# Ubuntu Packaging Guide

***Release 1.0.0 bzr655 ubuntu14.04.1***

**Ubuntu Developers**

09.04.2018

Inhaltsverzeichnis

Welcome to the Ubuntu Packaging and Development Guide! We are currently developing codename Bionic Beaver, which is to be released in April 2018 as Ubuntu 18.04 LTS.

This is the official place for learning all about Ubuntu Development and packaging. After reading this guide you will have:

- Heard about the most important players, processes and tools in Ubuntu development,
- Your development environment set up correctly,
- A better idea of how to join our community,
- Fixed an actual Ubuntu bug as part of the tutorials.

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

Das heißt, Du kannst 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 Du mit machen willst. Wir freuen uns über Fragen und wenn Du anfängst, Ubuntu zusammen mit uns besser zu machen.

Solltest Du Probleme haben: keine Panik! Schau Dir den *Artikel über Kommunikation* an und Du wirst sehen, wie Du am leichtesten in Kontakt mit anderen Entwicklern kommst.

Die Anleitung teilt sich in zwei Bereiche auf:

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

KAPITEL 1

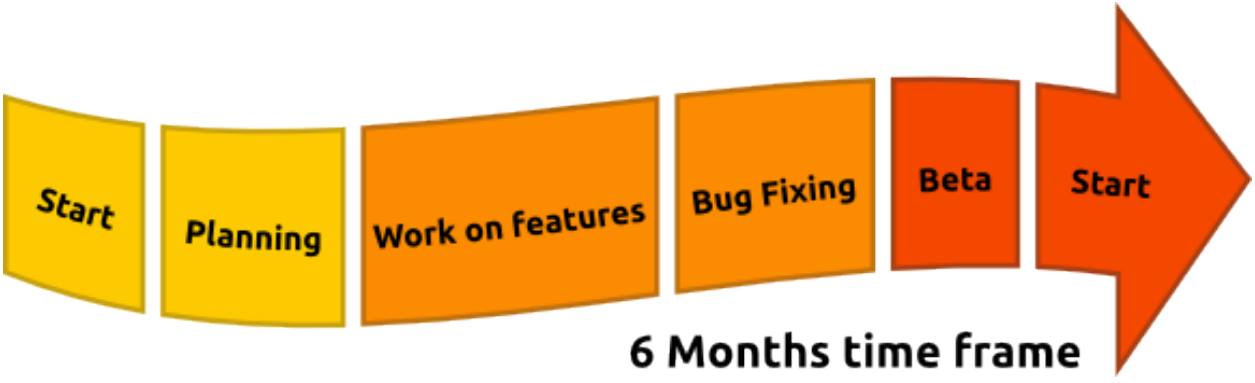
# Artikel

## 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.

Every time a new version of an application is released, or when someone makes a change to the source code that goes into Ubuntu, the source package must be uploaded to Launchpad's build machines to be compiled. The resulting binary packages then are distributed to the archive and its mirrors in different countries. The URLs in `/etc/apt/sources.list` point to an archive or mirror. Every day images are built for a selection of different Ubuntu flavours. They can be used in various circumstances. There are images you can put on a USB key, you can burn them on DVDs, you can use netboot images and there are images suitable for your phone and tablet. Ubuntu Desktop, Ubuntu Server, Kubuntu and others specify a list of required packages that get on the image. These images are then used for installation tests and provide the feedback for further release planning.

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.

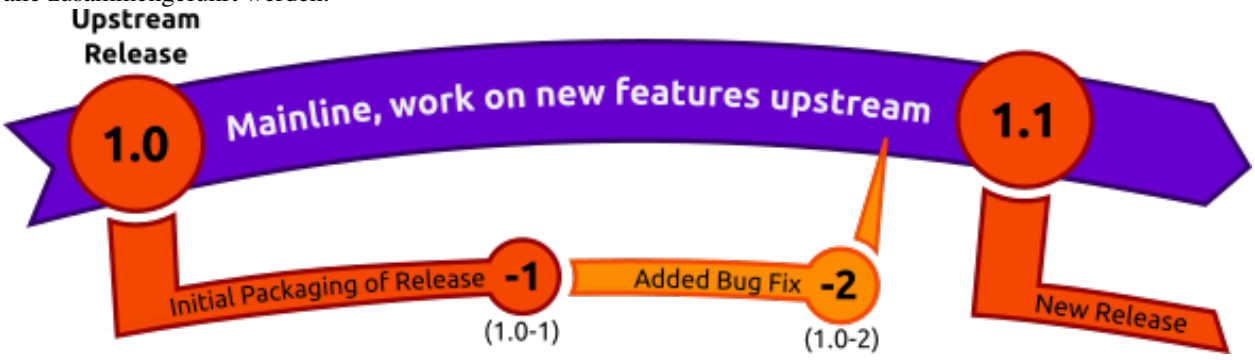
Ein weiteres wichtiges Werkzeug, welches die Kommunikation betrifft ist, sind Fehlerberichte. Wann immer ein Defekt in einem Paket oder einem Teil der Infrastruktur gefunden wird, wird ein Fehlerbericht auf Launchpad eingereicht. Jede Information ist diesem Bericht vereinigt und seine Wichtigkeit, Status 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 is 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, musst Du dir 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 du das Paket getestet hast, kannst du deine Änderung einfach zur Aufnahme in den aktuellen Ubuntu Entwicklungszweig vorschlagen. Ein Entwickler mit dem Recht zum hochladen wird deine Änderung für dich bewerten und anschließend es für dich in Ubuntu integrieren.


Wenn Du eine Lösung suchst 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.

Good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on `freenode`.. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

## 1.2 Die Programme einrichten

Es gibt einiges zu tun, bevor Du mit der Ubuntu-Entwicklung loslegen kannst. Dieser Artikel wird Dir helfen, dein System so einzurichten, dass Du mit Paketen arbeiten und Deine Pakete auf Ubuntu's Hosting-Platform, Launchpad, hochladen kannst. Hierüber werden wir reden:

- Paketierungs-Software installieren. Dies beinhaltet:
  - Ubuntu-spezifische Paketierungs-Werkzeuge
  - Verschlüsselungssoftware so dass Deine Arbeit als Deine eigene verifiziert werden kann
  - Weitere Verschlüsselungssoftware so dass Du sichere Dateitransfers machen kannst
- Deinen Account auf Lauchpad erstellen und einrichten
- Deine Entwicklungsumgebung aufsetzen, so dass Du lokale Builds von Paketen machen, mit anderen Entwicklern interagieren und Deine Änderungswünsche in Launchpad unterbreiten kannst.

---

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

Don't want to install the latest development version of Ubuntu? Spin up an LXD container.

---

### 1.2.1 Grundlegende Paketierungs-Software installieren

There are a number of tools that will make your life as an Ubuntu developer much easier. You will encounter these tools later in this guide. To install most of the tools you will need run this command:

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

Dieser Befehl wird folgende Software installieren

- `gnupg` – GNU Privacy Guard contains tools you will need to create a cryptographic key with which you will sign files you want to upload to Launchpad.
- `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.

#### Deinen GPG-Schlüssel erstellen

GPG stands for GNU Privacy Guard and it implements the OpenPGP standard which allows you to sign and encrypt messages and files. This is useful for a number of purposes. In our case it is important that you can sign files with your key so they can be identified as something that you worked on. If you upload a source package to Launchpad, it will only accept the package if it can absolutely determine who uploaded the package.

Um Deinen GPG-Schlüssel zu erstellen, starte:

```
$ 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 dich generieren, was ein bisschen dauern kann. Es braucht zufällige Bytes, also ist eine gute Idee das System ein wenig auszulasten. Beweg den Mauszeiger hin und her, schreib ein paar Zeilen und lade ein paar Webseiten.

Wenn das getan ist, wirst Du 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 muss du den öffentlichen Teil deines Schlüssels auf einen Keyserver hochladen, sodass man Nachrichten und Dateien dir zuordnen kann. Um das zu tun, führe folgenden Befehl aus:

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

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

#### Deinen 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 Deinen SSH-Schlüssel zu erstellen, starte:

```
$ 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.
- Du brauchst nicht alle nötigen Build-Abhängigkeiten lokal installieren.
- Du kannst mehrere Instanzen für diverse Ubuntu- und Debian-Versionen einrichten

`pbuilder` einzurichten ist sehr einfach, starte:

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

where <release> is for example *xenial*, *zesty*, *artful* or in the case of Debian maybe *sid* or *buster*. This will take a while as it will download all the necessary packages for a "minimal installation". These will be cached though.

### 1.2.2 Alles einrichten um mit Launchpad arbeiten zu können

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

- Was Launchpad ist und einen Launchpad Account erstellen
- Deinen GPG- und SSH-Schlüssel auf Launchpad hochladen
- Configure your shell to recognize you (for putting your name in changelogs)

#### 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. Du wirst Launchpad ebenso dazu benutzen um deine Lösungen zu veröffentlichen und andere Ubuntu-Entwickler dazu zu bringen sie zu überprüfen und zu finanzieren.

Du wirst dich mit Launchpad anmelden und ein Minumum an Informationen preisgeben müssen. Das gibt dir 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 siehe im Launchpad Hilfe Wiki.

### Einen Launchpad Account erstellen

If you don't already have a Launchpad account, you can easily create one. If you have a Launchpad account but cannot remember your Launchpad id, you can find this out by going to https://launchpad.net/~ and looking for the part after the ~ in the URL.

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 dich besser kennenzulernen.

Sobald du einen neuen Account registrierst, wird dir Launchpad eine E-Mail mit einem Weblink senden, der deine E-Mail-Adresse bestätigt. Falls du keine E-Mail erhältst, solltest du deinen Spam-Ordner überprüfen.

The new account help page on Launchpad has more information about the process and additional settings you can change.

### Deinen GPG-Schlüssel auf Launchpad hochladen

First, you will need to get your fingerprint and key ID.

Um Deinen GPG-Fingerabdruck zu finden, starte:

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

und es wird Dir 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
```

Then run this command to submit your key to Ubuntu keyserver:

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

where `43CDE61D` should be replaced by your key ID (which is in the first line of output of the previous command). Now you can import your key to Launchpad.

Gehe auf https://launchpad.net/~/+editpgpkeys und kopiere den "Key fingerprint" in das Textfeld. Im oben genannten Beispiel wäre das `5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D`. Klicke 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 encrypts the email, using your public key, so that it can be sure that the key is yours. If you are using Thunderbird, the default Ubuntu email client, you can install the Enigmail plugin to easily decrypt the message. If your email software does not support OpenPGP encryption, copy the encrypted email's contents, type `gpg` in your terminal, then paste the email contents into your terminal window.

Zurück auf der Launchpad-Webseite, drück die Schaltfläche zur Bestätigung und Launchpad wird den Import deines OpenPGP-Schlüssels abschließen.

Weitere Informationen findest Du auf https://help.launchpad.net/YourAccount/ImportingYourPGPKey

#### Deinen SSH-Schlüssel auf Launchpad hochladen

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

For more information on this process, visit the creating an SSH keypair page on Launchpad.

#### Deine Shell einrichten

The Debian/Ubuntu packaging tools need to learn about you as well in order to properly credit you in the changelog. Simply open your *~/.bashrc* in a text editor and add something like this to the bottom of it:

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

Speichere die Datei jetzt und starte entweder Dein Terminal neu oder starte:

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

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

## 1.3 Einen Bug in Ubuntu beheben

### 1.3.1 Einleitung

Wenn Du die Anweisungen in *Sich für Ubuntu-Entwicklung einrichten* befolgt hast, solltest Du 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.

Take a look at the bitesize bugs in Launchpad, and that might give you an idea of something to work on. It might also interest you to look at the bugs triaged by the Ubuntu One Hundred Papercuts team.

### 1.3.3 Herausfinden, was repariert werden muss

Wenn Du das Quellpaket, das den Fehler beinhaltet nicht kennst, aber Dir der Pfad zu dem betroffenen Programm auf Deinem System bekannt ist, dann kannst Du das Quellpaket selbst ermitteln, an dem Du arbeiten musst.

Let's say you've found a bug in Bumprace, a racing game. The Bumprace application can be started by running `/usr/bin/bumprace` on the command line. To find the binary package containing this application, use this command:

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

Dies gibt aus:

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

Note that the part preceding the colon is the binary package name. It's often the case that the source package and binary package will have different names. This is most common when a single source package is used to build multiple different binary packages. To find the source package for a particular binary package, type:

```
$ 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 bestaetigen

Once you have figured out which package the problem is in, it's time to confirm that the problem exists.

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 waere `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 Du lösen wolltest, 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 du im Web danach suchst, sollte es in den meisten Fällen einfach zu finden sein.

### 1.3.6 Hilfe anbieten

Wenn du einen offenen Fehler gefunden hast, dieser noch nicht zugewiesen ist und du die Möglichkeit hast ihn zu beheben, solltest du einen Kommetar mit deinem Lösungsvorschlag abgeben. Gib so viele Informationen wie möglich an: Unter welchen Umständen tritt der Fehler auf? Wie hast du den Fehler behoben? Hast du die Lösung getestet?

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

Es ist toll wenn Du Hilfe anbieten kannst und man wird dafür sehr dankbar sein.

### 1.3.7 Den Quellcode herunterladen

Once you know the source package to work on, you will want to get a copy of the code on your system, so that you can debug it. The ubuntu-dev-tools package has a tool called `pull-lp-source` that a developer can use to grab the

source code for any package. For example, to grab the source code for the tomboy package in `xenial`, you can type this:

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

If you do not specify a release such as `xenial`, it will automatically get the package from the development version.

Once you've got a local clone of the source package, you can investigate the bug, create a fix, generate a debdiff, and attach your debdiff to a bug report for other developers to review. We'll describe specifics in the next sections.

### 1.3.8 Arbeiten an einer Fehlerbehebung

Es wurden ganze Bücher darüber verfasst wie man Fehler findet, sie behebt, testet, usw. Wenn du ein Anfänger im Programmieren bist, versuche zuerst einfache Fehler wie beispielsweise in der Rechtschreibung zu beheben. Versuche die Änderungen so minimal wie möglich zu halten und dokumentiere deine Änderungen und Annahmen übersichtlich.

Bevor du dich daran machst, einen Fehler selbst du behen, solltest du 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

You may want to create a patch which includes the fix. The command `edit-patch` is a simple way to add a patch to a package. Run:

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

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

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

Nachdem du die Datei bearbeitet hast, schreibe `exit` oder drücke `Strg+D` um die zeitweilige Umgebung zu verlassen. Der neue Patch wird dann unter `debian/patches` eingefügt.

You must then add a header to your patch containing meta information so that other developers can know the purpose of the patch and where it came from. To get the template header that you can edit to reflect what the patch does, type this:

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

This will open the template in a text editor. Follow the template and make sure to be thorough so you get all the details necessary to describe the patch.

In this specific case, if you just want to edit `debian/control`, you do not need a patch. Put `Homepage: http://www.linux-games.com/bumprace/` at the end of the first section and the bug should be fixed.

#### Die Lösung dokumentieren

Es ist sehr wichtig Deine Änderung ausreichend zu dokumentieren, so dass Entwickler, die sich den Code zukünftig ansehen, nicht raten müssen, was Deine 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 Dich erstellen, wo Du die Lücken ausfüllen kannst. 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 dich 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 Änderungsprotokolles selbst konzentrieren: Es ist sehr wichtig festzuhalten:

1. Where the change was done.
2. What was changed.
3. Where the discussion of the change happened.

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.

In order to get it sponsored in the next section, you need to file a bug report in Launchpad (if there isn't one already, if there is, use that) and explain why your fix should be included in Ubuntu. For example, for tomboy, you would file a bug here (edit the URL to reflect the package you have a fix for). Once a bug is filed explaining your changes, put that bug number in the changelog.

### 1.3.9 Die Lösung testen

Um ein Testpaket mit Deinen Änderungen zu bauen, durchlaufe diese Kommandos:

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

This will create a source package from the branch contents (`-us -uc` will just omit the step to sign the source package and `-d` will skip the step where it checks for build dependencies, pbuilder will take care of that) and `pbuilder-dist` will build the package from source for whatever `release` you choose.

---

**Bemerkung:** If `debuild` errors out with "Version number suggests Ubuntu changes, but Maintainer: does not have Ubuntu address" then run the `update-maintainer` command (from ubuntu-dev-tools) and it will automatically fix this for you. This happens because in Ubuntu, all Ubuntu Developers are responsible for all Ubuntu packages, while in Debian, packages have maintainers.

---

In this case with bumprace, run this to view the package information:

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

As expected, there should now be a `Homepage:` field.

---

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

### 1.3.10 Submitting the fix and getting it included

With the changelog entry written and saved, run `debuild` one more time:

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

and this time it will be signed and you are now ready to get your diff to submit to get sponsored.

In a lot of cases, Debian would probably like to have the patch as well (doing this is best practice to make sure a wider audience gets the fix). So, you should submit the patch to Debian, and you can do that by simply running this:

```
$ submittodebian
```

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

Kommunikation ist wichtig, also solltest Du mehr Beschreibung mitliefern, wenn Du darum bittest die Änderung einzupflegen. Sei freundlich, erkläre es ausreichend.

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

It might be beneficial to just get it included in Debian and have it flow down to Ubuntu, in which case you would not follow the below process. But, sometimes in the case of security updates and updates for stable releases, the fix is already in Debian (or ignored for some reason) and you would follow the below process. If you are doing such updates, please read our *Security and stable release updates* article. Other cases where it is acceptable to wait to submit patches to Debian are Ubuntu-only packages not building correctly, or Ubuntu-specific problems in general.

But if you're going to submit your fix to Ubuntu, now it's time to generate a "debdiff", which shows the difference between two Debian packages. The name of the command used to generate one is also `debdiff`. It is part of the `devscripts` package. See `man debdiff` for all the details. To compare two source packages, pass the two dsc files as arguments:

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

In this case, `debdiff` the dsc you downloaded with `pull-lp-source` and the new dsc file you generated. This will generate a patch that your sponsor can then apply locally (by using `patch -p1 < /path/to/debdiff`). In this case, pipe the output of the debdiff command to a file that you can then attach to the bug report:

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

The format shown in `1-1.0-1ubuntu1.debdiff` shows:

1. `1-` tells the sponsor that this is the first revision of your patch. Nobody is perfect, and sometimes follow-up patches need to be provided. This makes sure that if your patch needs work, that you can keep a consistent naming scheme.
2. `1.0-1ubuntu1` shows the new version being used. This makes it easy to see what the new version is.
3. `.debdiff` is an extension that makes it clear that it is a debdiff.

While this format is optional, it works well and you can use this.

Next, go to the bug report, make sure you are logged into Launchpad, and click "Add attachment or patch" under where you would add a new comment. Attach the debdiff, and leave a comment telling your sponsor how this patch can be applied and the testing you have done. An example comment can be:

```
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.
```

Make sure you mark it as a patch (the Ubuntu Sponsors team will automatically be subscribed) and that you are subscribed to the bug report. You will then receive a review anywhere between several housr from submitting the patch to several weeks. If it takes longer than that, please join `#ubuntu-motu` on `freenode` and mention it there. Stick around until you get an answer from someone, and they can guide you as to what to do next.

Once you have received a review, your patch was either uploaded, your patch needs work, or is rejected for some other reason (possibly the fix is not fit for Ubuntu or should go to Debian instead). If your patch needs work, follow the same steps and submit a follow-up patch on the bug report, otherwise submit to Debian as shown above.

Remember: good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on freenode. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

### 1.3.11 Weitere Überlegungen

Wenn Du ein Paket findest und es sich herausstellt, dass Du mehrere triviale Dings darin beheben kannst, mach es ruhig in einer Änderung. Das wird die Code-Review und das Einpflegen beschleunigen.

Sollte es mehrere große Dinge geben, die Du beheben willst, mag es Sinn machen 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-Artikle 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 du nicht die benötigten Bibliotheken und Entwicklungsdateien besitzt. Installiere die benötigten Pakete und Entwicklungsdateien bis der Befehl erfolgreich ausgeführt wird.:

```
$ ./configure
```

Jetzt kannst Du 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
```

Du solltest 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`.

Much of the package building work is done by a series of scripts called `debhelper`. The exact behaviour of `debhelper` changes with new major versions, the compat file instructs `debhelper` which version to act as. You will generally want to set this to the most recent version which is `9`.

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 deiner Meinung nach in dem entgültigen Paket enthalten sein sollten.

`README.source` und `README.Debian` werden nur benötigt, wenn dein 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 kommitte den Code zu deinem 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.

Du kannst Dir den Inhalt eines Paketes mit folgendem Befehl ansehen:

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

Install the package and check it works (later you will be able to uninstall it using `sudo apt-get remove hello` if you want):

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

You can also install all packages at once using:

```
$ sudo debi
```

### 1.4.4 Nächste Schritte

Even if it builds the .deb binary package, your packaging may have bugs. Many errors can be automatically detected by our tool `lintian` which can be run on the source .dsc metadata file, .deb binary packages or .changes file:

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

To see verbose description of the problems use `--info` lintian flag or `lintian-info` command.

For Python packages, there is also a `lintian4python` tool that provides some additional lintian checks.

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
```

You can ask for reviews in `#ubuntu-motu` IRC channel, or on the MOTU mailing list. There might also be a more specific team you could ask such as the GNU team for more specific questions.

### 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 is for the mentoring of new and prospective Debian Developers. It is where you can find a sponsor to upload your package to the archive.
- Work-Needing and Prospective Packages - Information on how to file “Intent to Package” and “Request for Package” bugs as well as list of open ITPs and RFPs.
- Debian Developer’s Reference, 5.1. New packages - The entire document is invaluable for both Ubuntu and Debian packagers. This section documents processes for submitting new packages.

In some cases, it might make sense to go directly into Ubuntu first. For instance, Debian might be in a freeze making it unlikely that your package will make it into Ubuntu in time for the next release. This process is documented on the “New Packages” section of the Ubuntu wiki.

### 1.4.6 Bildschirmfotos

Hast du einmal ein Paket zu Debian hochgeladen, solltest du 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

Fixing security bugs in Ubuntu is not really any different than *fixing a regular bug in Ubuntu*, and it is assumed that you are familiar with patching normal bugs. To demonstrate where things are different, we will be updating the dbus package in Ubuntu 12.04 LTS (Precise Pangolin) for a security update.

### Den Quelltext bekommen

In this example, we already know we want to fix the dbus package in Ubuntu 12.04 LTS (Precise Pangolin). So first you need to determine the version of the package you want to download. We can use the `rmadison` to help with this:

```
$ 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
```

Typically you will want to choose the highest version for the release you want to patch that is not in -proposed or -backports. Since we are updating Precise's dbus, you'll download 1.4.18-1ubuntu1.4 from precise-updates:

```
$ 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
```

This will apply the existing patches and put the packaging in a temporary directory. Now edit the files needed to fix the vulnerability. Often upstream will have provided a patch so you can apply that patch:

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

After making the necessary changes, you just hit Ctrl-D or type exit to leave the temporary shell.

### Das Changelog und die Patches formatieren

After applying your patches you will want to update the changelog. The `dch` command is used to edit the `debian/changelog` file and `edit-patch` will launch `dch` automatically after un-applying all the patches. If you are not using `edit-patch`, you can launch `dch -i` manually. Unlike with regular patches, you should use the following format (note the distribution name uses precise-security since this is a security update for Precise) for security updates:

```
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]
...
```

Update your patch to use the appropriate patch tags. Your patch should have at a minimum the Origin, Description and Bug-Ubuntu tags. For example, edit debian/patches/99-fix-a-vulnerability.patch to have something like:

```
## 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
...
```

Multiple vulnerabilities can be fixed in the same security upload; just be sure to use different patches for different vulnerabilities.

#### Deine Arbeit testen und einsenden

At this point the process is the same as for *fixing a regular bug in Ubuntu*. Specifically, you will want to:

1. Build your package and verify that it compiles without error and without any added compiler warnings
2. Upgrade to the new version of the package from the previous version
3. Test that the new package fixes the vulnerability and does not introduce any regressions
4. Submit your work via a Launchpad merge proposal and file a Launchpad bug being sure to mark the bug as a security bug and to subscribe `ubuntu-security-sponsors`

If the security vulnerability is not yet public then do not file a merge proposal and ensure you mark the bug as private.

The filed bug should include a Test Case, i.e. a comment which clearly shows how to recreate the bug by running the old version then how to ensure the bug no longer exists in the new version.

The bug report should also confirm that the issue is fixed in Ubuntu versions newer than the one with the proposed fix (in the above example newer than Precise). If the issue is not fixed in newer Ubuntu versions you should prepare updates for those versions too.

### 1.5.2 Updates fuer stabile Releases

We also allow updates to releases where a package has a high impact bug such as a severe regression from a previous release or a bug which could cause data loss. Due to the potential for such updates to themselves introduce bugs we only allow this where the change can be easily understood and verified.

The process for Stable Release Updates is just the same as the process for security bugs except you should subscribe `ubuntu-sru` to the bug.

The update will go into the `proposed` archive (for example `precise-proposed`) where it will need to be checked that it fixes the problem and does not introduce new problems. After a week without reported problems it can be moved to `updates`.

See the Stable Release Updates wiki page for more information.

## 1.6 Patches für Pakete

Sometimes, Ubuntu package maintainers have to change the upstream source code in order to make it work properly on Ubuntu. Examples include, patches to upstream that haven't yet made it into a released version, or changes to the upstream's build system needed only for building it on Ubuntu. We could change the upstream source code directly, but doing this makes it more difficult to remove the patches later when upstream has incorporated them, or extract the change to submit to the upstream project. Instead, we keep these changes as separate patches, in the form of diff files.

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
```

The patches are kept in `debian/patches`. This package has one patch `kubuntu_01_fix_qmax_on_armel.diff` to fix a compile failure on ARM. The patch has been given a name to describe what it does, a number to keep the patches in order (two patches can overlap if they change the same file) and in this case the Kubuntu team adds their own prefix to show the patch comes from them rather than from Debian.

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

### 1.6.1 Patches mit Quilt

Bevor Du mit Quilt arbeitest, musst Du spezifizieren, wo die Patches liegen. Füge dies in Deine `~/.bashrc` hinzu:

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

Und verwende `source`, um die neuen Exports anzuwenden:

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

Standardmäßig werden alle Patches angewandt, wenn Du UDD Checkouts or heruntergeladene Quellpakete benutzt. Du kannst dies folgendermaßen überprüfen:

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

Wenn Du den Patch entfernen wolltest, würdest du `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, verwendst du `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 du ihn vergisst, 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. Du solltest 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 beispielswese 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 du jeden Patch mit den DEP-3 Kopfzeilen versiehst. Hier sind einige Einträge, die du verwenden kannst:

**Description** Description of what the patch does. It is formatted like `Description` field in `debian/control`: first line is short description, starting with lowercase letter, the next lines are long description, indented with a space.

**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, füge 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, kannst du 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 fahre 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 committe wie ueblich:

```
$ 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

Other patch systems used by packages include `dpatch` and `cdbs simple-patchsys`, these work similarly to Quilt by keeping patches in `debian/patches` but have different commands to apply, un-apply or create patches. You can find out which patch system is used by a package by using the `what-patch` command (from the `ubuntu-dev-tools` package). You can use `edit-patch`, shown in *previous chapters*, as a reliable way to work with all systems.

In even older packages changes will be included directly to sources and kept in the `diff.gz` source file. This makes it hard to upgrade to new upstream versions or differentiate between patches and is best avoided.

Ändere 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 dir 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 is fehlgeschlagen, das kann meist durch eine Neuerstellung des Pakets behoben werden. Bitte 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 überprüfe zuerst das Erstellungsprotokoll.

### 1.7.1 Erste Schritte

Als erstes solltest du versuchen das FTBFS selbst zu reproduzieren. Hol dir 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, erstelle es in einer schroot.

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

### 1.7.2 Debian überprüfen

Kannst du das Problem reproduzieren, ist es an der Zeit eine Lösung zu finden. Falls das Paket auch in Debian enthalten ist, kannst du überprüfen ob es dort erstellt werden kann, indem du auf http://packages.qa.debian.org/PACKAGE gehst. Falls Debian eine aktuelle Version einsetzt, solltest du sie mergen. Falls nicht, überprüfe 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, du wirst sie sicher auch bei der Lösungssuche einbeziehen wollen.

### 1.7.3 Andere Gründe warum die Paketerstellung fehlsschlägt

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

### 1.7.4 Das Problem beheben

Hast du erst einmal eine Lösung für das Problem gefunden, folge demselben Prozess wie bei jedem anderen Fehler. Erstelle einen Patch, hänge ihn an einen bzr-Zweig oder -Fehler an, informiere ubuntu-sponsors, und versuche 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 musss. 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, starte:

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

The SONAME is `libnova-0.12.so.2`, which matches the file name (usually the case but not always). Here upstream has put the upstream version number as part of the SONAME and given it an ABI version of `2`. Library package names should follow the SONAME of the library they contain. The library binary package is called `libnova-0.12-2` where `libnova-0.12` is the name of the library and `2` is our ABI number.

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 is 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 files are needed on some non-Linux systems with poor library support but usually cause more problems than they solve on Debian systems. It is a current Debian goal to remove .la files and we should help with this.

### 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

When a package builds against a library the `shlibs` mechanism will add a package dependency on that library. This is why most programs will have `Depends: ${shlibs:Depends}` in `debian/control`. That gets replaced with the library dependencies at build time. However shlibs can only make it depend on the major ABI version number, `2` in our libnova example, so if new symbols get added in libnova 2.1 a program using these symbols could still be installed against libnova ABI 2.0 which would then crash.

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. Beginne 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 du selbst anwenden kannst:

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

Which will create a file named similar to `dpkg-gensymbolsnY_WWI` that lists all the symbols. It also lists the current package version. We can remove the packaging version from that listed in the symbols file because new symbols are not generally added by new packaging versions, but by the upstream developers:

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

Nun verschiebe die Datei an ihren Ort, comitte und mache eine 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 würdest du erneut dpkg-gensymbols ausführen und es liefert eine Auflistung der Unterschiede um die Symboldatei zu aktualisieren.

### 1.8.4 C++-Bibliothek-Symboldateien

C++ has even more exacting standards of binary compatibility than C. The Debian Qt/KDE Team maintain some scripts to handle this, see their Working with symbols files page for how to use them.

### 1.8.5 Weiterführende Literatur

Junichi Uekawa's Debian Library Packaging Guide goes into this topic in more detail.

## 1.9 Zurückportieren von Software-Aktualisierungen

Sometimes you might want to make new functionality available in a stable release which is not connected to a critical bug fix. For these scenarios you have two options: either you upload to a PPA or prepare a backport.

### 1.9.1 Persönliches Paketarchiv (PPA)

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

The PPA documentation on Launchpad is fairly comprehensive and should get you up and running in no time.

### 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 Du Dich dazu entschlossen hast, ein Paket durch eine Backport in einen stabilen Release zu portieren, wirst Du 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, kannst du 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 schaue 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 du einen GPG-Schlüssel besitzt (siehe: *Erste Schritte*), stelle sicher dass du den selben Alias und E-Mail Adresse im `changelog` verwendest wie in deinem 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 Du ein neues Paket ohne Vorlage erstellst, kannst Du 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 Packetieren 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`).

For further information, see the changelog section (Section 4.4) of the Debian Policy Manual.

### 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.

Note that in Ubuntu, we set the `Maintainer` field to a general address because anyone can change any package (this differs from Debian where changing packages is usually restricted to an individual or a team). Packages in Ubuntu should generally have the `Maintainer` field set to `Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>`. If the Maintainer field is modified, the old value should be saved in the `XSBC-Original-Maintainer` field. This can be done automatically with the `update-maintainer` script available in the `ubuntu-dev-tools` package. For further information, see the Debian Maintainer Field spec on the Ubuntu wiki.

Jeder weitere Abschnitte beschreibt ein Binärpaket, welches gebaut wird.

For further information, see the control file section (Chapter 5) of the Debian Policy Manual.

### 2.2.3 Die Copyright-Datei

This file gives the copyright information for both the upstream source and the packaging. Ubuntu and Debian Policy (Section 12.5) require that each package installs a verbatim copy of its copyright and license information to `/usr/share/doc/$(package_name)/copyright`.

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. Eine 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'.
```

This example follows the Machine-readable debian/copyright format. You are encouraged to use this format as well.

### 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 $@
```

Lass 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, benutze:

```
$ 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 mit anderen Optionen ausführen oder komplett andere Befehle stattdessen ausführen. (Zu beachten ist, dass du für die Erstellung debhelper 7.0.50 oder höher verwenden solltest.)

Wirf 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 wir dvon dh_install verwendet um Dateien in das Binärpacket zu installieren. Es hat zwei gängige Einsatzmöglichkeiten:

- Um Dateien in dein 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 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.

Siehe 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
```

If your tarballs live on Launchpad, the debian/watch file is a little more complicated (see Question 21146 and Bug 231797 for why this is). In that case, use something like:

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

Für weitere Informationen, schaue Dir 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, schau Dir Ubuntu External Health Status an.

### Die Datei source/format

Diese Datei spezifiziert das Format des Quellpakets. Es sollt 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. Solltest Du Dich 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 summarizes additional information concerning the switch to the 3.0 source package formats.

See `man dpkg-source` and the source/format section (Section 5.21) of the Debian New Maintainers' Guide for additional details.

### 2.2.6 Weiterführende Quellen

In addition to the links to the Debian Policy Manual in each section above, the Debian New Maintainers' Guide has more detailed descriptions of each file. Chapter 4, "Required files under the debian directory" further discusses the control, changelog, copyright and rules files. Chapter 5, "Other files under the debian directory" discusses additional files that may be used.

## 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 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

While the test script can be easily executed on its own, it is strongly recommended to actually use `autopkgtest` from the `autopkgtest` package for verifying that your test works; otherwise, if it fails in the Ubuntu Continuous Integration (CI) system, it will not land in Ubuntu. This also avoids cluttering your workstation with test packages or test configuration if the test does something more intrusive than the simple example above.

The README.running-tests (online version) documentation explains all available testbeds (schroot, LXD, QEMU, etc.) and the most common scenarios how to run your tests with `autopkgtest`, e. g. with locally built binaries, locally modified tests, etc.

Das Ubuntu CI-System benutzt den QEMU-Runner und führt alle Tests der Pakete in des Archievs 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 erstelle eine Testumgebung mit:

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

(Please see its manpage and `--help` output for selecting different releases, architectures, output directory, or using proxies). This will build e. g. `adt-trusty-amd64-cloud.img`.

Then run the tests of a source package like `libpng` in that QEMU image:

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

The Ubuntu CI system runs packages with only selected packages from `-proposed` available (the package which caused the test to be run); to enable that, run:

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

or to run with all packages from `-proposed`:

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

The `autopkgtest` manpage has a lot more valuable information on other testing options.

### 2.4.4 Weitere Beispiele

Diese Liste ist nicht vollständig, aber wird dir 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.
- The gtk+3.0 tests also do a compile/link/run check in the "build" test. There is an additional "python3-gi" test which verifies that the GTK library can also be used through introspection.
- In den ubiquity Tests wird die Upstream Testsammlung ausgeführt.
- The gvfs tests have comprehensive testing of their functionality and are very interesting because they emulate usage of CDs, Samba, DAV and other bits.

### 2.4.5 Ubuntu Infrastruktur

Packages which have `autopkgtest` enabled will have their tests run whenever they get uploaded or any of their dependencies change. The output of automatically run autopkgtest tests can be viewed on the web and is regularly updated.

Debian also uses `autopkgtest` to run package tests, although currently only in schroots, so results may vary a bit. Results and logs can be seen on http://ci.debian.net. So please submit any test fixes or new tests to Debian as well.

### 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:

- Führe `bzr branch ubuntu:<Paketname>` aus,
- 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 du machen kannst

The Ubuntu Engineering team put together a list of required test-cases, where packages which need tests are put into different categories. Here you can find examples of these tests and easily assign them to yourself.

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

## 2.5 Chroots benutzen

Wenn Du eine Version von Ubuntu benutzt, aber an Paketen für andere Versionen arbeitest, kannst Du eine Umgebung dieser Version mit einer `Chroot` erzeugen.

Eine `Chroot` erlaubt dir 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

Benutze 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 Deine Version von `debootstrap` Trusty nicht kennt, kannst Du versuchen die Version in `backports` zu upgraden.

Du kannst dann in der Chroot arbeiten:

```
$ sudo chroot trusty
```

Wo Du alle Pakete installieren oder löschen kannst ohne Dein Hauptsystem zu berühren.

Vielleicht möchtest Du auch deine GPG/ssh-Schlüssel und Bazaarkonfiguration in das Chroot kopieren, um so einfacher auf Pakete zugreifen zu 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.

For more information on chroots see our Debootstrap Chroot wiki page.

### 2.5.2 Alternativen

SBuild is a system similar to PBuilder for creating an environment to run test package builds in. It closer matches that used by Launchpad for building packages but takes some more setup compared to PBuilder. See the Security Team Build Environment wiki page for a full explanation.

Full virtual machines can be useful for packaging and testing programs. TestDrive is a program to automate syncing and running daily ISO images, see the TestDrive wiki page for more information.

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

Amazon's EC2 cloud computers allow you to hire a computer paying a few US cents per hour, you can set up Ubuntu machines of any supported version and package on those. This is useful when you want to compile many packages at the same time or to overcome bandwidth restraints.

## 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

Packaging of KDE programs in Ubuntu is managed by the Kubuntu and MOTU teams. You can contact the Kubuntu team on the Kubuntu mailing list and `#kubuntu-devel` Freenode IRC channel. More information about Kubuntu development is on the Kubuntu wiki page.

Our packaging follows the practices of the Debian Qt/KDE Team and Debian KDE Extras Team. Most of our packages are derived from the packaging of these Debian teams.

### 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 Packet eine `Message.sh` Datei einbinden. Wenn dem nicht so ist, beschwere dich upstream. Ob es funktioniert kannst Du überprüfen, in dem Du `extract-messages.sh` ausführst, welches eine oder mehrere Dateien namens `.pot` in `po/` erstellen sollte. Dies wird während des Bauens automatisch getan, wenn Du `dh``mit der Option ``--width=kde` benutzt.

Upstream will usually have also put the translation `.po` files into the `po/` directory. If they do not, check if they are in separate upstream language packs such as the KDE SC language packs. If they are in separate language packs Launchpad will need to associate these together manually, contact David Planella to do this.

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

Library symbols are tracked in `.symbols` files to ensure none go missing for new releases. KDE uses C++ libraries which act a little differently compared to C libraries. Debian’s Qt/KDE Team have scripts to handle this. See Working with symbols files for how to create and keep these files up to date.

KAPITEL 3

# Weiterführende Literatur

You can read this guide offline in different formats, if you install one of the binary packages.

Wenn Du mehr über das Bauen von Debian-Paketen lernen willst, sind hier einige Debian-Resourcen, die Du hilfreich finden könntest.

- How to package for Debian;
- Debian Policy Manual;
- Debian New Maintainers' Guide — available in many languages;
- Packaging tutorial (also available as a package);
- Guide for Packaging Python Modules.

We are always looking to improve this guide. If you find any problems or have some suggestions, please report a bug on Launchpad. If you'd like to help work on the guide, grab the source there as well.