Ubuntu Packaging Guide

Release 1.0.2 bzr685 ubuntu14.04.1

Ubuntu Developers

27.04.2019

Inhaltsverzeichnis

Willkommen beim Ubuntu Paketierungs- und Entwicklungshandbuch!

Dies ist der offizielle Ort um alles über die Ubuntu Entwicklung und Paketierung zu lernen. Nach dem Lesen dieses Handbuch, werden Sie folgendes haben:

- Ein Verständnis der wichtigsten Mitspieler, Prozesse und Werkzeuge in der Ubuntu-Entwicklung,
- Ihre Entwicklungsumgebung richtig eingerichtet,
- Einen besseren Eindruck wie man sich unserer Gemeinschaft anschließt,
- Einen echten Ubuntu Bug als Teil des Kurses behoben haben.

Ubuntu ist nicht nur ein freies Open Source Betriebssystem, seine Plattform ist auch offen und wird in einer transparenten Art entwickelt. Der Quellcode für jede einzelnen Komponenten kann einfach besorgt werden und jede einzelne Änderung an der Ubuntu Plattform kann angeschaut werden.

Das heißt, Sie können selbst aktiv werden und die Dinge verbessern. Die Gemeinschaft der Ubuntu Plattform Entwickler ist immer an neuen Kollegen, die gerade anfangen, interessiert.

Ubuntu ist außerdem eine Gemeinschaft von tollen Leuten, die an freie Software glauben und dass jeder Zugang dazu haben sollte. Die Mitglieder sind einladend und freuen sich, wenn Sie mit machen wollen. Wir freuen uns über Fragen und wenn Sie anfangen, Ubuntu zusammen mit uns besser zu machen.

Sollten Sie Probleme haben: keine Panik! Schau Sie den *Artikel über Kommunikation* an und Sie werden sehen, wie Sie am leichtesten in Kontakt mit anderen Entwicklern kommen.

Die Anleitung teilt sich in zwei Bereiche auf:

- Eine Liste von Artikel, die spezifische Aufgaben betreffen.
- Eine Sammlung von Knowledge-Base Artikel, die ins Detail gehen und Werkzeuge und Workflows erklären.

KAPITEL 1

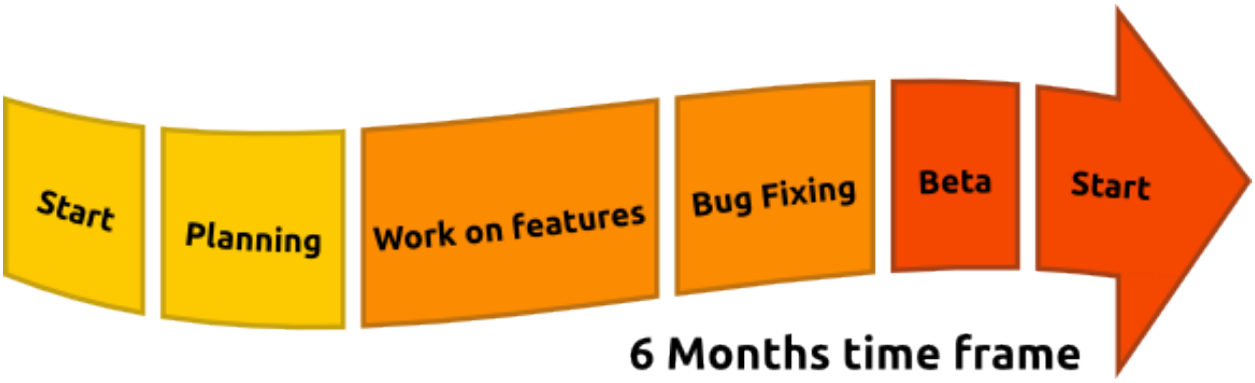
Artikel

1.1 Einführung in die Ubuntu-Entwicklung

Ubuntu besteht aus tausenden verschiedenen Komponenten, geschrieben in vielen verschiedenen Programmiersprachen. Jede Komponente – sei es eine Softwarebibliothek, ein Werkzeug oder eine grafische Anwendung – ist als Quellpaket erhältlich. Quellpakete bestehen in den meisten Fällen aus zwei Teilen: dem eigentlichen Quellcode und den Metadaten. Metadaten enthalten die Abhängigkeiten des Pakets, Informationen zum Urheberrecht und zur Lizenz, und Einweisungen wie das Paket erstellt werden soll. Ist das Quellpaket einmal kompiliert, werden durch den Erstellungsprozess Binärpakete zur Verfügung gestellt, welche der Benutzer in Form von .deb Dateien installieren kann.

Jedes Mal wenn eine neue Version einer Anwendung veröffentlicht wird oder jemand eine Änderung am Sourcecode macht, der in Ubuntu geht muss das Quellpaket auf Launchpads Buildrechner hochgeladen und kompiliert werden. Das entstandene Binärpaket wird dann ins Archiv verteilt sowie an Spiegelserver in verschiedenen Ländern. Die URLs in `/etc/apt/sources.list` zeigen auf ein Archiv oder Spielgelserver. Jeden Tag werden Abbilder für unterschiedliche Ubuntu Richtungen gebaut. Diese können in verschiedenen Umständen verwendet werden. Dies sind Abbilder, die man auf eine USB-Stick schreiben, auf DVDs brennen kann oder man kann netboot Abilder verwenden. Ebenso gibt es Abbilder, die für Ihr Telefon und Tablet passen. Ubuntu Desktop, Ubuntu Server, Kubuntu und andere geben eine Liste von benötigten Paketen an, die auf das Abbild kommen. Diese Abbilder werden dann zu Installationstests verwendet oder Feedback für die weitere Release-Planung verwendet.

Die Entwicklung von Ubuntu ist sehr stark abhängig vom aktuelle Veröffentlichungszyklus. Alle 6 Monate wird eine neue Version von Ubuntu veröffentlicht. Dies ist aber nur möglich, weil es fest definierte Enddaten für den Eingang neuer Paketversionen gibt. Ab diesem Datum sind Entwickler dazu angehalten, nur noch kleinere Veränderungen vorzunehmen. Nach der Hälfte des Entwicklungszeitraumes ist das sogenannte »Feature Freeze« erreicht, bis zu dem alle neuen Funktionalitäten implementiert sein müssen. In der restlichen Zeit wird hauptsächlich an der Fehlerbehebung gearbeitet. Zu diesem Zeitpunkt werden die Benutzeroberfläche, die Dokumentation, der Kernel usw. gesperrt, die »Beta-Phase« ist erreicht, in der sehr viele Tests durchgeführt werden. Von nun an werden nur noch kritische Fehler behoben und eine Vorveröffentlichung wird erstellt. Sobald keine größeren Probleme mehr auftreten, wird daraus die letztendliche neue Versionsveröffentlichung.



Tausende Quellpakete, Millionen von Codezeilen und hunderte beteiligte Personen benötigen eine gute Kommunikation und Planung, um einen hohen Qualitätsstandard zu halten. Am Anfang sowie in der Mitte eines Veröffentlichungszyklus findet eine Veranstaltung Namens »Ubuntu Developer Summit« statt, bei dem Entwickler und sonstige beteiligte Personen zusammentreffen und zukünftige Funktionalitäten der nächsten Version planen. Die zuständigen Projektgruppen diskutieren dort über diese Funktionalitäten und stellen eine Spezifikation auf, in der alle detaillierten Informationen, Erwartungen, Implementierungen, nötigen Veränderungen an anderen Stellen und Testbedingungen enthalten sind. Der komplette Prozess ist dabei transparent und für jeden einsehbar, sodass Sie auch teilnehmen können, ohne direkt anwesend zu sein. Dazu gibt es Videoübertragungen, Chats mit Teilnehmern oder auch das Abonnement der Änderungen an Spezifikationen. Sie sind also immer auf dem neusten Stand.

Aber nicht jede einzelne Änderung kann bei einem solchen Treffen diskutiert werden, besonders, weil Ubuntu auch von Änderungen in vielen anderen Projekten abhängt. Deshalb stehen alle an Ubuntu beteiligten Menschen dauerhaft in Kontakt. Die meisten Projekte benutzen eigene externe Mailinglisten, um nicht den Überblick in der Hauptarbeit zu verlieren. Für eilige Änderungen benutzen Entwickler und alle Beitragenden außerdem noch den Internet Relay Chat (IRC). Jegliche Diskussionen sind offen und öffentlich einsehbar.

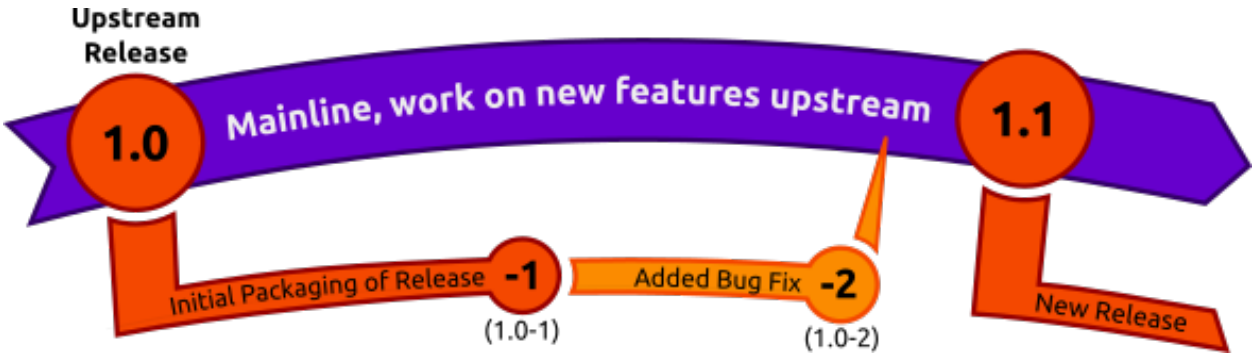
Ein weiteres wichtiges Werkzeug, welches die Kommunikation betrifft ist, sind Fehlerberichte. Wann immer ein Defekt in einem Paket oder einem Teil der Infrastruktur gefunden wird, wird ein Fehlerbericht auf Launchpad eingereicht. Jede Information ist diesem Bericht vereinigt und seine Wichtigkeit, Status und Bearbeiter werden bei Bedarf angepasst. Das macht es zu einem effektiven Werkzeug den Fehlern in einem Paket oder Projekt Herr zu bleiben und den Arbeitsaufwand zu organisieren.

Die meiste in Ubuntu erhältliche Software ist nicht von den Ubuntu-Entwicklern selbst geschrieben, sondern von Entwicklern anderer Open-Source-Projekte und die dann in Ubuntu eingefügt wird. Diese Projekte werden “Upstreams” genannt, weil ihr Quellcode in Ubuntu einfließt, wo wir ihn “nur” integrieren. Die Beziehung zu den Upstreams ist von kritischer Wichtigkeit für Ubuntu. Es ist nicht nur der Code den wir von ihnen bekommen, sondern sie bekommen von uns auch Benutzer, Fehlerberichte und Fehlerbehebungen von Ubuntu (und anderen Distributionen).

Der wichtigste Upstream für Ubuntu ist Debian. Debian ist die Distribution auf der Ubuntu aufbaut und viele der Entscheidungen über das Designs der Paket-Infrastruktur werden dort getroffen. Traditionell hatte Debian schon immer eigene Betreuer oder ganze Entwicklerteams für jedes einzelne Paket. In Ubuntu gibt es auch Teams die ein Interesse an einer Einheit von Paketen zeigen und natürlicherweise hat jeder Entwickler ein Spezialgebiet. Jedoch sind Teilnahme (und Rechte zum Hochladen) generell offen für jeden der Fähigkeiten und Willen zeigt.

Selber zu Ubuntu beizutragen ist nicht so schwierig wie es scheint und kann durchaus eine lohnenswerte Erfahrung sein. Dabei geht es nicht nur darum, etwas neues und spannendes zu lernen, sondern auch um das Lösen von Problemen und so das Helfen von Millionen von Menschen.

Quelloffene Entwicklung geschieht in einer dezentralen Art mit unterschiedlichen Zielen. Beispielsweise kommt es vor, dass ein Entwickler gerne eine neue großartige Funktion implementieren möchte, während Ubuntu, gebunden an den Veröffentlichungszyklus, ein Hauptaugenmerk auf ein stabiles System mit guter Fehlerbehebung legt. Darum wird hier das Prinzip der “aufgeteilten Arbeit” angewendet, wo in vielen Zweigen gleichzeitig entwickelt wird, die am Ende alle zusammengeführt werden.



In dem oben gezeigtem Beispiel würde es Sinn machen, Ubuntu mit der bestehenden Version des Projektes ausliefern, die Fehlerbehebung hinzufügen, diese für die nächste Veröffentlichung Upstream hinzuzufügen und es mit der (wenn geeignet) nächsten Ubuntu-Veröffentlichung auszuliefern. Dieses ist der bestmögliche Kompromiss und es würde jeder Gewinnen.

Um einen Fehler in Ubuntu zu reparieren, müssen Sie sich zuerst den Quelltext des Paketes besorgen. Dann den Fehler beheben und so dokumentieren, dass es einfach für andere Entwickler und Benutzer zu verstehen ist und schließlich das Paket bauen um es zu testen. Nachdem Sie das Paket getestet haben, können Sie Ihre Änderung einfach zur Aufnahme in den aktuellen Ubuntu Entwicklungszweig vorschlagen. Ein Entwickler mit dem Recht zum hochladen wird Ihre Änderung für Sie bewerten und anschließend es für Sie in Ubuntu integrieren.

Wenn Sie eine Lösung suchen, ist es eine gute Idee zu prüfen, ob Upstream das Problem bekannt ist. Wenn es noch keine Lösung gibt, macht es Sinn, daran gemeinsam zu arbeiten.

Zusätzliche Schritte könnten beinhalten, die Änderung auf einen älteren, immer noch unterstützten Release zurückzuportieren oder die Änderungen an Upstream weiterzuleiten.

Die wichtigsten Anforderungen für den Erfolg in der Ubuntu-Entwicklung sind: Der Drang »Dinge wieder zum Laufen zu bringen«, keine Angst davor zu haben Dokumentationen zu lesen und Fragen zu stellen, Teamgeist zu zeigen und ein wenig Detektivarbeit genießen zu können.

Gute Orte um Ihre Fragen zu stellen sind `ubuntu-motu@lists.`.ubuntu.com‘‘ und `#ubuntu-motu` auf freenode. Sie werden leicht viele neue Freunde finden sowie Leute mit der gleichen Leidenschaft, wie Sie sie haben: Die Welt einen besseren Ort durch die Herstellung besserer Open Source Software machen.

1.2 Die Programme einrichten

Es gibt einiges zu tun, bevor Sie mit der Ubuntu-Entwicklung loslegen können. Dieser Artikel wird Ihnen helfen, Ihr System so einzurichten, dass Sie mit Paketen arbeiten und Ihre Pakete auf Ubuntu’s Hosting-Platform, Launchpad, hochladen können. Hierüber werden wir reden:

- Paketierungs-Software installieren. Dies beinhaltet:
  - Ubuntu-spezifische Paketierungs-Werkzeuge
  - Verschlüsselungssoftware so dass Ihre Arbeit als Ihre eigene verifiziert werden kann
  - Weitere Verschlüsselungssoftware so dass Sie sichere Dateitransfers machen können
- Ihren Account auf Lauchpad erstellen und einrichten

- Ihre Entwicklungsumgebung aufsetzen, so dass Sie lokale Builds von Paketen machen, mit anderen Entwicklern interagieren und Ihre Änderungswünsche in Launchpad unterbreiten können.

Bemerkung: Es macht Sinn, Paketierungaufgaben direkt in der Entwicklungsversion von Ubuntu zu machen. Dies wird es Ihnen erlauben Ihre Änderungen in derselben Umgebung zu testen, wo diese später eingepflegt und verwendet werden.

Sie möchten nicht die neueste Ubuntu Entwicklerversion installieren? Richten Sie einen LXD container ein.

1.2.1 Grundlegende Paketierungs-Software installieren

Es gibt eine Reihe von Werkzeugen, die Ihr Leben als Ubuntu Entwickler viel leichter machen. Auf diese Werkzeuge stossen Sie später in diesem Handbuch. Um den Großteil der Werkzeuge zu installieren brauchen Sie diesen Befehl:

```
$ sudo apt install gnupg pbuilder ubuntu-dev-tools apt-file
```

Dieser Befehl wird folgende Software installieren:

- `gnupg` – GNU Privacy Guard enthält Werkzeuge die Sie brauchen werden, um einen kryptografischen Schlüssel zu erstellen mit dem Sie Dateien unterzeichnen werden, die Sie auf Launchpad hochladen möchten.
- `pbuilder` – ein Werkzeug um reproduzierbare Builds eines Pakets in einer sauberen und isolierten Umgebung zu machen.
- `ubuntu-dev-tools` (und `devscripts`, eine direkte Abhängkigkeit) – eine Sammlung von Werkzeugen, die viele Paketierungsaufgaben einfacher machen.
- `apt-file` ist eine einfache Möglichkeit das Binärpaket zu finden welches eine gegebene Datei enthält.

Ihren GPG-Schlüssel erstellen

GPG steht für GNU Privacy Guard und es implementiert den OpenPGP Standard, der es möglich macht, Nachrichten und Dateien zu signieren und zu verschlüsseln. Dies ist für verschiedene Zwecke wichtig. In unserem Falle ist es wichtig, dass Sie Dateien mit Ihrem Schlüssel signieren können, so dass sie identifizieren, dass es etwas ist woran Sie gearbeitet haben. Wenn Sie ein Quellpaket auf Launchpad hochladen, wird dieses das Paket nur akzeptieren wenn es genau bestimmen kann wer das Paket hochgeladen hat.

Um Ihren GPG-Schlüssel zu erstellen, starten Sie:

```
$ gpg --gen-key
```

GPG fragt zuerst, welche Art von Schlüssel Sie generieren möchten. Die Standardauswahl (RSA und DSA) ist eine gute Wahl. Als nächstes werden Sie nach der Schlüsselgröße gefragt. Die Standardauswahl (zur Zeit 2048) ist zwar eine gute Wahl, aber 4096 bietet deutlich mehr Sicherheit. Danach müssen Sie angeben, ob der Schlüssel zu einem bestimmten Zeitpunkt ungültig werden soll. Auch hier ist die Stadardauswahl »0« ein guter Wert, der bedeutet, dass der Schlüssel unbegrenzt gültig ist. Zum Schluss werden noch Ihr Name sowie Ihre E-Mail-Adresse abgefragt. Geben Sie hier die E-Mail-Adresse an, mit der Sie bei der Entwicklung bei Ubuntu tätig sein möchten. Bei Bedarf können später weitere hinzugefügt werden. Die letzte Angabe ist eine Passphrase (eine Passphrase ist ein Passwort, in dem auch Leerzeichen enthalten sein dürfen). Sie sollte sehr sicher sein.

Jetzt wird GPG einen Schlüssel für Sie generieren, was ein bisschen dauern kann. Es braucht zufällige Bytes, also ist eine gute Idee das System ein wenig auszulasten. Bewegen Sie den Mauszeiger hin und her, schreiben ein paar Zeilen und laden ein paar Webseiten.

Wenn das getan ist, werden Sie in etwa solch eine Meldung erhalten:


```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

In diesem Fall ist `43CDE61D` die *key ID*.

Als nächstes müssen Sie den öffentlichen Teil Ihres Schlüssels auf einen Keyserver hochladen, so dass man Nachrichten und Dateien Ihnen zuordnen kann. Um das zu tun, führen Sie folgenden Befehl aus:

```
$ gpg --send-keys --keyserver keyserver.ubuntu.com <KEY ID>
```

Dies wird Ihren Schlüssel auf den Ubuntu-Schlüsselserver laden, aber ein Netzwerk aus Schlüsselservern wird den Schlüssel untereinander weiterreichen. Ist der Vorgang erst einmal abgeschlossen, ist Ihr unterschriebener öffentlicher Schlüssel bereit alle Ihre Beiträge auf der ganzen Welt zu verifizieren.

Ihren SSH-Schlüssel erstellen

SSH steht für *Secure Shell* und ist ein Protokoll, welches es Ihnen erlaubt, Daten sicher über ein Netzwerk auszutauschen. Es ist üblich per SSH Zugang zur Kommandozeile eines andern Rechners zu erhalten und es zur sicheren Dateiübertragung zu verwenden. Für unsere Zwecke nutzen wir SSH hauptsächlich zum sicheren Hochladen von Paketen zu Launchpad.

Um Ihren SSH-Schlüssel zu erstellen, starten Sie:

```
$ ssh-keygen -t rsa
```

Der Standardname ergibt normalerweise Sinn, deshalb kann er auch einfach so bleiben. Aus Sicherheitsgründen ist es ratsam eine Passphrase zu verwenden.

pbuilder einrichten

`pbuilder` ermöglicht es Ihnen, Pakete lokal auf Ihrem Rechner zu erstellen. Das dient mehreren Zwecken:

- Die Erstellung wird in einer minimalen und sauberen Umgebung vorgenommen. Das hilft sicherzustellen, dass der Vorgang in einer wiederholbaren Weise abläuft ohne Ihr System zu modifizieren.
- Sie brauchen nicht alle nötigen Build-Abhängigkeiten lokal installieren.
- Sie können mehrere Instanzen für diverse Ubuntu- und Debian-Versionen einrichten

`pbuilder` einzurichten ist sehr einfach, starten Sie:

```
$ pbuilder-dist <release> create
```

wobei <release> beispielsweise *xenial*, *zesty*, *artful* sein kann bzw. im Falle von Debian *sid* oder *buster*. Es wird einige Zeit dauern da es alle nötigen Pakete für eine "minimale Installation" herunterlädt. Diese werden aber im Cache gespeichert werden.

1.2.2 Alles einrichten um mit Launchpad arbeiten zu können

Mit einer grundlegenden lokalen Konfiguration vor Ort ist Ihr nächster Schritt Ihr System auf die Arbeit mit Launchpad einzurichten. Dieses Kapitel legt den Schwerpunkt auf folgende Themen:

- Was Launchpad ist und einen Launchpad Account erstellen
- Ihren GPG- und SSH-Schlüssel auf Launchpad hochladen

- Konfigurieren Ihrer Shell, Sie zu erkennen (um Ihren Namen im Changelogs zu bringen)

Über Launchpad

Launchpad ist das Kernstück der Infrastruktur in Ubuntu. Es speichert nicht nur unsere Pakete und unseren Code, sondern auch Dinge wie Übersetzungen, Fehlerberichte, Informationen über Personen die an Ubuntu arbeiten sowie deren Mitgliedschaften in Entwicklerteams. Sie werden Launchpad ebenso dazu benutzen um Ihre Lösungen zu veröffentlichen und andere Ubuntu-Entwickler dazu zu bringen sie zu überprüfen und zu fördern.

Sie werden sich mit Launchpad anmelden und ein Minumum an Informationen preisgeben müssen. Das gibt Ihnen die Möglichkeit Code runter- und hochzuladen, Fehlerberichte auszustellen und vieles mehr.

Neben Ubuntu kann Launchpad jedes beliebige Projekt mit freier Software beherbergen. Für weitere Informationen sehen Sie im Launchpad Hilfe Wiki.

Einen Launchpad Account erstellen

Wenn Sie noch nicht bereits einen Launchpad Account besitzt, können Sie leicht einen erstellen. Wenn Sie ein Launchpad-Konto besitzt aber Ihre Launchpad ID nicht wissen, können Sie durch Besuch auf https://launchpad.net/~ herauszufinden und nach dem Teil nach der ~ in der URL schauen.

Der Registrierungsprozess von Launchpad wird nach einem Anzeigenamen verlangen. Es ist empfehlenswert hier seinen echten Namen zu verwenden, damit die anderen Ubuntu-Entwickler eine Möglichkeit haben Sie besser kennenzulernen.

Sobald Sie einen neuen Account registrieren, wird Ihnen Launchpad eine E-Mail mit einem Weblink senden, der Ihre E-Mail-Adresse bestätigt. Falls Sie keine E-Mail erhälten, sollten Sie Ihren Spam-Ordner überprüfen.

Die new account help page auf Launchpad hat mehr Informationen über den Prozess und zusätzliche Einstellungen, die Sie ändern können.

Ihren GPG-Schlüssel auf Launchpad hochladen

Als erstes müssen Sie Ihren Fingerprint und key ID erhalten.

Um Ihren GPG-Fingerabdruck zu finden, starten Sie:

```
$ gpg --fingerprint email@address.com
```

und es wird Ihnen etwa sowas ausgegeben:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

Starten Sie dann diesen Befehl um Ihren Schlüssel an den Ubuntu Keyserver zu schicken:

```
$ gpg --keyserver keyserver.ubuntu.com --send-keys 43CDE61D
```

wobei `43CDE61D` durch Ihre Schlüssel-ID ersetzt werden sollte (welche die erste Zeile der Ausgabe des vorherigen Befehls ist). Nun können Sie Ihren Schlüssel auf Launchpad importieren.

Gehen Sie auf https://launchpad.net/~/+editpgpkeys und kopieren den “Key fingerprint” in das Textfeld. Im oben genannten Beispiel wäre das `5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D`. Klicken Sie jetzt auf “Import Key”.

Launchpad benutzt den Fingerabdruck, um Ihren Schlüssel am Ubuntu-Schlüsselserver abzufragen. Bei Erfolg erhalten Sie eine verschlüsselte E-Mail mit der Bitte, den Schlüsselimport zu bestätigen. Durchsuchen Sie dazu Ihr E-Mail Postfach nach dieser E-Mail. *Unterstützt Ihr E-Mail Anbieter OpenPGP-Verschlüsselung, werden Sie nach dem Passwort gefragt, dass Sie bei der Erstellung des Schlüssels verwendet haben. Geben Sie dieses ein und klicken Sie auf den Link, um zu bestätigen, dass dies Ihr Schlüssel ist.*

Launchpad verschlüsselt die E-Mail mit Ihrem öffentlichen Schlüssel und so kann es davon sicher ausgehen, dass der Schlüssel Ihnen gehört. Wenn Sie Thunderbird, den Standard-E-Mail-Client verwenden, können Sie das Enigmail Plugin verwenden, um schnell die Nachricht zu entschlüsseln. Wenn Ihre E-Mail-Software nicht OpenPGP unterstützt kopieren Sie die Inhalte der verschlüsselten E-Mail, geben `gpg` in Ihrem Terminal ein und fügen die E-Mail-Inhalte in Ihr Terminalfenster ein.

Zurück auf der Launchpad-Webseite, drücken Sie die Schaltfläche zur Bestätigung und Launchpad wird den Import Ihres OpenPGP-Schlüssels abschließen.

Weitere Informationen finden Sie auf https://help.launchpad.net/YourAccount/ImportingYourPGPKey

Ihren SSH-Schlüssel auf Launchpad hochladen

Öffnen Sie https://launchpad.net/~/+editsshkeys in einem Webbrowser, und öffnen auch `~/.ssh/id_rsa.pub` in einem Texteditor. Dies ist der öffentliche Teil Ihres SSH-Schlüssels, also ist es sicher ihn auf Launchpad zu veröffentlichen. Kopieren Sie den Inhalt der Datei und fügen sie in die Textbox ein die mit "Add an SSH key" überschrieben ist. Klicken Sie jetzt "Import Public Key".

Für weitere Informationen über diesen Prozess besuchen Sie die Seite creating an SSH keypair auf Launchpad.

Ihre Shell einrichten

Die Debian/Ubuntu Paketierungswerkzeuge müssen auch mehr über Sie erfahren um Sie entsprechend im Changelog aufzulisten. Öffnen Sie einfach Ihre *~/.bashrc* in einem Texteditor und fügen etwas wie folgt am Ende davon ein:

```
export DEBFULLNAME="Bob Dobbs"
export DEBEMAIL="subgenius@example.com"
```

Speichern Sie die Datei jetzt und starten entweder Ihr Terminal neu oder starten:

```
$ source ~/.bashrc
```

(Falls Sie nicht die Standardanwendung *bash* benutzt, passen Sie bitte die Konfigurationsdatei für diese Anwendung dementsprechend an.)

1.3 Einen Bug in Ubuntu beheben

1.3.1 Einleitung

Wenn Sie die Anweisungen in *sich für Ubuntu-Entwicklung einrichten* befolgt haben, sollten Sie bereit sein loszulegen.

Wie man in der obigen Abbildung sehen kann, gibt es wenig Überraschungen im Prozess der Fehlerbehebung: man findet ein Problem, lädt den Quellcode herunter, arbeitet an einer Lösung, lädt die notwendigen Änderungen nach Launchpad und bittet um einen Merge. In diesem Handbuch werden wir alle nötigen Schritte nacheinander behandeln.

1.3.2 Das Problem finden

Es gibt viele verschiedene Wege Dinge zu finden an denen man arbeiten kann. Es kann ein Fehlerbericht sein, der einen selbst betrifft (was das Testen vereinfacht), oder ein Problem, das man woanders beobachtet hat, vielleicht auch in einem Fehlerbericht.

Schauen Sie die the bitesize bugs in Launchpad an, was Ihnen eine Vorstellung davon gibt woran man arbeiten kann. Es interessiert Sie vielleicht auch, sich die Bugs anzuschauen, die vom Ubuntu Hundred Papercuts Team triaged wurden.

1.3.3 Herausfinden, was repariert werden muss

Wenn Sie das Quellpaket, das den Fehler beinhaltet nicht kennen, aber der Pfad zu dem betroffenen Programm auf Ihrem System bekannt ist, dann können Sie das Quellpaket selbst ermitteln, an dem Sie arbeiten müssen.

Nehmen wir an, dass Sie einen Bug in Bumprace, einem Rennspiel, gefunden haben. Die Bumprace Anwendung kann durch Aufruf von `/usr/bin/bumprace` auf der Kommandozeile gestartet werden. Um das Binärpaket zu finden, welches diese Anwendung enthält, verwenden Sie diesen Befehl:

```
$ apt-file find /usr/bin/bumprace
```

Dies gibt aus:

```
bumprace: /usr/bin/bumprace
```

Beachten Sie dass der Teil vor der Spalte das Binärpaketname ist. Oft ist es so, dass das Quellpaket und das Binärpaket unterschiedliche Namen haben werden. Dies passiert am häufigsten wenn ein einzelnes Quellpaket verwendet wird um verschiedene unterschiedliche Binärpakete bauen soll. Um das Quellpaket für ein bestimmtes Binärpaket zu finden, geben Sie folgendes ein:

```
$ apt-cache showsrc bumprace | grep ^Package:
Package: bumprace
$ apt-cache showsrc tomboy | grep ^Package:
Package: tomboy
```

`apt-cache` ist Teil der Standardinstallation von Ubuntu.

1.3.4 Das Problem bestätigen

Sobald Sie herausgefunden haben, in welchem Paket sich das Problem befindet, wird es Zeit zu bestätigen, dass das Problem existiert.

Sagen wir das Paket `bumprace` nennt keine Homepage in seiner Paketbeschreibung. In einem ersten Schritt würde man prüfen, ob das Problem nicht vielleicht bereits behoben ist. Das ist leicht herauszufinden, entweder man schaut in Software Center oder führt folgendes aus:

```
apt-cache show bumprace
```

Die Ausgabe sollte etwa so aussehen:

```
Package: bumprace
Priority: optional
Section: universe/games
Installed-Size: 136
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XNBC-Original-Maintainer: Christian T. Steigies <cts@debian.org>
Architecture: amd64
Version: 1.5.4-1
Depends: bumprace-data, libc6 (>= 2.4), libsdl-image1.2 (>= 1.2.10),
         libsdl-mixer1.2, libsdl1.2debian (>= 1.2.10-1)
Filename: pool/universe/b/bumprace/bumprace_1.5.4-1_amd64.deb
Size: 38122
MD5sum: 48c943863b4207930d4a2228cedc4a5b
SHA1: 73bad0892be471bbc471c7a99d0b72f0d0a4babc
SHA256: 64ef9a45b75651f57dc76aff5b05dd7069db0c942b479c8ab09494e762ae69fc
Description-en: 1 or 2 players race through a multi-level maze
 In BumpRacer, 1 player or 2 players (team or competitive) choose among 4
 vehicles and race through a multi-level maze. The players must acquire
 bonuses and avoid traps and enemy fire in a race against the clock.
 For more info, see the homepage at http://www.linux-games.com/bumprace/
Description-md5: 3225199d614fba85ba2bc66d5578ff15
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

Ein Gegenbeispiel wäre `gedit`, welches eine Homepage gesetzt hat:

```
$ apt-cache show gedit | grep ^Homepage
Homepage: http://www.gnome.org/projects/gedit/
```

Manchmal stellt sich heraus, dass jemand das Problem, das Sie lösen wollten, schon bearbeitet hat. Um doppelte und nutzlose Arbeit zu verhindern, macht es Sinn zuerst ein wenig Detektivarbeit zu leisten.

1.3.5 Die Fehlersituation ansehen

Zuerst sollte man überprüfen ob bereits ein Bericht über das Problem in Ubuntu zu finden ist. Vielleicht arbeitet bereits jemand an einer Fehlerbehebung oder wir können zu der Lösung beitragen. Für Ubuntu werfen wir einen kurzen Blick auf https://bugs.launchpad.net/ubuntu/+source/bumprace und es gibt dort keinen offenen Problembericht.

Bemerkung: Für Ubuntu wird die URL `https://bugs.launchpad.net/ubuntu/+source/<package>` einen immer auf die Bug-Seite des jeweiligen Pakets führen.

Für Debian, welches die Hauptquelle für Ubuntu's Pakete ist, schauen wir auf http://bugs.debian.org/src:bumprace und finden auch keinen bestehenden Fehlerbericht für unser Problem.

Bemerkung: Für Debian wird die URL `http://bugs.debian.org/src:<package>` einen immer auf die Bug-Seite des jeweiligen Pakets führen.

Wir arbeiten an einem besonderen Problem, da es nur die für das Paketieren relevanten Teile des `bumprace` betrifft. Wenn es ein Problem im Quellcode wäre, würde es hilfreich sein auch den Upstream-Bugtracker zu überprüfen. Das

ist leider von Paket zu Paket unterschiedlich; aber wenn Sie im Web danach suchen, sollte es in den meisten Fällen einfach zu finden sein.

1.3.6 Hilfe anbieten

Wenn Sie einen offenen Fehler gefunden haben, dieser noch nicht zugewiesen ist und Sie die Möglichkeit haben ihn zu beheben, sollten Sie einen Kommentar mit Ihrem Lösungsvorschlag abgeben. Geben Sie so viele Informationen wie möglich an: Unter welchen Umständen tritt der Fehler auf? Wie haben Sie den Fehler behoben? Haben Sie die Lösung getestet?

Wenn noch kein Fehlerbericht eingesandt wurde, können Sie das tun. Was Sie im Kopf behalten solltn ist: ist das Problem klein genug, dass man vielleicht einfach jemand direkt darum bittet einen Patch hoch zu laden? Haben Sie es vielleicht geschafft einen Teil des Problems zu beheben und wollen dies mitteilen?

Es ist toll wenn Sie Hilfe anbieten können und man wird dafür sehr dankbar sein.

1.3.7 Den Quellcode herunterladen

Sobald das Quellpaket bekannt ist auf dem gearbeitet werden soll, möchten Sie wahrscheinlich eine Kopie des Codes haben damit Sie diesen verbessern können. Das ubuntu-dev-tools Paket beinhaltet ein Werkzeug namens `pull-lp-source` welches ein Entwickler nutzen kann um den Quelltext jedes Pakets zu bekommen. Beispiel: Um den Sourcecode für das das Paket tomboy zu in `xenial` zu bekommen, können Sie dies eingeben:

```
$ pull-lp-source bumprace xenial
```

Wird keine Veröffentlichung wie `xenial` angeben, wird automatisch das Paket der Entwicklungsversion geholt.

Sobald man eine lokale Kopie des Quellpakets hat, kann man den Bug untersuchen, eine Behebung erstellen, ein debdiff erstellen und sein debdiff an einen Bug-Report zur Überprüfung durch andere Entwickler anhängen. Wir werden die Besonderheiten in den nächsten Abschnitten beschreiben.

1.3.8 Arbeiten an einer Fehlerbehebung

Es wurden ganze Bücher darüber verfasst, wie man Fehler findet, sie behebt, testet, usw. Wenn Sie ein Anfänger im Programmieren sind, versuchen Sie zuerst einfache Fehler wie beispielsweise in der Rechtschreibung zu beheben. Versuchen Sie die Änderungen so minimal wie möglich zu halten und dokumentieren Sie Ihre Änderungen und Annahmen übersichtlich.

Bevor Sie sich daran machen, einen Fehler selbst zu beheben, sollten Sie sicherstellen, dass nicht schon ein anderer diesen Fehler behoben hat oder gerade daran arbeitet. Anlaufstellen dafür sind:

- Upstream (und Debian) Fehlererfassung (offene and geschlossene Fehler),
- Die Upstream-Revisionseinträge (oder eine neue Veröffentlichung) haben möglicherweise das Problem behoben,
- Fehler oder Paket-Uploads von Debian oder einer anderen Distribution

Sie möchten vielleicht einen Patch erstellen, der die Behebung enthält. Der Befehl `edit-patch` ist eine einfache Art einen Patch einem Paket hinzuzufügen. Starten Sie:

```
$ edit-patch 99-new-patch
```

Das wird die Paketierung in temporäres Verzeichnis kopieren. Sie können nun die Dateien mit einem Texteditor bearbeiten oder die Patches vom Upstream anwenden, zum Beispiel:


```
$ patch -p1 < ../bugfix.patch
```

Nachdem Sie die Datei bearbeitet haben, schreiben Sie `exit` oder drücken `Strg+D` um die zeitweilige Umgebung zu verlassen. Der neue Patch wird dann unter `debian/patches` eingefügt.

Sie müssen dann einen Header zu Ihrem Patch inklusive Metainformationen hinzufügen, so dass andere Entwickler den Zweck des Patches kennen und woher dieser stammt. Um den Vorlagen-Header zum Editieren zu bekommen, um zu zeigen was der Patch macht, geben Sie folgendes ein:

```
$ quilt header --dep3 -e
```

Dies wird die Vorlage in einem Texteditor öffnen. Folgen Sie der Vorlage und vergewisseren sich, dass Sie ausführlich sind, so dass alle nötigen Details, die den Patch beschreiben, vorhanden sind.

Wenn Sie nur‘‘debian/control‘‘ in diesem spezifischen Fall bearbeiten möchten brauchen Sie keinen Patch. Tragen Sie `Homepage: http://www.linux-games.com/bumprace/` am Ende der ersten Sektion ein und der Bug sollte behoben sein.

Die Lösung dokumentieren

Es ist sehr wichtig Ihre Änderung ausreichend zu dokumentieren, so dass Entwickler, die sich den Code zukünftig ansehen, nicht raten müssen, was Ihre Gründe und Annahmen gewesen sind. Jedes Debian- und Ubuntu-Quellpaket beinhaltet die Datei `debian/changelog`, wo die Änderungen jedes hochgeladenen Paketes dokumentiert werden.

Die einfachste Möglichkeit um zu updaten ist den Befehl auszuführen:

```
$ dch -i
```

Dies wird eine Vorlage für einen Changelog-Eintrag für Sie erstellen, wo Sie die Lücken ausfüllen können. Ein Beispiel dafür wäre etwa:

```
specialpackage (1.2-3ubuntu4) trusty; urgency=low

  * debian/control: updated description to include frobnicator (LP: #123456)

 -- Emma Adams <emma.adams@isp.com>  Sat, 17 Jul 2010 02:53:39 +0200
```

`dch` sollte die erste und letzte Zeile von solch einem Änderungsprotokolleintrag bereits für Sie ausgefüllt haben. Zeile 1 enthält den Quellpaketnamen, die Versionsnummer, die Ubuntu-Zielversion und die Dringlichkeit (welche meistens »low« ist). Die letzte Zeile enthält immer den Namen des Autors, E-Mail-Adresse und Zeitstempel (im Format **RFC 5322**) der Änderung.

Wenn dieses Hindernis beseitigt ist, können wir uns auf den eigentlichen Eintrag des Änderungsprotokolls selbst konzentrieren: Es ist sehr wichtig festzuhalten:

1. Wo die Änderung gemacht wurde.
2. Was geändert wurde.
3. Wo die Diskussion um die Änderung statt fand.

In unserem (sehr dürftigen) Beispiel ist der letzte Punkt mit `(LP: #123456)` abgedeckt, was sich auf den Launchpad-Fehler 123456 bezieht. Fehlerberichte, Beiträge zu Mailing-Listen oder Spezifikationen sind allgemein gute Informationen, die als Rationale für eine Änderung angegeben werden können. Zusätzlich wird im Falle der Verwendung von `LP: #<Nummer>` der angegebene Fehler automatisch geschlossen, wenn das Paket zu Ubuntu hochgeladen wird.

Um Unterstützung im nächsten Abschnitt zu bekommen ist es nötig, einen Bugreport in Lauchpad zu erstellen (falls noch keiner vorhanden ist, falls doch – verwenden Sie diesen) und zu erklären warum Ihre Behebung in Ubuntu enthalten sein sollte. Beispielsweise bei tomboy würde man einen Bug hier angeben (bearbeiten Sie die URL für das

Paket für das Sie eine Behebung haben). Sobald ein Bug aufgegeben wurde, der Ihre Änderungen erklärt, geben Sie diese Bugnummer im Changelog ein.

1.3.9 Die Lösung testen

Um ein Testpaket mit Ihren Änderungen zu bauen, durchlaufen Sie diese Kommandos:

```
$ debuild -S -d -us -uc
$ pbuilder-dist <release> build ../<package>_<version>.dsc
```

Dies wird ein Quellpaket von den Zweiginhalten (`-us -uc` wird nur den Schritt der Unterzeichnung des Quellpakets überspringen und‘‘-d‘‘ wird den Schritt der Prüfung nach Build-Abhängigkeiten auslassen, pbuilder wird sich darum kümmern) und `pbuilder-dist` wird das Paket aus den Quellen für jegliches gewählte `release` bauen.

Bemerkung: Wenn `debuild` folgende Fehlermeldung ausgibt “Version number suggests Ubuntu changes, but Maintainer: does not have Ubuntu address” starten Sie den `update-maintainer` Befehl (aus ubuntu-dev-tools) und es wird dies automatisch für Sie beheben. Dies passiert weil in Ubuntu alle Ubuntu Entwickler für alle Ubuntu-Pakete verantwortlich sind. In Debian hingegen haben die Pakete Maintainer.

Im Falle von bumprace, starten Sie folgendes um die Paketinformationen zu sehen:

```
$ dpkg -I ~/pbuilder/*_result/bumprace_*.deb
```

Wie erwartet sollte nun ein `Homepage:` Feld dort sein.

Bemerkung: In vielen Fällen werden Sie das Paket auch installieren müssen, um zu prüfen, dass alles funktioniert. In unserem Fall ist das viel einfacher. Wenn der Build erfolgreich war, werden Sie die Binärpakete in `~/pbuilder/<release>_result` finden. Installieren Sie sie einfach per `sudo dpkg -i <package>.deb` oder indem Sie auf sie im Dateimanager doppelt-klicken.

1.3.10 Einreichen der Behebung und es einbringen

Sobald der changelog Eintrag geschrieben und gespeichert ist, starten Sie `debuild` noch einmal:

```
$ debuild -S -d
```

und dieses Mal wird es unterschrieben werden und Sie sind nun bereit, Ihr diff einzuschicken und Unterstützung zu bekommen.

In vielen Fällen möchte Debian eventuell den Patch ebenfalls haben (dies zu tun ist vorbildlich um sicherzustellen dass ein weiterer Kreis die Behebung bekommt). Somit sollten Sie den Patch an Debian schicken und Sie können dies wie folgt tun:

```
$ submittodebian
```

Dies wird Sie durch eine Reihe von Schritten führen, um sicherzustellen, dass der Bug an der richtigen Stelle landet. Überprüfen Sie das Diff noch einmal, um sicherzustellen, dass es keine unerwünschten Änderungen enthält, die Sie vorher gemacht haben.

Kommunikation ist wichtig, also sollten Sie mehr Beschreibung mitliefern, wenn Sie darum bitten die Änderung einzupflegen. Seien Sie freundlich, erklären Sie es ausreichend.

Wenn alles geklappt hat, sollten Sie eine Mail von Debian’s Bug-Tracker mit weiteren Informationen bekommen. Dies kann manchmal einige Minuten dauern.

Es mag vorteilhaft sein, es einfach in Debian einzubringen und es zu Ubuntu fließen zu lassen. In diesem Falle folgt man nicht dem untenstehenden Prozess. Aber manchmal im Falle von Security-Updates und Updates für stabile Veröffentlichungen ist die Behebung bereits in Debian enthalten (oder wird aus einem bestimmten Grund ignoriert) und Sie möchten dem untenstehenden Prozess folgen. Wenn Sie solche Updates machen, lesen Sie bitte unseren Artikel *Security and stable release updates*. Andere Fälle wo es möglich ist zu warten Patches an Debian zu schicken, sind reine Ubuntu Pakete, die nicht korrekt bauen oder Ubuntu-spezifische allgemeine Probleme.

Aber wenn Sie vorhaben Ihre Behebung an Ubuntu zu schicken wird es nun Zeit, ein "debdiff" zu erstellen. Dieses soll den Unterschied zwischen zwei Debian Paketen zeigen. Der Name des Befehls um eins zu erstellen ist ebenfalls `debdiff`. Es ist Teil des `devscripts` Pakets. Siehen Sie `man debdiff` für alle Details. Um zwei Quellpakete zu vergleichen, geben Sie die zwei dsc-Dateien als Argumente an:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc
```

In diesem Fall wenden Sie ein `debdiff` auf das heruntergeladene dsc mit `pull-lp-source` an sowie auf die neue dsc-Datei die Sie erstellt haben. Dies wird einen Patch erstellen, die Ihr Sponst lokal anwenden kann (durch `patch -p1 < /path/to/debdiff`). In diesem Fall wenden Sie eine Pipe auf die Ausgabe des debdiff Befehls auf eine Datei an, die man dann dem Bugreport anhängen kann:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc > 1-1.0-1ubuntu1.debdiff
```

Das Format, welches in `1-1.0-1ubuntu1.debdiff` gezeigt wird, zeigt:

1. `1-` informiert den Betreuer, dass dies die erste Revision Ihres Patches ist. Niemand ist perfekt und machmal müssen Nachfolge-Patches angeboten werden. Dies stellt sicher, dass falls Ihr Patch mehr Arbeit benötigt, man ein konsistentes Namensschema verwenden kann.
2. `1.0-1ubuntu1` zeigt die neue Version, die verwendet wird. Dies macht es einfach festzustellen, welche die neue Version ist.
3. `.debdiff` ist eine Erweiterung, die illustriert, dass es ein debdiff ist.

Während dieses Format optional ist, funktioniert es gut und Sie können dies verwenden.

Gehen Sie anschließend zu dem Fehlerbericht, vergewisseren Sie sich, dass Sie auf Launchpad eingeloggt sind und klicken auf "Add attachment or patch" wo Sie einen neuen Kommentar hinzufügen. Hängen Sie das debdiff an und hinterlassen einen Kommentar, der Ihrem Betreuer sagt, wie dieser Patch angewendet werden kann und welche Tests erfolgt sind. Ein Beispielkommentar kann wie folgt aussehen:

```
This is a debdiff for Artful applicable to 1.0-1. I built this in pbuilder
and it builds successfully, and I installed it, the patch works as intended.
```

Vergewissern Sie sich, dass Sie es als einen Patch markieren (das Ubuntu Sponsors Team wird automatisch abonniert) und dass Sie den Fehlerbericht abonniert haben. Sie werden dann eine Durchsicht irgendwo zwischen mehreren Stunden seit dem Einreichen des Patches zu mehreren Wochen bekommen. Wenn dies länger dauern sollte, treten Sie bitte `#ubuntu-motu` auf `freenode` bei und erwähnen es dort. Bleiben Sie so lange da, bist Sie eine Antwort von jemand bekommen und diese Leute Sie anweisen, was als nächstes zu tun ist.

Sobald Sie einen Review erhalten haben, wurde Ihr Patch entweder hochgeladen, Ihr Patch braucht noch Arbeit oder wurde aus anderen Gründen abgelehnt (eventuell passt die Behebung nicht für Ubuntu und sollte stattdessen zu Debian gehen). Wenn den Patch noch etwas Arbeit benötigt, folgen Sie den gleichen Schritten und senden einen Folge-Patch im Bugreport. In anderen Fällen senden Sie diesen an Debian wie oben gezeigt.

Bedenken Sie: Gute Orte um Ihre Fragen zu stellen sind `ubuntu-motu@lists.ubuntu.com` und `#ubuntu-motu` auf freenode. Sie werden leicht viele neue Freunde finden sowie Leute mit der gleichen Leidenschaft, wie Sie sie haben: Die Welt einen besseren Ort durch die Herstellung besserer Open Source Software machen.

1.3.11 Weitere Überlegungen

Wenn Sie ein Paket finden und es sich herausstellt, dass Sie mehrere triviale Dinge darin beheben können, machen Sie es ruhig in einer Änderung. Das wird die Code-Review und das Einpflegen beschleunigen.

Sollte es mehrere große Dinge geben, die Sie beheben wollen, mag es Sinn ergeben stattdessen mehrere Patches oder Merge Proposals einzuschicken. Sollte es bereits individuelle Bugs dafür geben, macht das die Sache sogar noch einfacher.

1.4 Neue Software paketieren

Auch wenn bereits tausende Pakete in den Ubuntu-Archiven vorhanden sind, gibt es noch immer sehr viele Programme, die nicht dort zu finden sind. Wenn es ein neues interessantes Programm gibt, das eine größere Verbreitung haben sollte, möchten Sie vielleicht versuchen ein Paket für Ubuntu zu erzeugen oder ein PPA einzurichten. Dieser Leitfaden hilft Ihnen Schritt für Schritt dabei die neuen Softwarepakete zu erstellen.

Sie sollten als erstes den :doc:‘Vorbereitung<./getting-set-up>‘s-Artikel lesen, um Ihre Entwicklungsumgebung vorzubereiten.

1.4.1 Das Programm überprüfen

Der erste Schritt in der Paketerstellung ist das Besorgen des veröffentlichen Quelltextes von Upstream (die Autoren von Software werden als “Upstream” bezeichnet) und das Überprüfen auf Kompilier- und Ausführbarkeit.

Dieser Leitfaden führt Sie anhand einer einfachen Anwendung namens GNU Hello, die von GNU.org veröffentlicht wurde, duch den Prozess des Paketbaus.

Download GNU Hello:

```
$ wget -O hello-2.10.tar.gz "http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz"
```

Now uncompress it:

```
$ tar xf hello-2.10.tar.gz
$ cd hello-2.10
```

Diese Anwendung nutzt das autoconf-Build-System, also wollen wir `./configure` ausführen, um uns für das Kompilieren vorzubereiten.

Dies wird die benötigten Erstellungsabhängigkeiten überprüfen. Um `hello` als einfaches Beispiel zu nehmen, `build-essential` sollte alles bereitstellen was wir brauchen. An komplexeren Programmen wird dieser Befehl scheitern, wenn Sie nicht die benötigten Bibliotheken und Entwicklungsdateien besitzen. Installieren Sie die benötigten Pakete und Entwicklungsdateien bis der Befehl erfolgreich ausgeführt wird.:

```
$ ./configure
```

Jetzt können Sie den Quelltext kompilieren:

```
$ make
```

Wenn das Kompilieren erfolgreich war, können Sie folgende Anwendung installieren und starten:

```
$ sudo make install
$ hello
```


1.4.2 Ein Paket starten

`bzr-builddeb` includes a plugin to create a new package from a template. The plugin is a wrapper around the `dh_make` command. Run the command providing the package name, version number, and path to the upstream tarball:

```
$ sudo apt-get install dh-make bzr-builddeb
$ cd ..
$ bzr dh-make hello 2.10 hello-2.10.tar.gz
```

Wenn Sie nach dem Pakettyp gefragt werden, wählen Sie mit der Eingabe von `s` den Typ einzelne Binärdatei. Der Quelltext wird in einen Zweig importiert und das `debian/` Paketverzeichnis wird hinzugefügt. Sehen Sie sich einmal den Inhalt an. Die meisten Dateien werden nur für spezielle Pakete (beispielsweise Emacs Module) benötigt, sodass Sie mit dem Entfernen der nicht benötigten Dateien anfangen können:

```
$ cd hello/debian
$ rm *ex *EX
```

Sie sollten nun jede der Dateien anpassen.

In `debian/changelog` change the version number to an Ubuntu version: `2.10-0ubuntu1` (upstream version 2.10, Debian version 0, Ubuntu version 1). Also change `unstable` to the current development Ubuntu release such as `trusty`.

Vieles Paketbauarbeit wird von einer Reihe von Skripten übernommen, die `debhelper` heißen. Das exakte Verhalten von `debhelper` ändert sich mit neuen Majorversionen. Die compat Datei weist `debhelper` an unter welcher Version dies geschehen soll. Im allgemeinen empfiehlt es sich, dies auf die neueste Version zu stellen welche `9` ist.

Unter `control` sind alle Metadaten eines Paketes enthalten. Der erste Abschnitt beschreibt das Quellpaket. Der zweite Abschnitt beschreibt die Binärpakete, die erstellt werden. Unter `Build-Depends:` müssen alle Pakete eingetragen werden, von denen die Kompilierung abhängt. Für `hello` werden mindestens die folgenden benötigt:

```
Build-Depends: debhelper (>= 9)
```

Sie müssen außerdem eine Beschreibung der Anwendung das Feld `Description:` eintragen.

`copyright` muss ausgefüllt werden um der Lizenz des Upstreams gerecht zu werden. Der Datei hello/COPYING nach ist das die GNU GPL 3 oder neuer.

`docs` enthält alle Dokumentationsdateien des Upstreams, die Ihrer Meinung nach in dem entgültigen Paket enthalten sein sollten.

`README.source` und `README.Debian` werden nur benötigt, wenn Ihr Paket nicht nur Standardfunktionen hat. Das trifft hier nicht zu, also können sie gelöscht werden.

`source/format` kann beibehalten werden, es beschreibt das Versionsformat des Quellpakets und sollte `3.0 (quilt)` sein.

Die umfangreichste Datei ist `rules`. Hierbei handelt es sich um eine Make-Datei, die den Quelltext kompiliert und in ein Binärpaket verwandelt. Erfreulicherweise wird dabei heutzutage die meiste Arbeit von `debhelper 7` erledigt, sodass das universale `%` Make-Dateiziel nur das `dh` Script ausführt, das alles benötigte durchführt.

Alle Dateien sind in größerer Ausführlichkeit in der *Übersicht im Artikel über das Debian Verzeichnis* beschrieben.

Schlussendlich committen Sie den Code zu Ihrem Paketier-Zweig:

```
$ bzr add debian/source/format
$ bzr commit -m "Initial commit of Debian packaging."
```


1.4.3 Das Paket bauen

Jetzt wird überprüft, ob das Programm kompiliert und das -deb Binärpaket erfolgreich gebaut werden kann:

```
$ bzr builddeb -- -us -uc
$ cd ../../
```

Mit dem Befehl `bzr builddeb` wird das Paket im aktuellen Verzeichnis gebaut. Die Option `-us -uc` ist die Anweisung, dass das Paket nicht GPG signiert werden muss. Das Ergebnis wird in `..` ausgegeben.

Sie können sich den Inhalt eines Paketes mit folgendem Befehl ansehen:

```
$ lesspipe hello_2.10-0ubuntu1_amd64.deb
```

Installieren Sie das Paket und prüfen ob es funktioniert (später werden Sie in der Lage sein, es mit `sudo apt-get remove hello` zu deinstallieren sofern gewünscht):

```
$ sudo dpkg --install hello_2.10-0ubuntu1_amd64.deb
```

Sie können auch alle Pakete sofort wie folgt installieren:

```
$ sudo debi
```

1.4.4 Nächste Schritte

Selbst wenn es das .deb Binärpaket baut kann Ihre Paketierung Bugs haben. Viele Fehler können automatisch durch unser Werkzeug `lintian` entdeckt werden. Dies kann gegen die Quell- .dsc Metadaten Datei, .deb Binärpakete oder .changes Datei gestartet werden:

```
$ lintian hello_2.10-0ubuntu1.dsc
$ lintian hello_2.10-0ubuntu1_amd64.deb
```

Um ausführlichere Beschreibungen der Probleme zu sehen verwenden Sie das `--info` lintian Flag oder den Befehl `lintian-info`.

Für Python Pakete gibt es auch ein `lintian4python` Werkzeug, welches einige zusätzliche lintian Überprüfungen bietet.

Nachdem der Fehler im Bauprozess behoben wurde, können Sie mit der Option `-nc` “no clean” das Paket erneut bauen, ohne mit der Kompilierung des Quelltextes starten zu müssen:

```
$ bzr builddeb -- -nc -us -uc
```

Nachdem sichergestellt ist, dass das Paket auf dem eigenen Rechner erfolgreich gebaut werden kann, sollten Sie überprüfen, ob dies auch auf einem frisch installierten System funktioniert. Zu diesem Zweck kann `pbuilder` eingesetzt werden. Da das gebaute Paket in ein PPA (Personal Package Archive) hochgeladen werden soll, ist es erforderlich, das Paket zu *signieren*. So kann Launchpad überprüfen, dass dieses Paket wirklich von Ihnen stammt (die hochzuladende Datei muss signiert werden, da die `-us` und `-uc` Markierungen nicht wie zuvor an `bzr builddeb` übergeben wurden). Für das Signieren benötigen Sie eine funktionierende Version von GPG. Sollten Sie `pbuilder-dist` oder GPG noch nicht installiert haben, *so machen Sie dies jetzt*:

```
$ bzr builddeb -S
$ cd ../build-area
$ pbuilder-dist trusty build hello_2.10-0ubuntu1.dsc
```

Wenn Sie mit Ihrem Paket zufrieden sind, wird es vermutlich Ihre Absicht sein, dass andere Benutzer Ihr Paket überprüfen. Dazu können Sie den Zweig in Launchpad hochladen:


```
$ bzr push lp:~<lp-username>/+junk/hello-package
```

Das Hochladen in ein PPA hat mehrere Vorteile: Sie können einfach den Paketbauvorgang auf Fehlerfreiheit überprüfen und andere können die Binärpakete testen. Hierfür benötigen Sie ein PPA in Launchpad, in das Sie ihre Dateien mittels `dput` hochladen:

```
$ dput ppa:<lp-username>/<ppa-name> hello_2.10-0ubuntu1.changes
```

Sie können nach Feedback im `#ubuntu-motu` IRC Kanal, oder auf der‘MOTU Mailingliste <ubuntu-motu>‘_. fragen. Dort kann auch ein spezielles Team wie das GNU Team für detailliertere Fragen gefragt werden.

1.4.5 Zur Einbindung einsenden

Es gibt eine Vielzahl von Wegen, auf denen ein Paket in Ubuntu einziehen kann. In den meisten Fällen ist der Weg über Debian der beste Weg. Auf diesem Weg wird versichert, dass das Paket die größte Anzahl an Nutzern hat da es nicht nur in Debian und Ubuntu vorhanden sein wird, sondern auch in ihren Derivaten. Hier sind einige nützliche Links für das Hinzufügen neuer Pakete im Debianprojekt:

- Debian Mentors FAQ – debian-mentors dient dem Mentoring von neuen und zukünftigen Debianentwicklern. Hier können Sie einen Sponsor finden um Ihr Paket ins Archiv hochzuladen.
- Work-Needing and Prospective Packages – Informationen wie man “Intent to Package” und “Request for Package” Bugs meldet und offene of open ITPs und RFPs auflistet.
- Debian Developer’s Reference, 5.1. New packages – Das gesamte Dokument ist äußerst wichtig sowohl für Ubuntu als auch Debian Paketierer. Diese Sektion dokumentiert wie man neue Pakete einreicht.

In manchen Fällen mag es Sinn ergeben direkt zuerst in Ubuntu zu gehen. Beispielsweise mag Debian in einem Freeze sein, was es wiederum unwahrscheinlich macht, dass Ihr Paket es rechtzeitig in Ubuntu für die nächste Veröffentlichung schafft. Dieser Prozess ist in der Sektion ” <NewPackages>‘_ des Ubuntu Wikis dokumentiert.

1.4.6 Bildschirmfotos

Haben Sie einmal ein Paket zu Debian hochgeladen, sollten Sie Bildschirmfotos bereitstellen, um zukünftigen Benutzern einen Einblick in das Programm zu geben. Diese sollen auf http://screenshots.debian.net/upload hochgeladen werden.

1.5 Security und Stable Release Updates

1.5.1 Einen sicherheitstechnischen Fehler in Ubuntu beheben

Einleitung

Beheben von Sicherheitslücken und Bugs ist nicht wirklich anders als *fixing a regular bug in Ubuntu* und es wird angenommen, dass Ihnen das Patchen von normalen Bugs bekannt ist. Um zu demonstrieren, wo Dinge anders ablaufen, werden wir das dbus Paket in Ubuntu 12.04 LTS (Precise Pangolin) einem Sicherheitsupdate unterziehen.

Den Quelltext bekommen

In diesem Beispiel wissen wir bereits, dass wir das dbus-Paket in Ubuntu 12.04 LTS (Precise Pangolin) beheben möchten. Somit müssen Sie zuerst die Version des Pakets, welches Sie herunterladen möchten, bestimmen: Wir können `rmadison` hier zur Hilfe verwenden:


```
$ rmadison dbus | grep precise
dbus | 1.4.18-1ubuntu1   | precise          | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-security | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-updates  | source, amd64, armel, armhf, i386, powerpc
```

Normalerweise sollten Sie die höchste Version für die Veröffentlichung die Sie patchen möchten wählen, die nicht in -proposed oder -backports ist. Da wird Precise's dbus aktualisieren, laden Sie 1.4.18-1ubuntu1.4 aus precise-updates herunter:

```
$ bzr branch ubuntu:precise-updates/dbus
```

Den Quelltext patchen

Now that we have the source package, we need to patch it to fix the vulnerability. You may use whatever patch method that is appropriate for the package, but this example will use `edit-patch` (from the ubuntu-dev-tools package). `edit-patch` is the easiest way to patch packages and it is basically a wrapper around every other patch system you can imagine.

Um einen Patch mit `edit-patch` anzulegen:

```
$ cd dbus
$ edit-patch 99-fix-a-vulnerability
```

Dies wird die existerenden Patches anwenden und die Paketierung in einem temporären Verzeichnis platzieren. Bearbeiten Sie nun die Dateien, die benötigt werden, die Lücke zu beheben. Oft bietet Upsteam einen Patch an so dass Sie diesen anwenden können:

```
$ patch -p1 < /home/user/dbus-vulnerability.diff
```

Nachdem die nötigen Änderungen gemacht sind, drücken Sie einfach Strg-D oder geben exit ein um die temporäre Shell zu verlassen.

Das Changelog und die Patches formatieren

Nachdem Ihre Patches angewendet wurden, ist es sinnvoll, das Changelog zu überarbeiten. Der `dch` Befehl wird verwendet um die `debian/changelog` Datei zu bearbeiten und `edit-patch` wird `dch``automatisch starten sobald alle Patches rück-angewendet wurden. Wenn Sie ``edit-patch` nicht verwenden, können Sie `dch -i` manuell starten. Nicht wie bei regulären Patches sollten Sie das folgende Format benutzen (beachten Sie, dass der Distributionsname precise-security für Sicherheitsupdates verwendet, da es ein Sicherheitsupdate für Precise ist):

```
dbus (1.4.18-2ubuntu1.5) precise-security; urgency=low

  * SECURITY UPDATE: [DESCRIBE VULNERABILITY HERE]
    - debian/patches/99-fix-a-vulnerability.patch: [DESCRIBE CHANGES HERE]
    - [CVE IDENTIFIER]
    - [LINK TO UPSTREAM BUG OR SECURITY NOTICE]
    - LP: #[BUG NUMBER]
...
```

Aktualisieren Sie Ihren Patch und verwenden die passenden Patchtags. Ihr Patch sollte mindest die Origin, Description und Bug-Ubuntu-Tags haben. Editieren Sie beispielsweise debian/patches/99-fix-a-vulnerability.patch dass es etwas wie folgt enthält:


```
## Description: [DESCRIBE VULNERABILITY HERE]
## Origin/Author: [COMMIT ID, URL OR EMAIL ADDRESS OF AUTHOR]
## Bug: [UPSTREAM BUG URL]
## Bug-Ubuntu: https://launchpad.net/bugs/[BUG NUMBER]
Index: dbus-1.4.18/dbus/dbus-marshal-validate.c
...
```

Mehrere Sicherheitslücken können im gleichen Security Upload behoben werden; vergewisseren Sie sich nur, dass Sie verschiedene Patches für verschedene Sicherheitslücken verwendest.

Ihre Arbeit testen und einsenden

Ab diesem Zeitpunkt entspricht der Prozess dem in *fixing a regular bug in Ubuntu*. Insbesondere sollten Sie:

1. Ihre Pakete bauen und überprüfen, dass es fehlerfrei kompiliert und ohne zusätzliche Compilerwarnungen
2. Aktualisieren Sie auf die neueste Version des Pakets von der vorherigen Version
3. Testen Sie, dass das neue Paket die Verwundbarkeit behebt und keine neue Regressionen einführt
4. Reichen Sie Ihre Arbeit über einen Launchpad Merge Proposal ein und geben einen Launchpad Bug ein, wobei Sie sich vergewissern sollten, dass dieser Bug als Security Bug markiert wird und `ubuntu-security-sponsors` abonniert wird.

Wenn die Sicherheitslücke noch nicht öffentlich ist, dann geben Sie keinen Merge-Vorschlag auf und vergewisseren sich, dass Sie den Bug als privat markieren.

Der aufgegebene Bug sollte einen Testfall einhalten, beispielsweise einen Kommentar der klar zeigt, wie der Bug durch Aufruf der alten Version nachgestellt werden kann und dann wie der Bug nicht mehr in der neuen Version existiert.

Der Bugreport sollte auch bestätigen, dass diese Problematik in Ubuntu-Versionen behoben ist, die neuer sind als die Version mit der vorgeschlagenen Behebung (im oberen Beispiel neuer als Precise). Wenn diese Problematik nicht in neueren Ubuntu-Versionen behoben ist, sollten Sie Aktualisierungen für diese Versionen ebenfalls vorbereiten.

1.5.2 Updates für stabile Releases

Wir erlauben auch Updates auf Veröffentlichungen, wo ein Paket einen Bug mit großem Einfluss hat, wie beispielsweise schwere Regressionen von einer vorherigen Veröffentlichung oder einen Bug der Datenverlust verursachen könnte. Aufgrund der Schwere für solche Aktualisierungen auf sich selbst, wo sie selbst Bugs einführen erlauben wir dies nur, wenn die Änderung leicht verständlich und überprüft ist.

Der Prozess für Stable Release Updates ist genau gleich wie der Prozess für Sicherheitsprobleme nur dass man `ubuntu-sru` beim Bug abonnieren sollte.

Die Aktualisierung wird das `proposed` Archiv gehen (beispielsweise `precise-proposed`). Dort wird eine Überprüfung notwendig, dass das Problem behoben wird und keine weiteren Probleme mit sich bringt. Nach einer Woche ohne gemeldete Probleme kann es in `updates` verschoben werden.

Schauen Sie sich die Stable Release Updates Wikiseite für weitere Informationen an.

1.6 Patches für Pakete

Manchmal müssen Ubuntu Paketmaintainer den Upstream Sourcecode ändern damit dieser einwandfrei auf Ubuntu funktioniert. Beispiele sind Patches an Upstream, die noch nicht in eine veröffentlichte Version eingebunden wurden oder Änderungen auf das Buildsystem die nur verwendet werden müssen um auf Ubuntu zu kompilieren. Wir könnten den Upstream Sourcecode direkt ändern aber dies macht es schwieriger die Patches später zu entfernen wenn Upstream

diese eingebunden hat bzw. die Änderung zu extrahieren um es beim Upstream-Projekt einzureichen. Stattdessen behalten wir diese Änderungen als separate Patches in Form von diff-Dateien.

Man kann Patches in Debian-Paketen auf unterschiedliche Arten handhaben, glücklicherweise stellt sich Quilt als Standard heraus, weil es von den meisten Paketen verwendet wird.

Schauen wir uns ein Beispiel an: `kamoso` in Trusty:

```
$ bzr branch ubuntu:trusty/kamoso
```

Die Patches werden in `debian/patches` gespeichert. Dieses Paket hat einen Patch `kubuntu_01_fix_qmax_on_armel.diff` um einen Komplierfehler auf ARM zu beheben. Dem Patch wurde ein Name gegeben um zu beschreiben was er macht, eine Nummer um die Patches in einer Reihe zu halten (zwei Patches können sich überschneiden wenn sie die gleiche Datei ändern) und in diesem Falle fügt das Kubuntu Team ihre eigene Vorsilbe hinzu um zu zeigen, dass der Patch von ihnen kommt und nicht von Debian.

Die Reihenfolge in der Patches angewandt werden ist in `debian/patches/series` definiert.

1.6.1 Patches mit Quilt

Bevor Sie mit Quilt arbeiten, müssen Sie spezifizieren, wo die Patches liegen. Fügen Sie dies in Ihre `~/.bashrc` hinzu:

```
export QUILT_PATCHES=debian/patches
```

Und verwenden Sie `source`, um die neuen Exports anzuwenden:

```
$ . ~/.bashrc
```

Standardmäßig werden alle Patches angewandt, wenn Sie UDD Checkouts oder heruntergeladene Quellpakete benutzen. Sie können dies folgendermaßen überprüfen:

```
$ quilt applied
kubuntu_01_fix_qmax_on_armel.diff
```

Wenn Sie den Patch entfernen wollten, würden Sie `pop` ausführen:

```
$ quilt pop
Removing patch kubuntu_01_fix_qmax_on_armel.diff
Restoring src/kamoso.cpp

No patches applied
```

Und um einen Patch anzuwenden, verwenden Sie `push`:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp

Now at patch kubuntu_01_fix_qmax_on_armel.diff
```

1.6.2 Einen neuen Patch hinzufügen

Um einen neuen Patch hinzuzufügen, muss Quilt angewiesen werden, einen neuen Patch zu erstellen, sagen Sie ihm, welche Dateien dieser Patch ändern soll, bearbeiten Sie die Dateien und aktualisieren Sie den Patch:


```
$ quilt new kubuntu_02_program_description.diff
Patch kubuntu_02_program_description.diff is now on top
$ quilt add src/main.cpp
File src/main.cpp added to patch kubuntu_02_program_description.diff
$ sed -i "s,Webcam picture retriever,Webcam snapshot program,"
src/main.cpp
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

Der `quilt add` ist sehr wichtig, wenn Sie ihn vergessen, werden die Dateien nicht im Patch auftauchen.

Die Änderung wird jetzt in `debian/patches/kubuntu_02_program_description.diff` sein und die `series` Datei wird den Patch auflisten. Sie sollten die neue Datei zur Paketierung hinzufügen:

```
$ bzr add debian/patches/kubuntu_02_program_description.diff
$ bzr add .pc/*
$ dch -i "Add patch kubuntu_02_program_description.diff to improve the program description"
$ bzr commit
```

Quilt behält seine Metadaten im `.pc/` Verzeichnis, sodass dieses momentan ebenfalls für das Paketieren hinzugefügt werden muss. In der Zukunft sollte das verbessert werden.

Im Allgemeinen sollten Sie vorsichtig sein, Patches zu Programmen hinzuzufügen sofern diese nicht von Upstream kommen, es gibt häufig einen guten Grund warum die Änderung noch nicht vorgenommen wurde. Das oben genannte Beispiel ändert einen Benutzerschnittstellen-String beispielsweise und würde somit mit den gesamten Übersetzungen brechen. Wenn Zweifel bestehen, fragen Sie den Upstreamautor bevor ein Patch hinzugefügt wird.

1.6.3 Patch Headers

Wir empfehlen, dass Sie jeden Patch mit den DEP-3 Kopfzeilen versehen. Hier sind einige Einträge, die Sie verwenden können:

Description Beschreibung was der Patch macht. Diese wird wie das `Description` Feld in `debian/control` formatiert: die erste Zeile ist die Kurzbeschreibung, die mit einem Kleinbuchstaben anfängt, die nächsten Zeilen sind ausführliche Beschreibung, die mit einem Leerzeichen eingerückt sind.

Author Autor des Patches (i.e. "Jane Doe <packager@example.com>").

Origin Wo der Patch herkommt (z.B. "upstream"), wenn *Author* nicht verwendet wird.

Bug-Ubuntu Ein Link zu einem Launchpad-Bug, eine Kurzform wird vorgezogen (wie z.B. *https://bugs.launchpad.net/bugs/XXXXXXX*). Wenn es auch Bugs in Upstream-Bugtracker oder in Debian gibt, fügen Sie auch *Bug* oder *Bug-Debian* Einträge hinzu.

Forwarded Ob der Patch an Upstream weitergeleitet wurde. Entweder "yes", "no" oder "not-needed".

Last-Update Datum der letzten Revision (in der Form "YYYY-MM-DD").

1.6.4 Auf neue Upstream Versionen aktualisieren

Um zur neuen Version zu upgraden, können Sie das `bzr merge-upstream` Kommando verwenden:

```
$ bzr merge-upstream --version 2.0.2 https://launchpad.net/ubuntu/+archive/primary/+files/kamoso_2.0
```

Wenn Sie diesen Befehl starten, werden alle Patches nicht angewandt, da sie nicht mehr zu alt werden können. Sie müssen eventuell aktualisiert werden um der neuen Upstreamquelle zu entsprechen oder sie müssen eventuell in ihrer Gesamtheit entfernt werden. Um nach Problemen zu forschen, wenden Sie die Patches nacheinander an:


```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp
Hunk #1 FAILED at 398.
1 out of 1 hunk FAILED -- rejects in file src/kamoso.cpp
Patch kubuntu_01_fix_qmax_on_armel.diff can be reverse-applied
```

Wenn er umgekehrt angewendet werden kann, bedeutet das, dass der Patch bereits von Upstream angewendet wurde, damit kann er gelöscht werden:

```
$ quilt delete kubuntu_01_fix_qmax_on_armel
Removed patch kubuntu_01_fix_qmax_on_armel.diff
```

Dann fahren Sie fort:

```
$ quilt push
Applied kubuntu_02_program_description.diff
```

Es ist sinnvoll Refresh zu starten, dies wird den Patch entsprechend der veränderten Upstreamquelle aktualisieren:

```
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

Dann committen Sie wie üblich:

```
$ bzr commit -m "new upstream version"
```

1.6.5 Quilt im Paket verwenden

Moderne Pakete nutzen standardmäßig Quilt, welches in das Paketierungsformat eingebaut ist. Prüfen Sie `debian/source/format` um sich zu vergewissern, dass dort `3.0 (quilt)` steht.

Ältere Pakete, die das Quellformat 1.0 verwenden, werden explizit Quilt verwenden müssen, normalerweise in dem ein Makefile in `debian/rules` eingebunden wird.

1.6.6 Quilt konfigurieren

Die `~/.quiltrc` Datei kann verwendet werden um quilt zu konfigurieren. Hier sind einige Optionen, die nützlich sein können, für die Verwendung von quilt mit debian/packages:

```
# Set the patches directory
QUILT_PATCHES="debian/patches"
# Remove all useless formatting from the patches
QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
# The same for quilt diff command, and use colored output
QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
```

1.6.7 Andere Patch-Systeme

Andere Patchsysteme, die von Paketen verwendet werden beinhalten `dpatch` und `cdbs simple-patchsys`. Diese arbeiten ähnlich wie Quilt indem Patches in `debian/patches` gehalten werden aber unterschiedliche Befehle haben Patches anzuwenden, rückabzuwickeln oder Patches zu erstellen. Sie können herausfinden welches Patchsystem von einem Paket verwendet wird in dem Sie den `what-patch` Befehl (aus dem Paket `ubuntu-dev-tools`) verwenden. Sie können `edit-patch`, gezeigt in *vorherigen Kapiteln*, als einen zuverlässigen Weg verwenden, um mit allen Systemen zu arbeiten.

In noch älteren Paketen werden Änderungen direkt in den Quellen beinhaltet und in der `diff.gz` Quelldatei gespeichert. Dies erschwert es, auf neue Upstream Versionen zu aktualisieren oder zu zwischen Patches zu unterscheiden und sollte am besten nicht gemacht werden.

Änderen Sie nicht das Patch-System eines Pakets ohne das mit dem Debian-Maintainer oder dem zuständigen Ubuntu-Team zu besprechen. Wenn derzeit keines existiert, ist es Ihnen freigestellt Quilt hinzuzufügen.

1.7 FTBFS-Pakete reparieren

Bevor ein Paket in Ubuntu verwendet werden kann, muss es aus den Quellen erstellt werden können. Falls dies nicht gelingt wird es in -proposed warten und ist daher nicht in den Ubuntu-Archiven verfügbar. Eine vollständige Liste solcher Pakete finden Sie unter http://qa.ubuntuwire.org/ftbfs/. Auf der Seite werden 5 Kategorien angezeigt:

- Paket konnte nicht erstellt werden (F): Beim Erstellen trat ein Fehler auf.
- Erstellung abgebrochen (X): Die Erstellung wurde aus einem Grund abgebrochen. Davon sollten folgende von Anfang vermieden werden.
- Paket wartet auf ein anderes Paket (M): Dieses Paket wartet bis ein anderes anderes Paket erstellt oder aktualisiert wird oder (wenn das Paket in main ist) eine seiner Abhängigkeiten ist im falschen Archiv-Teil.
- Fehler im chroot (C): Ein Teil des chroot ist fehlgeschlagen, das kann meist durch eine Neuerstellung des Pakets behoben werden. Bitten Sie einen Entwickler eine Neuerstellung vorzunehmen.
- Fehler beim Hochladen (U): Das Paket konnte nicht hochgeladen werden. Das ist normalerweise ein Fall um eine Neuerstellung zu bitten, aber überprüfen Sie zuerst das Erstellungsprotokoll.

1.7.1 Erste Schritte

Als erstes sollten Sie versuchen das FTBFS selbst zu reproduzieren. Holen Sie sich den Code entweder per `bzr branch lp:ubuntu/PACKAGE` und dann den Tarball oder per `dget PACKAGE_DSC` auf der ".dsc"-Datei von der Launchpad-Seite. Ist das einmal erledigt, erstellen Sie es in einer schroot.

Sie sollten in der Lage sein das FTBFS zu reproduzieren. Wenn nicht, überprüfen Sie ob gerade eine fehlende Abhängigkeit heruntergeladen wird, was bedeutet das Sie sie zu einer Erstellungsabhängigkeit in debain/control machen müssen. Ein Paket lokal zu erstellen hilft auch ein Problem ausfindig zu machen, was durch eine fehlende, nicht aufgeführte Abhängigkeit verursacht wird (funktioniert lokal aber nicht in einer schroot).

1.7.2 Debian überprüfen

Können Sie das Problem reproduzieren, ist es an der Zeit eine Lösung zu finden. Falls das Paket auch in Debian enthalten ist, können Sie überprüfen ob es dort erstellt werden kann, indem Sie auf http://packages.qa.debian.org/PACKAGE gehen. Falls Debian eine aktuelle Version einsetzt, solltest Sie diese mergen. Falls nicht, überprüfen Sie die Erstellungs- und Fehlerberichte, die auf der Seite verlinkt sind, für zusätzliche Infos über FTBFS oder Patches. Debian betreut auch eine Liste von Befehl-FTBFS und wie sie behoben werden können; sie ist unter https://wiki.debian.org/qa.debian.org/FTBFS zu finden, Sie werden sie sicher auch bei der Lösungssuche einbeziehen wollen.

1.7.3 Andere Gründe warum die Paketerstellung fehlschlägt

Wenn ein Paket in main enthalten ist, aber eine dessen Abhängigkeiten nicht in main vorhanden ist, sollte ein MIR-Fehlerbericht eingereicht werden. https://wiki.ubuntu.com/MainInclusionProcess erklärt den Ablauf.

1.7.4 Das Problem beheben

Haben Sie erst einmal eine Lösung für das Problem gefunden, folgen Sie demselben Prozess wie bei jedem anderen Fehler. Erstellen Sie einen Patch, hängen ihn an einen bzr-Zweig oder -Fehler an, informieren ubuntu-sponsors, und versuchen dann das der Patch von Upstream und/oder Debian aufgenommen wird.

1.8 Gemeinsame Bibliotheken

Gemeinsame Bibliotheken sind kompilierter Code, der von vielen verschiedenen Programmen gemeinsam genutzt wird. Sie werden als `.so`-Dateien unter `/usr/lib/` zur Verfügung gestellt.

Eine Bibliothek exportiert Symbole, welche die kompilierten Versionen von Funktion, Klassen und Variablen sind. Eine Bibliothek wird als SONAME bezeichnet und beinhaltet eine Versionsnummer. Dieser SONAME entspricht nicht zwingend der öffentlichen Versionsbezeichnung. Ein Programm wird gegen eine gegebene SONAME-Version der Bibliothek kompiliert. Wenn eines der Symbole entfernt oder geändert wird, muss die Versionsnummer angepasst werden, was dazu führt, dass jedes Paket welches die Bibliothek benutzt, wieder gegen die neue Version kompiliert werden muss. Versionsnummern werden normalerweise vom Upstream festgelegt und wir folgen ihnen mit unseren Binärpaketnamen, auch ABI-Nummer genannt. Aber manchmal benutzt der Upstream keine vernünftigen Versionsnummern und die Paketierer müssen einer getrennten Nummerierung folgen.

Bibliotheken werden normalerweise von Upstream als Einzelreleases verteilt. Manchmal werden sie auch als Teil einen Programms herausgegeben. In diesem Fall können sie einfach mit in das Programm-Paket integriert werden (wenn zu erwarten ist, dass kein anderes Programm diese Bibliothek benutzt). Meistens sollten sie jedoch getrennt werden und in gesonderte Pakete gepackt werden.

Die Bibliotheken selbst werden in einem Binärpaket mit dem Namen `libfoo1` abgelegt, wobei `foo` der Name der Bibliothek und `1` die SONAME-Version ist. Entwicklungsdateien aus dem Paket, beispielsweise Kopfdateien, die benötigt werden, um Programme gegen die Bibliothek zu übersetzen, werden in einem Paket mit dem Namen `libfoo-dev` abgelegt.

1.8.1 Ein Beispiel

Wir werden libnova als Beispiel verwenden:

```
$ bzr branch ubuntu:trusty/libnova
$ sudo apt-get install libnova-dev
```

Um den SONAME der Bibliothek herauszufinden, starten Sie:

```
$ readelf -a /usr/lib/libnova-0.12.so.2 | grep SONAME
```

Der SONAME ist `libnova-0.12.so.2`, die dem Dateinamen entspricht (üblicherweise der Fall aber nicht immer). Hier hat Upstream die Versionsnummer als Teil des SONAME eingesetzt und ihm eine ABI-Version von `2` gegeben. Bibliothekspaketnamen sollten dem SONAME der Bibliothek folgenden, die sie beinhalten. Das Binärbibliothekspaket heißt `libnova-0.12-2` wobei `libnova-0.12` der Name der Bibliothek und `2` unsere ABI-Nummer ist.

Wenn im Upstream inkompatible Änderungen an den Bibliotheken durchgeführt werden, müssen sie eine neue Revision des SONAME erstellen und wir werden unsere Bibliothek umbenennen müssen. Jedes andere Paket welches unsere Bibliothek verwendet, muss dann wieder gegen die neue Version kompiliert werden; das nennt man Transition und kann einigen Aufwand verursachen. Im besten Fall bleibt unsere ABI-Nummer passend zu SONAME des Upstreams, aber manchmal führen sie Inkompatibilitäten ein ohne ihre Versionsnummer zu ändern und so müssen wir unsere anpassen.

Wenn wir uns debian/libnova-0.12-2.install ansehen, stellen wir fest, dass dort zwei Dateien eingebunden werden:


```
usr/lib/libnova-0.12.so.2
usr/lib/libnova-0.12.so.2.0.0
```

Das letzte ist die eigentliche Bibliothek, komplett mit Unter- und Punkversionsnummer. Das erste ist ein Symlink welcher auf die eigentliche Bibliothek verweist. Der Symlink ist das, nach dem die Programme welche Bibliothek benutzen suchen, sie kümmern sich nicht um die Unterversionsnummern.

`libnova-dev.install` beinhaltet alle benötigten Dateien um das Programm mit dieser Bibliothek zu kompilieren. Kopfdateien, eine Konfigurationsbinärdatei, die libtool Datei `.la` und `libnova.so`, welche als weiterer Symlink zu der Bibliothek führt; Programme die gegen die Bibliothek kompiliert werden kümmern sich nicht um die Hauptversionsnummer (jedoch wird das die Binärdatei in die sie kompilieren berücksichtigen).

`.la` libtool Datein werden auf manchen nicht-Linuxsystemen benötigt, die mangelnde Bibliotheksunterstützung haben aber verursachen normalerweise mehr Probleme auf Debiansystemen als sie lösen. Es ist ein aktuelles Debian Ziel, .la-Dateien zu entfernen und wir sollten dabei helfen.

1.8.2 Statische Bibliotheken

Das -dev Paket stellt außerdem `usr/lib/libnova.a` bereit. Dies ist eine statische Bibliothek, eine Alternative zu gemeinsamen Bibliotheken. Jedes gegen diese statische Bibliothek kompilierte Programm beinhaltet diesen Quelltext. Damit entfällt das Problem, die Bibliothek könnte als Binärdatei inkompatibel sein. Allerdings bedeutet dies auch, dass alle Fehler, die Sicherheitsprobleme inbegriffen, nicht behoben werden, solange das Programm nicht erneut kompiliert wird. Aus diesem Grund sind Programme, die statische Bibliotheken verwenden, zu vermeiden.

1.8.3 Symbol-Dateien

Wenn ein Paket gegen eine Bibliothek baut wird der `shlibs` Mechanismus eine Paketabhängigkeit auf diese Bibliothek hinzuzufügen. Aus diesem Grund haben die meisten Programme `Depends: ${shlibs:Depends}` in `debian/control`. Dies wird mit den Bibliotheksabhängigkeiten zur Bauzeit ersetzt werden. Trotzdem kann shlibs es nur auf die Major ABI Versionsnummer abhängen lassen, `2` in unsersem libnova Beispiel. Falls also neue Symbole in libnova 2.1 hinzugefügt werden kann ein Programm welches diese Symbole benutzt trotzdem gegen libnova ABI 2.0 installiert werden was dann einen Absturz zur Folge hat.

Um die Bibliotheksabhängigkeiten präzise zu halten, verwenden wir `.symbols`-Dateien, die alle Symbole in einer Bibliothek auflisten und in welcher Version sie zuerst auftauchten.

libnova hat keine Symboldatei, also können wir eine erstellen. Beginnen Sie damit das Paket zu kompilieren:

```
$ bzr builddeb -- -nc
```

Die Option `-nc` sorgt dafür, dass nach dem Kompilieren die Dateien des Programms, die währenddessen erzeugten wurden, nicht entfernt werden. Wechseln Sie in das Verzeichnis des Kompilierungsprozesses und führen Sie `dpkg-gensymbols` für das Bibliothekspaket aus:

```
$ cd ../build-area/libnova-0.12.2/
$ dpkg-gensymbols -plibnova-0.12-2 > symbols.diff
```

Dies erstellt eine Diff-Datei welche Sie selbst anwenden können:

```
$ patch -p0 < symbols.diff
```

Dies wird eine Datei erstellen die ähnlich `dpkg-gensymbolsnY_WWI` benannt wird, welche alle Symbole auflistet. Es listet auch die aktuelle Paketversion auf. Wir können die Paketierungsversion von derjenigen, die in der symbols Datei aufgelistet ist, entfernen da neue Symbole nicht allgemein durch neue Paketierungsversionen hinzugefügt werden sondern von den Upstream Entwicklern:


```
$ sed -i s,-0ubuntu2,, dpkg-gensymbolsnY_WWI
```

Nun verschieben Sie die Datei an ihren Ort, comitten und machen einen Testlauf:

```
$ mv dpkg-gensymbolsnY_WWI ../../libnova/debian/libnova-0.12-2.symbols
$ cd ../../libnova
$ bzr add debian/libnova-0.12-2.symbols
$ bzr commit -m "add symbols file"
$ bzr builddeb
```

Wenn es erfolgreich kompiliert, ist die Symboldatei in Ordnung. Mit der nächsten Upstream-Version von libnova würden Sie erneut dpkg-gensymbols ausführen und es liefert eine Auflistung der Unterschiede um die Symboldatei zu aktualisieren.

1.8.4 C++-Bibliothek-Symboldateien

C++ hat sogar noch genauere Binärkompatibilitäts-Standards als C. Das Debian Debian Qt/KDE Team wartet einige Skripte um dies zu verwalten. Schauen Sie ihre Working with symbols files Seite an wie man diese verwendet.

1.8.5 Weiterführende Literatur

Junichi Uekawa's Debian Library Packaging Guide geht bei diesem Thema stärker ins Detail.

1.9 Zurückportieren von Software-Aktualisierungen

Manchmal möchten Sie eine neue Funktionalität in einer stabilen Veröffentlichung machen, die sich nicht auf eine kritische Fehlerbehebung bezieht. Für diese Fälle haben Sie zwei Optionen: Entweder Sie laden in ein PPA hoch oder bereiten einen Backport vor.

1.9.1 Persönliches Paketarchiv (PPA)

Ein PPA bringt viele Vorteile. Es ist leicht zu benutzen und Sie benötigen keinerlei Genehmigung oder Überprüfung anderer. Aber andererseits werden die Benutzer sie manuell einrichten müssen, da sie keine Standard-Paketquelle ist.

Die PPA Dokumentation auf Launchpad ist sehr ausführlich und sollte Sie schnell zu Ergebnissen führen.

1.9.2 Offizielle Ubuntu-Backports

Das Zurückportieren ist ein Mittel um den Nutzern neue Funktionen bereitzustellen. Wegen der Gefahr von Kompabilitätsproblemen erhalten Nutzer keine zurückportierten Pakete ohne ausdrückliche Maßnahmen ihrerseits. Das macht Zurückportieren grundsätzlich zu einer schlechten Wahl für Fehlerbehebungen. Wenn ein Paket in einer Ubuntu-Veröffentlichung einen Fehler aufweist, sollte er entweder durch *ein Sicherheitsupdate oder den "Stable-Release"-Updateprozess* angemessen behoben werden.

Wenn Sie sich dazu entschlossen haben, ein Paket durch eine Backport in einen stabilen Release zu portieren, werden Sie zumindest einen Test-Build ausführen müssen und das Paket in demjenigen Release zu testen. `pbuilder-dist` (im `ubuntu-dev-tools` Paket) ist ein tolles Werkzeug um das problemlos zu tun.

Um eine Anfrage auf Zurückportierung zu stellen und sie vom Team bearbeitet zu bekommen, können Sie das Werkzeug `requestbackport` verwenden (auch in dem Paket `ubuntu-dev-tools` enthalten). Es wird feststellen

welche dazwischenliegende Veröffentlichungen das Paket benötigt um zurückportiert werden zu können, listet alle zurückliegenden Abhängigkeiten auf und stellt eine offizielle Anfrage. Es wird außerdem dem Fehlerbericht eine Prüfliste für Tests anhängen.

KAPITEL 2

Wissensdatenbank

2.1 Kommunikation in der Ubuntu-Entwicklung

In einem Projekt, bei dem tausende Zeilen Quelltext geändert werden, viele Entscheidungen getroffen werden und hunderte Leute täglich zusammenspielen, ist es wichtig effizient zu kommunizieren.

2.1.1 Mailinglisten

Mailinglisten sind ein sehr bedeutendes Werkzeug um Ideen an ein größeres Team zu kommunizieren und sicherzustellen, dass man jeden erreicht, auch über Zeitzonen hinweg.

Bezüglich der Entwicklung, sind dies die Wichtigsten:

- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce (ausschließlich Ankündigungen, hier findet man nur die wichtigsten Ankündigungen bezüglich der Entwicklung)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel (allgemeine Diskussionen zur Ubuntu Entwicklung)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-motu (MOTU Team Diskussion, eine gute Anlaufstelle für Hilfe bei der Paketierung)

2.1.2 IRC-Kanäle

Für Echtzeit-Diskussionen schauen Sie auf dem IRC Server irc.freenode.net in einem oder mehreren der folgenden Kanäle vorbei:

- #ubuntu-devel (für allgemeine Diskussion zur Entwicklung)
- #ubuntu-motu (für MOTU Team Diskussionen und genereller Hilfe)

2.2 Allgemeine Übersicht über das `debian/` Verzeichnis

Dieser Artikel gibt eine kurze Übersicht über die verschiedenen Dateien im `debian/` Verzeichnis, welche für das Paketieren von Ubuntu Paketen wichtig sind. Die wichtigsten Dateien sind `changelog`, `control`, `copyright` und `rules`. Diese Dateien werden für alle Pakete benötigt. Anhand weiterer Dateien im `debian/` Verzeichnis kann das Verhalten der Pakete angepasst und konfiguriert werden. Während einige dieser Dateien in diesem Artikel beschrieben werden, ist er nicht als vollständige Übersicht gedacht.

2.2.1 Das Änderungsprotokoll

Die Datei ist, wie sich schon am Namen erkennen lässt, eine Liste von Änderungen die in jeder Version gemacht wurden. Sie hat ein spezielles Format aus dem man Paketname, Version, Distribution, Änderungen, Autor und Zeitpunkt herauslesen kann. Falls Sie einen GPG-Schlüssel besitzen (siehe: *Erste Schritte*), stellen Sie sicher dass Sie den selben Alias und E-Mail Adresse im `changelog` verwenden wie in Ihrem Schlüssel. Das folgende ist eine `changelog` Vorlage:

```
package (version) distribution; urgency=urgency

 * change details
  - more change details
 * even more change details

-- maintainer name <email address>[two spaces]  date
```

Das Format (besonders das Datumsformat) ist wichtig. Das Datum sollte im **RFC 5322** Format sein, sodass es dann durch Eingabe von `date -R` abgerufen werden kann. Zur Erleichterung kann der Befehl ‘‘dch‘‘genutzt werden, um den Changelog zu editieren. Er wird das Datum automatisch aktualisieren.

Unterpunkte werden durch einen Strich “-” dargestellt, während Hauptpunkte durch einen Stern “*” gekennzeichnet werden.

Falls Sie ein neues Paket ohne Vorlage erstellen, können Sie mit `dch --create` (`dch` befindet sich im `devscripts` Paket) Standard-Daten für `debian/changelog` anlegen.

Dies ist ein Beispiel für eine `changelog` Datei des hello Pakets:

```
hello (2.8-0ubuntu1) trusty; urgency=low

  * New upstream release with lots of bug fixes and feature improvements.

-- Jane Doe <packager@example.com>  Thu, 21 Oct 2013 11:12:00 -0400
```

Erwähnenswert ist, dass die Version ein `-0ubuntu1` Suffix angehängt hat, das die Distributions-Änderung wiederspiegelt. Sie wird benutzt damit das Paketieren in der gleichen Quellversion aktualisiert werden kann (z.B. für Fehlerbehebungen).

Ubuntu und Debian haben leicht unterschiedliche Versionsbezeichnungen um Konflikte mit demselben Quellpaket zu vermeiden. Wenn ein Debian-Paket unter Ubuntu geändert wurde, wird ein `ubuntuX` (wobei `X` für die Ubuntu-Revision steht) an das Ende der Debianversion angehängt. Also wenn das Debian-Paket hello `2.6-1` unter Ubuntu geändert wurde, würde die Versionsbezeichnung `2.6-1ubuntu1` sein. Falls ein Paket dieser Anwendung nicht für Debian existiert, dann ist die Debian-Revision `0` (z.B. `2.6-0ubuntu1`).

Für weitere Informationen, schauen Sie die changelog section (Section 4.4) des Debian Policy Manual an.

2.2.2 Die control Datei

Die `control` Datei enthält Informationen, die der Paketmanager (also z.B. `apt-get`, `synaptic` und `adept`) verwendet, build-spezifische Abhängigkeiten, Betreuerinformationen und vieles andere.

Für das Ubuntupaket `hello`, sieht die Datei `control` folgendermaßen aus:

```
Source: hello
Section: devel
Priority: optional
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Jane Doe <packager@example.com>
```



```
Standards-Version: 3.9.5
Build-Depends: debhelper (>= 7)
Vcs-Bzr: lp:ubuntu/hello
Homepage: http://www.gnu.org/software/hello/

Package: hello
Architecture: any
Depends: ${shlibs:Depends}
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting. It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them. Seriously, though: this is
 an example of how to do a Debian package. It is the Debian version of
 the GNU Project's 'hello world' program (which is itself an example
 for the GNU Project).
```

Der erste Absatz beschreibt in dem Feld `Build-Depends` das Quellpaket inklusive aller benötigten Paketabhängigkeiten, die nötig sind um das Paket aus dem Quellcode zu bauen. Es enthält außerdem einige Metadaten wie den Namen des Verantwortlichen, die Version der Debian-Richtlinien, der Ort der Paketversionkontrolle und die Upstream-Homepage.

Beachten Sie bitte, dass wir in Ubuntu das''Maintainer'' Feld auf eine allgemeine Adresse setzen, da jeder jedes Paket ändern kann (dies unterscheidet sich von Debian wo die Veränderung von Paketen auf eine Person oder ein Team beschränkt ist. Pakete in Ubuntu sollten allgemein das `Maintainer` Feld auf `Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>` setzen. Wenn das Maintainer-Feld modifiziert wird, sollte das alte Wert im `XSBC-Original-Maintainer` Feld gespeichert werden. Dies kann automatisch mit dem `update-maintainer` Skript im `ubuntu-dev-tools` Paket erfolgen. Für weitere Informationen schauen Sie Debian Maintainer Field spec im Ubuntu-Wiki an.

Jeder weitere Abschnitte beschreibt ein Binärpaket, welches gebaut wird.

Für weitere Informationen schauen Sie die control file Section (Kapitel 5) des Debian Policy Manual an.

2.2.3 Die Copyright-Datei

Diese Datei zeigt die Copyright Informationen für sowohl die Upstream Quelle und die Paketierung an. Ubuntu und Debian Policy (Sektion 12.5) verlangen, dass jedes Paket eine Originalkopie seiner Copyright and Lizenzinformation in `/usr/share/doc/$(package_name)/copyright` installiert.

Im allgemeinen findet man Urheberrechtsinformationen in der Datei `COPYING` in dem Quellverzeichnis des Programms. Diese Datei sollte Auskünfte über die Namen der Autoren und der Paketierer, die URL des ursprünglichen Programms, eine Zeile die das Jahr und den Inhaber der Urheberrechtsansprüche, und den Text des Urheberrechts an sich, geben. Ein Beispiel wäre:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: Hello
Source: ftp://ftp.example.com/pub/games

Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+

Files: debian/*
Copyright: Copyright 1998 Jane Doe <packager@example.com>
License: GPL-2+

License: GPL-2+
```

```
This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.
.
This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for more
details.
.
You should have received a copy of the GNU General Public
License along with this package; if not, write to the Free
Software Foundation, Inc., 51 Franklin St, Fifth Floor,
Boston, MA  02110-1301 USA
.
On Debian systems, the full text of the GNU General Public
License version 2 can be found in the file
'/usr/share/common-licenses/GPL-2'.
```

Dieses Beispiel folgt dem Machine-lesbaren debian/copyright Format. Dabei wollen wir ermutigen, dieses Format ebenfalls zu benutzen.

2.2.4 Die rules Datei

Die letzte Datei, die wir uns anschauen ist `rules`. Hier geschieht alle Arbeit, um das Paket zu erzeugen. Es ist ein Makefile mit Targets, um die Anwendung zu kompilieren und zu installieren, dann die `.deb` Datei von den installierten Dateien zu erzeugen. Es enthält auch ein Target um "aufzuräumen", so dass das Quellpaket wieder auf dem ursprünglichen Stand ist.

Hier ist eine vereinfachte Version der rules Datei, die von `dh_make` erzeugt wurde (erhältlich im `dh-make` Paket):

```
#!/usr/bin/make -f
# -*- makefile -*-

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

%:
	dh  $@
```

Lassen Sie uns durch diese Datei ein bisschen genauer durchgehen. Sie sorgt dafür, dass jedes Target, welches von `debian/rules` aufgerufen wird, als Argument an `/usr/bin/dh` weitergegeben wird, welches selbst wiederum alle nötigen `dh_*`-Befehle aufrufen wird.

`dh` durchläuft eine Sequenz von debhelper Kommandos. Die unterstützen Sequenzen korrespondieren mit den Targets einer `debian/rules`-Datei: "build", "clean", "install", "binary-arch", "binary-indep" und "binary". Um zu sehen, welche Kommandos als Teil welchen Targets durchlaufen werden, benutzen Sie:

```
$ dh binary-arch --no-act
```

Befehlen in der Sequenz binary-indep wird die Option "-i" mitgegeben um sicherzustellen, dass sie nur mit binär-unabhängigen Paketen funktionieren und Befehlen in der Sequenz binary-arch wird die Option "-a" mitgegeben um sicherzustellen, dass sie nur mit Architektur-unabhängigen Paketen funktionieren.

Jeder debhelper Befehl wird aufgenommen wenn er erfolgreich in `debian/package.debhelper.log` ausgeführt wird. (Welcher dh_clean löscht.) So kann dh mitteilen welche Befehle bereits für welches Paket ausgeführt

wurden und überspringt diejenigen, die nochmals ausgeführt werden sollen.

Jedes Mal wenn dh ausgeführt wird, untersucht es die Logdatei und findet den zuletzt verwendeten Befehl welcher in der gegebenen Sequenz enthalten ist. Es springt danach zum nächsten Befehl in der Sequenz. Die Optionen --until, --before, --after und --remaining können dieses Verhalten beeinflussen.

Falls debian/rules ein Target mit einem Namen wie override_dh_command enthält, dann wird dh, sobald es an dem Befehl in der Sequenz angekommen ist, das Target von dieser Anweisungsdatei verwenden statt den eigentlichen Befehl auszuführen. Das neue Target kann dann den Befehl mit anderen Optionen ausführen oder komplett andere Befehle stattdessen ausführen. (Zu beachten ist, dass Sie für die Erstellung debhelper 7.0.50 oder höher verwenden sollten.)

Werfen Sie einen Blick in /usr/share/doc/debhelper/examples/ und man dh für weitere Beispiele. Außerdem ist der Regelkatalog (Abschnitt 4.9) der Debian-Grundsatzanweisung hilfreich.

2.2.5 Zusätzliche Dateien

Die Datei install

Die Datei install wird von dh_install verwendet, um Dateien in das Binärpacket zu installieren. Es hat zwei gängige Einsatzmöglichkeiten:

- Um Dateien in Ihr Paket zu installieren, die nicht vom Upstream-Build-System installiert werden.
- Aufteilen eines einzelnen großen Quellpaketes in mehrere Binärpakete.

Im ersten Fall sollte die Datei install eine Zeile für jede installierte Datei enthalten, die sowohl das Datei- als auch Installationsverzeichnis festlegt. Zum Beispiel, die folgende install-Datei würde das das Skript foo in das Stammverzeichnis des Quellpakets nach usr/bin und eine Desktop-Datei in das debian-Verzeichnis nach usr/share/applications installieren:

```
foo usr/bin
debian/bar.desktop usr/share/applications
```

Wenn ein Quellpaket mehrere Binärpakete produziert, wird dh die Dateien in debian/tmp statt direkt in debian/<package> installieren. Dateien aus debian/tmp können dann in getrennte Binärpakete mithilfe mehrerer $package_name.install-Dateien verschoben werden. Dies wird oft dazu benutzt um große Mengen architekturunabhängiger Daten aus architekturabhängigen Paketen herauszulösen und sie in Architecture: all-Pakete zu integrieren. In diesem Fall werden nur der Name der zu installierenden Dateien (oder Ordner) benötigt und nicht das Installationsverzeichnis. Zum Beispiel könnte foo.install mit ausschließlich architekturabhängigen Dateien so ausschauen:

```
usr/bin/
usr/lib/foo/*.so
```

Während foo-common.install mit der architekturunabhängigen Datei so ausschauen könnte:

```
/usr/share/doc/
/usr/share/icons/
/usr/share/foo/
/usr/share/locale/
```

Dies würde zwei Binärpakete erzeugen, foo und foo-common. Beide würden ihren eigenen Abschnitt in debian/control benötigen.

Sehen Sie man dh_install und den Abschnitt zur Installationsdatei (Abschnitt 5.11) der Debian-Anleitung für neue Maintainers für zusätzliche Informationen.

Die Datei 'watch'

Die Datei debian/watch erlaubt uns automatisch mithilfe des Werkzeuges uscan in dem Paket devscripts zu überprüfen, ob neue Upstream-Versionen vorhanden sind. Die erste Zeile der "watch"-Datei muss die Format-Version bezeichnen (3, zum Zeipunkt dieser Textfassung), während die folgenden Zeilen die zu parsenden URLs enthalten. Zum Beispiel:

```
version=3

http://ftp.gnu.org/gnu/hello/hello-(.*).tar.gz
```

Der Befehl uscan im Wurzelverzeichnis des Quellcodes wird jetzt die Upstream-Versionsnummer in debian/changelog mit der neusten verfügbaren Upstream-Version vergleichen. Wenn eine neue Upstream-Version gefunden wurde, wird sie automatisch heruntergeladen. Zum Beispiel:

```
$ uscan
hello: Newer version (2.7) available on remote site:
  http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz
  (local version is 2.6)
hello: Successfully downloaded updated package hello-2.7.tar.gz
    and symlinked hello_2.7.orig.tar.gz to it
```

Wenn Ihre Tarballs auf Launchpad leben, ist die debian/watch etwas komplizierter (siehe Frage 21146 und Bug 231797 für die Gründe). Verwenden Sie in diesem Falle etwas wie:

```
version=3
https://launchpad.net/flufl.enum/+download http://launchpad.net/flufl.enum/.*/flufl.enum-(.+).tar.gz
```

Für weitere Informationen, schauen Sie man uscan und die watch file section (Section 4.11) im Debian Policy Manual an.

Um eine Liste der Pakete zu sehen, deren watch Datei eine neuere Version Upstream berichtet, schauen Sie sich Ubuntu External Health Status an.

Die Datei source/format

Diese Datei spezifiziert das Format des Quellpakets. Es soll eine einzige Zeile enthalten, die das gewünschte Format beschreibt.

- 3.0 (native) für native Debian-Pakete (keine Upstreamversion)
- 3.0 (quilt) für Pakete mit einem separaten Upstream-Tarball
- 1.0 für Pakete, die explizit das Standard-Format wünschen

Momentan wird das Paket-Quellformat standardmäßig auf 1.0 gesetzt, sollte die Datei nicht existieren. Das kann man jedoch auch explizit in der source/format Datei angeben. Sollten Sie sich dagegen entscheiden, wird lintian eine Warnung wegen der fehlenden Datei ausgeben. Diese Warnung hat lediglich Informationscharakter und kann sicher ignoriert werden.

Entwickler werden ermutigt, das neuere 3.0 Quellformat zu verwenden. Es stellt eine Reihe von Features bereit:

- Unterstützung für zusätzliche Komprimierungsformate: bzip2, lzma, xz
- Ünterstützung für mehrere Upstream-Tarballs
- Nicht nötig den Upsteam-Tarball neu zu packen um das Debian-Verzeichnis zu löschen
- Debian-spezifische Änderung sind nicht länger in einem einzelnen .diff.gz sondern stattdessen in mehreren Patches kompatibel mit quilt unter debian/patches/

https://wiki.debian.org/Projects/DebSrc3.0 fasst weitere zusätzliche Informationen bezüglich der Umstellung auf die 3.0 Quellpaketformate zusammen.

Siehe `man dpkg-source` sowie source/format section (Sektion 5.21) des Debian New Maintainers' Guide für weitere Details.

2.2.6 Weiterführende Quellen

Zusätzlich zu den Links im Debian Policy Manual in jeder übergeordneten Sektion bietet der Debian New Maintainers' Guide detailliertere Beschreibungen jeder Datei. Chapter 4, "Required files under the debian directory" beschreibt weiterhin die control, changelog, copyright und rules Dateien. Chapter 5, "Other files under the debian directory" beschreibt zusätzliche Dateien, die verwendet werden können.

2.3 ubuntu-dev-tools: Tools for Ubuntu developers

`ubuntu-dev-tools` package is a collection of 30 tools created for making packaging work much easier for Ubuntu developers. It's similar in scope to Debian `devscripts` package.

2.3.1 Setting up packaging environment

`setup-packaging-environment` command allows to interactively set up packaging environment, including setting environment variables, installing required packages and ensuring that required repositories are enabled.

2.3.2 Environment variables

Introducing yourself

`ubuntu-dev-tools` configurations can be set using environment variables. It's used for example in changelogs. For example, to set e-mail address (and full name), use `UBUMAIL` variable. It overrides the `DEBEMAIL` and `DEBFULLNAME` variables used by `devscripts`. To learn `ubuntu-dev-tools` about you, open *~/.bashrc* in text editor and add something like this:

```
export UBUMAIL="Marcin Mikołajczak <marcin@example.org>"
```

Now, save this file and restart your terminal or use `source ~/.bashrc`.

Changing preferred builder

Default builder is specified by `UBUNTUTOOLS_BUILDER` variable. To set between *pbuilder* (default), *pbuilder-dist*, and *sbuild*, change this variable. Example:

```
export UBUNTUTOOLS_BUILDER=sbuild
```

Save file and restart terminal.

You can also check whether to update the builder every time before building, by changing `UBUNTUTOOLS_UPDATE_BUILDER` from `no` (default) to `yes`.

2.3.3 Downloading source packages

`ubuntu-dev-tools` comes with `pull-lp-source` command, allowing to download source packages from Launchpad. Its usage is simple. To download latest source package for ubuntu-settings, use:

```
$ pull-lp-source ubuntu-settings
```

You can also specify release from which you want to download source or specify version of source package. `-d` option allows to download source package without extracting. A slightly more complex example would look like this:

```
$ pull-lp-source brisk-menu 0.5.0-1 -d
```

`pull-debian-source` package allows to do the same for Debian source packages. It has similar syntax.

2.3.4 Backporting packages

`ubuntu-dev-tools` provides `backportpackage` allowing us to backport a package from specified release of Ubuntu or Debian. For example, to backport `bzr` package from latest development release for your installed Ubuntu version, simply:

```
$ backportpackage -w . bzr
```

This command allows to use more options. To specify Ubuntu release for which you are going to backport a package, use `-d dest` or `--destination=DEST` parameter, where `DEST` is Ubuntu release, for example `xenial`. You can specify more than one destination. In turn, `-s SOURCE` and `--source=SOURCE` specifies the Ubuntu or Debian release from which you are going to backport a package. `-w DIR` and `--workdir=DIR` specifies directory, where package files will be downloaded, unpacked and built. By default, it will create temporary directory that will be automatically deleted. `-U` or `--update` allows to update build environment before building package. `-u` or `--upload` allows to upload package after building (for example to PPAs) using `dput`.

2.3.5 Requesting backports

`requestbackport` command makes creating backports through Launchpad bugs much easier. It creates testing checklist that will be included in the bug. For example, to request backporting libqt5webkit5 from latest development branch to current stable release (without optional parameters):

```
$ requestbackport libqt5webkit5
```

You should fulfill the checklist if you have already tested the backport.

Additional options allows to specify destination of backport and its source, by using `-d DEST` or `--destination=DEST` and `s SRC` or `--source=SRC`.

2.3.6 Other simple commands

`ubuntu-dev-tools` also includes small utilities allowing to do simple tasks like checking whether .iso file is an Ubuntu installation media.

ubuntu-iso

To do this, use `ubuntu-iso <pathtoiso>`, for example:

```
$ ubuntu-iso ~/Downloads/ubuntu.iso
```


bitesize

"Bitesize" tag is used on Launchpad to describe tasks that are suitable for begineers who want to contribute to one of the projects. `bitesize` command allows to add "bitesize" tag to Launchpad bug with just simple command, by providing its number, like:

```
$ bitesize 1735410
```

404main

`404main` allows to check whether all of package build dependencies are included in main repository of specified Ubuntu distribution. Example:

```
$ 404main libqt5webkit5 xenial
```

If any of the required packages isn't part of Ubuntu main repository, you can check whether the package fulfill Ubuntu main inclusion requirements and request it.

Further reading

`ubuntu-dev-tools` manpages are covering more about usage of this package.

2.4 autopkgtest: Automatische Tests für Pakete

Die DEP 8 Spezifikation definiert wie automatisches Testen sehr einfach in Paketen eingebunden werden kann. Alles was es braucht um einen Test in einem Paket einzubinden:

- erstellen Sie eine Datei namens `debian/tests/control`, die die Anforderungen für die Testumgebung festlegt,
- fügen Sie die Tests in `debian/tests/` ein.

2.4.1 Anforderungen für die Testumgebung

In `debian/tests/control` wird festgelegt, was die Testumgebung leisten muss. Zum Beispiel werden alle für den Test benötigten Pakete aufgeführt, ob die Testumgebung während der Erstellung verloren geht oder ob `root` Privilegien gebraucht werden. Die DEP 8 Spezifikation listet alle möglichen Optionen auf.

Im Folgenden schauen wir uns das Quellpaket `glib2.0` an. Im einfachsten Fall sieht die Datei so aus:

```
Tests: build
Depends: libglib2.0-dev, build-essential
```

Das stellt für den Test `debian/tests/build` sicher, dass die Pakete `libglib2.0-dev` und `build-essential` installiert sind.

Bemerkung: Sie können in der `Depends`-Zeile `@` benutzen, um festzulegen, dass Sie alle Pakete installiert haben wollen, die aus dem jeweiligen Quellpaket erzeugt werden.

2.4.2 Die eigentlichen Tests

Der passende Test für das obige Beispiel könnte folgender sein:

```
#!/bin/sh
# autopkgtest check: Build and run a program against glib, to verify that the
# headers and pkg-config file are installed correctly
# (C) 2012 Canonical Ltd.
# Author: Martin Pitt <martin.pitt@ubuntu.com>

set -e

WORKDIR=$(mktemp -d)
trap "rm -rf $WORKDIR" 0 INT QUIT ABRT PIPE TERM
cd $WORKDIR
cat <<EOF > glibtest.c
#include <glib.h>

int main()
{
    g_assert_cmpint (g_strcmp0 (NULL, "hello"), ==, -1);
    g_assert_cmpstr (g_find_program_in_path ("bash"), ==, "/bin/bash");
    return 0;
}
EOF

gcc -o glibtest glibtest.c `pkg-config --cflags --libs glib-2.0`
echo "build: OK"
[ -x glibtest ]
./glibtest
echo "run: OK"
```

An dieser Stelle wird ein sehr einfaches Stück C-Code in ein temporäres Verzeichnis geschrieben. Dies wird mit den Systembibliotheken übersetzt (wobei die Flags und Bibliothekpfade benutzt werden, wie sie uns von *pkg-config* geliefert werden). Dann wird der resultierende Binär-Code ausgeführt, der grundlegende Funtionen in glib testet.

Während der Test selbst sehr klein und einfach ist, umfasst er doch eine Menge: Es wird sichergestellt, dass das -dev Paket alle nötigen Abhängigkeiten besitzt, dass von dem Paket funktionierende pkg-Konfigurationsdateien installiert werden, dass die Header und Bibliotheken richtig platziert werden, oder dass der Kompiler und Linker funktionieren. Das hilft dabei, kritische Fehler schon sehr früh aufzudecken.

2.4.3 Den Test ausführen

Während das Testskript leicht selbst ausgeführt werden kann, wird dringend empfohlen, `autopkgtest` aus dem `autopkgtest` Paket zu verwenden. Somit kann überprüft werden, dass Ihr Test funktioniert. In anderen Fällen wird es der Test wenn er im Ubuntu Continuous Integration (CI) System fehlschlägt, nicht in Ubuntu landen. Dies verhindet eben so, dass Ihre Workstation mit Testpaketen oder Testkonfiguration zugemüllt wird, wenn der Test etwas weiteres als das einfache Beispiel oben macht.

Die README.running-tests (online version) Dokumentation erklärt alle verfügbaren Testumgebund (schroot, LXE, QEMU etc.) und die üblichsten Szenarien um Ihre Tests mit `autopkgtest` laufen zu lassen. Beispielsweise mit lokal erstellten Binärdateien, lokal modifizierten Tests etc.

Das Ubuntu CI-System benutzt den QEMU-Runner und führt alle Tests der Pakete in des Archivs aus, welche `-proposed` aktiviert haben. Um genau die selbe Umgebung zu reproduzieren, müssen zuerst die nötigen Pakten installiert werden:


```
sudo apt install autopkgtest qemu-system qemu-utils autodep8
```

Nun erstellen Sie eine Testumgebung mit:

```
autopkgtest-buildvm-ubuntu-cloud -v
```

(Bitte beachten Sie die Manpage und die `--help` Ausgabe zur Auswahl von verschiedenen Veröffentlichungen, Architekturen, Ausgabeverzeichnis oder die Nutzung von Proxies). Dies baut beispielsweise *adt-trusty-amd64-cloud.img*‘.

Dann starte die Tests eines Quellpakets wie `libpng` in diesem QEMU-Abbild:

```
autopkgtest libpng --- qemu adt-trusty-amd64-cloud.img
```

Das Ubuntu CI System startet Pakete nur mit ausgewählten Paketen von `-proposed` die verfügbar sind (das Paket welches den Start des Tests auslöst); um dies zu aktivieren starten Sie:

```
autopkgtest libpng -U --apt-pocket=proposed=src:foo --- qemu adt-release-amd64-cloud.img
```

oder mit allen Paketen von `-proposed` laufen lassen:

```
autopkgtest libpng -U --apt-pocket=proposed --- qemu adt-release-amd64-cloud.img
```

Die `autopkgtest` Manpage beinhaltet eine Menge an weiteren wertvollen Informationen über andere Testoptionen.

2.4.4 Weitere Beispiele

Diese Liste ist nicht vollständig, aber wird Ihnen sicher einen guten Einblick geben wie automatisierte Testdurchläufe in Ubuntu implementiert und benutzt werden.

- Die libxml2 Tests sind sehr ähnlich. Sie führen auch eine Testerstellung aus ein bisschen einfachem C-Code durch und führen sie aus.
- Die gtk+3.0 tests leisten auch einen Kompilier/Link/Start-Check im “build” Test. Es gibt einen zusätzlichen “python3-gi” Test welcher sicherstellt, dass die GTK Bibliothek auch über Introspektion genutzt werden kann.
- In den ubiquity Tests wird die Upstream Testsammlung ausgeführt.
- Die gvfs tests haben ausführliches Testen ihrer Funktionalität und sind sehr interessant, da die die Nutzung von CDs, Samba, DAV und anderen Teilen emulieren.

2.4.5 Ubuntu Infrastruktur

Pakete , die‘‘autopkgtest‘‘ aktiviert haben werden ihre Tests starten sobald sie hochgeladen werden oder eine ihrer Abhängigkeiten sich ändert. Die Ausgabe von automatically run autopkgtest tests kann im Web eingesehen werden und wird stetig aktualisiert.

Debian benutzt ebenfalls `autopkgtest` für Pakettests wobei dies momentan nur in schroots erfolgt. Somit können die Ergebnisse ein wenig variieren. Ergebnisse und Logs können auf http://ci.debian.net eingesehen werden. Bitte senden Sie daher jegliche Testbehebungen oder neue Tests gleichzeitig an Debian.

2.4.6 Die Tests in Ubuntu bekommen

Der Prozess, um einen autopkgtest in Ubuntu einzubringen ist größtenteils identisch mit *fixing a bug in Ubuntu*. Im Prinzip muss man einfach:

- `bzr branch ubuntu:<Paketname>` ausführen,

- bearbeiten Sie `debian/control` um die Tests zu aktivitieren,
- fügen Sie ein `debian/tests`-Verzeichnis hinzu,
- bearbeite `debian/tests/control` basierend auf der DEP 8 Spezifikation,
- fügen Sie Ihre(n) Test(s) zu `debian/tests` hinzu,
- die Änderungen committen, sie nach Launchpad hochladen, und einen Merge vorschlagen, sie dann überprüfen lassen, genau wie jede andere Änderung an einem Quellpaket auch.

2.4.7 Was Sie machen können

Das Ubuntu Engineering Team hat eine *Liste von benötigten Testfällen<requiredtests_>* zusammengestellt wo Pakete, die Tests benötigen in verschiedene Kategorien gestellt werden. Hier finden Sie Beispiele dieser Tests und können sie sich selbst zuweisen.

Sollten Probleme auftreten, kann man auf dem #ubuntu-quality IRC Kanal Kontakt zu Entwicklern herstellen, die helfen können.

2.5 Chroots benutzen

Wenn Sie eine Version von Ubuntu benutzen, aber an Paketen für andere Versionen arbeiten, können Sie eine Umgebung dieser Version mit einer `Chroot` erzeugen.

Eine `Chroot` erlaubt Ihnen ein Dateisystem einer anderen Distribution zu haben in dem man nahezu normal arbeiten kann. Dadurch vermeidet man den Overhead des Laufens einer vollen virtuellen Maschine.

2.5.1 Eine Chroot-Umgebung anlegen

Benutzen Sie das Kommando `debootstrap` um eine neue Chroot zu erzeugen:

```
$ sudo debootstrap trusty trusty/
```

Dies erstellt ein Verzeichnis `trusty` und installiert darin ein minimales Trusty-System.

Wenn Ihre Version von `debootstrap` Trusty nicht kennt, können Sie versuchen die Version in `backports` zu upgraden.

Sie können dann in der Chroot arbeiten:

```
$ sudo chroot trusty
```

Wo Sie alle Pakete installieren oder löschen können ohne Ihr Hauptsystem zu berühren.

Vielleicht möchten Sie auch Ihre GPG/ssh-Schlüssel und Bazaarkonfiguration in das Chroot kopieren, um so einfacher auf Pakete zugreifen und sie signieren zu können:

```
$ sudo mkdir trusty/home/<username>
$ sudo cp -r ~/.gnupg ~/.ssh ~/.bazaar trusty/home/<username>
```

Damit apt und andere Programme sich nicht mehr über fehlende Locales beschweren, kann man die relevanten Language Packs installieren:

```
$ apt-get install language-pack-en
```


Um X-Window Programme in der Chroot Umgebung nutzen zu können, muss das /tmp Verzeichnis in die Umgebung gebunden werden. Dies geht außerhalb des Chroot mit folgendem Kommando:

```
$ sudo mount -t none -o bind /tmp trusty/tmp
$ xhost +
```

Für einige Programme kann es nötig sein die Verzeichnisse /dev und /proc in das Chroot zu binden.

Für weitere Informationen über chroots schauen Sie unsere Debootstrap Chroot Wikiseite. an.

2.5.2 Alternativen

SBuild ist ein System ähnlich PBuilder um eine Umgebung zu erstellen in der Testpaket-Builds laufen gelassen werden können. Es ähnelt eher dem System welches von Launchpad genutzt wird um Pakete zu erstellen aber benötigt mehr Einrichtung als PBuilder. Schauen Sie die Security Team Build Environment Wikiseite für eine vollständige Erklärung an.

Vollständige virtuelle Maschinen können zur Paketierung und Testen von Programmen nützlich sein. TestDrive ist ein Programm um die Synchronisation und starten von Daily ISO-Abbildern zu betreiben. Schauen Sie die TestDrive Wikiseite für weitere Informationen an.

Sie können pbuilder auch so konfigurieren, dass er anhält, wenn er ein Problem findet. Kopieren Sie C10shell von /usr/share/doc/pbuilder/examples in ein Verzeichnis und benutzen Sie das `--hookdir=` Argument um darauf zu verweisen.

Amazons EC2 Cloudrechner ermöglichen es einen Rechner für ein paar US-Cents pro Stunde zu mieten. Man kann Ubuntu Rechner jeder unterstützten Version einrichten und auf diesen paketieren. Dies ist nützlich wenn man mehrere Pakete gleichzeitig paketieren möchte oder Bandbreiteneinschränkungen überwinden möchte.

2.6 Setting up sbuild

`sbuild` simplifies building Debian/Ubuntu binary package from source in clean environment. It allows to try debugging packages in environment similar (as opposed to `pbuild`) to builders used by Launchpad.

It works on different architectures and allows to build packages for other releases. It needs kernel supporting overlayfs.

2.6.1 Installing sbuild

To use sbuild, you need to install sbuild and other required packages and add yourself to the `sbuild` group:

```
$ sudo apt install debhelper sbuild schroot ubuntu-dev-tools
$ sudo adduser $USER sbuild
```

Create `.sbuildrc` in your home directory with following content:

```
# Name to use as override in .changes files for the Maintainer: field
# (mandatory, no default!).
$maintainer_name='Your Name <user@example.org>';

# Default distribution to build.
$distribution = "bionic";
# Build arch-all by default.
$build_arch_all = 1;

# When to purge the build directory afterwards; possible values are "never",
```



```
# "successful", and "always".  "always" is the default. It can be helpful
# to preserve failing builds for debugging purposes.  Switch these comments
# if you want to preserve even successful builds, and then use
# "schroot -e --all-sessions" to clean them up manually.
$purge_build_directory = 'successful';
$purge_session = 'successful';
$purge_build_deps = 'successful';
# $purge_build_directory = 'never';
# $purge_session = 'never';
# $purge_build_deps = 'never';

# Directory for writing build logs to
$log_dir=$ENV{HOME}."/ubuntu/logs";

# don't remove this, Perl needs it:
1;
```

Replace "Your Name <user@example.org>" with your name and e-mail address. Change default distribution if you want, but remember that you can specify target distribution when executing command.

If you haven't restarted your session after adding yourself to the `sbuild` group, enter:

```
$ sg sbuild
```

Generate GPG keypair for sbuild and create chroot for specified release:

```
$ sbuild-update --keygen
$ mk-sbuild bionic
```

This will create chroot for your current architecture. You might want to specify another architecture. For this, you can use `--arch` option. Example:

```
$ mk-sbuild xenial --arch=i386
```

2.6.2 Using schroot

Entering schroot

You can use `schroot -c <release>-<architecture> [-u <USER>]` to enter newly created chroot, but that's not exactly the reason why you are using sbuild:

```
$ schroot -c bionic-amd64 -u root
```

Using schroot for package building

To build package using sbuild chroot, we use (surprisingly) the `sbuild` command. For example, to build `hello` package from x86_64 chroot, after applying some changes:

```
apt source hello
cd hello-*
sed -i -- 's/Hello/Goodbye/g' src/hello.c   # some
sed -i -- 's/Hello/Goodbye/g' tests/hello-1 #
dpkg-source --commit
dch -i                                      #
update-maintainer                           # changes
sbuild -d bionic-amd64
```


To build package from source package (`.dsc`), use location of the source package as second parameter:

```
sbuild -d bionic-amd64 ~/packages/goodbye_*.dsc
```

To make use of all power of your CPU, you can specify number of threads used for building using standard `-j<threads>`:

```
sbuild -d bionic-amd64 -j8
```

2.6.3 Maintaining schroots

Listing chroots

To get list of all your sbuild chroots, use `schroot -l`. The `source:` chroots are used as base of new schroots. Changes here aren't recommended, but if you have specific reason, you can open it using something like:

```
$ schroot -c source:bionic-amd64
```

Updating schroots

To upgrade the whole schroot:

```
$ sbuild-update -ubc bionic-amd64
```

Expiring active schroots

If because of any reason, you haven't stopped your schroot, you can expire all active schroots using:

```
$ schroot -e --all-sessions
```

2.6.4 Further reading

There is Debian wiki page covering sbuild usage.

Ubuntu Wiki also has article about basics of sbuild.

`sbuild` manpages are covering details about sbuild usage and available features.

2.7 KDE Paketierung

Paketierung von KDE Programmen in Ubuntu wird vom Kubuntu und MOTU-Teams verwaltet. Sie können das Kubuntu Team auf der Kubuntu Mailingliste und `#kubuntu-devel` Freenode IRC Kanal kontaktieren. Mehr Informationen über Kubuntu-Entwicklung ist auf der Kubuntu Wikiseite.

Unsere Paketierung folgt der Praxis des Debian Qt/KDE Team und dem Debian KDE Extras Team. Die meisten unserer Pakete werden aus der Paketierung dieser Debian-Teams abgeleitet.

2.7.1 Richtlinien für Fehlerkorrekturen

Kubuntu führt keine Fehlerkorrekturen an KDE-Programmen durch solange sie nicht von Upstream-Autoren oder dortigen Einreichungen stammen, mit der Absicht sie bald in das Programm einfließen zu lassen, oder das Problem mit den Upstream-Autoren abgesprochen wurde.

Kubuntu ändert keine Paketbezeichnungen mit Ausnahme dort, wo Upstream dies erwartet (wie das Logo des Kickoff-Menüs links oben) oder es der Vereinfachung dient (wie der Entfernung des Begrüßungsbildschirms).

2.7.2 debian/rules

Debian-Pakete verwenden Zusätze zur herkömmlichen Debhelper-Verwendung. Diese sind im Paket *pkg-kde-tools‘* enthalten.

Pakete, welche Debhelper 7 verwenden, sollten die Option `--with=kde` anhängen. Dies stellt sicher, dass die richtigen Flags zum Bauen benutzt werden und Optionen zum Umgang mit kdeinit-Stubs and Übersetzungen hinzugefügt werden:

```
%:
    dh $@ --with=kde
```

Einige neuere KDE-Packete verwenden das `dhmk`-System, eine alternative zu `dh`, welches von dem Debian Qt/KDE-Team entwickelt wurde. Sie können sich darüber in /usr/share/pkg-kde-tools/qt-kde-team/2/README informieren. Pakete die dieses benutzen, werden `/usr/share/pkg-kde-tools/qt-kde-team/2/debian-qt-kde.mk enthalten` anstelle des Ausführens von `dh`.

2.7.3 Übersetzungen

Übersetzungen von Paketen in main werden in Launchpad importiert und werden von Launchpad in Ubuntu’s Language-Packs exportiert.

Daher muss jedes KDE-Paket Übersetzungsvorlagen generieren, Upstream-Übersetzungen verwenden oder bereitstellen und Übersetzungen für `.desktop`-Dateien interpretieren können.

Um Übersetzungsvorlagen zu generieren, muss das Paket eine `Message.sh` Datei einbinden. Wenn dem nicht so ist, beschweren Sie sich Upstream. Ob es funktioniert, können Sie überprüfen, in dem Sie `extract-messages.sh` ausführen, welches eine oder mehrere Dateien namens `.pot` in `po/` erstellen sollte. Dies wird während des Bauens automatisch getan, wenn Sie `dh``mit der Option ``--width=kde` benutzen.

Upstream wird normalerweise auch die Übersetzungs `.po` Dateien in das `po/` Verzeichnis packen müssen. Falls sie das nicht tun, prüfen Sie ob diese in separaten Upsteam Sprachpaketen wie dem KDE SC Sprachpaket sind. Wenn diese in separaten Sprachpaketen sind, muss Launchpad diese manuell zusammenfügen. Kontaktieren Sie David Planella dies durchzuführen.

Wenn ein Paket von universe zu main verschoben wird, muss es erneut hochgeladen werden bevor die Übersetzung in Launchpad importiert wird.

```
.desktop``Dateien benötigen ebenfalls Übersetzungen. Wir patchen KDELibs,
dass die Übersetzungen aus ``.po Dateien geholt werden, auf die durch eine Zeile
X-Ubuntu-Gettext-Domain= gezeigt wird und den .desktop``Dateien zur Buildzeit
hinzugefügt werden. Eine .pot-Datei für jedes Paket wird während des
Builds erzeugt und die .po-Dateien müssen von Upstream heruntergeladen
und in das Paket bzw. unsere Sprachpakete gebracht werden. Die Liste von
.po-Dateien, die aus KDEs Repositories heruntergeladen werden soll, ist in
``/usr/lib/kubuntu-desktop-i18n/desktop-template-list.
```


2.7.4 Bibliothekssymbole

Bibliothekssymbole werden in `.symbols` Dateien verfolgt um sicherzustellen, dass keine für neue Veröffentlichungen verloren gehen. KDE verwendet C++ Bibliotheken die etwas anders funktionieren als C-Bibliotheken. Debian's Qt/KDE Team hat Skripte, die dies bewältigen. Schauen Sie sich Working with symbols files wie man diese Dateien erstellt und aktuell hält.

KAPITEL 3

Weiterführende Literatur

Sie können dieses Handbuch offline in verschiedenen Formaten lesen, wenn man eins der Binärpakete installiert.

Wenn Sie mehr über das Bauen von Debian-Paketen lernen wollen, sind hier einige Debian-Resourcen, die Sie hilfreich finden könnten.

- Wie man für Debian paketiert;
- Debian Policy Manual;
- Debian New Maintainers' Guide — verfügbar in vielen Sprachen;
- Paketierungskurs (auch als Paket) verfügbar;
- Handbuch zur Paketierung von Python Modulen.

Wir suchen ständig nach Wegen, dieses Handbuch zu verbessern. Wenn Sie auf Probleme stossen oder einige Vorschläge haben, bitte melden Sie einen Bug auf Launchpad. Wenn Sie gerne an dem Hanbuch arbeiten möchten holen Sie die Quellen ebenfalls dort.