen).

Zeile 3 beschreibt, wie wichtig es ist, dass der Benutzer das Paket installiert. 3

- Die Priorität `optional` ist normalerweise für neue Pakete sinnvoll, die nicht mit anderen Paketen der Prioritäten `required`, `important` oder `standard` kollidieren.

Bereich und Priorität werden von Oberflächen wie **aptitude** benutzt, um Pakete zu sortieren und Standardparameter auszuwählen. Wenn Sie das Paket für Debian hochladen, können die Werte dieser beiden Felder von den Archivbetreuern überschrieben werden. Sie erhalten dann eine Nachricht darüber per E-Mail.

Da es sich um ein Paket mit normaler Priorität handelt und es nichts anderes stört, ändern wir die Priorität auf »`optional`«.

Zeile 4 ist der Name und die E-Mail-Adresse des Betreuers. Sie müssen sicherstellen, dass dieses Feld eine gültige »`To`«-Kopfzeile für eine E-Mail enthält, weil nach dem Hochladen die Fehlerdatenbank diesen Eintrag nutzt, um die Fehler-E-Mails an Sie zuzustellen. Benutzen Sie keine Kommata, kaufmännische Und-Zeichen (&) oder Klammern.

Zeile 5 enthält die Liste der Pakete, die zum Bauen des Pakets benötigt werden, im Feld `Build-Depends`. Sie können hier ebenso das Feld `Build-Depends-Indep` in einer zusätzlichen Zeile benutzen [4]. Einige Pakete wie `gcc` und `make` werden impliziert, weil sie vom Paket `build-essential` benötigt werden. Falls Sie andere Werkzeuge zum Bauen Ihres Pakets brauchen, müssen Sie sie zu diesen Feldern hinzufügen. Mehrere Einträge werden durch Kommata getrennt; mehr über die Syntax dieser Zeilen finden Sie in den Erläuterungen der binären Paketabhängigkeiten.

- Bei allen Paketen, die auf den Befehl **dh** in der Datei `debian/rules` zugreifen, müssen Sie »`debhelper (>=9)`« in das Feld `Build-Depends` eintragen, um die Debian-Richtlinien für das Ziel `clean` zu erfüllen.
- Quellpakete, die binäre Pakete mit »`Architecture: any`« erstellen, werden vom automatischen Bausystem (»Autobuilder«) neu gebaut. Da während dieser Autobuilder-Prozedur nur die in `Build-Depends` aufgeführten Pakete vor der Ausführung von `debian/rules build` installiert werden (siehe Abschnitt 6.2), muss das Feld `Build-Depends` praktisch alle erforderlichen Pakete auflisten. Das Feld `Build-Depends-Indep` wird daher selten benutzt.
- Quellpakete, die nur binäre Pakete mit »`Architecture: all`« erstellen, können im Feld `Build-Depends-Indep` alle erforderlichen Pakete auflisten, es sei denn, diese sind bereits im Feld `Build-Depends` aufgeführt, um die Debian-Richtlinie für das Ziel `clean` zu erfüllen.

Wenn Sie sich nicht sicher sind, welches von beiden Sie benutzen sollten, verwenden Sie das Feld `Build-Depends`, um auf der sicheren Seite zu sein. [5]

Um herauszufinden, welche Pakete Ihr Paket zum Bauen benötigt, führen Sie diesen Befehl aus:

```
$ dpkg-depcheck -d ./configure
```

Um die genauen Build-Abhängigkeiten für ***/usr/bin/foo*** manuell herauszufinden, führen Sie

```
$ objdump -p /usr/bin/foo | grep NEEDED
```

2Lesen Sie das Debian Policy Manual, Kapitel 2.4 »Sections« (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) und Liste der Abschnitte in `sid` (http://packages.debian.org/unstable/) .

3Lesen Sie das Debian Policy Manual, Kapitel 2.5 »Priorities« (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) .

4Lesen Sie das Debian Policy Manual, Kapitel 7.7 »Relationships between source and binary packages - Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep« (http://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps)

5Diese etwas merkwürdige Situation ist eine Besonderheit, die in dem Debian Policy Manual, Fußnote 55 (http://www.debian.org/doc/debian-policy/-footnotes.html#f55) sehr gut dokumentiert ist. Es liegt nicht an der Verwendung des Befehls **dh** in der Datei `debian/rules`, sondern daran, wie das Programm **dpkg-buildpackage** arbeitet. Dieselbe Situation gilt auch für das »auto build system« von Ubuntu (https://bugs.launchpad.net/launchpad-buildd/+bug/238141) .

aus. Rufen Sie dann für jede aufgelistete Bibliothek (beispielsweise **libfoo.so.6**) diesen Befehl auf:

```
$ dpkg -S libfoo.so.6
```

Dann nehmen Sie einfach die Entwicklerversion (`-dev`) jedes Pakets als einen `Build-Depends`-Eintrag. Falls Sie dafür **ldd** benutzen, werden auch indirekt abhängende Bibliotheken aufgelistet, die wiederum zu exzessiven Bauabhängigkeiten führen können.

`Gentoo` benötigt noch `xlibs-dev`, `libgtk1.2-dev` und `libglib1.2-dev`, um gebaut werden zu können, deshalb hängen wir diese direkt hinter `debhelper` an.

Zeile 6 enthält die Version des Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) , dessen Standards dieses Paket entspricht, also die Version, die Sie gelesen haben, während Sie Ihr Paket erstellten.

In Zeile 7 können Sie die URL der Homepage der Originalautoren der Software notieren.

Zeile 9 enthält den Namen des Binärpakets. Üblicherweise ist dies der gleiche Name wie der des Quellpakets, aber das muss nicht immer so sein.

Zeile 10 beschreibt die Architekturen, für die das binäre Paket kompiliert werden kann. Dieser Wert ist normalerweise einer der folgenden, abhängig von der Art des binären Pakets: [6]

- `Architecture: any`
  - Das erstellte Binärpaket ist architekturabhängig, typischerweise in einer kompilierten Sprache.
- `Architecture: all`
  - Das erstellte Binärpaket ist architekturunabhängig, besteht typischerweise aus Text, Bilder oder Skripten in einer interpretierten Sprache.

Wir lassen Zeile 10 unverändert, da das Programm in C geschrieben ist. dpkg-gencontrol(1) wird den geeigneten Architekturwert für jede Maschine, auf der diese Quellen gebaut werden, einfügen.

Falls Ihr Paket unabhängig von der Architektur funktioniert (beispielsweise ein Shell- oder Perl-Skript oder Dokumentation), ändern Sie dies in »`all`« und lesen Sie später unter Abschnitt 4.4 über die Benutzung der Regel `binary-indep` statt `binary-arch` für den Bau des Pakets.

Zeile 11 zeigt eine der mächtigsten Eigenschaften des Paketsystems von Debian. Pakete können auf verschiedene Arten miteinander in Beziehung stehen. Neben `Depends` (hängt ab von) gibt es noch die Beziehungsfelder `Recommends` (empfiehlt), `Suggests` (schlägt vor), `Pre-Depends` (setzt voraus), `Breaks` (beschädigt), `Conflicts` (kollidiert mit), `Provides` (enthält) und `Replaces` (ersetzt).

Die verschiedenen Paketverwaltungswerkzeuge verhalten sich normalerweise bei der Behandlung dieser Beziehungen gleich; wenn nicht, wird dies erklärt (siehe dpkg(8), dselect(8), apt(8), aptitude(1) usw.).

Es folgt eine vereinfachte Beschreibung der Paketbeziehungen: [7]

- `Depends`

  Das Paket wird erst installiert, wenn die hier aufgelisteten Pakete ebenfalls installiert sind. Benutzen Sie dies, falls ihr Programm ohne ein bestimmtes Paket überhaupt nicht läuft (oder schwere Schäden verursacht).

- `Recommends`

  Benutzen Sie dies für Pakete, die nicht absolut notwendig sind, die aber typischerweise mit Ihrem Programm verwendet werden. Wenn ein Benutzer Ihr Programm installiert, fragen wahrscheinlich alle Oberflächen nach, ob die empfohlenen Pakete auch installiert werden sollen. **aptitude** und **apt-get** installieren alle empfohlenen Pakete zusammen mit Ihrem Paket (aber der Benutzer kann diese Voreinstellung deaktivieren). **dpkg** ignoriert dieses Feld.

[6] Lesen Sie Debian Policy Manual, Kapitel 5.6.8 »Architecture« (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) für die genauen Details.

[7] Lesen Sie Debian Policy Manual, Kapitel 7 »Declaring relationships between packages« (http://www.debian.org/doc/debian-policy/ch-relationships.html) .

- `Suggests`

  Benutzen Sie dies für Pakete, die mit Ihrem Programm gut zusammenarbeiten, aber absolut nicht notwendig sind. Wenn ein Benutzer Ihr Programm installiert, werden sie wahrscheinlich nicht gefragt, ob die vorgeschlagenen Pakete auch installiert werden sollen. **aptitude** kann so konfiguriert werden, dass es vorgeschlagene Pakete zusammen mit Ihrem Paket installiert, aber dies ist nicht die Voreinstellung. **dpkg** und **apt-get** ignorieren dieses Feld.

- `Pre-Depends`

  Dies ist stärker als `Depends`. Das Paket wird erst installiert, wenn die hier aufgelisteten Pakete fertig installiert *und richtig konfiguriert* sind. Benutzen Sie dies *sehr* sparsam und nur, nachdem auf der Mailingliste debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) darüber diskutiert wurde. Lies: Verwenden Sie es überhaupt nicht. :-)

- `Conflicts`

  Das Paket wird erst installiert, wenn alle Pakete, mit denen es kollidiert, entfernt wurden. Benutzen Sie dies, wenn ihr Programm überhaupt nicht läuft oder schwere Probleme verursacht, solange ein bestimmtes Paket installiert ist.

- `Breaks`

  Wenn dieses Paket installiert ist, werden alle aufgeführten Pakete als beschädigt markiert. Normalerweise gibt ein Eintrag unter `Breaks` an, dass er auf Versionen älter als einen bestimmten Wert zutrifft. Die Lösung besteht üblicherweise darin, höherwertige Paketverwaltungswerkzeuge zu verwenden, um ein Upgrade der aufgeführten Pakete durchzuführen.

- `Provides`

  Für einige Paketarten mit mehreren Alternativen wurden virtuelle Namen definiert. Die vollständige Liste dieser virtuellen Pakete finden Sie in der Datei virtual-package-names-list.txt.gz (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) . Benutzen Sie dies, wenn Ihr Paket die Funktionalität eines existierenden virtuellen Pakets bietet.

- `Replaces`

  Benutzen Sie dies, wenn Ihr Paket Dateien eines anderen Pakets überschreibt oder ein anderes Paket vollständig ersetzt (wird zusammen mit `Conflicts` benutzt). Dateien der genannten Pakete werden mit den Dateien aus Ihrem Paket überschrieben.

All diese Felder haben eine einheitliche Syntax. Es ist jeweils eine durch Kommata getrennte Liste der Paketnamen. Diese Paketnamen können auch aus einer Liste von alternativen Paketnamen bestehen, die durch senkrechte Striche »|« (»pipe«-Zeichen) getrennt werden.

Die Anwendung der Felder kann auf bestimmte Versionen eines genannten Pakets beschränkt werden. Die Einschränkung jedes einzelnen Pakets wird in Klammern nach seinem Namen aufgeführt und sollte einen Vergleichsoperator aus der folgenden Liste enthalten, gefolgt von einem Versionsnummernwert. Die erlaubten Vergleiche sind `<<`, `<=`, `=`, `>=` und `>>` für strikt niedriger, niedriger oder gleich, genau gleich, höher oder gleich und strikt höher. Ein Beispiel:

```
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (<< 5), quux-foo (<= 7.6)
```

Die letzte Funktionalität, über die Sie Bescheid wissen müssen, ist `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}` usw.

dh_shlibdeps(1) berechnet die Abhängigkeiten von gemeinsam benutzten Bibliotheken für Binärpakete. Es erstellt eine Liste von ELF-Programmen und gemeinsam benutzten Bibliotheken, die es für jedes Binärpaket gefunden hat. Diese Liste wird zur Ersetzung von `${shlibs:Depends}` verwandt.

dh_perl(1) berechnet Perl-Abhängigkeiten. Es erstellt für jedes Binärpaket eine Liste von Abhängigkeiten von `perl` oder `perlapi`. Diese Liste wird zur Ersetzung von `${perl:Depends}` verwandt.

Einige `debhelper`-Befehle können dazu führen, dass das erstellte Paket von einigen zusätzlichen Paketen abhängt. Alle diese Befehle erstellen eine Liste von benötigten Paketen für jedes Binärpaket. Diese Liste wird zur Ersetzung von `${misc:Depends}` verwandt.

dh_gencontrol(1) erstellt die Datei `DEBIAN/control` für jedes Binärpaket und ersetzt dabei `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}`, usw.

Nach all diesen Erklärungen können wir das Feld `Depends` aber exakt so belassen, wie es jetzt ist und eine weitere Zeile dahinter einfügen, in der »`Suggests: file`« steht, weil `gentoo` einige Funktionen des Pakets `file` nutzen kann.

Zeile 9 ist die Homepage-URL. Nehmen wir an, diese sei http://www.obsession.se/gentoo/.

Zeile 12 enthält eine Kurzbeschreibung. Terminals sind typischerweise 80 Zeichen breit, also sollte die Kurzbeschreibung nicht länger als etwa 60 Zeichen sein. Ich ändere es in `fully GUI-configurable, two-pane X file manager`.

In die Zeile 13 kommt eine ausführliche Beschreibung. Sie sollte aus einem kleinen Text bestehen, der mehr über das Paket verrät. Die erste Spalte jeder Zeile muss leer sein. Es dürfen keine leeren Zeilen vorkommen, Sie können aber welche simulieren, indem Sie einen einzelnen »`.`« (Punkt) in die Zeile einsetzen. Es darf nach der ausführlichen Beschreibung auch nicht mehr als eine Leerzeile vorkommen. [8]

Wir können zwischen die Zeilen 6 und 7 die Felder `Vcs-*` einfügen, um das Versionskontrollsystem (VCS) zu dokumentieren. [9] Wir nehmen an, dass das Paket `gentoo` sein VCS im Git-Service von Debian Alioth unter `git://git.debian.org/git/collab` hat.

Zum Schluss ist dies die aktualisierte Datei `control`:

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 4.0.0
 7 Vcs-Git: https://anonscm.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: https://anonscm.debian.org/git/collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
18  program itself. If you still prefer to hand-edit configuration files,
19  they’re fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(Die Zeilennummerierung habe ich hinzugefügt.)

4.2 `copyright`

Diese Datei enthält Informationen über das Copyright und die Lizenz der Quellen der Originalautoren. Debian Policy Manual, Kapitel 12.5 »Copyright information« (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) gibt den Inhalt vor und DEP-5: Machine-parseable `debian/copyright` (http://dep.debian.net/deps/dep5/) bietet Hilfestellungen für ihr Format.

[8] Diese Beschreibungen sind auf Englisch. Übersetzungen dieser Beschreibungen werden durch das Debian Description Translation Project - DDTP (http://www.debian.org/intl/l10n/ddtp) bereitgestellt.

[9] Lesen Sie Debian-Entwicklerreferenz, 6.2.5. »Ort des Versionsverwaltungssystems« (http://www.debian.org/doc/manuals/developers-reference/best-pkging-practices.html#bpp-vcs) .

Dh_make kann eine Vorlage für die Datei `copyright` erzeugen. Verwenden Sie hier die Option »`--copyright gpl2`«, um eine Vorlage für das Paket `gentoo` zu erhalten, das unter GPL-2 veröffentlicht wurde.

Sie müssen hier fehlende Informationen eintragen, um die Datei zu vervollständigen, beispielsweise die Quelle, von der Sie das Paket bezogen haben, die tatsächlichen Copyright-Vermerke und die Lizenz. Bei den verbreiteten Lizenzen von freier Software (GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0, 3-Clause BSD, CC0-1.0, MPL-1.1, MPL-2.0 oder der Artistic license) können Sie auf die entsprechende Datei im Verzeichnis `/usr/share/common-licenses/` verweisen, das auf jedem Debian-System existiert. Anderenfalls müssen Sie die vollständige Lizenz einfügen.

Zusammengefasst sollte die Datei `copyright` von `gentoo` so aussehen:

```
 1 Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
 2 Upstream-Name: gentoo
 3 Upstream-Contact: Emil Brink <emil@obsession.se>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Files: *
 7 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 8 License: GPL-2+
 9
10 Files: icons/*
11 Copyright: 1998 Johan Hanson <johan@tiq.com>
12 License: GPL-2+
13
14 Files: debian/*
15 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
16 License: GPL-2+
17
18 License: GPL-2+
19  This program is free software; you can redistribute it and/or modify
20  it under the terms of the GNU General Public License as published by
21  the Free Software Foundation; either version 2 of the License, or
22  (at your option) any later version.
23  .
24  This program is distributed in the hope that it will be useful,
25  but WITHOUT ANY WARRANTY; without even the implied warranty of
26  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
27  GNU General Public License for more details.
28  .
29  You should have received a copy of the GNU General Public License along
30  with this program; if not, write to the Free Software Foundation, Inc.,
31  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
32  .
33  On Debian systems, the full text of the GNU General Public
34  License version 2 can be found in the file
35  '/usr/share/common-licenses/GPL-2'.
```

(Die Zeilennummerierung habe ich hinzugefügt.)

Bitte befolgen Sie das »HOWTO«, das von den FTP-Masters zur Verfügung gestellt wird und an debian-devel-announce geschickt wurde: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3 `changelog`

Dies ist eine zwingend vorgeschriebene Datei, deren Format im Debian Policy Manual, Kapitel 4.4 »debian/changelog« (http://www.debia
doc/debian-policy/ch-source.html#s-dpkgchangelog) beschrieben wird. Dieses Format benötigen **dpkg** und andere Programme, um die Versionsnummer, Revision, Distribution und die Dringlichkeit Ihres Pakets zu bestimmen.

Für Sie ist die Datei ebenfalls wichtig, weil es sinnvoll ist, alle von Ihnen vorgenommenen Änderungen zu dokumentieren. Damit können Leute, die Ihr Paket herunterladen, einfacher herausfinden, ob es Probleme mit dem Paket gibt, die sie kennen sollten. Diese Datei wird im Binärpaket als `/usr/share/doc/gentoo/changelog.Debian.gz` gespeichert.

dh_make hat eine Standardvorlage erstellt, die so aussieht:

```
1  gentoo (0.9.12-1) unstable; urgency=medium
2
3   * Initial release. (Closes: #nnnn)  <nnnn ist die Fehlernummer Ihres ITP>
4
5  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
6
```

(Die Zeilennummerierung habe ich hinzugefügt.)

In der Zeile 1 stehen der Paketname, die Version, die Distribution und die Dringlichkeit. Der Name muss mit dem Namen des Quellpakets übereinstimmen, die Distribution sollte `unstable` sein und die Dringlichkeit sollte auf `medium` gesetzt werden, falls es keinen besonderen Grund für einen anderen Wert gibt.

Zeilen 3-5 sind ein Protokolleintrag, in denen Sie die in dieser Paketrevision gemachten Änderungen dokumentieren können (hier kommen keine Änderungen des Originalautors hinein; für diese Zwecke gibt es eine spezielle Datei, die von den Originalautoren erstellt wurde und die Sie später als `/usr/share/doc/gentoo/changelog.gz` installieren werden). Wir nehmen an, dass die Nummer Ihres ITP-Fehlerberichts (»Intent To Package«; Absicht, ein Paket zu erstellen) »`12345`« lautet. Neue Zeilen werden direkt unter der obersten Zeile, die mit einem Stern (»`*`«) beginnt, eingefügt. Sie können das mit dch(1) erledigen. Sie können dies per Hand mit einem Texteditor bearbeiten, solange Sie den von dch(1) verwandten Formatierungskonventionen folgen.

Um zu verhindern, dass ein Paket versehentlich hochgeladen wird, bevor es fertig ist, empfiehlt es sich, den Distributionswert auf einen ungültigen Ausdruck `UNRELEASED` zu setzen.

Schließlich kommen Sie zu einer Datei wie dieser hier:

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2
3   * Initial Release. Closes: #12345
4   * This is my first Debian package.
5   * Adjusted the Makefile to fix $(DESTDIR) problems.
6
7  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
8
```

(Die Zeilennummerierung habe ich hinzugefügt.)

Sobald Sie mit allen Änderungen zufrieden und sie im `changelog` dokumentiert haben, sollten Sie den Wert der Distribution von `UNRELEASED` auf den Ziel-Distributionswert `unstable` (oder sogar `experimental`) ändern. 10

Sie können später mehr über Aktualisierungen der Datei `changelog` in Kapitel 8 lesen.

4.4 `rules`

Wir werfen nun einen Blick auf die genauen Regeln, die dpkg-buildpackage(1) verwenden wird, um das Paket zu bauen. Diese Datei ist in Wirklichkeit ein weiteres `Makefile`, aber anders als das von den Originalautoren mitgelieferte. Im Unterschied zu den anderen Dateien im Verzeichnis `debian` ist diese Datei als ausführbar gekennzeichnet.

10Falls Sie den Befehl `dch -r` zur Durchführung dieser letzten Änderung verwenden, stellen Sie sicher, dass Sie die Datei `changelog` explizit im Editor speichern.

4.4.1 Ziele der Datei `rules`

Jede `rules`-Datei, wie jedes andere `Makefile`, besteht aus mehreren Regeln, von der jede ein Ziel und eine Ausführung definiert.[11] Eine neue Regel beginnt mit der Ziel-Deklaration in der ersten Spalte. Die folgenden Zeilen beginnen mit einem Tabulator (ASCII 9), nach dem das Rezept zur Durchführung des Ziels angegeben wird. Leere Zeilen und Zeilen, die mit einem # (Hash) anfangen, werden als Kommentare behandelt und ignoriert. [12]

Eine Regel, die Sie ausführen möchten, wird durch ihren Zielnamen als Befehlszeilenargument aufgerufen. Beispielsweise führen `debian/rules` *`build`* und `fakeroot make -f debian/rules` *`binary`* die Regeln für die Ziele *`build`* respektive *`binary`* aus.

Es folgt eine vereinfachte Erklärung der Ziele:

- `clean`-Ziel: Löschen aller kompilierten, erzeugten und nicht benötigten Dateien im Bauverzeichnis (erforderlich)
- `build`-Ziel: Bauen der kompilierten Programme und formatierten Dokumente aus den Quellen im Bauverzeichnis (erforderlich)
- `build-arch`-Ziel: Bauen der kompilierten architekturabhängigen Programme aus den Quellen im Bauverzeichnis (erforderlich)
- `build-indep`-Ziel: Bauen der architekturunabhängigen formatierten Dokumente aus den Quellen im Bauverzeichnis (erforderlich)
- `install`-Ziel: Installieren der Dateien in einen Verzeichnisbaum unterhalb des Verzeichnisses `debian` für jedes Binärpaket. Falls sie festgelegt wurden, hängen `binary*`-Ziele effektiv von diesem Ziel ab (optional)
- `binary`-Ziel: Erstellen aller Binärpakete (effektiv ist dies die Kombination der `binary-arch`- und `binary-indep`-Ziele) (erforderlich) [13]
- `binary-arch`-Ziel: Erstellen Architektur-abhängiger (`Architecture: any`) Binärpakete im übergeordneten Verzeichnis (erforderlich) [14]
- `binary-indep`-Ziel: Erstellen Architektur-unabhängiger (`Architecture: all`) Binärpakete im übergeordneten Verzeichnis (erforderlich) [15]
- `get-orig-source`-Ziel: Herunterladen der neuesten Version des Quellpakets von dem Archiv der Originalautoren (optional)

Sie sind jetzt wahrscheinlich überwältigt, aber nach der genaueren Betrachtung der Datei `rules`, die uns **dh_make** als Vorgabe erstellt hat, wird alles viel einfacher werden.

4.4.2 Die vorgegebene Datei `rules`

Neuere Versionen von **dh_make** erzeugen als Vorgabe eine sehr einfache, doch mächtige Datei `rules`, indem sie den Befehl **dh** verwenden:

```
1 #!/usr/bin/make -f
2 # See debhelper(7) (uncomment to enable)
3 # output every command that modifies files on the build system.
4 #DH_VERBOSE = 1
5
```

11 Sie können das Schreiben einer `Makefile` erlernen, indem Sie Debian Reference, 12.2. ”Make” (http://www.debian.org/doc/manuals/debian-reference/-ch12#_make) lesen. Die komplette Dokumentation ist als http://www.gnu.org/software/make/manual/html_node/index.html oder als Paket `make-doc` im Archivbereich `non-free` verfügbar.

12 Debian Policy Manual, Kapitel 4.9 »Main building script: debian/rules« (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) erklärt die Details.

13 Dieses Ziel wird von »`dpkg-buildpackage`« wie in Abschnitt 6.1 beschrieben benutzt.

14 Dieses Ziel wird von »`dpkg-buildpackage -B`« wie in Abschnitt 6.2 beschrieben benutzt.

15 Dieses Ziel wird von »`dpkg-buildpackage -A`« benutzt.

```
 6 # see FEATURE AREAS in dpkg-buildflags(1)
 7 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
 8
 9 # see ENVIRONMENT in dpkg-buildflags(1)
10 # package maintainers to append CFLAGS
11 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
12 # package maintainers to append LDFLAGS
13 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
14
15
16 %:
17         dh $@
```

(Die Zeilennummerierung habe ich hinzugefügt und auch einige Kommentare verkürzt. In der richtigen `rules`-Datei sind die führenden Leerzeichen Tabulatoren.)

Sie sind wahrscheinlich mit ähnlichen Zeilen wie der Zeile 1 aus Shell- oder Perl-Skripten vertraut. Sie teilt dem Betriebssystem mit, dass das Skript mit `/usr/bin/make` verarbeitet werden soll.

In Zeile 4 kann das Kommentarzeichen entfernt werden, um die Variable `DH_VERBOSE` auf 1 zu setzen. Dann gibt der Befehl **dh** aus, welche **dh_***-Befehle er ausführt. Sie können hier auch eine Zeile »`export DH_OPTIONS=-v`« hinzufügen. Dann gibt jeder **dh_***-Befehl aus, welche Befehle von jedem **dh_***-Befehl ausgeführt werden. Das hilft Ihnen dabei, zu verstehen, was genau hinter den Kulissen dieser einfachen `rules`-Datei passiert. So können Sie Probleme besser finden. Das neue **dh** ist als ein zentraler Bestandteil der `debhelper`-Werkzeuge entwickelt worden und versteckt nichts vor Ihnen.

In den Zeilen 16 und 17 wird die gesamte Arbeit mit einer impliziten Regel, die die Muster-Regel verwendet, erledigt. Das Prozentzeichen steht für ein »beliebiges Ziel«, das dann lediglich ein Programm aufruft, nämlich **dh** mit dem Namen des Ziels. [16] Der Befehl **dh** ist ein Skript, das abhängig vom übergebenen Argument die entsprechenden Sequenzen von **dh_***-Programmen ausführt. [17]

- »`debian/rules clean`« ruft »`dh clean`« auf, das wiederum folgendes ausführt:

```
dh_testdir
dh_auto_clean
dh_clean
```

- »`debian/rules build`« ruft »`dh build`« auf, das wiederum folgendes ausführt:

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- »`fakeroot debian/rules binary`« ruft »`fakeroot dh binary`« auf, das wiederum folgendes ausführt [18]:

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
```

[16] Dies verwendet die neuen Möglichkeiten von `debhelper` v7+. Dessen Designkonzepte werden in »Not Your Grandpa's Debhelper« (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) erklärt, das auf der DebConf9 vom `debhelper`-Betreuer präsentiert wurde. Unter `lenny` erzeugte **dh_make** eine wesentlich kompliziertere Datei `rules` mit expliziten Regeln und vielen **dh_***-Skripten für jede der Regeln, wobei die meisten jetzt unnötig sind (und das Alter des Pakets zeigen). Der neue **dh**-Befehl ist einfacher und befreit uns von der »manuellen« Durchführung dieser Routinearbeit. Sie haben mit den `override_dh_*`-Zielen weiterhin die vollständige Kontrolle über diesen Prozess mit den Zielen `override_dh_*`. Siehe Abschnitt 4.4.3. Er basiert lediglich auf dem Paket `debhelper` und verschleiert den Prozess des Paketbaus nicht, wie dies beim Paket `cdbs` sein kann.

[17] Sie können überprüfen, welche Sequenzen von **dh_***-Programmen für ein bestimmtes *`Ziel`* tatsächlich aufgerufen werden, indem Sie »`dh Ziel --no-act`« oder »`debian/rules -- 'Ziel --no-act'`« ausführen. Dadurch werden die Sequenzen nicht ausgeführt.

[18] Im folgenden Beispiel wird angenommen, dass `debian/compat` einen Wert gleich oder größer als 9 hat, um die Python-Unterstützungsbefehle automatisch aufzurufen.

```
dh_installchangelogs
dh_installexamples
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
dh_installmodules
dh_installlogcheck
dh_installlogrotate
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- »`fakeroot debian/rules binary-arch`« ruft »`fakeroot dh binary-arch`« auf, das wiederum dieselbe Sequenz ausführt wie »`fakeroot dh binary`«, allerdings wird an jeden Befehl die Option »`-a`« angehängt.
- »`fakeroot debian/rules binary-indep`« ruft »`fakeroot dh binary-indep`« auf, das wiederum fast dieselbe Sequenz ausführt wie »`fakeroot dh binary`«, allerdings wird an jeden Befehl die Option »`-i`« angehängt und die Befehle **dh_strip**, **dh_makeshlibs** und **dh_shlibdeps** werden weggelassen.

Die Funktion der Befehle **dh_*** sind größtenteils aus ihren Namen selbsterklärend. [19] Es gibt einige bemerkenswerte, für die es Sinn ergibt, hier unter der Annahme einer typischen Bauumgebung, basierend auf einem `Makefile`, eine stark vereinfachte Erläuterung bereitzustellen: [20]

- **dh_auto_clean** führt normalerweise folgendes aus, wenn ein `Makefile` existiert und das Ziel `distclean` enthält. [21]

```
make distclean
```

- **dh_auto_configure** führt normalerweise folgendes aus, wenn die Datei `configure` existiert (Argumente zur besseren Lesbarkeit abgekürzt).

[19] Für die komplette Information darüber, was die ganzen **dh_***-Skripte genau durchführen und wie ihre Optionen lauten, lesen Sie bitte deren Handbuchseiten und die Dokumentation von `debhelper`.

[20] Diese Befehle unterstützen andere Bauumgebungen wie `setup.py`, die durch Ausführen von `dh_auto_build --list` in einem Paketbauverzeichnis aufgelistet werden können.

[21] Tatsächlich wird nach dem ersten verfügbaren Ziel aus der Liste `distclean`, `realclean` oder `clean` in dem `Makefile` gesucht und dieses ausgeführt.

```
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
```

- **dh_auto_build** führt normalerweise folgendes aus, um das erste Ziel des `Makefile`s auszuführen, falls dieses existiert.

```
make
```

- **dh_auto_test** führt normalerweise folgendes aus, falls ein `Makefile` mit dem Ziel `test` existiert. 22

```
make test
```

- **dh_auto_install** führt normalerweise folgendes aus, falls ein `Makefile` mit dem Ziel `install` existiert (Zeile zur besseren Lesbarkeit umgebrochen).

```
make install \
  DESTDIR=/Pfad/zum/Paket_Version-Revision/debian/Paket
```

Alle Ziele, die den Befehl **fakeroot** erfordern, werden **dh_testroot** enthalten. Falls Sie nicht diesen Befehl benutzen, um vorzugeben, »root« zu sein, wird er mit einer Fehlermeldung beendet.

Das Wichtigste, was Sie über die Datei `rules` wissen müssen, die von **dh_make** erzeugt wurde, ist, dass sie lediglich ein Vorschlag ist. Sie funktioniert für die meisten Pakete, aber für etwas kompliziertere Pakete sollten Sie sich nicht scheuen, sie für Ihre Zwecke anzupassen.

Obwohl »`install`« kein erforderliches Ziel ist, wird es unterstützt. »`fakeroot dh install`« verhält sich wie »`fakeroot dh binary`«, stoppt aber nach **dh_fixperms**.

4.4.3 Anpassungen der Datei `rules`

Es gibt viele Arten, die `rules`-Datei anzupassen, die den neuen Befehl **dh** verwendet.

Der Befehl »`dh $@`« kann wie folgt angepasst werden: 23

- Unterstützung des Befehls **dh_python2** hinzufügen (die beste Wahl für Python). 24
  - Aufnahme des Pakets `python` in »`Build-Depends`«.
  - Verwenden Sie »`dh $@ --with python2`«.
  - Hiermit werden Python-Module mit dem Rahmenwerk `python` bearbeitet.
- Unterstützung für den Befehl **dh_pysupport** hinzufügen. (veraltet)
  - Aufnahme des Pakets `python-support` in »`Build-Depends`«.
  - Verwenden Sie »`dh $@ --with pysupport`«.
  - Hiermit werden Python-Module mit dem Rahmenwerk `python-support` bearbeitet.
- Unterstützung für den Befehl **dh_pycentral** hinzufügen. (veraltet)
  - Aufnahme des Pakets `python-central` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with python-central`«.
  - Hierdurch wird auch der Befehl **dh_pysupport** deaktiviert.
  - Hiermit werden Python-Module mit dem Rahmenwerk `python-central` bearbeitet.

22 Tatsächlich wird nach dem ersten verfügbaren Ziel aus der Liste `test` oder `check` in dem `Makefile` gesucht und dieses ausgeführt.

23 Falls ein Paket die Datei `/usr/share/perl5/Debian/Debhelper/Sequence/`*`Eigener_Name`*`.pm` installiert, können Sie dessen angepasste Funktion mittels »`dh $@ --with `*`Eigener-Name`*« aktivieren.

24 Die Benutzung des Befehls **dh_python2** wird gegenüber den Befehlen **dh_pysupport** oder **dh_pycentral** bevorzugt. Verwenden Sie nicht den Befehl **dh_python**.

- Unterstützung für den Befehl **dh_installtex** hinzufügen.
  - Aufnahme des Pakets `tex-common` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with tex`«.
  - Hiermit werden Type-1-Schriften, Muster für Silbentrennung und Formate für TeX registriert.
- Unterstützung für die Befehle **dh_quilt_patch** und **dh_quilt_unpatch** hinzufügen.
  - Aufnahme des Pakets `quilt` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with quilt`«.
  - Hiermit werden für ein Quellpaket im Format `1.0` aus Dateien im Verzeichnis `debian/patches` Patches auf die ursprünglichen Quellen angewendet und auch wieder rückgängig gemacht.
  - Dies wird nicht benötigt, falls Sie das neue Quellpaketformat `3.0 (quilt)` benutzen.
- Unterstützung für den Befehl **dh_dkms** hinzufügen.
  - Aufnahme des Pakets `dkms` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with dkms`«.
  - Hiermit wird die korrekte Verwendung von DKMS durch die Kernel-Modul-Pakete sichergestellt.
- Unterstützung für die Befehle **dh_autotools-dev_updateconfig** und **dh_autotools-dev_restoreconfig** hinzufügen.
  - Aufnahme des Pakets `autotools-dev` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with autotools-dev`«.
  - Hiermit werden die Dateien `config.sub` und `config.guess` aktualisiert und wiederhergestellt.
- Unterstützung für den Befehle **dh_autoreconf** und **dh_autoreconf_clean** hinzufügen.
  - Aufnahme des Pakets `dh-autoreconf` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with autoreconf`«.
  - Hiermit werden die Dateien des GNU-Bausystems aktualisiert und nach dem Bau wiederhergestellt.
- Unterstützung für den Befehl **dh_girepository** hinzufügen.
  - Aufnahme des Pakets `gobject-introspection` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with gir`«.
  - Das berechnet Abhängigkeiten für Pakete, die GObject-Introspektionsdaten ausliefern und erstellt die Ersetzungsvariable `${gir:Depends}` für die Paketabhängigkeiten.
- Unterstützung für die Vervollständigung in der **bash** hinzufügen.
  - Aufnahme des Pakets `bash-completion` in »`Build-Depends`«.
  - Verwenden Sie stattdessen »`dh $@ --with bash-completion`«.
  - Hiermit wird die Vervollständigung durch **bash** unter Verwendung einer Konfigurationsdatei in `debian/`*`Paket`*`.bash-complet` installiert.

Viele **dh_***-Befehle, die vom neuen **dh**-Befehl aufgerufen werden, können durch entsprechende Konfigurationsdateien im `debian`-Verzeichnis angepasst werden. Siehe Kapitel 5 sowie die Handbuchseite jedes Befehls für Anpassungen dieser Funktionalitäten.

Es kann nötig sein, über **dh** aufgerufene **dh_***-Befehle mit zusätzlichen Argumenten oder zusätzliche Befehle mit ihnen auszuführen oder sie ganz auszulassen. In solchen Fällen können Sie ein `override_dh_`*`foo`*-Ziel mit der entsprechenden Regel in der Datei `rules` erstellen, das ein `override_dh_`*`foo`*-Ziel für den Befehl **dh_*****foo*** definiert, den Sie ändern wollen. Im Grunde bedeutet das nur »*führe stattdessen mich aus*«. [2][5]

[2][5] Falls Sie unter `Lenny` das Verhalten eines **dh_***-Skripts ändern wollten, mussten Sie die entsprechende Zeile in der Datei `rules` aufsuchen und dort anpassen.

Bitte beachten Sie, dass die **dh_auto_***-Befehle dazu neigen, mehr als die in dieser stark vereinfachten Erklärung dargestellten Tätigkeiten zu erledigen, um Randfälle zu berücksichtigen. Es ist keine gute Idee, `override_dh_*`-Ziele zu verwenden, um die vereinfachten äquivalenten Befehle als Ersatz zu verwenden (außer beim Ziel `override_dh_auto_clean`), da damit solche pfiffigen `debhelper`-Funktionalitäten umgangen werden.

Falls Sie beispielsweise mittels der Autotools Systemkonfigurationsdaten des aktuellen `gentoo`-Pakets im Verzeichnis `/etc/gentoo` statt im normalen Verzeichnis `/etc` speichern wollen, können Sie das Vorgabeargument `--sysconfig=/etc` für **./configure** im Befehl **dh_auto_configure** durch folgendes außer Kraft setzen:

```
override_dh_auto_configure:
        dh_auto_configure -- --sysconfig=/etc/gentoo
```

Die nach dem `--` übergebenen Argumente werden zu den vorgegebenen Argumenten des automatisch ausgeführten Programms hinzugefügt, um sie zu überschreiben. Die Benutzung des Befehls **dh_auto_configure** ist besser als der direkte Aufruf von **./configure**, weil in diesem Fall lediglich das Argument `--sysconfig` überschrieben wird und andere, sehr wohl beabsichtigte, Argumente für den Befehl **./configure** erhalten bleiben.

Falls das `Makefile` in den Quellen von `gentoo` zum Bauen explizit das Ziel `build` benötigt, [2][6], können Sie ein Ziel namens `override_dh_auto_build` erstellen, um dies zu ermöglichen.

```
override_dh_auto_build:
        dh_auto_build -- build
```

Hiermit wird sichergestellt, dass `$(MAKE)` mit allen voreingestellten Argumenten des Befehls **dh_auto_build** plus dem Argument `build` ausgeführt wird.

Falls das `Makefile` in den Quellen von `gentoo` zum Aufräumen für das Debianpaket explizit das Ziel `packageclean` benötigt statt der Ziele `distclean` oder `clean`, können Sie ein Ziel namens `override_dh_auto_clean` erstellen, um dies zu nutzen.

```
override_dh_auto_clean:
        $(MAKE) packageclean
```

Falls das `Makefile` in den Quellen von `gentoo` das Ziel `test` enthält, das Sie im Paketbau-Prozess für Debian nicht ausführen wollen, können Sie ein leeres Ziel namens `override_dh_auto_test` erstellen, um dies zu übergehen.

```
override_dh_auto_test:
```

Wenn `gentoo` eine unübliche ursprüngliche Changelog-Datei namens `FIXES` enthält, wird diese standardmäßig von **dh_installchangelo** nicht installiert. Der Befehl **dh_installchangelogs** braucht den Namen `FIXES` als Argument, um die Datei zu installieren. [2][7]

```
override_dh_installchangelogs:
        dh_installchangelogs FIXES
```

Wenn Sie den neuen **dh**-Befehl benutzen, wird es schwierig, die genauen Effekte von expliziten Zielen wie den in Abschnitt 4.4.1 aufgelisteten zu verstehen. Eine Ausnahme stellt `get-orig-source` dar. Bitte beschränken Sie daher - soweit möglich - explizite Ziele auf solche mit den Namen `override_dh_*` sowie vollständig davon unabhängige.

[2][6] **dh_auto_build** ohne Argumente führt das erste Ziel in dem `Makefile` aus.

[2][7] Die Dateien `debian/changelog` und `debian/NEWS` werden immer automatisch installiert. Das Changelog der Originalautoren wird gefunden, indem die Dateinamen in Kleinbuchstaben umgewandelt werden und mit `changelog`, `changes`, `changelog.txt` und `changes.txt` verglichen werden.

Kapitel 5

Andere Dateien im Verzeichnis `debian`

Der Befehl **dh_make** wurde umfangreich aktualisiert, seitdem dieses alte Dokument geschrieben wurde. Daher treffen einige Teile dieses Dokuments nicht mehr zu.

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Der Befehl **debmake** wird anstelle des Befehls **dh_make** in meinem neuen Guide for Debian Maintainers (https://www.debian.org/-doc/devel-manuals#debmake-doc) verwandt.

Um kontrollieren zu können, was genau `debhelper` während des Paketbauens durchführt, erstellen Sie optionale Konfigurationsdateien im Verzeichnis `debian`. In diesem Kapitel finden Sie einen Überblick darüber, wofür die einzelnen Dateien benötigt werden und wie ihr jeweiliges Format aussieht. Im Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) und in der Debian-Entwicklerreferenz (http://www.debian.org/doc/devel-manuals#devref) finden Sie Anleitungen zum Thema Paketierung.

Der Befehl **dh_make** wird einige Vorlagenkonfigurationsdateien unter dem Verzeichnis `debian` erstellen. Schauen Sie sich alle an.

In diesem Kapitel werden zur Vereinfachung Dateien im Verzeichnis `debian` ohne das einleitende `debian/` referenziert, wann immer die Bedeutung eindeutig ist.

Der Befehl **dh_make** erstellt für `debhelper` nicht alle Konfigurationsdateien. Falls Sie diese benötigen, müssen Sie sie mit einem Editor erstellen.

Falls Sie eine dieser Dateien verwenden möchten oder müssen, machen Sie bitte folgendes:

- Benennen Sie die Konfigurationsdateien um, so dass sie den tatsächlichen Paketnamen anstatt *`Paket`* verwenden.
- Passen Sie die Inhalte der Vorlagedateien Ihren Bedürfnissen an.
- Löschen Sie Vorlagedateien, die Sie nicht benötigen.
- Passen Sie die Datei `control` an, falls notwendig (siehe Abschnitt 4.1).
- Passen Sie die Datei `rules` an, falls notwendig (siehe Abschnitt 4.4).

Alle `debhelper`-Konfigurationsdateien ohne ein *`Paket`*-Präfix, wie beispielsweise die Datei `install`, beziehen sich auf das erste Binärpaket. Wenn es mehrere Binärpakete gibt, können die jeweiligen Konfigurationen dadurch zugeordnet werden, dass der Name des Pakets als Präfix vor den Namen der Konfigurationsdatei gestellt wird. Beispiele sind *`Paket-1`*`.install`, *`Paket-2`*`.install` usw.

5.1 README.Debian

Alle zusätzlichen Details oder Unterschiede zwischen dem ursprünglichen Paket und Ihrer Debian-Version sollten hier dokumentiert werden.

dh_make erstellt eine Standardvorlage, die so aussieht:

```
gentoo for Debian
-----------------
<possible notes regarding this package - if none, delete this file>
 -- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

Falls Sie nichts zu dokumentieren haben, löschen Sie diese Datei. Siehe dh_installdocs(1).

5.2 compat

Die Datei `compat` legt die `debhelper`-Kompatibilitätsstufe fest. Derzeit sollten Sie `debhelper` V10 verwenden, indem Sie folgendes ausführen:

```
$ echo 10 > debian/compat
```

Unter bestimmten Umständen können Sie die Kompatibilitätsstufe V9 für die Kompatibilität mit älteren Systemen verwenden. Es wird aber empfohlen, unter keinen Umständen eine Stufe unterhalb von V9 zu verwenden; deren Verwendung in neuen Paketen sollte vermieden werden.

5.3 conffiles

Eine der ärgerlichsten Sachen bei Software ist es, wenn Sie richtig viel Zeit und Mühe in die Konfiguration eines Programms investieren und schon das nächste Upgrade alle Ihre Änderungen zunichte macht. Debian löst dieses Problem, indem diese Konfigurationsdateien als Conffiles markiert werden. 1. Wenn Sie ein Upgrade eines Pakets durchführen, werden Sie gefragt, ob Sie Ihre alte Konfigurationsdateien behalten wollen oder nicht.

dh_installdeb(1) markiert *automatisch* alle Dateien im Verzeichnis `/etc` als »conffiles«. Wenn Ihr Programm also nur dort Konfigurationsdateien besitzt, müssen Sie sie nicht in dieser Datei auflisten. Bei den meisten Paketarten sollte der einzige Ort, an dem sich jemals Conffiles befinden, `/etc` sein und daher muss diese Datei nicht existieren.

Falls Ihr Programm Konfigurationsdateien nutzt, diese aber auch selbst bearbeitet, ist es das Beste, diese nicht als Conffiles zu kennzeichnen, weil sonst **dpkg** den Benutzer jedes Mal auffordert, Änderungen zu bestätigen.

Falls es für das Programm, das Sie paketieren, erforderlich ist, dass jeder Benutzer die Konfigurationsdateien im Verzeichnis `/etc` anpassen muss, gibt es zwei populäre Arten, es so einzurichten, dass sie keine Conffiles sind, um **dpkg** ruhig zu stellen:

- Erstellen Sie einen symbolischen Link im Verzeichnis `/etc`, der auf eine Datei im Verzeichnis `/var` zeigt. Dieser kann von den Betreuerskripten erzeugt werden.
- Erstellen Sie eine Datei, die von den Betreuerskripten im Verzeichnis `/etc` erzeugt wird.

Für weitere Informationen über die Betreuerskripte lesen Sie Abschnitt 5.18.

1Lesen Sie dpkg(1) und das Debian Policy Manual, »D.2.5 Conffiles« (http://www.debian.org/doc/debian-policy/ap-pkg-controlfields.html#s-pkg-f-Conffiles) .

5.4 *Paket*.cron.*

Falls Ihr Paket immer wiederkehrende Aufgaben erledigen muss, um korrekt zu arbeiten, können Sie diese Dateien benutzen, um das einzurichten. Sie können regelmäßige Aufgaben erstellen, die stündlich, täglich, wöchentlich oder monatlich ausgeführt werden. Alternativ kann dies auch zu jedem anderen von Ihnen gewünschten Zeitpunkt stattfinden. Die Dateinamen lauten:

- *Paket*.cron.hourly - wird als /etc/cron.hourly/*Paket* installiert: Ausführung einmal pro Stunde.
- *Paket*.cron.daily - wird als /etc/cron.daily/*Paket* installiert: Ausführung einmal pro Tag.
- *Paket*.cron.weekly - wird als /etc/cron.weekly/*Paket* installiert: Ausführung einmal pro Woche.
- *Paket*.cron.monthly - wird als /etc/cron.monthly/*Paket* installiert: Ausführung einmal pro Monat.
- *Paket*.cron.d - wird als /etc/cron.d/*Paket* installiert: Ausführung zu anderen Zeiten.

Die meisten dieser Dateien sind Shellskripte. Die einzige Ausnahme stellt *Paket*.cron.d dar, das im Format einer crontab(5) vorliegen muss.

Es wird keine dedizierte cron.*-Datei für das Rotieren der Protokolldateien benötigt. Lesen Sie dazu bitte dh_installlogrotate(1) und logrotate(8).

5.5 dirs

In dieser Datei werden alle Verzeichnisse festgelegt, die wir brauchen, die aber von der normalen Installationsprozedur (»make install DESTDIR=...«, aufgerufen von »dh_auto_install«) aus irgendwelchen Gründen nicht erstellt werden. Dies bedeutet fast immer, dass es ein Problem mit dem Makefile gibt.

Für Dateien, die in der Datei install aufgelistet sind, müssen die Verzeichnisse nicht zuerst erstellt werden. Siehe Abschnitt 5.11.

Am besten ist es, wenn Sie zunächst die Installation ausprobieren und diesen Mechanismus nur dann benutzen, wenn es dabei Probleme gibt. Es gibt keinen einleitenden Schrägstrich bei den Verzeichnisnamen, die in der Datei dirs aufgeführt sind.

5.6 *Paket*.doc-base

Hat Ihr Programm außer Handbuch- und Info-Seiten noch andere Dokumentation, sollten Sie die Datei doc-base benutzen, um diese zu registrieren, damit der Benutzer sie mit Programmen wie dhelp(1), dwww(1) oder doccentral(1) finden kann.

Das schließt normalerweise HTML-, PS- und PDF-Dateien ein, die sich in /usr/share/doc/*Paketname*/ befinden.

So sieht die doc-base-Datei gentoo.doc-base für das Paket gentoo aus:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: File Management
Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

Informationen über das Format dieser Datei finden Sie in install-docs(8) und der Debian-Anleitung von doc-base in der lokalen Kopie /usr/share/doc/doc-base/doc-base.html/index.html, die vom Paket doc-base bereitgestellt wird.

Für weitere Details über das Installieren von zusätzlicher Dokumentation sehen Sie bitte in Abschnitt 3.3 nach.

5.7 docs

Diese Datei enthält die Dateinamen der Dokumentationsdateien, die dh_installdocs(1) für uns in das temporäre Verzeichnis installiert.

Standardmäßig schließt das alle Dateien im obersten Verzeichnis des Quellcodes ein, die da heißen »BUGS«, »README*«, »TODO« usw.

Für gentoo werden noch weitere Dateien hinzugefügt:

```
BUGS
CONFIG-CHANGES
CREDITS
NEWS
README
README.gtkrc
TODO
```

5.8 emacsen-*

Falls Ihr Paket Emacs-Dateien bereitstellt, die während der Installation des Pakets kompiliert werden, können Sie diese Dateien dafür nutzen.

Sie werden durch dh_installemacsen(1) ins temporäre Verzeichnis installiert.

Falls Sie dies nicht benötigen, löschen Sie die Dateien.

5.9 *Paket*.examples

Der Befehl dh_installexamples(1) installiert die in dieser Datei aufgelisteten Dateien und Verzeichnisse als Beispieldateien.

5.10 *Paket*.init und *Paket*.default

Falls Ihr Paket einen Daemon enthält, der beim Hochfahren des Systems gestartet werden muss, haben Sie offensichtlich meine anfängliche Empfehlung missachtet, stimmt's? :-)

Bitte lesen Sie dh_installinit(1) dh_installsystemd(1), um ein Hochfahr-Skript bereitzustellen.

Die Datei *Paket*.default wird als /etc/default/*Paket* installiert. Diese Datei legt Voreinstellungen fest, die vom Init-Skript eingelesen werden. Diese Datei *package*.default wird meistens zum Setzen einiger Vorgabewerte für Schalter oder Zeitüberschreitungen verwandt. Falls in Ihrem init-Skript bestimmte einstellbare Funktionen sind, können Sie diese in der Datei *package*.default statt im init-Skript selbst einstellen.

Falls Ihr Programm der Originalautoren eine Datei für ein Init-Skript bereitstellt, können Sie dies entweder benutzten oder ein eigenes erstellen. Falls Sie das mitgelieferte init-Skript nicht verwenden, erstellen Sie ein neues in *Paket*.init. Falls das von den Originalautoren mitgelieferte Init-Skript aber gut aussieht und an der richtigen Stelle installiert wird, müssen Sie trotzdem die symbolischen Links für rc* erzeugen. Dafür müssen Sie **dh_installinit** in der Datei rules mit den folgenden Zeilen überschreiben:

```
override_dh_installinit:
        dh_installinit --onlyscripts
```

Falls Sie das nicht benötigen, löschen Sie die Dateien.

5.11 `install`

Falls es Dateien gibt, die in Ihr Paket installiert werden müssen, die aber vom Standardaufruf »`make install`« nicht erfasst werden, schreiben Sie diese Dateinamen und Ziele in die Datei `install`. Sie werden dann von dh_install(1) installiert. 2 Sie sollten zunächst überprüfen, ob es nicht ein spezielleres Werkzeug gibt, das verwendet werden kann. Beispielsweise sollten Dokumente in der Datei `docs` stehen und nicht in dieser hier.

Diese Datei `install` enthält pro Zeile eine zu installierende Datei, zunächst den Namen der Datei (relativ zum obersten Verzeichnis des Paketbaus), dann ein Leerzeichen und zuletzt das Installationsverzeichnis (relativ zum Install-Verzeichnis). Ein Beispiel, wo dies benutzt werden kann, ist eine Binärdatei `src/`*`bar`*, die nicht installiert wurde. Die Datei `install` könnte so aussehen:

```
src/bar usr/bin
```

Wenn dieses Paket installiert ist, bedeutet dies, dass es einen ausführbaren Befehl `/usr/bin/`*`bar`* geben wird.

Alternativ kann die Datei `install` nur den Dateinamen ohne Installationsverzeichnis enthalten, wenn sich der relative Verzeichnispfad nicht ändert. Dieses Format wird üblicherweise für große Pakete benutzt, die das Ergebnis des Baus auf mehrere Binärpakete verteilen. Dafür verwenden diese *`Paket-1`*`.install`, *`Paket-2`*`.install` usw.

Der Befehl **dh_install** fällt darauf zurück, im Verzeichnis `debian/tmp` nach Dateien zu suchen, wenn er sie im aktuellen Verzeichnis nicht findet (oder wo auch immer Sie das Programm mit `--sourcedir` angewiesen haben, zu suchen).

5.12 *`Paket`*`.info`

Falls Ihr Paket »info«-Seiten hat, sollten Sie diese mit dh_installinfo(1) installieren, indem Sie sie in einer Datei *`Paket`*`.info` auflisten.

5.13 *`Paket`*`.links`

Falls Sie zusätzliche symbolische Links in den Paketbauverzeichnissen als Paketbetreuer erstellen müssen, sollten Sie diese mit dh_link(1) installieren, indem Sie deren komplette Pfade der Quell- und Zieldateien in einer Datei *`Paket`*`.links` aufführen.

5.14 {*`Paket`*`.|source/}lintian-overrides`

Falls die Debian-Richtlinien eine Ausnahme von einer Regel erlauben, erzeugt `lintian` eventuell eine falsche Meldung. Wenn dies der Fall ist, können Sie *`Paket`*`.lintian-overrides` oder `source/lintian-overrides` benutzen, um die Meldung zu unterdrücken. Bitte lesen Sie das Benutzerhandbuch von Lintian (`https://lintian.debian.org/manual/index.html`) und missbrauchen Sie diesen Mechanismus nicht.

`Paket``.lintian-overrides` ist für das Binärpaket *`Paket`* und wird als `usr/share/lintian/overrides/`*`Paket`* vom Befehl **dh_lintian** installiert.

`source/lintian-overrides` ist für das Quellpaket. Diese Datei wird nicht installiert.

5.15 `manpage.*`

Ihr(e) Programm(e) sollte(n) eine Handbuchseite haben. Ist keine vorhanden, sollten Sie sie erstellen. Der Befehl **dh_make** erzeugt einige Vorlagendateien für Handbuchseiten. Diese müssen für jeden Befehl kopiert und bearbeitet werden, dem eine Handbuchseite fehlt. Bitte löschen Sie alle nicht benutzten Vorlagen.

2Dies ersetzt den veralteten Befehl dh_movefiles(1), der durch die Datei `files` konfiguriert wurde.

5.15.1 `manpage.1.ex`

Handbuchseiten werden üblicherweise in nroff(1) geschrieben. Das Beispiel `manpage.1.ex` ist auch in **nroff** geschrieben. In der Handbuchseite von man(7) finden Sie eine kurze Erklärung, wie solche Dateien bearbeitet werden können.

Der endgültige Name der Handbuchseite sollte den Namen des Programms, das dokumentiert wird, angeben. Deshalb ändern wir den Namen von »`manpage`« nach »`gentoo`«. Der Dateiname muss auch eine »`.1`« als erstes Suffix erhalten, was bedeutet, dass es sich um eine Handbuchseite für einen Benutzerbefehl handelt. Vergewissern Sie sich, dass dieser Abschnitt tatsächlich richtig ist. Hier ist eine kurze Liste der Abschnitte für Handbuchseiten:

Abschnitt	Beschreibung	Hinweise
1	Dienstprogramme für Benutzer	Ausführbare Befehle oder Skripte
2	Systemaufrufe	Vom Kernel bereitgestellte Funktionen
3	Bibliotheksfunktionen	Funktionen innerhalb Systembibliotheken
4	Besondere Dateien	Normalerweise innerhalb von `/dev`
5	Dateiformate	Z.B. Format von `/etc/passwd`
6	Spiele	Spiele oder andere belanglose Programme
7	Makropakete	Wie **man**-Makros
8	Systemadministration	Programme, die typischerweise nur von Root ausgeführt werden
9	Kernelroutinen	Nicht standardisierte Aufrufe und Interna

Also bekommt `gentoo`s Handbuchseite den Namen `gentoo.1`. Falls es in den ursprünglichen Quellen keine Handbuchseite namens `gentoo.1` gibt, müssen Sie diese erstellen, indem Sie die Vorlage `manpage.1.ex` in `gentoo.1` umbenennen und sie bearbeiten. Dabei verwenden Sie die Informationen aus dem Beispiel und die Dokumentation des Originalautors.

Sie können auch den Befehl **help2man** benutzen, um eine Handbuchseite aus der Ausgabe des Programms mit den Optionen »`--help`« und »`--version`« zu erzeugen. 3

5.15.2 `manpage.sgml.ex`

Falls Sie es andererseits bevorzugen, in SGML anstatt **nroff** zu schreiben, können Sie die Vorlage `manpage.sgml.ex` benutzen. Dann müssen Sie Folgendes tun:

- Benennen Sie die Datei um, beispielsweise in `gentoo.sgml`.
- Installieren Sie das Paket `docbook-to-man`
- Fügen Sie `docbook-to-man` der Zeile `Build-Depends` in der Datei `control` hinzu.
- Fügen Sie das Target `override_dh_auto_build` in Ihrer Datei `rules` hinzu:

```
override_dh_auto_build:
        docbook-to-man debian/gentoo.sgml > debian/gentoo.1
        dh_auto_build
```

5.15.3 `manpage.xml.ex`

Falls Sie XML gegenüber SGML bevorzugen, können Sie die Vorlage `manpage.xml.ex` benutzen. Dann müssen Sie Folgendes tun:

- Umbenennen der Quelldatei in etwas wie `gentoo.1.xml`
- Installieren Sie das Paket `docbook-xsl` und einen XSLT-Prozessor wie `xsltproc` (empfohlen).

3Beachten Sie, dass die Platzhalter-Handbuchseite von **help2man** behaupten wird, dass detailliertere Informationen im **info**-System vorhanden seien. Falls dem Befehl eine Info-Seite fehlt, sollten Sie die von **help2man** erstellte Handbuchseite manuell bearbeiten.

- Fügen Sie `docbook-xsl`, `docbook-xml` und `xsltproc` der Zeile `Build-Depends` in der Datei `control` hinzu
- Fügen Sie das Target `override_dh_auto_build` in Ihrer Datei `rules` hinzu:

```
override_dh_auto_build:
        xsltproc --nonet \
         --param make.year.ranges 1 \
         --param make.single.year.ranges 1 \
         --param man.charmap.use.subset 0 \
         -o debian/ \
http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\
        debian/gentoo.1.xml
        dh_auto_build
```

5.16 *Paket*`.manpages`

Falls Ihr Paket Handbuchseiten hat, sollten Sie sie mit dh_installman(1) installieren, indem Sie sie in einer Dateie *Paket*`.manpages` auflisten.

Um `docs/gentoo.1` als Handbuchseite für das Paket `gentoo` zu installieren, erstellen Sie folgendermaßen eine Datei `gentoo.manpages`:

```
docs/gentoo.1
```

5.17 `NEWS`

Der Befehl dh_installchangelogs(1) installiert diese Datei.

5.18 `{post|pre}{inst|rm}`

Die Dateien `postinst`, `preinst`, `postrm` und `prerm` [4] werden *Betreuerskripte* (engl. »maintainer scripts«) genannt. Diese Skripte werden für die Steuerung des Pakets verwendet und von **dpkg** aufgerufen, wenn Ihr Paket installiert, aktualisiert oder entfernt wird.

Als neuer Betreuer sollten Sie das manuelle Bearbeiten dieser Betreuerskripte vermeiden, da sie problematisch sind. Für weitere Informationen lesen Sie in dem Debian Policy Manual, Kapitel 6 »Package maintainer scripts and installation procedure« (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) und sehen Sie sich die Beispieldateien an, die von **dh_make** erstellt wurden.

Falls Sie nicht auf mich gehört haben und eigene Betreuerskripte für ein Paket erstellt haben, müssen Sie sicherstellen, dass Sie sie nicht nur für den Aufruf mit **install** und **upgrade** getestet haben, sondern auch für **remove** und **purge**.

Upgrades auf die neue Version sollen ohne Rückfragen geschehen und keine Störungen oder Unterbrechungen zur Folge haben (Benutzer, die das Paket verwenden, sollen nicht bemerken, dass ein Upgrade durchgeführt wurde. Sie sollen nur feststellen, dass alte Fehler behoben wurden und das Paket eventuell neue Fähigkeiten hat).

Wenn das Upgrade nicht ohne Störung ablaufen kann (beispielsweise weil Konfigurationsdateien über verschiedene Home-Verzeichnisse verteilt sind, die einen vollkommen anderen Aufbau haben), können Sie als letzte Möglichkeit für das Paket eine sichere Voreinstellung wählen (beispielsweise den Service deaktivieren) und die von den Richtlinien verlangte ausführliche Information (`README.Debian` und `NEWS.Debian`) bereitstellen. Belästigen Sie den Benutzer nicht mit **debconf**-Hinweisen, die von den Betreuerskripten bei Upgrades angezeigt werden.

[4] Trotz der hier verwendeten Abkürzungssyntax `{pre,post}{inst,rm}` von **bash**, um die Dateinamen zu bezeichnen, sollten Sie reine POSIX-Syntax für diese Betreuerskripte verwenden, um kompatibel zu der Verwendung von **dash** als System-Shell zu bleiben.

Das Paket `ucf` stellt eine ähnliche Infrastruktur wie für Dateien zur Verfügung, die als *Conffile* gekennzeichnet wurden. Damit bleiben von Benutzern durchgeführte Änderungen an Dateien erhalten, auch wenn diese Dateien nicht als *Conffiles* gekennzeichnet werden können. Beispiele hierfür sind Dateien, die von den Betreuerskripten verwaltet werden. Hierdurch sollen Probleme in der Behandlung dieser Dateien minimiert werden.

Diese Betreuerskripte gehören zu den Errungenschaften von Debian, die erklären, **warum jemand Debian verwendet**. Sie müssen sehr darauf achten, diese Skripte nicht zu einem Grund für Ärger werden zu lassen.

5.19 *Paket*`.symbols`

Die Paketierung einer Bibliothek ist für einen neuen Betreuer nicht leicht und sollte vermieden werden. Hat Ihr Paket allerdings Bibliotheken, dann sollten Sie eine Datei `debian/`*`Paket`*`.symbols` haben. Lesen Sie Abschnitt A.2.

5.20 `TODO`

Der Befehl dh_installdocs(1) installiert diese Datei.

5.21 `watch`

Das Format der Datei `watch` ist in der Handbuchseite von uscan(1) dokumentiert. Die Datei `watch` konfiguriert das Programm **uscan** (aus dem Paket `devscripts`), um die Site zu überwachen, von der Sie die Originalquellen bezogen haben. Dies wird außerdem von dem Dienst Debian Package Tracker (https://tracker.debian.org/) benutzt.

Hier sind seine Inhalte:

```
# watch control file for uscan
version=3
http://sf.net/gentoo/gentoo-(.+)\.tar\.gz debian uupdate
```

Normalerweise würde mit einer `watch`-Datei die URL unter »`http://sf.net/gentoo`« heruntergeladen und nach Links im Format »`<a href=...>`« durchsucht. Der Basisname (nur der Teil nach dem letzten »`/`«) jeder verlinkten URL wird mit dem regulären Perl-Ausdruck (siehe perlre(1)) »`gentoo-(.+)\.tar\.gz`« verglichen. Von allen Dateien, auf die der Ausdruck passt, wird die Datei mit der höchsten Versionsnummer heruntergeladen. Das Programm **uupdate** wird dann ausgeführt, um daraus den aktualisierten Quelltextbaum zu erzeugen.

Dies stimmt zwar für andere Websites, doch das Herunterladen von SourceForge unter http://sf.net ist eine Ausnahme. Wenn die Datei `watch` eine URL enthält, auf die der reguläre Perl-Ausdruck »`^http://sf\.net/`« passt, wird diese vom Programm **uscan** durch »`http://qa.debian.org/watch/sf.php/`« ersetzt und erst dann die nachfolgenden Regeln angewendet. Die URL-Umleitungsfunktion unter http://qa.debian.org/ ist entwickelt worden, um einen stabilen Umleitungsservice für jedes `watch`-Muster der Form »`http://sf.net/`*`Projekt/tar-Name`*`-(.+)\.tar\.gz`« in der Datei `watch` bereitzustellen. Hierdurch wird das Problem gelöst, dass SourceForge-URLs regelmäßig geändert werden.

Falls die Originalautoren eine kryptographische Signatur des Tarballs bereitstellen, wird empfohlen, seine Authentizität mittels der Option `pgpsigurlmangle` zu prüfen, wie dies in uscan(1) beschrieben ist.

5.22 `source/format`

In der Datei `debian/source/format` soll eine einzelne Zeile stehen, in der das gewünschte Format für das Quellpaket angegeben wird (lesen Sie dpkg-source(1) für eine ausführliche Liste). Nach `Squeeze` sollte dort entweder:

- `3.0 (native)` für native Debian-Pakete oder

• `3.0 (quilt)` für alles andere stehen.

Das neuere Quellformat `3.0 (quilt)` zeichnet Änderungen in einer Patchserie im Verzeichnis `debian/patches` für **quilt** auf. Diese Änderungen werden dann während des Entpackens des Quellpakets automatisch angewendet. [5] Die Debian-spezifischen Änderungen werden einfach in einem Archiv namens `debian.tar.gz` gespeichert, das alle Dateien im Verzeichnis `debian` enthält. Mit diesem neuen Format können binäre Dateien wie PNG-Icons vom Paketbetreuer eingebunden werden, ohne Tricks anwenden zu müssen. [6]

Wenn **dpkg-source** ein Quellpaket im Quellformat `3.0 (quilt)` entpackt, werden automatisch alle Patches angewendet, die in `debian/patches/series` aufgeführt sind. Sie können das Anwenden der Patches nach dem Entpacken verhindern, indem Sie die Option `--skip-patches` benutzen.

5.23 source/local-options

Wenn Sie die Paketierungsarbeiten für Debian in einem Versionskontrollsystem verwalten wollen, erstellen Sie üblicherweise einen Zweig (z. B. `upstream`), in dem Sie die Quellen des Originalautors verfolgen und einen weiteren Zweig (z. B. üblicherweise `master` für Git), in dem Sie das Debian-Paket verfolgen. Für letzteres wollen Sie sicherlich die unveränderten Ursprungsquellen zusammen mit Ihren `debian/*`-Dateien für das Paketieren haben, um das Zusammenführen von neuen Ursprungsquellen zu vereinfachen.

Nachdem Sie ein Paket gebaut haben, bleiben die Patches im Quelltext normalerweise erhalten. Sie müssen sie manuell entfernen, indem Sie »`dquilt pop -a`« aufrufen, bevor Sie in den `master`-Zweig einchecken können. Sie können dies automatisieren, indem Sie die optionale Datei `debian/source/local-options` hinzufügen und dort »`unapply-patches`« hineinschreiben. Diese Datei wird nicht in das erzeugte Quellpaket aufgenommen und verändert nur das lokale Bauverhalten. Diese Datei kann auch »`abort-on-upstream-changes`« enthalten (siehe dpkg-source(1)).

```
unapply-patches
abort-on-upstream-changes
```

5.24 source/local-options

Die automatisch erstellten Dateien im Quellbaum können für das Paketieren recht störend wirken, da sie unbedeutende große Patch-Dateien hervorrufen. Es gibt angepasste Module wie **dh_autoreconf**, um dieses Problem zu vereinfachen. Dies wird in Abschnitt 4.4.3 beschrieben.

Sie können dem Optionsargument `--extend-diff-ignore` von dpkg-source(1) einen regulären Perl-Ausdruck übergeben, um Änderungen, die an den automatisch erstellten Dateien beim Erstellen des Quellpakets vorgenommen wurden, zu ignorieren.

Als allgemeine Lösung, um dieses Problem der automatisch erstellten Dateien zu adressieren, können Sie so ein Optionsargument an **dpkg-source** in der Datei `source/options` des Quellpakets speichern. Folgendes wird die Erstellung von Patch-Dateien für `config.sub`, `config.guess` und `Makefile` überspringen:

```
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

5.25 patches/*

Das alte Quellformat `1.0` erzeugte eine einzelne große `diff.gz`-Datei, in der sowohl die Paketbetreuungsdateien unter `debian` als auch Patches für den Quelltext enthalten waren. So ein Paket ist etwas schwierig zu untersuchen und zu verstehen, wenn später der Quelltext geändert werden soll. Das ist nicht so schön.

[5] Siehe DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) für eine Zusammenfassung zum Wechsel auf die neuen Quellformate `3.0 (quilt)` und `3.0 (native)`.

[6] Tatsächlich unterstützt dieses neue Format sogar mehrere ursprüngliche Tarbälle und mehr Kompressionsmethoden. Dies würde in diesem Dokument aber zu weit führen.

Das neuere Quellformat `3.0 (quilt)` speichert Patches in `debian/patches/*`-Dateien unter Verwendung des Befehls **quilt** ab. Diese Patches sowie andere Paketdaten bleiben alle innerhalb des Verzeichnisses `debian` und werden als `debian.tar.gz`-Datei gepackt. Da der Befehl **dpkg-source** in `3.0 (quilt)`-Quellen die Patch-Daten im **quilt**-Format verarbeiten kann, ohne auf das Paket `quilt` zurückzugreifen, wird keine `Build-Depends` für `quilt` benötigt. [7]

Der Befehl **quilt** wird in der Handbuchseite von quilt(1) erklärt. Er zeichnet Änderungen an den Quellen als Stapel von `-p1`-Patch-Dateien im Verzeichnis `debian/patches` auf, dadurch bleibt der Quelltextbaum außerhalb des `debian`-Verzeichnisses unangetastet. Die Reihenfolge der Patches wird in der Datei `debian/patches/series` gespeichert. Sie können die Patches leicht anwenden (=push), entfernen (=pop) und aktualisieren. [8]

Für den Abschnitt Kapitel 3 haben wir drei Patches in `debian/patches` erstellt.

Da die Debian-Patches in `debian/patches` enthalten sind, stellen Sie bitte sicher, dass der Befehl **dquilt** korrekt eingerichtet ist, so wie in Abschnitt 3.1 beschrieben.

Wenn später jemand (Sie selbst eingeschlossen) einen Patch namens *`foo`*`.patch` für die Quellen erstellt, ist die Veränderung eines `3.0 (quilt)`-Quellpakets recht einfach:

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../foo.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
... Patch beschreiben
```

Die Patches, die in dem neuen Quellformat `3.0 (quilt)` gespeichert werden, müssen frei von Ungenauigkeiten (*fuzz*) sein. Sie können dies mit »`dquilt pop -a; while dquilt push; do dquilt refresh; done`« sicherstellen.

[7] Es wurden verschiedene Methoden für die Organisation der Patches in Debian-Paketen vorgeschlagen und auch angewendet. Das **quilt**-System ist das bevorzugteste Organisationssystem. Weitere Systeme sind u.A. **dpatch**, **dbs** und **cdbs**. Viele von diesen Systemen speichern solche Patches in `debian/patches/*`-Dateien ab.

[8] Falls Sie einen Sponsor darum bitten, Ihr Paket hochzuladen, ist diese Art der klaren Aufteilung und Dokumentation Ihrer Änderungen sehr wichtig, um die Durchsicht Ihres Pakets durch den Sponsor zu beschleunigen.

Kapitel 6

Bau des Pakets

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Nun sollten wir soweit sein, das Paket zu bauen.

6.1 Kompletter (Neu-)Bau

Um ein Paket ordnungsgemäß komplett (neu) zu bauen, müssen Sie Folgendes installiert haben:

- Das Paket `build-essential`,
- die im Feld `Build-Depends` aufgelisteten Pakete (siehe Abschnitt 4.1) und
- die im Feld `Build-Depends-Indep` aufgelisteten Pakete (siehe Abschnitt 4.1).

Dann führen Sie den folgenden Befehl im Quellverzeichnis des Programms aus:

```
$ dpkg-buildpackage -us -uc
```

Hiermit wird alles für Sie erledigt, um vollständige Binärpakete und Quellpakete zu erstellen. Im Einzelnen:

- Aufräumen des Quellverzeichnisbaums (»`debian/rules clean`«),
- Bauen des Quellpakets (»`dpkg-source -b`«),
- Bauen des Programms (»`debian/rules build`«),
- Bauen der Binärpakete (»`fakeroot debian/rules binary`«),
- Erstellen der Datei `.dsc`,
- Erstellen der Datei `.changes`, mittels **dpkg-genchanges**.

Falls das Ergebnis des Baus den Erwartungen entspricht, signieren Sie die Dateien `.dsc` und `.changes` mit Ihrem privaten GPG-Schlüssel mit dem Befehl **debsign**. Sie müssen Ihre geheime Passphrase zweimal eingeben. 1

Für nicht native Debian-Pakete, z.B. `gentoo`, werden Sie die folgenden Dateien im übergeordneten Verzeichnis (`~/gentoo`) nach dem Paketbau sehen:

1Dieser GPG-Schlüssel muss von einem Debian-Entwickler signiert worden sein, um mit dem Vertrauensnetz verbunden zu sein, und muss im Debian-Schlüsselring (http://keyring.debian.org) registriert sein. Dies ermöglicht es, dass Ihre hochgeladenen Pakete im Debian-Archiv akzeptiert werden. Lesen Sie Creating a new GPG key (http://keyring.debian.org/creating-key.html) und Debian Wiki on Keysigning (http://wiki.debian.org/Keysigning) .

- `gentoo_0.9.12.orig.tar.gz`

  Dies ist der ursprüngliche Quelltext-Tarball der Originalautoren, lediglich umbenannt, um dem Debian-Standard zu genügen. Beachten Sie, dass diese Umbenennung beim initialen Aufruf von »`dh_make -f ../gentoo-0.9.12.tar.gz`« erfolgt ist.

- `gentoo_0.9.12-1.dsc`

  Dies ist eine Zusammenfassung des Inhalts des Quellcode-Pakets. Diese Datei wird aus Ihrer Datei `control` erzeugt und für das Entpacken des Quellcodes mittels dpkg-source(1) benötigt.

- `gentoo_0.9.12-1.debian.tar.gz`

  Dieser komprimierte Tarball enthält die Dateien aus Ihrem `debian`-Verzeichnis. Jede einzelne Änderung, die Sie am ursprünglichen Code vorgenommen haben, wird als **quilt**-Patch in `debian/patches` gespeichert.

  Wenn jemand Ihr Paket von Grund auf neu bauen will, kann er dafür einfach die drei oben genannten Dateien verwenden. Das Verfahren des Auspackens ist trivial: Kopieren Sie einfach die drei Dateien in ein Verzeichnis und führen Sie »`dpkg-source -x gentoo_0.9.12-1.dsc`« aus. 2

- `gentoo_0.9.12-1_i386.deb`

  Das ist Ihr fertiges Binärpaket. Sie können es mit **dpkg** installieren und wieder entfernen wie jedes andere Paket auch.

- `gentoo_0.9.12-1_i386.changes`

  Diese Datei beschreibt alle Änderungen in dieser Paket-Revision. Die Verwaltungsprogramme für Debians FTP-Archive benötigen diese Datei zur Installation der Binär- und Quellcodepakete im FTP-Archiv. Sie wird zum Teil aus den Dateien `changelog` und `.dsc` erzeugt.

  Wenn Sie weiter an dem Paket arbeiten, wird sich sein Verhalten ändern und neue Funktionen werden hinzugefügt. Leute, die Ihr Paket herunterladen, können sich diese Datei ansehen und feststellen, was sich geändert hat. Die Verwaltungsprogramme für das Debian-Archiv werden den Inhalt dieser Datei auch an die Mailingliste debian-devel-changes@lists.debian.org (http://lists.debian.org/debian-devel-changes/) schicken.

Die Dateien `gentoo_0.9.12-1.dsc` und `gentoo_0.9.12-1_i386.changes` müssen unter Verwendung des Befehls **debsign** mit Ihrem privaten GPG-Schlüssel, der sich im Verzeichnis `~/.gnupg/` befindet, signiert werden, bevor sie zum Debian-FTP-Archiv hochgeladen werden. Die GPG-Signatur stellt unter Verwendung Ihres öffentlichen GPG-Schlüssels den Nachweis bereit, dass diese Dateien tatsächlich von Ihnen sind.

Mit dem folgenden Eintrag in die Datei `~/.devscripts` kann der Befehl **debsign** dazu veranlasst werden, mit Ihrer angegebenen geheimen Schlüsselkennung zu signieren (was gut fürs Sponsern ist):

```
DEBSIGN_KEYID=Ihre_GPG-Schlüsselkennung
```

Die langen Zahlenreihen in den Dateien `.dsc` und `.changes` sind SHA1/SHA256-Prüfsummen der aufgelisteten Dateien. Jeder, der Ihr Paket herunterlädt, kann die enthaltenen Dateien mit sha1sum(1) oder sha256sum(1) testen. Wenn die Zahlen nicht übereinstimmen, weiß er, die geprüfte Datei ist beschädigt oder manipuliert.

6.2 Autobuilder

Debian unterstützt viele Portierungen (http://www.debian.org/ports/) mit dem Autobuilder-Netz (http://www.debian.org/devel/buildd/) , das **buildd**-Daemons auf vielen verschiedenen Rechnerarchitekturen ausführt. Obwohl Sie dies nicht selbst durchführen müssen, sollte Ihnen bewusst sein, was mit Ihren Paketen passiert. Lassen Sie uns einen kurzen Blick darauf werfen, wie Ihre Pakete vom Autobuilder-Netz für verschiedene Architekturen neu gebaut werden. 3

Für Pakete mit »`Architecture: any`« führt das Autobuilder-System eine Neuübersetzung durch. Es stellt folgende Installationen sicher:

2Sie können das Anwenden der **quilt**-Patches im `3.0 (quilt)`-Quellformat am Ende des Auspackens verhindern, indem Sie die Option `--skip-patches` benutzen. Alternativ können Sie »`dquilt pop -a`« nach dem normalen Auspacken aufrufen.

3Das tatsächliche Autobuilder-System besteht aus einem wesentlich komplizierteren Schema als dem hier dargestellten. Diese Details führen aber hier zu weit.

- des Pakets `build-essential` und
- der Pakete, die im Feld `Build-Depends` (siehe Abschnitt 4.1) aufgeführt sind.

Dann führt es den folgenden Befehl im Quellverzeichnis aus:

```
$ dpkg-buildpackage -B
```

Hiermit wird alles erledigt, um ein architekturabhängiges Binärpaket für eine andere Architektur zu erstellen. Im Einzelnen:

- Aufräumen des Quellverzeichnisbaums (»`debian/rules clean`«),
- Bauen des Programms (»`debian/rules build`«),
- Bauen der architekturabhängigen Binärpakete (»`fakeroot debian/rules binary-arch`«),
- Signieren der `.dsc`-Quelldatei mit **gpg**,
- Erstellen und Signieren der für das Hochladen notwendigen `.changes`-Datei mit **dpkg-genchanges** und **gpg**.

Das ist der Grund, weshalb Sie Ihr Paket auch für andere Architekturen sehen.

Obwohl Pakete, die im Feld `Build-Depends-Indep` aufgeführt sind, installiert sein müssen, wenn wir das Paket bauen (siehe Abschnitt 6.1), müssen diese auf dem Autobuilder-System nicht installiert sein, weil es nur architekturabhängige Binärpakete baut. [4] Diese Unterscheidung zwischen dem normalen Bau und der Situation bei dem Autobuilder-Ablauf entscheidet darüber, ob Sie solche erforderlichen Pakete im Feld `Build-Depends` oder `Build-Depends-Indep` der Datei `debian/control` auflisten (siehe Abschnitt 4.1).

6.3 Der Befehl debuild

Sie können den Prozess des Paketbauens rund um die Ausführung des Befehls **dpkg-buildpackage** weiter mit dem Befehl **debuild** automatisieren. Siehe debuild(1).

Der Befehl **debuild** führt den Befehl **lintian** aus, um nach dem Bau des Debian-Pakets eine statische Prüfungen durchzuführen. Der Befehl **lintian** kann mit dem folgenden Eintrag in der Datei `~/.devscripts` angepasst werden:

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

Jetzt kann das Säubern des Quellverzeichnisses und Neubauen des Pakets unter Ihrem Benutzerkonto einfach so durchgeführt werden:

```
$ debuild
```

Sie können das Quellverzeichnis einfach so säubern:

```
$ debuild -- clean
```

[4] Im Gegensatz zum `pbuilder`-Paket erzwingt die **chroot**-Umgebung im Paket `sbuild`, die vom Autobuilder-System verwendet wird, keine Verwendung einer Minimalumgebung. Daher können übriggebliebene Pakete installiert sein.

6.4 Das Paket pbuilder

Um die Abhängigkeiten für den Bau in einer sauberen Umgebung (**chroot**) zu überprüfen, ist das Paket pbuilder sehr gut geeignet. [5] Es stellt sicher, dass das Paket aus den Quellen auf einem Sid-Autobuilder für verschiedene Architekturen gebaut werden kann. Weiterhin wird hierdurch ein schwerwiegender FTBFS-Fehler (»Fails To Build From Source«, kann nicht aus den Quellen gebaut werden) verhindert, der immer als release-kritisch (RC) angesehen wird. [6]

Lassen Sie uns das Paket pbuilder folgendermaßen anpassen:

- Machen Sie das Verzeichnis /var/cache/pbuilder/result für Ihr Benutzerkonto schreibbar.
- Erstellen Sie ein Verzeichnis, z.B. */var/cache/pbuilder/hooks*, das ebenfalls für den Benutzer schreibbar ist, so dass Hook-Skripts dort abgelegt werden können.
- Konfigurieren Sie die Datei ~/.pbuilderrc oder /etc/pbuilderrc, so dass sie folgendes enthält:

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

Lassen Sie uns zuerst das lokale pbuilder-**chroot**-System wie folgt initialisieren:

```
$ sudo pbuilder create
```

Falls Sie ein Quellpaket schon komplett fertiggestellt haben, führen Sie die folgenden Befehle in dem Verzeichnis aus, in dem die Dateien *foo*.orig.tar.gz, *foo*.debian.tar.gz und *foo*.dsc liegen, um das lokale pbuilder-**chroot**-System zu aktualisieren und Binärpakete darin zu bauen:

```
$ sudo pbuilder --update
$ sudo pbuilder --build foo_Version.dsc
```

Die neu erstellten Pakete ohne GPG-Signaturen werden unter /var/cache/pbuilder/result/ mit einem nicht-root-Eigentümer abgelegt.

Die GPG-Signaturen für die Dateien .dsc und .changes können wie folgt erstellt werden:

```
$ cd /var/cache/pbuilder/result/
$ debsign foo_Version_Arch.changes
```

Falls Sie bereits ein aktualisiertes Quellverzeichnis haben, jedoch das zugehörige Quellpaket nicht erstellt haben, führen Sie stattdessen die folgenden Befehle in dem Quellverzeichnis aus, in dem das Verzeichnis debian enthalten ist:

```
$ sudo pbuilder --update
$ pdebuild
```

Sie können sich in der erstellten **chroot**-Umgebung anmelden, indem Sie den Befehl »pbuilder --login --save-after-logi verwenden und diese dann so einrichten, wie Sie wollen. Diese Umgebung kann gespeichert werden, indem die Shell-Eingabeaufforderung mittels ^D (Steuerung-D) verlassen wird.

Die aktuelle Version des Programms **lintian** kann in der chroot-Umgebung ausgeführt werden, indem das Hook-Skript */var/cache/pbuilder/hooks/*B90lintian wie folgt eingerichtet wird: [7]

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --allow-downgrades install "$@"
```

[5] Weil das Paket pbuilder ständig weiterentwickelt wird, sollten Sie die tatsächliche Konfiguration herausfinden, indem Sie die aktuelle offizielle Dokumentation durchlesen.

[6] Lesen Sie http://buildd.debian.org/ für weitere Informationen über das automatische Bauen von Debian-Paketen.

[7] Dies setzt HOOKDIR=/var/cache/pbuilder/hooks voraus. Sie finden viele Beispiele für Hook-Skripte im Verzeichnis /usr/share/doc/pbuilder/examples.

```
        }
install_packages lintian
echo "+++ Lintian-Ausgabe +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes" - pbuilder
# Verwenden Sie diese Version, falls Lintian beim Bau nicht fehlschlagen soll
#su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" - pbuilder
echo "+++ Ende der Lintian-Ausgabe +++"
```

Sie müssen Zugriff auf die neueste `Sid`-Umgebung haben, damit Sie Pakete korrekt für `Sid` bauen können. Allerdings kann es in `Sid` immer wieder mal Probleme geben, so dass Sie wahrscheinlich nicht Ihr gesamtes System umstellen wollen. Das Paket `pbuilder` bietet einen Ausweg für diese Situation.

Es kann vorkommen, dass Sie Ihre Pakete für `Stable` aktualisieren müssen, nachdem sie veröffentlicht worden sind, beispielsweise über `stable-proposed-updates`, `stable/updates` usw. [8] In einem solchen Fall sollten Sie Ihre Pakete zügig aktualisieren, die Ausrede, dass Sie ein `Sid`-System verwenden, ist nicht akzeptabel. Das Paket `pbuilder` kann Ihnen dabei helfen, auf nahezu jede Debian-basierte Distribution derselben Architektur zuzugreifen.

Siehe http://www.netfort.gr.jp/~dancer/software/pbuilder.html, pdebuild(1), pbuilderrc(5) und pbuilder(8).

6.5 Der Befehl git-buildpackage und ähnliche

Falls der Autor des ursprünglichen Programms ein Quelltext-Verwaltungsprogramm (VCS) [9] zur Betreuung des Codes einsetzt, sollten Sie darüber nachdenken, dies ebenfalls zu tun. Damit wird das Zusammenführen und Herauspicken von Patches des Originalautors wesentlich einfacher. Es gibt verschiedene Pakete, die spezialisierte Wrapper-Skripte enthalten, die das Bauen eines Debian-Pakets mit dem jeweiligen VCS erleichtern.

- `git-buildpackage`: Eine Suite zur Unterstützung von Debian-Paketen in Git-Depots.
- `svn-buildpackage`: Hilfsprogramme zur Betreuung von Debian-Paketen mit Subversion.
- `cvs-buildpackage`: Eine Sammlung von Skripten für Debian-Pakete in CVS-Quelltextbäumen.

Die Verwendung von `git-buildpackage` ist bei Debian-Entwicklern recht beliebt, um Debian-Pakete mit dem Git-Server auf alioth.debian.org (http://alioth.debian.org/) zu verwalten. 10 Dieses Paket bietet viele Befehle, um Paketierungsaktivitäten zu *automatisieren*:

- gbp-import-dsc(1): Import eines bestehenden Debian-Pakets in ein Git-Depot.
- gbp-import-orig(1): Import eines neuen Tars der Originalautoren in ein Git-Depot.
- gbp-dch(1): Erstellung des Debian-Changelogs aus Git-Commit-Nachrichten.
- git-buildpackage(1): Bau von Debian-Paketen aus einem Git-Depot.
- git-pbuilder(1): Bau von Debian-Paketen aus einem Git-Depot mit **pbuilder**/**cowbuilder**.

Diese Befehle verwenden drei Zweige, um die Paktierungsaktivitäten nachzuverfolgen:

- `main` für den Debian-Paketierungsquellbaum
- `upstream` für den Quellbaum der Originalautoren
- `pristine-tar` für Tarbälle der Originalautoren, die mit der Option `--pristine-tar` erstellt wurden.11

Sie können `git-buildpackage` mit `~/.gbp.conf` konfigurieren. Lesen Sie gbp.conf(5). 12

[8] Es gibt für Aktualisierungen Ihrer `Stable`-Pakete einige Einschränkungen.

[9] Lesen Sie Version control systems (http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) für mehr Informationen.

10 Das Dokument Debian wiki Alioth (http://wiki.debian.org/Alioth) dokumentiert, wie der Dienst alioth.debian.org (http://alioth.debian.org/) verwandt wird.

11 Die Option `--pristine-tar` ruft den Befehl **pristine-tar** auf, der eine exakte Kopie des ursprünglichen Tarballs erneut erstellt und dabei nur eine kleine binäre Deltadatei und den Inhalt des Tarballs verwendet, der typischerweise im Zweig `upstream` des VCS gehalten wird.

12 Hier sind einige Quellen im Web für fortgeschrittene Anwender.

6.6 Schneller Neubau

Bei einem großen Paket wollen Sie bestimmt nicht alles nach jeder kleiner Änderung in `debian/rules` neu kompilieren. Für Testzwecke können Sie folgendermaßen eine `.deb`-Datei erstellen, ohne alle Schritte durchzumachen[13]:

```
$ fakeroot debian/rules binary
```

Oder noch einfacher, falls Sie nur feststellen wollen, ob das Paket gebaut werden kann oder nicht:

```
$ fakeroot debian/rules build
```

Wenn Sie mit Ihren Anpassungen fertig sind, vergessen Sie nicht, das Paket gemäß der korrekten Prozedur neu zu bauen. Sie werden `.deb`-Dateien, die auf diese Weise gebaut wurden, nicht korrekt hochladen können.

6.7 Befehlshierarchie

Dies ist eine kurze Zusammenfassung, wie die vielen Befehle zum Paketbau in der Befehlshierarchie zusammenpassen. Es gibt viele Wege, das gleiche zu erledigen.

- `debian/rules` = Betreuerskript für den Paketbau
- **dpkg-buildpackage** = Kern des Paketbauwerkzeuges
- **debuild** = **dpkg-buildpackage** + **lintian** (unter bereinigten Umgebungsvariablen bauen)
- **pbuilder** = Kern des Debian-Chroot-Umgebungswerkzeuges
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (in der Chroot bauen)
- **cowbuilder** = Ausführung von **pbuilder** beschleunigen
- **git-pbuilder** = die einfach zu benutzende Syntax für **pdebuild** (von **gbp buildpackage** verwandt)
- **gbp** = Debian-Quellen in einem Git-Depot verwalten
- **gbp buildpackage** = **pbuilder** + **dpkg-buildpackage** + **gbp**

Obwohl die Verwendung von abstrakteren Befehlen wie **gbp buildpackage** und **pbuilder** eine perfekte Paketbauumgebung sicherstellen, ist es essenziell, zu verstehen, wie die systemnahen Befehle wie `debian/rules` und **dpkg-buildpackage** unter ihnen ausgeführt werden.

- »Building Debian Packages with git-buildpackage« (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`)
- »debian packages in git (https://honk.sigxcpu.org/piki/development/debian_packages_in_git/) «
- »Using Git for Debian Packaging (http://www.eyrie.org/~eagle/notes/debian/git.html) «
- »git-dpm: Debian packages in Git manager (http://git-dpm.alioth.debian.org/) «

13Umgebungsvariablen, die normalerweise auf vernünftige Werte gesetzt sind, werden bei dieser Methode nicht eingerichtet. Erstellen Sie niemals echte Pakete, die hochgeladen werden sollen, mit dieser **schnellen** Methode.

Kapitel 7

Überprüfen des Pakets auf Fehler

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Es gibt einige Techniken, die Sie zum Überprüfen eines Pakets auf Fehler vor dem Upload in das öffentliche Archiv wissen sollten.

Es empfiehlt sich auch, das Testen auf einem von Ihrer Maschine verschiedenen Rechner durchzuführen. Sie müssen genau auf alle Warnungen und Fehler für alle hier beschriebenen Tests achten.

7.1 Merkwürdige Änderungen

Falls Sie nach dem Bau Ihres nicht nativen Debian-Pakets im Format `3.0 (quilt)` einen neuen, automatisch erstellen Patch wie `debian-changes-*` im Verzeichnis `debian/patches` finden, besteht die Möglichkeit, dass Sie Dateien versehentlich geändert haben oder das Bauskript die Quellen der Originalautoren verändert hat. Falls es Ihr Fehler ist, korrigieren Sie ihn. Falls er vom Bauskript ausgelöst wurde, korrigieren Sie das dahinterliegende Problem mit **dh-autoreconf** wie in Abschnitt 4.4.3 oder umgehen Sie das Problem mit `source/options` wie in Abschnitt 5.24.

7.2 Überprüfen einer Paketinstallation

Sie müssen überprüfen, ob Ihr Paket ohne Probleme installiert werden kann. Der Befehl debi(1) hilft Ihnen bei der Testinstallation aller erstellten Binärpakete.

```
$ sudo debi gentoo_0.9.12-1_i386.changes
```

Um Installationsprobleme auf verschiedenen Systemen zu vermeiden, müssen Sie sicherstellen, dass es keine Dateinamenskonflikte mit anderen existierenden Paketen gibt, wobei Sie die vom Debian-Archiv heruntergeladene Datei `Contents-`*`i386`* verwenden. Der Befehl **apt-file** kann für diese Aufgabe praktisch sein. Falls es Kollisionen gibt, kümmern Sie sich um dieses echte Problem, indem Sie die Datei umbenennen, eine gemeinsame Datei in ein separates Paket verschieben, von dem verschiedene Pakete abhängen können, in Absprache mit den Betreuern anderer betroffener Pakete den Alternatives-Mechanismus nutzen (siehe update-alternatives(1)) oder indem Sie eine `Conflicts`-Beziehung in der Datei `debian/control` festlegen.

7.3 Überprüfen der Betreuerskripte eines Pakets

Alle Betreuerskripte (das heißt, die Dateien `preinst`, `prerm`, `postinst` und `postrm`) sind schwer korrekt zu schreiben, falls Sie nicht vom Programm `debhelper` automatisch erstellt werden. Verwenden Sie sie daher als neuer Betreuer nicht (siehe Abschnitt 5.18).

Falls das Paket nicht triviale Betreuerskripte verwendet, testen Sie nicht nur die Installation, sondern auch das Entfernen, das vollständige Entfernen und den Upgrade-Prozess. Viele Betreuerskriptfehler zeigen sich, wenn Pakete entfernt oder endgültig entfernt werden. Verwenden Sie für die Tests den Befehl **dpkg** wie folgt:

```
$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_Version-Revision_i386.deb
```

Dies sollte in einem Ablauf wie dem Folgenden passieren:

- Installieren Sie die vorhergehende Version (falls notwendig)
- Führen Sie ein Upgrade von der vorhergehenden Version durch
- Führen Sie ein Downgrade auf eine vorherige Version durch (optional)
- Entfernen Sie es vollständig
- Installieren Sie das neue Paket
- Entfernen Sie es
- Installieren Sie es wieder
- Entfernen Sie es vollständig

Falls dies Ihr erstes Paket ist, sollten Sie ein Pseudo-Paket mit einer anderen Version erstellen, um Ihr Paket vorab zu testen und zukünftige Probleme zu vermeiden.

Behalten Sie im Hinterkopf, dass die Benutzer ein Upgrade von der Version, die in der letzten Debian-Veröffentlichung enthalten war, durchführen werden, falls es schon mal veröffentlicht wurde. Denken Sie daran, auch Upgrades von dieser Version zu prüfen.

Obwohl ein Downgrade offiziell nicht unterstützt wird, ist es eine nette Geste, dies dennoch zu unterstützen.

7.4 `lintian` verwenden

Führen Sie lintian(1) mit Ihrer Datei `.changes` aus. Der Befehl **lintian** führt viele Skripte aus, um auf typische Paketierungsfehler zu prüfen. 1

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

Selbstverständlich müssen Sie den Dateinamen mit dem Namen der Datei `.changes`, die für Ihr Paket erstellt wurde, ersetzen. Die Ausgabe des Befehls **lintian** verwendet die folgenden Schalter:

- `E:` für Fehler; eine definitive Verletzung der Richtlinien oder ein Paketierungsfehler.
- `W:` für Warnungen; eine mögliche Verletzung der Richtlinien oder ein möglicher Paketierungsfehler.
- `I:` für Information; Informationen über bestimmte Aspekte der Paketierung.
- `N:` für Hinweis; eine detaillierte Meldung, um bei der Fehlersuche zu helfen.
- `O:` für »außer Kraft gesetzt«; eine Meldung, die durch Dateien `lintian-overrides` außer Kraft gesetzt, von der Option `--show-overrides` aber angezeigt wurde.

Wenn Sie Warnungen sehen, passen Sie Ihr Paket an, um diese zu vermeiden oder stellen Sie sicher, dass die Warnungen unberechtigt sind. Falls sie unberechtigt sind, legen Sie `lintian-overrides`-Dateien wie in Abschnitt 5.14 beschrieben an.

Beachten Sie, dass Sie das Paket mit **dpkg-buildpackage** bauen und **lintian** in einem Befehl darauf anwenden können, falls Sie debuild(1) oder pdebuild(1) verwenden.

1Sie müssen die Option `-i -I --show-overrides` von **lintian** angeben, falls Sie `/etc/devscripts.conf` oder `~/.devscripts` wie in Abschnitt 6.3 beschrieben angepasst haben.

7.5 Der Befehl debc

Sie können Dateien im binären Debian-Paket mit dem Befehl debc(1) auflisten.

```
$ debc package.changes
```

7.6 Der Befehl debdiff

Sie können Dateiinhalte in zwei Debian-Quellpaketen mit dem Befehl debdiff(1) vergleichen.

```
$ debdiff altes-Paket.dsc neues-Paket.dsc
```

Sie können Dateilisten in zwei Gruppen von binären Debian-Paketen mit dem Befehl debdiff(1) vergleichen.

```
$ debdiff altes-Paket.changes neues-Paket.changes
```

Sie sind nützlich, um zu identifizieren, wass in den Quellpaketen geändert wurde und auf unbeabsichtigte Änderungen beim Aktualisieren der Binärpakete zu prüfen, wie versehentlich fehlplatzierte oder entfernte Dateien.

7.7 Der Befehl interdiff

Sie können zwei `diff.gz`-Dateien mit dem Befehl interdiff(1) vergleichen. Dies ist zur Überprüfung, dass keine unbeabsichtigten Änderungen beim Aktualisieren von Paketen im alten `1.0`-Quellformat durch den Betreuer an den Quellen vorgenommen wurden, nützlich.

```
$ interdiff -z altes-Paket.diff.gz neues-Paket.diff.gz
```

Das neue Quellformat `3.0` speichert Änderungen in mehreren Patch-Dateien, wie in Abschnitt 5.25 beschrieben. Sie können die Änderungen jeder `debian/patches/*`-Datei auch mit **interdiff** nachverfolgen.

7.8 Der Befehl mc

Viele dieser Dateiüberprüfungsoperationen können in einen intuitiven Prozess mit einem Dateimanager wie mc(1) verwandelt werden. Er ermöglicht es Ihnen, nicht nur die Inhalte eines `*.deb`-Pakets anzuschauen, sondern auch die von `*.udeb`, `*.debian.tar.gz`, `*.diff.gz` und `*.orig.tar.gz`-Dateien.

Schauen Sie nach zusätzlichen, nicht benötigten Dateien oder solchen der Länge 0, sowohl im binären als auch im Quellpaket. Oft werden (Programm-)Reste nicht korrekt bereinigt; passen Sie Ihre Datei `rules` an, um dies zu ermöglichen.

Kapitel 8

Aktualisieren des Pakets

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Nachdem Sie ein Paket veröffentlichen, werden Sie es bald aktualisieren müssen.

8.1 Neue Debian-Revision

Nehmen wir an, dass ein Fehlerbericht für Ihr Paket unter `#654321` eingereicht wurde und er ein Problem beschreibt, das Sie lösen können. Sie können wie folgt eine neue Debian-Revision des Pakets erstellen:

- Falls dies als neuer Patch aufgezeichnet werden soll, führen Sie Folgendes aus:
  - `dquilt new` *`Fehlernummer.patch`*, um den Patch-Namen zu setzen
  - `dquilt add` *`defekte_Datei`*, um die zu korrigierende Datei anzugeben
  - Korrigieren Sie das Problem in den Paketquellen für den Fehler der Originalautoren
  - `dquilt refresh`, um es in *`Fehlernummer.patch`* aufzuzeichnen
  - `dquilt header -e`, um seine Beschreibung hinzuzufügen
- Falls hiermit ein existierender Patch aktualisiert wird, führen Sie Folgendes durch:
  - `dquilt pop` *`foo.patch`*, um den existierenden *`foo.patch`* aufzurufen
  - Korrektur des Problems in der alten *`foo.patch`*
  - `dquilt refresh`, um *`foo.patch`* zu aktualisieren
  - `dquilt header -e`, um seine Beschreibung zu aktualisieren
  - `while dquilt push; do dquilt refresh; done`, um alle Patches anzuwenden und dabei *fuzz* (Unschärfe) zu entfernen
- Fügen Sie eine neue Revision an den Anfang der Datei `changelog` hinzu, beispielsweise mit `dch -i` oder explizit mit `dch -v` *`Version-Revision`* und fügen Sie dann die Kommentare mit ihrem Lieblingseditor ein. [1]
- Fügen Sie eine kurze Beschreibung des Fehler und der Lösung in dem Changelog-Eintrag hinzu, gefolgt von `Closes: #654321`. Auf diese Art wird der Fehlerbericht *automatisch* geschlossen, sobald Ihre neue Paketversion im Archiv akzeptiert wird.

1 Um das Datum in das benötigte Format zu bekommen, verwenden Sie `LANG=C date -R`.

- Wiederholen Sie die obigen Schritte, um weitere Fehler zu beheben, und aktualisieren Sie dabei die Debian-Datei `changelog` mit `dch`.
- Wiederholen Sie die Tätigkeiten aus Abschnitt 6.1 und Kapitel 7.
- Sobald Sie zufrieden sind, sollten Sie den Wert für die Distribution im `changelog` von `UNRELEASED` auf den Ziel-Distributionswert `unstable` (oder ggf. `experimental`) setzen.[2]
- Laden Sie Ihr Paket wie in Kapitel 9 beschrieben hoch. Der Unterschied besteht darin, dass Sie diesmal das ursprüngliche Quellarchiv nicht beifügen, da es sich nicht geändert hat und im Debian-Archiv bereits existiert.

Ein schwieriger Fall kann auftreten, falls Sie ein lokales Paket erstellen, um mit dem Paketieren zu experimentieren, bevor Sie die normale Version in das offizielle Archiv hochladen, z.B. *`1.0.1-1`*. Für reibungslosere Upgrades ist es eine gute Idee, einen `changelog`-Eintrag mit einer Versionszeichenkette wie *`1.0.1-1~rc1`* zu erstellen. Sie können `changelog` bereinigen, indem Sie Ihre lokalen Änderungseinträge in einen einzigen Eintrag für das offizielle Paket zusammenfassen. Lesen Sie Abschnitt 2.6 für die Reihenfolge von Versionszeichenketten.

8.2 Überprüfung einer neuen Version der Originalautoren

Wenn Sie Pakete für eine neue Veröffentlichung der Originalautoren für das Debian-Archiv vorbereiten, müssen Sie zuerst die neue Veröffentlichung der Originalautoren prüfen.

Beginnen Sie damit, dass Sie `changelog`, `NEWS` und eventuell weitere mit der neuen Version veröffentlichte Dokumentation lesen.

Sie können dann Änderungen zwischen den alten und den neuen Quellen der Originalautoren wie folgt prüfen und dabei nach allem Verdächtigem Ausschau halten:

```
$ diff -urN foo-alteVersion foo-neueVersion
```

Änderungen durch Autotools an einigen automatisch erstellten Dateien wie `missing`, `aclocal.m4`, `config.guess`, `config.h.in`, `config.sub`, `configure`, `depcomp`, `install-sh`, `ltmain.sh` und `Makefile.in` können ignoriert werden. Sie können sie vor der Ausführung von **diff** in den Quellen für die Prüfung löschen.

8.3 Neue Version der Originalautoren

Falls ein Paket *`foo`* korrekt im neueren Format `3.0 (native)` oder `3.0 (quilt)` paketiert ist, besteht das Paketieren einer neuen Version der Originalautoren im wesentlichen im Verschieben des alten Verzeichnisses `debian` in die neuen Quellen. Dies kann durch Ausführung von `tar xvzf /Pfad/zu/foo_alteVersion.debian.tar.gz` in den entpackten Quellen passieren. [3] Natürlich müssen Sie einige offensichtliche Hausaufgaben machen:

- Erstellen Sie eine Kopie der Quellen der Originalautoren als `foo_`*`neueVersion`*`.orig.tar.gz`-Datei.
- Aktualisieren Sie die Debian-Datei `changelog` mit `dch -v` *`neueVersion-1`*.
  - Fügen Sie einen Eintrag mit `New upstream release` hinzu.
  - Beschreiben Sie genau die Änderungen *in der neuen Veröffentlichung der Originalautoren*, die berichtete Fehler schließen, und schließen Sie diese Fehler, indem Sie `Closes: #`*`Fehler_Nummer`* hinzufügen.
  - Beschreiben Sie genau die Änderungen *an der neuen Veröffentlichung der Originalautoren* durch den Betreuer, die berichtete Fehler schließen, und schließen Sie diese Fehler, indem Sie `Closes: #`*`Fehler_Nummer`* hinzufügen.

2Falls Sie den Befehl `dch -r` zur Durchführung dieser letzten Änderung verwenden, stellen Sie sicher, dass Sie die Datei `changelog` explizit im Editor speichern.

3Falls ein Paket *`foo`* im alten Format `1.0` paketiert ist, kann dies stattdessen durch `zcat /Pfad/nach/foo_alte_Version.diff.gz|patch -p1` in der neuen entpackten Quelle durchgeführt werden.

- `while dquilt push; do dquilt refresh; done`, um alle Patches anzuwenden und gleichzeitig *fuzz* (Unschärfe) zu entfernen.

Falls das Patchen/Zusammenfügen nicht reibungsfrei lief, prüfen Sie die Situation (Hinweise verbleiben in `.rej`-Dateien).

- Falls ein Patch, den Sie an den Quellen angewandt hatten, in die Quellen der Originalautoren integriert wurde,
  - führen Sie `dquilt delete` aus, um ihn zu entfernen.
- Falls ein Patch, den Sie auf die Quellen anwandten, mit den neuen Änderungen in den Quellen der Originalautoren in Konflikt steht,
  - wenden Sie `dquilt push -f` an, um die alten Patches anzuwenden und Rückweisungen als *`baz`*`.rej` zu erzwingen.
  - Bearbeiten Sie die Datei *`baz`* manuell, um den geplanten Effekt von *`baz`*`.rej` zu erreichen.
  - `dquilt refresh`, um den Patch zu aktualisieren
- Fahren Sie mit `while dquilt push; do dquilt refresh; done` wie gewohnt fort.

Dieser Prozess kann mit dem Befehl uupdate(1) wie folgt automatisiert werden:

```
$ apt-get source <i>foo</i>
...
dpkg-source: info: <i>foo</i> wird nach <i>foo</i>-<i>AlteVersion</i> extrahiert
dpkg-source: info: <i>foo</i>_<i>AlteVersion</i>.orig.tar.gz wird entpackt
dpkg-source: info: <i>foo</i>_<i>AlteVersion</i>-1.debian.tar.gz wird angewandt
$ ls -F
<i>foo</i>-<i>AlteVersion</i>/
<i>foo</i>_<i>AlteVersion</i>-1.debian.tar.gz
<i>foo</i>_<i>AlteVersion</i>-1.dsc
<i>foo</i>_<i>AlteVersion</i>.orig.tar.gz
$ wget http://example.org/<i>foo</i>/<i>foo</i>-<i>NeueVersion</i>.tar.gz
$ cd <i>foo</i>-<i>AlteVersion</i>
$ uupdate -v <i>NeueVersion</i> ../<i>foo</i>-<i>NeueVersion</i>.tar.gz
$ cd ../<i>foo</i>-<i>NeueVersion</i>
$ while dquilt push; do dquilt refresh; done
$ dch
b''…b'' Änderungen dokumentieren
```

Falls Sie eine wie in Abschnitt 5.21 beschriebene Datei `debian/watch` eingerichtet haben, könen Sie den Befehl **wget** überspringen. Sie führen einfach uscan(1) in dem Verzeichnis *`foo-alteVersion`* statt des Befehls **uupdate** aus. Damit werden *automatisch* die aktualisierten Quellen gesucht, heruntergeladen und der Befehl **uupdate** ausgeführt. [4]

Sie können diese aktualisierten Quellen veröffentlichen, indem Sie die Schritte aus Abschnitt 6.1, Kapitel 7 und Kapitel 9 wiederholen.

8.4 Den Paketstil aktualisieren

Die Aktualisierung des Paketierungsstils ist keine notwendige Aktivität beim Aktualisieren eines Pakets. Allerdings erlaubt dies Ihnen, die gesamten Möglichkeiten des modernen `debhelper`-Systems und des Quellformats `3.0` auszunutzen. [5]

- Falls Sie aus irgend einem Grund gelöschte Schablonendateien erneut erstellen müssen, können Sie **dh_make** mit der Option `--addmissing` in dem gleichen Debian-Quellverzeichnis aufrufen. Danach bearbeiten Sie diese entsprechend.

[4] Falls der Befehl **uscan** die aktualisierten Quellen herunterlädt, aber nicht den Befehl **uupdate** ausführt, sollten Sie die Datei `debian/watch` korrigieren, um `debian uupdate` am Ende der URL zu haben.

[5] Falls Ihr Sponsor oder andere Betreuer dem Aktualisieren des Paketierungsstils widersprechen, argumentieren Sie nicht. Es gibt wichtigere Dinge zu erledigen.

- Falls das Paket noch nicht aktualisiert wurde, um die v7+-**dh**-Syntax von `debhelper` für die Datei `debian/rules` zu verwenden, aktualisieren Sie es, um **dh** zu verwenden. Aktualisieren Sie die Datei `debian/control` entsprechend.
- Falls Sie die mit dem `Makefile`-Einschlussmechanismus des Common Debian Build System (`cdbs`) erzeugte Datei `rules` in die **dh**-Syntax umwandeln wollen, schauen Sie sich die folgenden Informationen an, um seine `DEB_*`-Konfigurationsvariablen zu verstehen:
  - lokale Kopie von `/usr/share/doc/cdbs/cdbs-doc.pdf.gz`
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)
- Falls Sie ein `1.0`-Quellpaket ohne die Datei *`foo`*`.diff.gz` haben, können Sie sie auf das neuere Quellformat `3.0 (native)` umstellen, indem Sie `debian/source/format` mit `3.0 (native)` erstellen. Der Rest der Dateien `debian/*` kann einfach so kopiert werden.
- Falls Sie ein `1.0`-Quellpaket mit der Datei *`foo`*`.diff.gz` haben, können Sie es in das neuere Quellformat `3.0 (quilt)` umwandeln, indem Sie die Datei `debian/source/format` mit `3.0 (quilt)` erstellen. Der Rest der Dateien `debian/*` kann einfach so kopiert werden. Importieren Sie die Datei `gross.diff`, die vom Befehl `filterdiff -z -x '*/debian/*' `*`foo`*`.diff.gz > gross.diff` erstellt wurde, in Ihr **quilt**-System, falls notwendig. [6]
- Falls es mit einem anderen Patch-System wie `dpatch`, `dbs` oder `cdbs` mit `-p0`, `-p1` oder `-p2` paketiert war, wandeln Sie es unter Verwendung des `deb3`-Skripts aus http://bugs.debian.org/581186 in das `quilt`-Format um.
- Falls es mit dem Befehl **dh** mit der Option `--with quilt` oder mit den Befehlen **dh_quilt_patch** und **dh_quilt_unpatch** paketiert wurde, entfernen Sie diese und sorgen Sie dafür, dass das neuere Format `3.0 (quilt)` verwandt wird.

Sie sollten die DEP - Debian Enhancement Proposals (http://dep.debian.net/) prüfen und akzeptierte (ACCEPTED) Vorschläge umsetzen.

Sie müssen auch andere in Abschnitt 8.3 beschriebene Aufgaben erledigen.

8.5 UTF-8-Umstellung

Falls die Dokumente der Originalautoren in alten Kodierungsschemata vorliegen, ist es eine gute Idee, sie in UTF-8 umzuwandeln.

- Verwenden Sie iconv(1) für die Konvertierung reiner Textdateien.

```
iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

- Verwenden Sie w3m(1) zur Umwandlung von HTML-Dateien in reine UTF-8-Textdateien. Stellen Sie dabei sicher, dass sie es unter einer UTF-8-Locale ausführen.

```
LC_ALL=en_US.UTF-8 w3m -o display_charset=UTF-8 \
        -cols 70 -dump -no-graph -T text/html \
        < foo_in.html > foo_out.txt
```

8.6 Erinnerungen für die Paketaktualisierung

Es folgen ein paar Erinnerungen für die Aktualisierung von Paketen:

- Erhalten Sie alte Einträge im `changelog` (klingt offensichtlich, aber es gab Fälle, in denen `dch` eingegeben wurde, wenn `dch -i` hätte verwandt werden sollen).

[6] Mit dem Befehl **splitdiff** können Sie `gross.diff` in viele kleine, inkrementelle Patches zerteilen.

- Existierende Debian-Änderungen müssen neu geprüft werden; entfernen Sie Dinge, die die Originalautoren integriert haben (in einer oder der anderen Form) und denken Sie daran, Dinge zu behalten, die noch nicht integriert wurden, falls es nicht doch überzeugende Gründe für die Entfernung gibt.
- Falls Änderungen am Bausystem vorgenommen wurden (hoffentlich wissen Sie es bei der Prüfung der Änderungen der Originalautoren), aktualisieren Sie `debian/rules` und die Bauabhängigkeiten in `debian/control`, falls notwendig.
- Prüfen Sie in der Fehlerdatenbank (BTS) (http://www.debian.org/Bugs/) , ob jemand Patches für derzeit offene Fehler bereitgestellt hat.
- Prüfen Sie den Inhalt der Datei `.changes`, um sicherzustellen, dass Sie in die korrekte Distribution hochladen, die richtigen Fehlerschließanweisungen im Feld `Closes` enthalten sind, die Felder `Maintainer` and `Changed-By` passen, die Datei GPG-signiert ist usw.

Kapitel 9

Das Paket hochladen

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Debian now requires source-only uploads for normal upload. So this page is outdated.

Nachdem Sie jetzt das neue Paket intensiv getestet haben, möchten Sie es zur gemeinsamen Benutzung in ein öffentliches Archiv hochladen.

9.1 In das Debian-Archiv hochladen

Sobald Sie ein offizieller Entwickler sind, [1] können Sie das Paket in das Debian-Archiv hochladen. [2]. Sie können dies manuell erledigen, aber es ist leichter, eines der bestehenden automatischen Werkzeuge zu verwenden, wie dupload(1) oder dput(1). Wir beschreiben, wie es mit dem Befehl **dupload** gemacht wird. [3]

Zuerst müssen Sie die Konfigurationsdatei von **dupload** einrichten. Sie können entweder die systemweite Datei `/etc/dupload.conf` bearbeiten oder durch Ihre persönliche Datei `~/.dupload.conf` die wenigen Dinge, die Sie ändern möchten, überschreiben.

Sie können die Handbuchseite dupload.conf(5) lesen, um zu verstehen, was jede dieser Optionen bedeutet.

Die Option `$default_host` bestimmt, welche der Upload-Warteschlangen standardmäßig benutzt wird. `anonymous-ftp-master` ist die primäre, aber es ist möglich, dass Sie eine andere benutzen möchten. [4]

Während Sie mit dem Internet verbunden Sie, können Sie Ihr Paket wie folgt hochladen:

```
$ dupload gentoo_0.9.12-1_i386.changes
```

dupload prüft, ob die SHA1/SHA256-Dateiprüfsummen zu den in der Datei `.changes` aufgeführten passen. Falls Sie nicht passen, wird es Sie warnen, dass sie wie in Abschnitt 6.1 beschrieben erneut bauen müssen, so dass es korrekt hochgeladen werden kann.

Falls Sie auf ein Problem beim Hochladen in ftp://ftp.upload.debian.org/pub/UploadQueue/ stoßen, können Sie dies durch manuelles Hochladen einer GPG-signierten Datei `*.commands` mit **ftp** korrigieren. [5] Beispielsweise mit der `hello.commands`:

1Siehe Abschnitt 1.1

2Es gibt öffentlich erreichbare Archive wie http://mentors.debian.net/, die genauso wie das Debian-Archiv funktionieren und einen Upload-Bereich für Nicht-DDs bereitstellen. Sie können ein äquivalentes Archiv selbst erstellen, indem Sie die unter http://wiki.debian.org/HowToSetupADebianRepository aufgeführten Werkzeuge verwenden. Daher ist dieser Abschnitt auch für Nicht-DDs nützlich.

3Das Paket `dput` scheint mehr Funktionalitäten zu enthalten und beliebter als das Paket `dupload` zu werden. Es verwendet die Datei `/etc/dput` für seine globale Konfiguration und die Datei `~/.dput.cf` für die benutzerspezifische Konfiguration. Es unterstützt auch standardmäßig Ubuntu-bezogene Dienste.

4Siehe Debian-Entwicklerreferenz 5.6, »Ein Paket hochladen« (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#upload) .

5Lesen Sie ftp://ftp.upload.debian.org/pub/UploadQueue/README. Alternativ können Sie den Befehl **dcut** aus dem Paket `dput` verwenden.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
 rm hello_1.0-1_i386.deb
 mv hello_1.0-1.dsx hello_1.0-1.dsc
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

[...]
-----END PGP SIGNATURE-----
```

9.2 Die Datei `orig.tar.gz` hochladen

Wenn Sie das Paket zum ersten Mal ins Archiv hochladen, müssen Sie auch die ursprüngliche Quelldatei `orig.tar.gz` einschließen. Falls die Debian-Revisionsnummer dieses Pakets weder `1` noch `0` ist, müssen Sie die Option `-sa` von **dpkg-buildpackage** verwenden.

Für den Befehl **dpkg-buildpackage**:

```
$ dpkg-buildpackage -sa
```

Für den Befehl **debuild**:

```
$ debuild -sa
```

Für den Befehl **pdebuild**:

```
$ pdebuild --debbuildopts -sa
```

Andererseits wird die Option `-sd` den Ausschluss der ursprünglichen Quelle `orig.tar.gz` erzwingen.

9.3 Übersprungene Uploads

Falls Sie durch Überspringen von Uploads mehrere Einträge in `debian/changelog` erzeugt haben, müssen Sie eine korrekte Datei `*_.changes` erzeugen, die sämtliche Änderungen seit dem letzten Upload enthält. Dies kann durch Angabe der Option `-v` von **dpkg-buildpackage** mit der Version, z.B. *`1.2`*, erfolgen.

Für den Befehl **dpkg-buildpackage**:

```
$ dpkg-buildpackage -v1.2
```

Für den Befehl **debuild**:

```
$ debuild -v1.2
```

Für den Befehl **pdebuild**:

```
$ pdebuild --debbuildopts "-v1.2"
```

Anhang A

Fortgeschrittene Paketierung

Die Überarbeitung dieser Anleitung mit aktualisierten Inhalten und weiteren praktischen Beispielen ist unter Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) verfügbar. Bitte verwenden Sie diese neue Anleitung als primäre Anleitung.

Es folgen einige Tipps und Verweise für fortgeschrittene Paketierungsfragen, mit denen Sie wahrscheinlich zu tun bekommen werden. Es wird Ihnen nachdrücklich empfohlen, alle hier vorgeschlagenen Referenzen zu lesen.

Es könnte sein, dass Sie die durch den Befehl **dh_make** erstellten Paketierungsvorlagedateien manuell bearbeiten müssen, um Punkte aus diesem Kapitel zu berücksichtigen. Der neuere Befehl **debmake** sollte diese Punkte besser berücksichtigen.

A.1 Laufzeit-Bibliothek

Bevor Sie Laufzeit-Bibliotheken paketieren, sollten Sie die folgenden Referenzen im Detail lesen:

- Debian Policy Manual, Kapitel 8 »Shared libraries« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html)
- Debian Policy Manual, Kapitel 9.1.1 »File System Structure« (http://www.debian.org/doc/debian-policy/ch-opersys.html#s-fhs)
- Debian Policy Manual, Kapitel 10.2 »Libraries« (http://www.debian.org/doc/debian-policy/ch-files.html#s-libraries)

Es folgen einige stark vereinfachte Tipps für den Anfang:

- Laufzeitbibliotheken (engl. »shared libraries«) sind ELF-Objektdateien, die übersetzten Code enthalten.
- Laufzeitbibliotheken werden als `*.so`-Dateien vertrieben, d.h. weder als `*.a`- noch als `*.la`-Dateien.
- Laufzeitbibliotheken werden hauptsächlich benutzt, um gemeinsamen Programmcode aus mehreren Programmen mittels des **ld**-Mechanismus' gemeinsam zu nutzen.
- Laufzeitbibliotheken werden manchmal dazu benutzt, mehrere Erweiterungen eines Programms mittels des **dlopen**-Mechanismus' bereitzustellen.
- Laufzeitbibliotheken exportieren Symbole, die übersetzte Objekte wie Variablen, Funktionen und Klassen darstellen und darauf Zugriff von dem verlinkten Programm ermöglichen.
- Der SONAME einer Laufzeitbibliothek `lib`*`foo`*`.so.`*`1`*: `objdump -p lib`*`foo`*`.so.`*`1` `| grep SONAME` [1]
- Der SONAME einer Laufzeitbibliothek passt typischerweise (aber nicht immer) auf den Dateinamen der Bibliothek.

[1] Alternativ: `readelf -d lib`*`foo`*`.so.`*`1` `| grep SONAME`

- Der SONAME von Laufzeitbibliotheken, die nach */usr/bin/foo* gelinkt sind: `objdump -p /usr/bin/foo | grep NEEDED` 2
- `lib`*`foo1`*: das Paket für die Laufzeitbibliothek `lib`*`foo`*`.so.`*`1`* mit der SONAME-ABI-Version *1*.3
- Die Paketbetreuerskripte des Bibliothekspakets müssen **ldconfig** unter den bestimmten Randbedingungen aufrufen, um die notwendigen symbolischen Links für den SONAME zu erzeugen.[4]
- `lib`*`foo1`*`-dbg`: das Fehlersuch-Symbol-Paket, das die Debugging-Symbole für das Laufzeitpaket `lib`*`foo1`* enthält.
- `lib`*`foo`*`-dev`: das Entwicklungspaket, das die Header-Dateien usw. für die Laufzeitbibliothek `lib`*`foo`*`.so` enthält.[5]
- Debian-Pakete sollten im Allgemeinen keine `*.la`-Libtool-Archive enthalten.[6]
- Debian-Pakete sollten im Allgemeinen keinen RPATH enthalten.[7]
- Obwohl er etwas veraltet und nur Sekundärliteratur ist, könnte der Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) noch nützlich sein.

A.2 `debian/`*`Paket`*`.symbols` verwalten

Wenn Sie eine Laufzeitbibliothek paketieren, sollten Sie eine Datei `debian/`*`Paket`*`.symbols` erstellen, um die minimale Version zu verwalten, die jedem Symbol für rückwärts-kompatible ABI-Änderungen unter dem gleichen SONAME der Bibliothek für den gleichen Laufzeitbibliotheksnamen zugeordnet ist.[8] Sie sollten die folgenden primären Referenzen im Detail lesen:

- Debian Policy Manual, Kapitel 8.6.3 »The symbols system« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-symbols) [9]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

Es folgt ein grobes Beispiel, wie das Paket `libfoo1` aus der Version `1.3` der Originalautoren mit der korrekten Datei `debian/libfoo1.symbols` zu erstellen ist:

- Bereiten Sie das Gerüst des debianisierten Quellbaums mit der Datei der Originalautoren `libfoo-1.3.tar.gz` vor.
  - Falls dies das erstmalige Paketieren des Pakets `libfoo1` ist, erstellen Sie die Datei `debian/libfoo1.symbols` mit leerem Inhalt.
  - Falls die vorherige Version `1.2` der Originalautoren im Paket `libfoo1` mit der korrekten `debian/libfoo1.symbols` in seinem Quellpaket paketiert war, verwenden sie diese wieder.

2 Alternativ: `readelf -d lib`*`foo`*`.so.`*`1`* `| grep NEEDED`

3 Siehe Debian Policy Manual, Kapitel 8.1 »Run-time shared libraries« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) .

4 Siehe Debian Policy Manual, Kapitel 8.1.1 »ldconfig« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) .

5 Siehe Debian Policy Manual, Kapitel 8.3 »Static libraries« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-static) und Debian Policy Manual, Kapitel 8.4 »Development files« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-dev) .

6 Siehe Debian wiki ReleaseGoals/LAFileRemoval (http://wiki.debian.org/ReleaseGoals/LAFileRemoval) .

7 Siehe Debian-Wiki RpathIssue (http://wiki.debian.org/RpathIssue) .

8 Rückwärts-inkompatible ABI-Änderungen verlangen normalerweise von Ihnen, dass Sie den SONAME der Bibliothek und den Namen des Laufzeitbibliothekspakets in neue SONAMEN bzw. Namen ändern.

9 Für C++-Bibliotheken und andere Fälle, bei denen das Nachverfolgen individueller Symbole zu schwierig ist, folgen Sie stattdessen dem Debian Policy Manual, Kapitel 8.6.4 »The shlibs system« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) .

– Falls die vorhergehende Version `1.2` der Originalautoren nicht mit `debian/libfoo1.symbols` paketiert worden war, erstellen Sie sie als Datei `symbols` von allen verfügbaren Binärpaketen des selben Laufzeitbibliotheksnamens, der den gleichen SONAME der Bibliothek enthält, beispielsweise Version `1.1-1` und `1.2-1`. 10

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

- Bauen Sie mit Werkzeugen wie **debuild** und **pdebuild** versuchsweise den Quellbaum. (Falls das aufgrund fehlender Symbole usw. fehlschlägt, gibt es einige rückwärtsinkompatible ABI-Änderungen, die es notwendig machen, den Paketnamen der Laufzeitbibliothek auf etwas wie `libfoo1a` zu erhöhen. Sie sollten dann wieder von vorne anfangen.)

```
$ cd libfoo-1.3
$ debuild
...
dpkg-gensymbols: Warnung: einige neue Symbole sind in der Symboldatei aufgetaucht: b''…
    b''
 lesen Sie die folgende Diff-Ausgabe
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx        2012-11-11 02:24:53.609667389 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
  foo_get_type@Base 1.1
+ foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...
```

- Falls Sie wie oben den von **dpkg-gensymbols** ausgegebenen Diff sehen, extrahieren Sie die korrekt aktualisierte `symbols`-Datei aus dem erstellten Binärpaket der Laufzeitbibliothek. 11

```
$ cd ..
$ dpkg-deb -R  libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1\.3-1/1\.3/' libfoo1-tmp/DEBIAN/symbols \
       >libfoo-1.3/debian/libfoo1.symbols
```

- Bauen Sie mit Werkzeugen wie **debuild** und **pdebuild** die Veröffentlichungspakete.

```
$ cd libfoo-1.3
$ debuild -- clean
$ debuild
b''…b''
```

Zusätzlich zu den obigen Beispielen müssen wir die ABI-Kompatibilität weiter prüfen und die Versionen für einige Symbole wo notwendig erhöhen. 12

Obwohl es nur eine nachrangige Referenz ist, könnte UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) im Debian Wiki sowie die darin verlinkten Webseiten hilfreich sein.

10Alle vorhergehenden Versionen eines Pakets sind unter http://snapshot.debian.org/ (http://snapshot.debian.org/) erhältlich. Die Debian-Revision wird von der Version entfernt, um das Rückportieren eines Paketes zu erleichtern: `1.1` << `1.1-1~bpo70+1` << `1.1-1` and `1.2` << `1.2-1~bpo70+1` << `1.2-1`

11Die Debian-Revision wird von der Version entfernt, um das Rückportieren eines Paketes zu erleichtern: `1.3` << `1.3-1~bpo70+1` << `1.3-1`

12Siehe Debian Policy Manual, Kapitel 8.6.2 »Shared library ABI changes« (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-updates) .

A.3 Multiarch

Die Multiarch-Funktionalität, die mit Debian Wheezy eingeführt wurde, unterstützt die architekturübergreifende Installation von Binärpaketen (insbesondere `i386`<->`amd64`, aber auch andere Kombinationen) in `dpkg` und `apt`. Sie sollten die folgenden Referenzen im Detail lesen:

- MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) im Ubuntu Wiki (Originalautoren)
- Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) im Debian Wiki (Debian-Situation)

Es verwendet ein Triplet wie `i386-linux-gnu` und `x86_64-linux-gnu` für den Installationspfad der Laufzeitbibliotheken. Der tatsächliche Triplett-Pfad wird durch dpkg-architecture(1) für jeden Bau dynamisch in `$(DEB_HOST_MULTIARCH)` gesetzt. Beispielsweise werden die Pfade zu Multiarch-Bibliotheken wie folgt geändert:[13]

Alter Pfad	I386-Multiarch-Pfad	Amd64-Multiarch-Pfad
`/lib/`	`/lib/i386-linux-gnu/`	`/lib/x86_64-linux-gnu/`
`/usr/lib/`	`/usr/lib/i386-linux-gnu/`	`/usr/lib/x86_64-linux-gnu/`

Es gibt einige typische Beispiele für Multiarch-Paketaufteilungsszenarien:

- eine Bibliotheksquelle `lib`*`foo`*`-1.tar.gz`
- eine in einer übersetzten Sprache geschriebene Werkzeugquelle *`bar`*`-1.tar.gz`
- eine in einer interpretierten Sprache geschriebene Werkzeugquelle *`baz`*`-1.tar.gz`

Paket	Architecture:	Multi-Arch:	Paketinhalt
`lib`*`foo1`*	any	same	die Laufzeitbibliothek, koinstallierbar
`lib`*`foo1`*`-dbg`	any	same	die Debug-Symbole der Laufzeitbibliothek, koinstallierbar
`lib`*`foo`*`-dev`	any	same	die Laufzeitbibliothek-Header-Dateien usw., koinstallierbar
`lib`*`foo`*`-tools`	any	foreign	die Laufzeitunterstützungsprogramme, nicht koinstallierbar
`lib`*`foo`*`-doc`	all	foreign	die Dokumentationsdateien der Laufzeitbibliothek
`bar`	any	foreign	die übersetzten Programmdateien, nicht koinstallierbar
`bar``-doc`	all	foreign	die Dokumentationsdateien für das Programm
`baz`	all	foreign	die interpretierten Programmdateien

Beachten Sie, dass die Entwicklungspakete einen Symlink für die zugehörige Laufzeitbibliothek **ohne eine Versionsnummer** enthalten sollten, z.B. `/usr/lib/x86_64-linux-gnu/libfoo.so` -> `libfoo.so.1`

A.4 Erstellen eines Laufzeitbibliothekspakets

Sie können ein Debian-Laufzeitbibliothekspaket mit Unterstützung von Multiarch mit dh(1) wie folgt bauen:

- `debian/control` aktualisieren
  - `Build-Depends: debhelper (>=10)` zum Quellpaketabschnitt hinzufügen
  - `Pre-Depends: ${misc:Pre-Depends}` für jedes Laufzeitbibliothekspaket hinzufügen
  - Fügen Sie den `Multi-Arch:`-Absatz zu jedem Binärpaketabschnitt hinzu.
- `debian/compat` auf »10« setzen

13 Alte Spezialbibliothekspfade wie `/lib32/` und `/lib64/` werden nicht mehr benutzt.

- Passen Sie für alle Paketierungsskripte die Pfade vom normalen `/usr/lib/` auf die Multiarch-Variante `/usr/lib/$(DEB_HOST_` an.
  - Zuerst `DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` in `debian/rules` aufrufen, um die Variable `DEB_HOST_MULTIARCH` zu setzen.
  - Ersetzen Sie in `debian/rules` `/usr/lib/` durch `/usr/lib/$(DEB_HOST_MULTIARCH)/`.
  - Falls `./configure` als Teil des Ziels `override_dh_auto_configure` in `debian/rules` verwandt wird, stellen Sie sicher, dass Sie es durch `dh_auto_configure --` ersetzen. [14]
  - Ersetzen Sie in `debian/`*`foo`*`.install`-Dateien alle Vorkommen von `/usr/lib/` durch `/usr/lib/*/`.
  - Erstellen Sie dynamisch durch Hinzufügen eines Skripts im Ziel `override_dh_auto_configure` in `debian/rules` Dateien wie `debian/`*`foo`*`.links` aus `debian/`*`foo`*`.links.in`.

```
override_dh_auto_configure:
        dh_auto_configure
        sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
                debian/foo.links.in > debian/foo.links
```

Bitte stellen Sie sicher, dass das Laufzeitbibliothekspaket nur die erwarteten Dateien enthält und dass Ihr -dev-Paket noch funktioniert.

Alle Dateien, die simultan aus dem Multiarch-Paket in den gleichen Dateipfad installiert werden, sollten exakt den gleichen Inhalt haben. Sie müssen vorsichtig mit Unterschieden sein, die durch die Byte-Reihenfolge und den Kompressionsalgorithmus entstehen.

A.5 Natives Debian-Paket

Falls ein Paket nur für Debian oder möglicherweise nur für lokale Benutzung betreut wird, können seine Quellen alle Dateien aus `debian/*` enthalten. Es gibt zwei Möglichkeiten, dies zu paketieren:

Sie können einen Tarball der Originalautoren erstellen, bei dem Sie die `debian/*`-Dateien ausschließen und es als nicht natives Debian-Paket wie in Abschnitt 2.1 paketieren. Dies ist die normale Art, zu der einige Leute raten.

Die Alternative wäre der Arbeitsablauf für ein natives Debian-Paket:

- Erstellen Sie ein natives Debian-Quellpaket im Format `3.0 (native)` mit einer einzelnen komprimierten Tar-Datei, in der alle Dateien enthalten sind.
  - *`Paket_Version`*`.tar.gz`
  - *`Paket_Version`*`.dsc`
- Bauen Sie Debian-Binärpakete aus den nativen Debian-Quellpaketen.
  - *`Paket_Version_Arch`*`.deb`

Falls Sie beispielsweise Quelldateien in `~/mypackage-1.0` ohne die `debian/*`-Dateien haben, können Sie ein natives Debian-Paket mittels des Befehls **dh_make** wie folgt erstellen:

```
$ cd ~/MeinPaket-1.0
$ dh_make --native
```

[14] Alternativ können Sie die Argumente `--libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` und `--libexecdir=\$${prefix}/lib/$(DEB_` zu `./configure` hinzufügen. Beachten Sie, dass `--libexecdir` den Standardpfad angibt, in den ausführbare Programme installiert werden, die von anderen Programmen (nicht von Benutzern) ausgeführt werden. Die Vorgabe von Autotools ist `/usr/libexec/`, die Debian-Vorgabe ist allerdings `/usr/lib/`.

Dann werden das Verzeichnis `debian` und seine Inhalte genau wie in Abschnitt 2.8 erstellt. Dies erstellt keinen Tarball, da dies ein natives Debian-Paket ist. Das ist aber auch der einzige Unterschied. Der Rest der Paketierungsaktivitäten ist praktisch identisch.

Nach der Ausführung des Befehls **dpkg-buildpackage** werden Sie die folgenden Dateien im übergeordneten Verzeichnis finden:

- `meinpaket_1.0.tar.gz`

  Dies ist der Quellcode-Tarball, der aus dem Verzeichnis `MeinPaket-1.0` durch den Befehl **dpkg-source** erstellt wurde. (Seine Endung ist nicht `orig.tar.gz`.)

- `meinpaket_1.0.dsc`

  Dies ist eine Zusammenfassung der Inhalte des Quellcodes, wie in dem nicht nativen Debian-Paket. (Es gibt keine Debian-Revision.)

- `meinpaket_1.0_i386.deb`

  Dies ist Ihr komplettes Binärpaket wie in dem nicht nativen Debian-Paket. (Es gibt keine Debian-Revision.)

- `meinpaket_1.0_i386.changes`

  Diese Datei beschreibt wie in dem nicht nativen Debian-Paket alle Änderungen, die in der aktuellen Paketversion erfolgten. (Es gibt keine Debian-Revision.)