Ubuntu Packaging Guide

Release 0.3.2 bzr343 quantal1

Ubuntu Developers

March 28, 2013

CONTENTS

Welcome to the Ubuntu Packaging and Development Guide! 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 is not only a free and open source operating system, its platform is also open and developed in a transparent fashion. The source code for every single component can be obtained easily and every single change to the Ubuntu platform can be reviewed.

This means you can actively get involved in improving it and the community of Ubuntu platform developers is always interested in helping peers getting started.

Ubuntu is also a community of great people who believe in free software and that it should be accessible for everyone. Its members are welcoming and want you to be involved as well. We want you to get involved, to ask questions, to make Ubuntu better together with us.

If you run into problems: don't panic! Check out the *communication article* and you will find out how to most easily get in touch with other developers.

The guide is split up into two sections:

- A list of articles based on tasks, things you want to get done.
- A set of knowledge-base articles that dig deeper into specific bits of our tools and workflows.

This guide focuses on the Ubuntu Distributed Development packaging method. This is a new way of packaging which uses Distributed Revision Control branches. It currently has some limitations which mean many teams in Ubuntu use *traditional packaging* methods. See the *UDD Introduction* page for an introduction to the differences.

CHAPTER

ONE

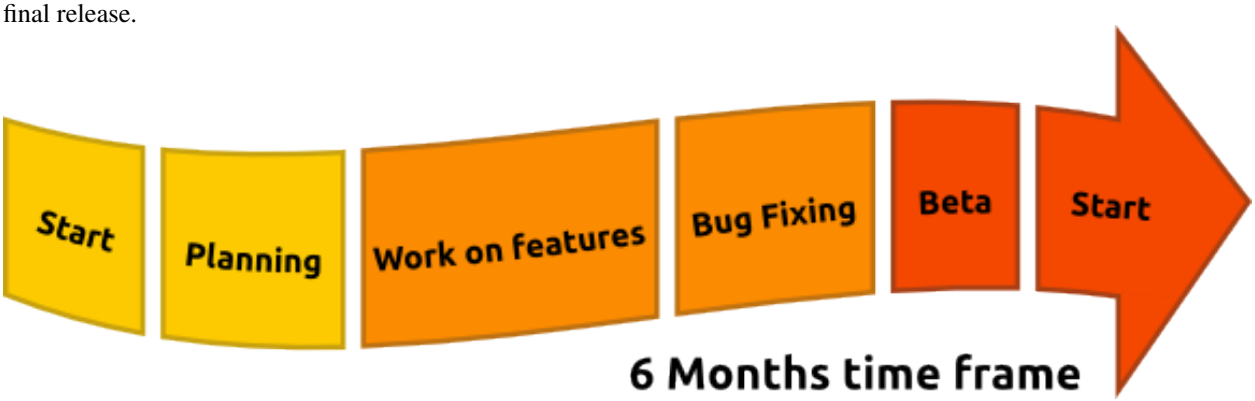
ARTICLES

1.1 Introduction to Ubuntu Development

Ubuntu is made up of thousands of different components, written in many different programming languages. Every component - be it a software library, a tool or a graphical application - is available as a source package. Source packages in most cases consist of two parts: the actual source code and metadata. Metadata includes the dependencies of the package, copyright and licensing information, and instructions on how to build the package. Once this source package is compiled, the build process provides binary packages, which are the .deb files users can install.

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 CD images are built for a selection of different Ubuntu flavours. Ubuntu Desktop, Ubuntu Server, Kubuntu and others specify a list of required packages that get on the CD. These CD images are then used for installation tests and provide the feedback for further release planning.

Ubuntu's development is very much dependent on the current stage of the release cycle. We release a new version of Ubuntu every six months, which is only possible because we have established strict freeze dates. With every freeze date that is reached developers are expected to make fewer, less intrusive changes. Feature Freeze is the first big freeze date after the first half of the cycle has passed. At this stage features must be largely implemented. The rest of the cycle is supposed to be focused on fixing bugs. After that the user interface, then the documentation, the kernel, etc. are frozen, then the beta release is put out which receives a lot of testing. From the beta release onwards, only critical bugs get fixed and a release candidate release is made and if it does not contain any serious problems, it becomes the final release.



Thousands of source packages, billions of lines of code, hundreds of contributors require a lot of communication and planning to maintain high standards of quality. At the beginning of each release cycle we have the Ubuntu Developer Summit where developers and contributors come together to plan the features of the next releases. Every feature is discussed by its stakeholders and a specification is written that contains detailed information about its assumptions, implementation, the necessary changes in other places, how to test it and so on. This is all done in an open and

transparent fashion, so even if you cannot attend the event in person, you can participate remotely and listen to a streamcast, chat with attendants and subscribe to changes of specifications, so you are always up to date.

Not every single change can be discussed in a meeting though, particularly because Ubuntu relies on changes that are done in other projects. That is why contributors to Ubuntu constantly stay in touch. Most teams or projects use dedicated mailing lists to avoid too much unrelated noise. For more immediate coordination, developers and contributors use Internet Relay Chat (IRC). All discussions are open and public.

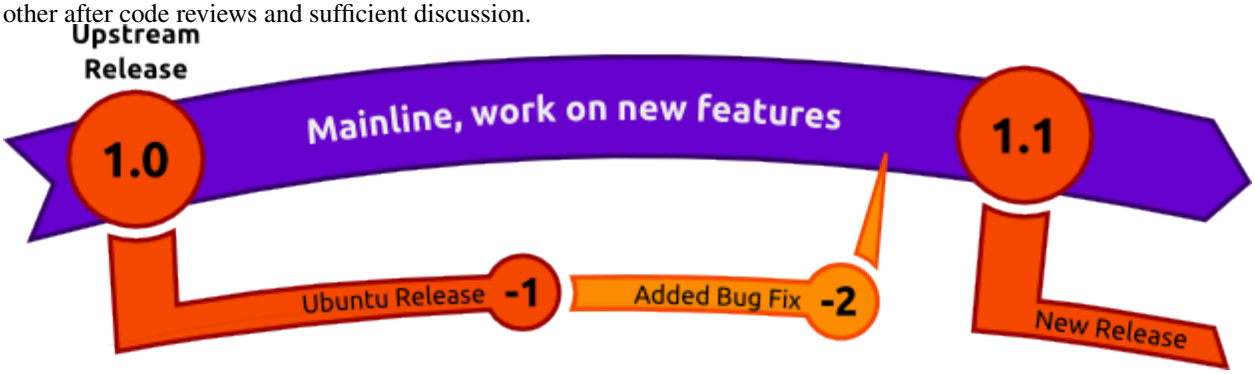
Another important tool regarding communication is bug reports. Whenever a defect is found in a package or piece of infrastructure, a bug report is filed in Launchpad. All information is collected in that report and its importance, status and assignee updated when necessary. This makes it an effective tool to stay on top of bugs in a package or project and organise the workload.

Most of the software available through Ubuntu is not written by Ubuntu developers themselves. Most of it is written by developers of other Open Source projects and then integrated into Ubuntu. These projects are called "Upstreams", because their source code flows into Ubuntu, where we "just" integrate it. The relationship to Upstreams is critically important to Ubuntu. It is not just code that Ubuntu gets from Upstreams, but it is also that Upstreams get users, bug reports and patches from Ubuntu (and other distributions).

The most important Upstream for Ubuntu is Debian. Debian is the distribution that Ubuntu is based on and many of the design decisions regarding the packaging infrastructure are made there. Traditionally, Debian has always had dedicated maintainers for every single package or dedicated maintenance teams. In Ubuntu there are teams that have an interest in a subset of packages too, and naturally every developer has a special area of expertise, but participation (and upload rights) generally is open to everyone who demonstrates ability and willingness.

Getting a change into Ubuntu as a new contributor is not as daunting as it seems and can be a very rewarding experience. It is not only about learning something new and exciting, but also about sharing the solution and solving a problem for millions of users out there.

Open Source Development happens in a distributed world with different goals and different areas of focus. For example there might be the case that a particular Upstream is interested in working on a new big feature while Ubuntu, because of the tight release schedule, is interested in shipping a solid version with just an additional bug fix. That is why we make use of "Distributed Development", where code is being worked on in various branches that are merged with each other after code reviews and sufficient discussion.



In the example mentioned above it would make sense to ship Ubuntu with the existing version of the project, add the bugfix, get it into Upstream for their next release and ship that (if suitable) in the next Ubuntu release. It would be the best possible compromise and a situation where everybody wins.

To fix a bug in Ubuntu, you would first get the source code for the package, then work on the fix, document it so it is easy to understand for other developers and users, then build the package to test it. After you have tested it, you can easily propose the change to be included in the current Ubuntu development release. A developer with upload rights will review it for you and then get it integrated into Ubuntu.

When trying to find a solution it is usually a good idea to check with Upstream and see if the problem (or a possible solution) is known already and, if not, do your best to make the solution a concerted effort.

Additional steps might involve getting the change backported to an older, still supported version of Ubuntu and forwarding it to Upstream.

The most important requirements for success in Ubuntu development are: having a knack for "making things work again," not being afraid to read documentation and ask questions, being a team player and enjoying some detective work.

Good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on `irc.freenode.net`. 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 Getting Set Up

There are a number of things you need to do to get started developing for Ubuntu. This article is designed to get your computer set up so that you can start working with packages, and upload your packages to Ubuntu's hosting platform, Launchpad. Here's what we'll cover:

- Installing packaging-related software. This includes:
  - Ubuntu-specific packaging utilities
  - Encryption software so your work can be verified as being done by you
  - Additional encryption software so you can securely transfer files
- Creating and configuring your account on Launchpad
- Setting up your development environment to help you do local builds of packages, interact with other developers, and propose your changes on Launchpad.

Note: It is advisable to do packaging work using the current development version of Ubuntu. Doing so will allow you to test changes in the same environment where those changes will actually be applied and used.

Don't worry though, the Ubuntu development release wiki page shows a variety of ways to safely use the development release.

1.2.1 Install basic packaging software

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-get install gnupg pbuilder ubuntu-dev-tools bzr-builddeb apt-file
```

Note: Since Ubuntu 11.10 "Oneiric Ocelot" (or if you have Backports enabled on a currently supported release), the following command will install the above and other tools which are quite common in Ubuntu development:

```
$ sudo apt-get install packaging-dev
```

This command will install the following software:

- `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` – a tool to do reproducible builds of a package in a clean and isolated environment.
- `ubuntu-dev-tools` (and `devscripts`, a direct dependency) – a collection of tools that make many packaging tasks easier.
- `bzr-builddeb` (and `bzr`, a dependency) – distributed version control with Bazaar, a new way of working with packages for Ubuntu that will make it easy for many developers to collaborate and work on the same code while keeping it trivial to merge each other's work.
- `apt-file` provides an easy way to find the binary package that contains a given file.

Create your GPG key

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.

To generate a new GPG key, run:

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

GPG will first ask you which kind of key you want to generate. Choosing the default (RSA and DSA) is fine. Next it will ask you about the keysize. The default (currently 2048) is fine, but 4096 is more secure. Afterwards, it will ask you if you want it to expire the key at some stage. It is safe to say "0", which means the key will never expire. The last questions will be about your name and email address. Just pick the ones you are going to use for Ubuntu development here, you can add additional email addresses later on. Adding a comment is not necessary. Then you will have to set a passphrase, choose a safe one (a passphrase is just a password which is allowed to include spaces).

Now GPG will create a key for you, which can take a little bit of time; it needs random bytes, so if you give the system some work to do it will be just fine. Move the cursor around, type some paragraphs of random text, load some web page.

Once this is done, you will get a message similar to this one:

```
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 this case `43CDE61D` is the *key ID*.

Next, you need to upload the public part of your key to a keyserver so the world can identify messages and files as yours. To do so, enter:

```
$ gpg --send-keys <KEY ID>
```

This will send your key to one keyserver, but a network of keyservers will automatically sync the key between themselves. Once this syncing is complete, your signed public key will be ready to verify your contributions around the world.

Create your SSH key

SSH stands for *Secure Shell*, and it is a protocol that allows you to exchange data in a secure way over a network. It is common to use SSH to access and open a shell on another computer, and to use it to securely transfer files. For our purposes, we will mainly be using SSH to securely upload source packages to Launchpad.

To generate an SSH key, enter:

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

The default file name usually makes sense, so you can just leave it as it is. For security purposes, it is highly recommended that you use a passphrase.

Set up pbuilder

`pbuilder` allows you to build packages locally on your machine. It serves a couple of purposes:

- The build will be done in a minimal and clean environment. This helps you make sure your builds succeed in a reproducible way, but without modifying your local system
- There is no need to install all necessary *build dependencies* locally
- You can set up multiple instances for various Ubuntu and Debian releases

Setting `pbuilder` up is very easy, run:

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

where <release> is for example *natty*, *maverick*, *lucid* or in the case of Debian maybe *sid*. 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 Get set up to work with Launchpad

With a basic local configuration in place, your next step will be to configure your system to work with Launchpad. This section will focus on the following topics:

- What Launchpad is and creating a Launchpad account
- Uploading your GPG and SSH keys to Launchpad
- Configuring Bazaar to work with Launchpad
- Configuring Bash to work with Bazaar

About Launchpad

Launchpad is the central piece of infrastructure we use in Ubuntu. It not only stores our packages and our code, but also things like translations, bug reports, and information about the people who work on Ubuntu and their team memberships. You will also use Launchpad to publish your proposed fixes, and get other Ubuntu developers to review and sponsor them.

You will need to register with Launchpad and provide a minimal amount of information. This will allow you to download and upload code, submit bug reports, and more.

Besides hosting Ubuntu, Launchpad can host any Free Software project. For more information see the Launchpad Help wiki.

Get a Launchpad account

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.

Launchpad's registration process will ask you to choose a display name. It is encouraged for you to use your real name here so that your Ubuntu developer colleagues will be able to get to know you better.

When you register a new account, Launchpad will send you an email with a link you need to open in your browser in order to verify your email address. If you don't receive it, check in your spam folder.

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

Upload your GPG key to Launchpad

To find about your GPG fingerprint, run:

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

and it will print out something like:

```
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
```

Head to https://launchpad.net/~/+editpgpkeys and copy the "Key fingerprint" into the text box. In the case above this would be `5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D`. Now click on "Import Key".

Launchpad will use the fingerprint to check the Ubuntu key server for your key and, if successful, send you an encrypted email asking you to confirm the key import. Check your email account and read the email that Launchpad sent you. *If your email client supports OpenPGP encryption, it will prompt you for the password you chose for the key when GPG generated it. Enter the password, then click the link to confirm that the key is yours.*

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.

Back on the Launchpad website, use the Confirm button and Launchpad will complete the import of your OpenPGP key.

Find more information at https://help.launchpad.net/YourAccount/ImportingYourPGPKey

Upload your SSH key to Launchpad

Open https://launchpad.net/~/+editsshkeys in a web browser, also open `~/.ssh/id_rsa.pub` in a text editor. This is the public part of your SSH key, so it is safe to share it with Launchpad. Copy the contents of the file and paste them into the text box on the web page that says "Add an SSH key". Now click "Import Public Key".

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

Configure Bazaar

Bazaar is the tool we use to store code changes in a logical way, to exchange proposed changes and merge them, even if development is done concurrently. It is used for the new Ubuntu Distributed Development method of working with Ubuntu packages.

To tell Bazaar who you are, simply run:

```
$ bzr whoami "Bob Dobbs <subgenius@example.com>"
$ bzr launchpad-login subgenius
```

whoami will tell Bazaar which name and email address it should use for your commit messages. With *launchpad-login* you set your Launchpad ID. This way code that you publish in Launchpad will be associated with you.

Note: If you can not remember the ID, go to https://launchpad.net/~ and see where it redirects you. The part after the "~" in the URL is your Launchpad ID.)

Configure your shell

Similar to Bazaar, the Debian/Ubuntu packaging tools need to learn about you as well. 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"
```

Now save the file and either restart your terminal or run:

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

(If you do not use the default shell, which is *bash*, please edit the configuration file for that shell accordingly.)

1.3 Ubuntu Distributed Development — Introduction

This guide focuses on packaging using the *Ubuntu Distributed Development* (UDD) method.

Ubuntu Distributed Development (UDD) is a new technique for developing Ubuntu packages that uses tools, processes, and workflows similar to generic distributed version control system (DVCS) based software development. The DVCS used for UDD is Bazaar.

1.3.1 Traditional Packaging Limitations

Traditionally Ubuntu packages have been kept in tar archive files. A traditional source package is made up of the upstream source tar, a "debian" tar (or compressed diff file for older packages) containing the packaging and a .dsc meta-data file. To see a traditional package run:

```
$ apt-get source kdetoys
```

This will download the upstream source `kdetoys_4.6.5.orig.tar.bz2`, the packaging `kdetoys_4.6.5-0ubuntu1.debian.tar.gz` and the meta-data `kdetoys_4.6.5-0ubuntu1~ppa1.dsc`. Assuming you have dpkg-dev installed it will extract these and give you the source package.

Traditional packaging would edit these files and upload. However this gives limited opportunity to collaborate with other developers, changes have to be passed around as diff files with no central way to track them and two developers can not make changes at the same time. So most teams have moved to putting their packaging in a revision control

system. This makes it easier for several developers to work on a package together. However there is no direct connection between the revision control system and the archive packages so the two must be manually kept in sync. Since each team works in its own revision control system a prospective developer must first work out where that is and how to get the packaging before they can work on the package.

1.3.2 Ubuntu Distributed Development

With Ubuntu Distributed Development all packages in the Ubuntu (and Debian) archive are automatically imported into Bazaar branches on our code hosting site Launchpad. Changes can be made directly to these branches in incremental steps and by anyone with commit access. Changes can also be made in forked branches and merged back in with Merge Proposals when they are large enough to need review or if they are by someone without direct commit access.

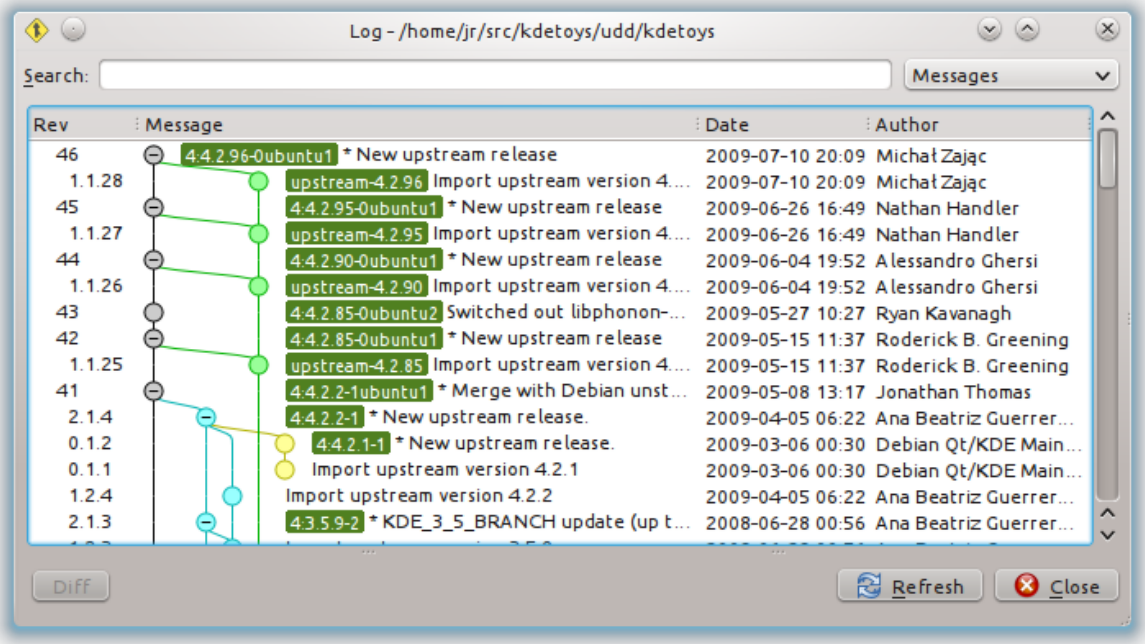
UDD branches are all in a standard location, so doing a checkout is easy:

```
$ bzr branch ubuntu:kdetoys
```

The merge history includes two separate branches, one for the upstream source and one which adds the `debian/` packaging directory:

```
$ cd kdetoys
$ bzr qlog
```

(This command uses *qbzr* for a GUI, run `log` instead of `qlog` for console output.)



This UDD branch of *kdetoys* shows the full packaging for each version uploaded to Ubuntu with grey circles and the upstream source versions with green circles. Versions are tagged with either the version in Ubuntu such as `4:4.2.29-0ubuntu1` or for the upstream branch with the upstream version `upstream-4.2.96`.

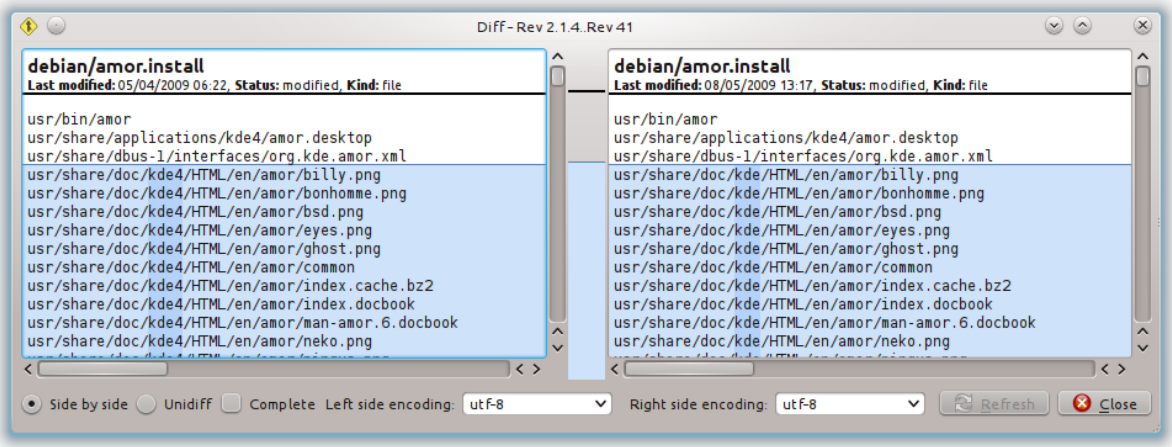
Many Ubuntu packages are based on the packages in Debian, UDD also imports the Debian package into our branches. In the *kdetoys* branch above the Debian versions from *unstable* are from the merge with blue circles while those from

Debian experimental are from the merge with yellow circles. Debian releases are tagged with their version number, e.g., `4:4.2.2-1`.

So from a UDD branch you can see the complete history of changes to the package and compare any two versions. For example, to see the changes between version 4.2.2 in Debian and the 4.2.2 in Ubuntu use:

```
$ bzr qdiff -r tag:4:4.2.2-1..tag:4:4.2.2-1ubuntu1
```

(This command uses *qbzr* for a GUI, run `diff` instead of `qdiff` for console output.)



From this we can clearly see what has changed in Ubuntu compared to Debian, very handy.

1.3.3 Bazaar

UDD branches use Bazaar, a distributed revision control system intended to be easy to use for those familiar with popular systems such as Subversion while offering the power of Git.

To do packaging with UDD you will need to know the basics of how to use Bazaar to manage files. For an introduction to Bazaar see the Bazaar Five Minute Tutorial and the Bazaar Users Guide.

1.3.4 Limitations of UDD

Ubuntu Distributed Development is a new method for working with Ubuntu packages. It currently has some notable limitations:

- Doing a full branch with history can take a lot of time and network resources. You may find it quicker to do a lightweight checkout `bzr checkout --lightweight ubuntu:kdetoys` but this will need a network access for any further bzr operations.
- Working with patches is fiddly. Patches can be seen as a branched revision control system, so we end up with RCS on top of RCS.
- There is no way to build directly from branches. You need to create a source package and upload that.
- Some packages have not been successfully imported into UDD branches. Recent versions of Bazaar will automatically notify you when this is the case. You can also check the status of the package importer manually before working on a branch.

All of the above are being worked on and UDD is expected to become the main way to work on Ubuntu packages soon. However currently most teams within Ubuntu do not yet work with UDD branches for their development. However

because UDD branches are the same as the packages in the archive any team should be able to accept merges against them.

1.4 Fixing a bug in Ubuntu

1.4.1 Introduction

If you followed the instructions to *get set up with Ubuntu Development*, you should be all set and ready to go.

As you can see in the image above, there is no surprises in the process of fixing bugs in Ubuntu: you found a problem, you get the code, work on the fix, test it, push your changes to Launchpad and ask for it to be reviewed and merged. In this guide we will go through all the necessary steps one by one.

1.4.2 Finding the problem

There are a lot of different ways to find things to work on. It might be a bug report you are encountering yourself (which gives you a good opportunity to test the fix), or a problem you noted elsewhere, maybe in a bug report.

Harvest is where we keep track of various TODO lists regarding Ubuntu development. It lists bugs that were fixed upstream or in Debian already, lists small bugs (we call them 'bitesize'), and so on. Check it out and find your first bug to work on.

1.4.3 Figuring out what to fix

If you don't know the source package containing the code that has the problem, but you do know the path to the affected program on your system, you can discover the source package that you'll need to work on.

Let's say you've found a bug in Tomboy, a note taking desktop application. The Tomboy application can be started by running `/usr/bin/tomboy` on the command line. To find the binary package containing this application, use this command:

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

This would print out:

```
tomboy: /usr/bin/tomboy
```

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 tomboy | grep ^Package:
Package: tomboy
$ apt-cache showsrc python-vigra | grep ^Package:
Package: libvigraimpex
```

`apt-cache` is part of the standard installation of Ubuntu.

1.4.4 Getting the code

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. In Ubuntu Distributed Development this is done by *branching the source package* branch corresponding to the source package. Launchpad maintains source package branches for all the packages in Ubuntu.

Once you've got a local branch of the source package, you can investigate the bug, create a fix, and upload your proposed fix to Launchpad, in the form of a Bazaar branch. When you are happy with your fix, you can *submit a merge proposal*, which asks other Ubuntu developers to review and approve your change. If they agree with your changes, an Ubuntu developer will upload the new version of the package to Ubuntu so that everyone gets the benefit of your excellent fix - and you get a little bit of credit. You're now on your way to becoming an Ubuntu developer!

We'll describe specifics on how to branch the code, push your fix, and request a review in the following sections.

1.4.5 Work on a fix

There are entire books written about finding bugs, fixing them, testing them, etc. If you are completely new to programming, try to fix easy bugs such as obvious typos first. Try to keep changes as minimal as possible and document your change and assumptions clearly.

Before working on a fix yourself, make sure to investigate if nobody else has fixed it already or is currently working on a fix. Good sources to check are:

- Upstream (and Debian) bug tracker (open and closed bugs),
- Upstream revision history (or newer release) might have fixed the problem,
- bugs or package uploads of Debian or other distributions.

You now 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
```

This will copy the packaging to a temporary directory. You can now edit files with a text editor or apply patches from upstream, for example:

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

After editing the file type `exit` or press `control-d` to quit the temporary shell. The new patch will have been added into `debian/patches`.

1.4.6 Testing the fix

To build a test package with your changes, run these commands:

```
$ bzr builddeb -- -S -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 `pbuilder-dist` will build the package from source for whatever `release` you choose.

Once the build succeeds, install the package from `~/pbuilder/<release>_result/` (using `sudo dpkg -i <package>_<version>.deb`). Then test to see if the bug is fixed.

Documenting the fix

It is very important to document your change sufficiently so developers who look at the code in the future won't have to guess what your reasoning was and what your assumptions were. Every Debian and Ubuntu package source includes `debian/changelog`, where changes of each uploaded package are tracked.

The easiest way to update this is to run:

```
$ dch -i
```

This will add a boilerplate changelog entry for you and launch an editor where you can fill in the blanks. An example of this could be:

```
specialpackage (1.2-3ubuntu4) natty; 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` should fill out the first and last line of such a changelog entry for you already. Line 1 consists of the source package name, the version number, which Ubuntu release it is uploaded to, the urgency (which almost always is 'low'). The last line always contains the name, email address and timestamp (in **RFC 5322** format) of the change.

With that out of the way, let's focus on the actual changelog entry itself: it is very important to document:

1. where the change was done
2. what was changed
3. where the discussion of the change happened

In our (very sparse) example the last point is covered by `(LP: #123456)` which refers to Launchpad bug 123456. Bug reports or mailing list threads or specifications are usually good information to provide as a rationale for a change. As a bonus, if you use the `LP: #<number>` notation for Launchpad bugs, the bug will be automatically closed when the package is uploaded to Ubuntu.

Committing the fix

With the changelog entry written and saved, you can just run:

```
debcommit
```

and the change will be committed (locally) with your changelog entry as a commit message.

To push it to Launchpad, as the remote branch name, you need to stick to the following nomenclature:

```
lp:~<yourlpid>/ubuntu/<release>/<package>/<branchname>
```

This could for example be:

```
lp:~emmaadams/ubuntu/natty/specialpackage/fix-for-123456
```

So if you just run:

```
bzr push lp:~emmaadams/ubuntu/natty/specialpackage/fix-for-123456
bzr lp-propose
```

you should be all set. The push command should push it to Launchpad and the second command will open the Launchpad page of the remote branch in your browser. There find the "(+) Propose for merging" link, click it to get the change reviewed by somebody and included in Ubuntu.

Our article about *seeking sponsorship* goes into more detail about getting feedback for your proposed changes.

If your branch fixes issues in stable releases or it is a security fix, you might want to have a look at our *Security and stable release updates* article.

1.5 Tutorial: Fixing a bug in Ubuntu

While the mechanics for *fixing a bug* are the same for every bug, every problem you look at is likely to be different from another. An example of a concrete problem might help to get an idea what to consider generally.

Note: At the time of writing this article this was not fixed yet. When you are reading the article this might actually be fixed. Take this as an example and try to adapt it to the specific problem you are facing.

1.5.1 Confirming the problem

Let's say the package `bumprace` does not have a homepage in its package description. As a first step you would check if the problem is not solved already. This is easy to check, either take a look at Software Center or run:

```
apt-cache show bumprace
```

The output should be similar to this:

```
Package: bumprace
Priority: optional
Section: universe/games
Installed-Size: 136
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
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
```

A counter-example would be `gedit`, which has a homepage set:

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

Sometimes you will find that a particular problem you are looking into is already fixed. To avoid wasting efforts and duplicating work it makes sense to first do some detective work.

1.5.2 Research bug situation

First we should check if a bug for the problem exists in Ubuntu already. Maybe somebody is working on a fix already, or we can contribute to the solution somehow. For Ubuntu we have a quick look at https://bugs.launchpad.net/ubuntu/+source/bumprace and there is no open bug with our problem there.

Note: For Ubuntu the URL `https://bugs.launchpad.net/ubuntu/+source/<package>` should always take to the bug page of the source package in question.

For Debian, which is the major source for Ubuntu's packages, we have a look at http://bugs.debian.org/src:bumprace and can't find a bug report for our problem either.

Note: For Debian the URL `http://bugs.debian.org/src:<package>` should always take to the bug page of the source package in question.

The problem we are working on is special as it only concerns the packaging-related bits of `bumprace`. If it was a problem in the source code it would be helpful to also check the Upstream bug tracker. This is unfortunately often different for every package you have a look at, but if you search the web for it, you should in most cases find it pretty easily.

1.5.3 Offering help

If you found an open bug and it is not assigned to somebody and you are in a position to fix it, you should comment on it with your solution. Be sure to include as much information as you can: Under which circumstances does the bug occur? How did you fix the problem? Did you test your solution?

If no bug report has been filed, you can file a bug for it. What you might want to bear in mind is: Is the issue so small that just asking for somebody to commit it is good enough? Did you manage to only partially fix the issue and you want to at least share your part of it?

It is great if you can offer help and will surely be appreciated.

1.5.4 Fixing the issue

For this specific example it is enough to search the web for `bumprace` and find the homepage of it. Be sure it is a live site and not just a software catalogue. http://www.linux-games.com/bumprace/ looks like it is the proper place.

To address the issue in the source package, we first need the source and we can easily get it by running:

```
bzr branch ubuntu:bumprace
```

If you read *the Debian Directory Overview* before, you might remember, that the homepage for a package is specified in the first part of `debian/control`, the section which starts with `Source:`.

So what we do next is run:

```
cd bumprace
```

and edit `debian/control` to add `Homepage: http://www.linux-games.com/bumprace/`. At the end of the first section should be a good place for it. Once you have done this, save the file.

If you now run:

```
bzr diff
```

you should see something like this:

```
=== modified file 'debian/control'
--- debian/control      2012-05-14 23:38:14 +0000
+++ debian/control      2012-09-03 15:45:30 +0000
@@ -12,6 +12,7 @@
                libtool,
                zlib1g-dev
 Standards-Version: 3.9.3
+Homepage: http://www.linux-games.com/bumprace/

 Package: bumprace
 Architecture: any
```

The diff is pretty simple to understand. The + indicates a line which was added. In our cases it was added just before the second section, starting with `Package`, which indicates a resulting binary package.

1.5.5 Documenting the fix

It is important to explain to your fellow developers what exactly you did. If you run:

```
dch -i
```

this will start an editor with a boilerplate changelog entry which you just have to fill out. In our case something like `debian/control: Added project's homepage.` should do. Then save the file. To double-check this worked out, run:

```
bzr diff debian/changelog
```

and you will see something like this:

```
=== modified file 'debian/changelog'
--- debian/changelog    2012-05-14 23:38:14 +0000
+++ debian/changelog    2012-09-03 15:53:52 +0000
@@ -1,3 +1,9 @@
+bumprace (1.5.4-1ubuntu1) UNRELEASED; urgency=low
+
+  * debian/control: Added project's homepage.
+
+ -- Peggy Sue <peggy.sue@example.com>  Mon, 03 Sep 2012 17:53:12 +0200
+
 bumprace (1.5.4-1) unstable; urgency=low

   * new upstream version, sound and music have been removed (closes: #613344)
```

A few additional considerations:

- If you have a reference to a Launchpad bug which is fixed by the issue, add (`LP: #<bug number>`) to the changelog entry line, ie: `(LP: #123456)`.

- If you want to get your fix included in Debian, for a Debian bug the syntax is `(Closes: #<bug number>)`, ie: `(Closes: #123456)`.
- If it is a reference to an upstream or Debian bug or a mailing list discussion, mention it as well.
- Try to wrap your lines at 80 characters.
- Try to be specific, not an essay, but enough for somebody (who did not deeply look into the issue) to understand.
- Mention how you fixed the issue and where.

1.5.6 Testing the fix

To test the fix, you need to *have your development environment set up*, then to build the package, install it and verify the problem is solved. In our case this would be:

```
bzr bd -- -S
pbuilder-dist <current Ubuntu release> build ../bumprace_*.dsc
dpkg -I ~/pbuilder/*_result/bumprace_*.deb
```

In step one we build the source package from the branch, then build it by using `pbuilder`, then inspect the resulting package to check if the Homepage field was added properly.

Note: In a lot of cases you will have to actually install the package to make sure it works as expected. Our case is a lot easier. If the build succeeded, you will find the binary packages in `~/pbuilder/<release>_result`. Install them via `sudo dpkg -i <package>.deb` or by double-clicking on them in your file manager.

As we verified, the problem is now solved, so the next step is sharing our solution with the world.

1.5.7 Getting the fix included

It makes to get fix included as Upstream as possible. Doing that you can guarantee that everybody can take the Upstream source as-is and don't need to have local modifications to fix it.

In our case we established that we have a problem with the packaging, both in Ubuntu and Debian. As Ubuntu is based on Debian, we will send the fix to Debian. Once it is included there, it will be picked up by Ubuntu eventually. The issue in our tutorial is clearly non-critical, so this approach makes sense. If it is important to fix the issue as soon as possible, you will need to send the solution to multiple bug trackers. Provided the issue affects all parties in question.

To submit the patch to Debian, simply run:

```
submittodebian
```

This will take you through a series of steps to make sure the bug ends up in the correct place. Be sure to review the diff again to make sure it does not include random changes you made earlier.

Communication is important, so when you add some more description to it to the inclusion request, be friendly, explain it well.

If everything went well you should get a mail from Debian's bug tracking system with more information. This might sometimes take a few minutes.

Note: If the problem is just in Ubuntu, you might want to consider *Seeking Review and Sponsorship* to get the fix included.

1.5.8 Additional considerations

If you find a package and find that there are a couple of trivial things you can fix at the same time, do it. This will speed up review and inclusion.

If there are multiple big things you want to fix, it might be advisable to send individual patches or merge proposals instead. If there are individual bugs filed for the issues already, this makes it even easier.

1.6 Packaging New Software

While there are thousands of packages in the Ubuntu archive, there are still a lot nobody has gotten to yet. If there is an exciting new piece of software that you feel needs wider exposure, maybe you want to try your hand at creating a package for Ubuntu or a PPA. This guide will take you through the steps of packaging new software.

You will want to read the *Getting Set Up* article first in order to prepare your development environment.

1.6.1 Checking the Program

The first stage in packaging is to get the released tar from upstream (we call the authors of applications "upstream") and check that it compiles and runs.

This guide will take you through packaging a simple application called GNU Hello which has been posted on GNU.org.

If you don't have the build tools lets make sure we have them first. Also if you don't have the required dependencies lets install those as well.

Install build tools:

```
$ sudo apt-get install build-essential
```

Download main package:

```
$ wget -O hello-2.7.tar.gz "http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz"
```

Now uncompress main package:

```
$ tar xf hello-2.7.tar.gz
$ cd hello-2.7
```

This application uses the autoconf build system so we want to run `./configure` to prepare for compilation.

This will check for the required build dependencies. As `hello` is a simple example, `build-essential` should provide everything we need. For more complex programs, the command will fail if you do not have the needed libraries and development files. Install the needed packages and repeat until the command runs successfully.:

```
$ ./configure
```

Now you can compile the source:

```
$ make
```

If compilation completes successfully you can install and run the program:

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

1.6.2 Starting a Package

`bzr-builddeb` includes a plugin to create a new package from a template. The plugin is a wrapper around the `dh_make` command. You should already have these if you installed `packaging-dev`. Run the command providing the package name, version number, and path to the upstream tarball:

```
$ sudo apt-get install dh-make
$ cd ..
$ bzr dh-make hello 2.7 hello-2.7.tar.gz
```

When it asks what type of package type `s` for single binary. This will import the code into a branch and add the `debian/` packaging directory. Have a look at the contents. Most of the files it adds are only needed for specialist packages (such as Emacs modules) so you can start by removing the optional example files:

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

You should now customise each of the files.

In `debian/changelog` change the version number to an Ubuntu version: `2.7-0ubuntu1` (upstream version 2.7, Debian version 0, Ubuntu version 1). Also change `unstable` to the current development Ubuntu release such as `precise`.

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 `8`.

`control` contains all the metadata of the package. The first paragraph describes the source package. The second and following paragraphs describe the binary packages to be built. We will need to add the packages needed to compile the application to `Build-Depends:`. For `hello`, make sure that it includes at least:

```
Build-Depends: debhelper (>= 8.0.0)
```

You will also need to fill in a description of the program in the `Description:` field.

`copyright` needs to be filled in to follow the licence of the upstream source. According to the hello/COPYING file this is GNU GPL 3 or later.

`docs` contains any upstream documentation files you think should be included in the final package.

`README.source` and `README.Debian` are only needed if your package has any non-standard features, we don't so you can delete them.

`source/format` can be left as is, this describes the version format of the source package and should be `3.0 (quilt)`.

`rules` is the most complex file. This is a Makefile which compiles the code and turns it into a binary package. Fortunately most of the work is automatically done these days by `debhelper 7` so the universal `%` Makefile target just runs the `dh` script which will run everything needed.

All of these file are explained in more detail in the *overview of the debian directory* article.

Finally commit the code to your packaging branch:

```
$ bzr commit -m "Initial commit of Debian packaging."
```

1.6.3 Building the package

Now we need to check that our packaging successfully compiles the package and builds the .deb binary package:

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

`bzr builddeb` is a command to build the package in its current location. The `-us -uc` tell it there is no need to GPG sign the package. The result will be placed in ...

You can view the contents of the package with:

```
$ lesspipe hello_2.7-0ubuntu1_amd64.deb
```

Install the package and check it works:

```
$ sudo dpkg --install hello_2.7-0ubuntu1_amd64.deb
```

1.6.4 Next Steps

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 both the source .dsc metadata file and the .deb binary package:

```
$ lintian hello_2.7-0ubuntu1.dsc
$ lintian hello_2.7-0ubuntu1_amd64.deb
```

A description of each of the problems it reports can be found on the lintian website.

After making a fix to the packaging you can rebuild using `-nc` "no clean" without having to build from scratch:

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

Having checked that the package builds locally you should ensure it builds on a clean system using `pbuilder`. Since we are going to upload to a PPA (Personal Package Archive) shortly, this upload will need to be *signed* to allow Launchpad to verify that the upload comes from you (you can tell the upload will be signed because the `-us` and `-uc` flags are not passed to `bzr builddeb` like they were before). For signing to work you need to have set up GPG. If you haven't set up `pbuilder-dist` or GPG yet, *do so now*:

```
$ bzr builddeb -S
$ cd ../build-area
$ pbuilder-dist precise build hello_2.7-0ubuntu1.dsc
```

When you are happy with your package you will want others to review it. You can upload the branch to Launchpad for review:

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

Uploading it to a PPA will ensure it builds and give an easy way for you and others to test the binary packages. You will need to set up a PPA in Launchpad and then upload with `dput`:

```
$ dput ppa:<lp-username> hello_2.7-0ubuntu1.changes
```

See *uploading* for more information.

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.

1.6.5 Submitting for inclusion

There are a number of paths that a package can take to enter Ubuntu. In most cases, going through Debian first can be the best path. This way ensures that your package will reach the largest number of users as it will be available in not

just Debian and Ubuntu but all of their derivatives as well. Here are some useful links for submitting new packages to Debian:

- 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.

1.7 Security and Stable Release Updates

1.7.1 Fixing a Security Bug in Ubuntu

Introduction

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 10.04 LTS (Lucid Lynx) for a security update.

Obtaining the source

In this example, we already know we want to fix the dbus package in Ubuntu 10.04 LTS (Lucid Lynx). 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 lucid
dbus | 1.2.16-2ubuntu4 |          lucid | source, amd64, i386
dbus | 1.2.16-2ubuntu4.1 | lucid-security | source, amd64, i386
dbus | 1.2.16-2ubuntu4.2 | lucid-updates | source, amd64, i386
```

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 Lucid's dbus, you'll download 1.2.16-2ubuntu4.2 from lucid-updates:

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

Patching the source

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, including *UDD techniques*, 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.

To create your patch using `edit-patch`:

```
$ 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.

Formatting the changelog and patches

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 lucid-security since this is a security update for Lucid) for security updates:

```
dbus (1.2.16-2ubuntu4.3) lucid-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-1.2.16/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.

Test and Submit your work

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 Lucid). If the issue is not fixed in newer Ubuntu versions you should prepare updates for those versions too.

1.7.2 Stable Release Updates

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 `lucid-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.

1.8 Patches to Packages

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.

There are a number of different ways of handling patches in Debian packages, fortunately we are standardizing on one system, Quilt, which is now used by most packages.

Let's look at an example package, `kamoso` in Natty:

```
$ bzr branch ubuntu:natty/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.

The order of patches to apply is kept in `debian/patches/series`.

1.8.1 Patches with Quilt

Before working with Quilt you need to tell it where to find the patches. Add this to your `~/.bashrc`:

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

And source the file to apply the new export:

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

By default all patches are applied already to UDD checkouts or downloaded packages. You can check this with:

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

If you wanted to remove the patch you would run `pop`:

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

No patches applied
```

And to apply a patch you use `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
```

1.8.2 Adding a New Patch

To add a new patch you need to tell Quilt to create a new patch, tell it which files that patch should change, edit the files then refresh the 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
```

The `quilt add` step is important, if you forget it the files will not end up in the patch.

The change will now be in `debian/patches/kubuntu_02_program_description.diff` and the `series` file will have had the new patch added to it. You should add the new file to the packaging:

```
$ 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 keeps its metadata in the `.pc/` directory, so currently you need to add that to the packaging too. This should be improved in future.

As a general rule you should be careful adding patches to programs unless they come from upstream, there is often a good reason why that change has not already been made. The above example changes a user interface string for example, so it would break all translations. If in doubt, do ask the upstream author before adding a patch.

1.8.3 Upgrading to New Upstream Versions

When you upgrade to a new upstream version, patches will often become out of date. They might need to be refreshed to match the new upstream source or they might need to be removed altogether.

You should start by ensuring no patches are applied. Unfortunately a commit is needed before you can merge in the new upstream (this is bug 815854):

```
$ quilt pop -a
$ bzr commit -m "remove patches"
```

Then upgrade to the new version:

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

Then apply the patches one at a time to check for problems:

```
$ 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
```

If it can be reverse-applied this means the patch has been applied already by upstream, so we can delete the patch:

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

Then carry on:

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

It is a good idea to run refresh, this will update the patch relative to the changed upstream source:

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

Then commit as usual:

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

1.8.4 Making A Package Use Quilt

Modern packages use Quilt by default, it is built into the packaging format. Check in `debian/source/format` to ensure it says `3.0 (quilt)`.

Older packages using source format 1.0 will need to explicitly use Quilt, usually by including a makefile into `debian/rules`.

1.8.5 Other Patch Systems

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.

Do not change a package's patch system without discussing it with the Debian maintainer or relevant Ubuntu team. If there is no existing patch system then feel free to add Quilt.

1.9 Shared Libraries

Shared libraries are compiled code which is intended to be shared among several different programs. They are distributed as `.so` files in `/usr/lib/`.

A library exports symbols which are the compiled versions of functions, classes and variables. A library has a name called an SONAME which includes a version number. This SONAME version does not necessarily match the public release version number. A program gets compiled against a given SONAME version of the library. If any of the symbols is removed or changes then the version number needs to be changed which forces any packages using that library to be recompiled against the new version. Version numbers are usually set by upstream and we follow them in our binary package names called an ABI number, but sometimes upstreams do not use sensible version numbers and packagers have to keep separate version numbers.

Libraries are usually distributed by upstream as standalone releases. Sometimes they are distributed as part of a program. In this case they can be included in the binary package along with the program (this is called bundling) if you do not expect any other programs to use the library, more often they should be split out into separate binary packages.

The libraries themselves are put into a binary package named `libfoo1` where `foo` is the name of the library and `1` is the version from the SONAME. Development files from the package, such as header files, needed to compile programs against the library are put into a package called `libfoo-dev`.

1.9.1 An Example

We will use libnova as an example:

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

To find the SONAME of the library run:

```
$ 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.

If upstream makes incompatible changes to their library they will have to reversion their SONAME and we will have to rename our library. Any other packages using our library package will need to recompiled against the new version, this is called a transition and can take some effort. Hopefully our ABI number will continue to match upstream's SONAME but sometimes they introduce incompatibilities without changing their version number and we will need to change ours.

Looking in debian/libnova-0.12-2.install we see it includes two files:

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

The last one is the actual library, complete with minor and point version number. The first one is a symlink which points to the actual library. The symlink is what programs using the library will look for, the running programs do not care about the minor version number.

`libnova-dev.install` includes all the files needed to compile a program with this library. Header files, a config binary, the `.la` libtool file and `libnova.so` which is another symlink pointing at the library, programs compiling against the library do not care about the major version number (although the binary they compile into will).

`.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.

1.9.2 Static Libraries

The -dev package also ships `usr/lib/libnova.a`. This is a static library, an alternative to the shared library. Any program compiled against the static library will include the code directory into itself. This gets round worrying about binary compatibility of the library. However it also means that any bugs, including security issues, will not be updated along with the library until the program is recompiled. For this reason programs using static libraries are discouraged.

1.9.3 Symbol Files

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.

To make the library dependencies more precise we keep `.symbols` files that list all the symbols in a library and the version they appeared in.

libnova has no symbols file so we can create one. Start by compiling the package:

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

The `-nc` will cause it to finish at the end of compilation without removing the built files. Change to the build and run `dpkg-gensymbols` for the library package:

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

This makes a diff file which you can self apply:

```
$ 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
```

Now move the file into its location, commit and do a test build:

```
$ 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
```

If it successfully compiles the symbols file is correct. With the next upstream version of libnova you would run dpkg-gensymbols again and it will give a diff to update the symbols file.

1.9.4 C++ Library Symbols Files

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.

1.9.5 Further Reading

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

1.10 Backporting software updates

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.

1.10.1 Personal Package Archive (PPA)

Using a PPA has a number of benefits. It is fairly straight-forward, you don't need approval of anyone, but the downside of it is that your users will have to manually enable it. It is a non-standard software source.

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

1.10.2 Official Ubuntu Backports

The Backports Project is a means to provide new features to users. Because of the inherent stability risks in backporting packages, users do not get backported packages without some explicit action on their part. This generally makes backports an inappropriate avenue for fixing bugs. If a package in an Ubuntu release has a bug, it should be fixed either through the *Security Update or the Stable Release Update process*, as appropriate.

Once you determined you want a package to be backported to a stable release, you will need to test-build and test it on the given stable release. `pbuilder-dist` (in the `ubuntu-dev-tools` package) is a very handy tool to do this easily.

To report the backport request and get it processed by the Backporters team, you can use the `requestbackport` tool (also in the `ubuntu-dev-tools` package). It will determine the intermediate releases that package needs to be backported to, list all reverse-dependencies, and file the backporting request. Also will it include a testing checklist in the bug.

CHAPTER

TWO

KNOWLEDGE BASE

2.1 Communication in Ubuntu Development

In a project where thousands of lines of code are changed, lots of decisions are made and hundreds of people interact every day, it is important to communicate effectively.

2.1.1 Mailing lists

Mailing lists are a very important tool if you want to communicate ideas to a broader team and make sure that you reach everybody, even across timezones.

In terms of development, these are the most important ones:

- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce (announce-only, the most important development announcements go here)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel (general Ubuntu development discussion)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-motu (MOTU Team discussion, get help with packaging)

2.1.2 IRC Channels

For real-time discussions, please connect to irc.freenode.net and join one or any of these channels:

- #ubuntu-devel (for general development discussion)
- #ubuntu-motu (for MOTU team discussion and generally getting help)

2.2 Basic Overview of the `debian/` Directory

This article will briefly explain the different files important to the packaging of Ubuntu packages which are contained in the `debian/` directory. The most important of them are `changelog`, `control`, `copyright`, and `rules`. These are required for all packages. A number of additional files in the `debian/` may be used in order to customize and configure the behavior of the package. Some of these files are discussed in this article, but this is not meant to be a complete list.

2.2.1 The changelog

This file is, as its name implies, a listing of the changes made in each version. It has a specific format that gives the package name, version, distribution, changes, and who made the changes at a given time. If you have a GPG key (see: *Getting set up*), make sure to use the same name and email address in `changelog` as you have in your key. The following is a template `changelog`:

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

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

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

The format (especially of the date) is important. The date should be in **RFC 5322** format, which can be obtained by using the command `date -R`. For convenience, the command `dch` may be used to edit changelog. It will update the date automatically.

Minor bullet points are indicated by a dash "-", while major points use an asterisk "*".

If you are packaging from scratch, `dch --create` (`dch` is in the `devscripts` package) will create a standard `debian/changelog` for you.

Here is a sample `changelog` file for hello:

```
hello (2.6-0ubuntu1) natty; urgency=low

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

-- Jane Doe <packager@example.com>  Thu, 21 Apr 2011 11:12:00 -0400
```

Notice that the version has a `-0ubuntu1` appended to it, this is the distro revision, used so that the packaging can be updated (to fix bugs for example) with new uploads within the same source release version.

Ubuntu and Debian have slightly different package versioning schemes to avoid conflicting packages with the same source version. If a Debian package has been changed in Ubuntu, it has `ubuntuX` (where `X` is the Ubuntu revision number) appended to the end of the Debian version. So if the Debian hello `2.6-1` package was changed by Ubuntu, the version string would be `2.6-1ubuntu1`. If a package for the application does not exist in Debian, then the Debian revision is `0` (e.g. `2.6-0ubuntu1`).

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

2.2.2 The control file

The `control` file contains the information that the package manager (such as `apt-get`, `synaptic`, and `adept`) uses, build-time dependencies, maintainer information, and much more.

For the Ubuntu `hello` package, the `control` file looks something like this:

```
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: 3.9.1
Build-Depends: debhelper (>= 7)
Bzr-Vcs: 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).
```

The first paragraph describes the source package including the list of packages required to build the package from source in the `Build-Depends` field. It also contains some meta-information such as the maintainer's name, the version of Debian Policy that the package complies with, the location of the packaging version control repository, and the upstream home page.

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.

Each additional paragraph describes a binary package to be built.

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

2.2.3 The copyright file

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`.

Generally, copyright information is found in the `COPYING` file in the program's source directory. This file should include such information as the names of the author and the packager, the URL from which the source came, a Copyright line with the year and copyright holder, and the text of the copyright itself. An example template would be:

```
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 The rules file

The last file we need to look at is `rules`. This does all the work for creating our package. It is a Makefile with targets to compile and install the application, then create the `.deb` file from the installed files. It also has a target to clean up all the build files so you end up with just a source package again.

Here is a simplified version of the rules file created by `dh_make` (which can be found in the `dh-make` package):

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

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

%:
	dh  $@
```

Let us go through this file in some detail. What this does is pass every build target that `debian/rules` is called with as an argument to `/usr/bin/dh`, which itself will call all the necessary `dh_*` commands.

`dh` runs a sequence of debhelper commands. The supported sequences correspond to the targets of a `debian/rules` file: "build", "clean", "install", "binary-arch", "binary-indep", and "binary". In order to see what commands are run in each target, run:

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

Commands in the binary-indep sequence are passed the "-i" option to ensure they only work on binary independent packages, and commands in the binary-arch sequences are passed the "-a" option to ensure they only work on architecture dependent packages.

Each debhelper command will record when it's successfully run in `debian/package.debhelper.log`. (Which dh_clean deletes.) So dh can tell which commands have already been run, for which packages, and skip running those commands again.

Each time `dh` is run, it examines the log, and finds the last logged command that is in the specified sequence. It then continues with the next command in the sequence. The `--until`, `--before`, `--after`, and `--remaining` options can override this behavior.

If `debian/rules` contains a target with a name like `override_dh_command`, then when it gets to that command in the sequence, `dh` will run that target from the rules file, rather than running the actual command. The override target

can then run the command with additional options, or run entirely different commands instead. (Note that to use this feature, you should Build-Depend on debhelper 7.0.50 or above.)

Have a look at `/usr/share/doc/debhelper/examples/` and `man dh` for more examples. Also see the rules section (Section 4.9) of the Debian Policy Manual.

2.2.5 Additional Files

The install file

The `install` file is used by `dh_install` to install files into the binary package. It has two standard use cases:

- To install files into your package that are not handled by the upstream build system.
- Splitting a single large source package into multiple binary packages.

In the first case, the `install` file should have one line per file installed, specifying both the file and the installation directory. For example, the following `install` file would install the script `foo` in the source package's root directory to `usr/bin` and a desktop file in the `debian` directory to `usr/share/applications`:

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

When a source package is producing multiple binary packages `dh` will install the files into `debian/tmp` rather than directly into `debian/<package>`. Files installed into `debian/tmp` can then be moved into separate binary packages using multiple `$package_name.install` files. This is often done to split large amounts of architecture independent data out of architecture dependent packages and into `Architecture: all` packages. In this case, only the name of the files (or directories) to be installed are needed without the installation directory. For example, `foo.install` containing only the architecture dependent files might look like:

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

While `foo-common.install` containing only the architecture independent file might look like:

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

This would create two binary packages, `foo` and `foo-common`. Both would require their own paragraph in `debian/control`.

See `man dh_install` and the install file section (Section 5.11) of the Debian New Maintainers' Guide for additional details.

The watch file

The `debian/watch` file allows us to check automatically for new upstream versions using the tool `uscan` found in the `devscripts` package. The first line of the watch file must be the format version (3, at the time of this writing), while the following lines contain any URLs to parse. For example:

```
version=3

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

Running `uscan` in the root source directory will now compare the upstream version number in `debian/changelog` with the latest available upstream version. If a new upstream version is found, it will be automatically downloaded. For example:

```
$ 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
```

For further information, see `man uscan` and the watch file section (Section 4.11) of the Debian Policy Manual.

For a list of packages where the `watch` file reports they are not in sync with upstream see Ubuntu External Health Status.

The source/format file

This file indicates the format of the source package. It should contain a single line indicating the desired format:

- `3.0 (native)` for Debian native packages (no upstream version)
- `3.0 (quilt)` for packages with a separate upstream tarball
- `1.0` for packages wishing to explicitly declare the default format

Currently, the package source format will default to 1.0 if this file does not exist. You can make this explicit in the source/format file. If you choose not to use this file to define the source format, Lintian will warn about the missing file. This warning is informational only and may be safely ignored.

You are encouraged to use the newer 3.0 source format. It provides a number of new features:

- Support for additional compression formats: bzip2, lzma and xz
- Support for multiple upstream tarballs
- Not necessary to repack the upstream tarball to strip the debian directory
- Debian-specific changes are no longer stored in a single .diff.gz but in multiple patches compatible with quilt under `debian/patches/`

http://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 Additional Resources

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 autopkgtest: Automatic testing for packages

The DEP 8 specification defines how automatic testing can very easily be integrated into packages. To integrate a test into a package, all you need to do is:

- add the following to the Source section in `debian/control`:

  ```
  XS-Testsuite: autopkgtest
  ```

- add a file called `debian/tests/control` which specifies the requirements for the testbed,
- add the tests in `debian/tests/`.

2.3.1 Testbed requirements

In `debian/tests/control` you specify what to expect from the testbed. So for example you list all the required packages for the tests, if the testbed gets broken during the build or if `root` permissions are required. The DEP 8 specification lists all available options.

Below we are having a look at the `glib2.0` source package. In a very simple case the file would look like this:

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

For the test in `debian/tests/build` this would ensure that the packages `libglib2.0-dev` and `build-essential` are installed.

Note: You can use `@` in the `Depends` line to indicate that you want all the packages installed which are built by the source package in question.

2.3.2 The actual tests

The accompanying test for the example above might be:

```
#!/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"
```

Here a very simple piece of C code is written to a temporary directory. Then this is compiled with system libraries (using flags and library paths as provided by *pkg-config*). Then the compiled binary, which just exercises some parts of core glib functionality, is run.

While this test is very small and basic, it tests quite a number of core components on a system. This may help to uncover critical issues early on.

2.3.3 Executing the test

The test script can be easily executed on its own, but if you want to make sure that the testbed is properly set up, you might want to use `adt-run` from the `autopkgtest` package to execute the test. The easiest way to do this is to run this command in the source tree:

```
sudo adt-run --no-built-binaries --built-tree=. --- adt-virt-null
```

The downside of this approach is that you test it locally, but can't ensure that this will work in a minimal environment. For example will it be hard to ensure that all the required packages are installed for the tests. With lp:auto-package-testing we have a more comprehensive testing tool. It uses a pristine virtual machine to run the tests. To set it up, firstly install the needed dependencies:

```
sudo apt-get install qemu-utils kvm eatmydata
```

Then, get the source code from Launchpad:

```
bzr branch lp:auto-package-testing
cd auto-package-testing
```

And provision a Raring AMD64 system:

```
./bin/prepare-testbed -r raring amd64
```

This command will create a pristine Raring AMD64 VM from a cloud image. To run the tests, simply run:

```
./bin/run-adt-test -r raring -a amd64 \
    -S file:///tmp/glib2.0-2.35.7/ glib2.0
```

This would use the source package in `/tmp/glib2.0-2.35.7/` and run the tests from this tree against the package `glib2.0` from the archive. The option `-S` also supports schemes for bzr, git, and apt sources. If you only specify a source with `-S` but do not specify a package name, this will instead build the branch and install the binaries from that build; this is useful if you want to run tests on a newer version than the one packaged in Ubuntu, or the package is not in Ubuntu at all. If use the `-k` flag you can log into the virtual machine after the tests were run. This makes it very easy to debug issues.

The auto-package-testing documentation has a lot more valuable information on other testing options.

2.3.4 Further examples

This list is not comprehensive, but might help you get a better idea of how automated tests are implemented and used in Ubuntu.

- The libxml2 tests are very similar. They also run a test-build of a simple piece of C code and execute it.

- 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 the ubiquity tests the upstream test-suite is executed.
- The gvfs tests have comprehensive testing of their functionality and are very interesting because they emulate usage of CDs, Samba, DAV and other bits.

2.3.5 Ubuntu infrastructure

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

While Debian does not have an automatic testing infrastructure set up yet, they should still be submitted to Debian, as DEP-8 is a Debian specification and Debian developers or users can still manually run the tests.

Packages in Debian with a testsuite header will also be automatically added when they are synced to Ubuntu.

2.3.6 Getting the test into Ubuntu

The process of submitting an autopkgtest for a package is largely similar to *fixing a bug in Ubuntu*. Essentially you simply:

- run `bzr branch ubuntu:<packagename>`,
- edit `debian/control` to enable the tests,
- add the `debian/tests` directory,
- write the `debian/tests/control` based on the DEP 8 Specification,
- add your test case(s) to `debian/tests`,
- commit your changes, push them to Launchpad, propose a merge and get it reviewed just like any other improvement in a source package.

2.3.7 What you can do

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.

If you should run into any problems, you can join the #ubuntu-quality IRC channel to get in touch with developers who can help you.

2.4 Getting the Source

2.4.1 Source package URLs

Bazaar provides some very nice shortcuts for accessing Launchpad's source branches of packages in both Ubuntu and Debian.

To refer to source branches use:

```
ubuntu:package
```

where *package* refers to the package name you're interested in. This URL refers to the package in the current development version of Ubuntu. To refer to the branch of Tomboy in the development version, you would use:

```
ubuntu:tomboy
```

To refer to the version of a source package in an older release of Ubuntu, just prefix the package name with the release's code name. E.g. to refer to Tomboy's source package in Maverick use:

```
ubuntu:maverick/tomboy
```

Since they are unique, you can also abbreviate the distro-series name:

```
ubuntu:m/tomboy
```

You can use a similar scheme to access the source branches in Debian, although there are no shortcuts for the Debian distro-series names. To access the Tomboy branch in the current development series for Debian use:

```
debianlp:tomboy
```

and to access Tomboy in Debian Lenny use:

```
debianlp:lenny/tomboy
```

2.4.2 Getting the source

Every source package in Ubuntu has an associated source branch on Launchpad. These source branches are updated automatically by Launchpad, although the process is not currently foolproof.

There are a couple of things that we do first in order to make the workflow more efficient later. Once you are used to the process you will learn when it makes sense to skip these steps.

Creating a shared repository

Say that you want to work on the Tomboy package, and you've verified that the source package is named `tomboy`. Before actually branching the code for Tomboy, create a shared repository to hold the branches for this package. The shared repository will make future work much more efficient.

Do this using the *bzr init-repo* command, passing it the directory name we would like to use:

```
$ bzr init-repo tomboy
```

You will see that a *tomboy* directory is created in your current working area. Change to this new directory for the rest of your work:

```
$ cd tomboy
```

Getting the trunk branch

We use the *bzr branch* command to create a local branch of the package. We'll name the target directory *tomboy.dev* just to keep things easy to remember:

```
$ bzr branch ubuntu:tomboy tomboy.dev
```

The tomboy.dev directory represents the version of Tomboy in the development version of Ubuntu, and you can always `cd` into this directory and do a *bzr pull* to get any future updates.

Ensuring the version is up to date

When you do your `bzr branch` you will get a message telling you if the packaging branch is up to date. For example:

```
$ bzr branch ubuntu:tomboy
Most recent Ubuntu version: 1.8.0-1ubuntu1.2
Packaging branch status: CURRENT
Branched 86 revisions.
```

Occasionally the importer fails and packaging branches do not match what is in the archive. A message saying:

```
Packaging branch status: OUT-OF-DATE
```

means the importer has failed. You can find out why on http://package-import.ubuntu.com/status/ and file a bug on the UDD project to get the issue resolved.

Upstream Tar

You can get the upstream tar by running:

```
bzr get-orig-source
```

This will try a number of methods to get the upstream tar, firstly by recreating it from the `upstream-x.y` tag in the bzr archive, then by downloading from the Ubuntu archive, lastly by running `debian/rules get-orig-source`. The upstream tar will also be recreated when using bzr to build the package:

```
bzr builddeb
```

The *builddeb* plugin has several configuration options.

Getting a branch for a particular release

When you want to do something like a stable release update (SRU), or you just want to examine the code in an old release, you'll want to grab the branch corresponding to a particular Ubuntu release. For example, to get the Tomboy package for Maverick do:

```
$ bzr branch ubuntu:m/tomboy maverick
```

Importing a Debian source package

If the package you want to work on is available in Debian but not Ubuntu, it's still easy to import the code to a local bzr branch for development. Let's say you want to import the *newpackage* source package. We'll start by creating a shared repository as normal, but we also have to create a working tree to which the source package will be imported (remember to cd out of the *tomboy* directory created above):

```
$ bzr init-repo newpackage
$ cd newpackage
$ bzr init debian
$ cd debian
$ bzr import-dsc http://ftp.de.debian.org/debian/pool/main/n/newpackage/newpackage_1.0-1.dsc
```

As you can see, we just need to provide the remote location of the dsc file, and Bazaar will do the rest. You've now got a Bazaar source branch.

2.5 Working on a Package

Once you have the source package branch in a shared repository, you'll want to create additional branches for the fixes or other work you plan to do. You'll want to base your branch off the package source branch for the distro release that you plan to upload to. Usually this is the current development release, but it may be older releases if you're backporting to an SRU for example.

2.5.1 Branching for a change

The first thing to do is to make sure your source package branch is up-to-date. It will be if you just checked it out, otherwise do this:

```
$ cd tomboy.dev
$ bzr pull
```

Any updates to the package that have uploaded since your checkout will now be pulled in. You do not want to make changes to this branch. Instead, create a branch that will contain just the changes you're going to make. Let's say you want to fix bug 12345 for the Tomboy project. When you're in the shared repository you previously created for Tomboy, you can create your bug fix branch like this:

```
$ bzr branch tomboy.dev bug-12345
$ cd bug-12345
```

Now you can do all my work in the `bug-12345` directory. You make changes there as necessary, committing as you go along. This is just like doing any kind of software development with Bazaar. You can make intermediate commits as often as you like, and when your changes are finished, you will use the standard `dch` command (from the `devscripts` package):

```
$ dch -i
```

This will drop you in an editor to add an entry to the *debian/changelog* file. When you added your `debian/changelog` entry, you should have included a bug fix tag that indicated which Launchpad bug issue you're fixing. The format of this textual tag is pretty strict: `LP: #12345`. The space between the `:` and the `#` is required and of course you should use the actual bug number that you're fixing. Your `debian/changelog` entry might look something like:

```
tomboy (1.5.2-1ubuntu5) natty; urgency=low

    * Don't fubar the frobnicator. (LP: #12345)

 -- Bob Dobbs <subgenius@example.com>  Mon, 10 Jan 2011 16:10:01 -0500
```

Commit with the normal:

```
bzr commit
```

A hook in bzr-builddeb will use the debian/changelog text as the commit message and set the tag to mark bug #12345 as fixed.

This only works with bzr-builddeb 2.7.5 and bzr 2.4, for older versions use `debcommit`.

2.5.2 Building the package

Along the way, you'll want to build your branch so that you can test it to make sure it does actually fix the bug.

In order to build the package you can use the `bzr builddeb` command from the `bzr-builddeb` package. You can build a source package with:

```
$ bzr builddeb -S
```

(`bd` is an alias for `builddeb`.) You can leave the package unsigned by appending `-- -uc -us` to the command.

It is also possible to use your normal tools, as long as they are able to strip the .bzr directories from the package, e.g.:

```
$ debuild -i -I
```

If you ever see an error related to trying to build a native package without a tarball, check to see if there is a `.bzr-builddeb/default.conf` file erroneously specifying the package as native. If the changelog version has a dash in it, then it's not a native package, so remove the configuration file. Note that while `bzr builddeb` has a `--native` switch, it does not have a `--no-native` switch.

Once you've got the source package, you can build it as normal with `pbuilder-dist` (or `pbuilder` or sbuild).

2.6 Seeking Review and Sponsorship

One of the biggest advantages to using the UDD workflow is to improve quality by seeking review of changes by your peers. This is true whether or not you have upload rights yourself. Of course, if you don't have upload rights, you will need to seek sponsorship.

Once you are happy with your fix, and have a branch ready to go, the following steps can be used to publish your branch on Launchpad, link it to the bug issue, and create a *merge proposal* for others to review, and sponsors to upload.

2.6.1 Pushing to Launchpad

We previously showed you how to *associate your branch to the bug* using `dch` and `bzr commit`. However, the branch and bug don't actually get linked until you push the branch to Launchpad.

It is not critical to have a link to a bug for every change you make, but if you are fixing reported bugs then linking to them will be useful.

The general form of the URL you should push your branch to is:

```
lp:~<user-id>/ubuntu/<distroseries>/<package>/<branch-name>
```

For example, to push your fix for bug 12345 in the Tomboy package for Natty, you'd use:

```
$ bzr push lp:~subgenius/ubuntu/natty/tomboy/bug-12345
```

The last component of the path is arbitrary; it's up to you to pick something meaningful.

However, this usually isn't enough to get Ubuntu developers to review and sponsor your change. You should next submit a *merge proposal*.

To do this open the bug page in a browser, e.g.:

```
$ bzr lp-open
```

If that fails, then you can use:

```
$ xdg-open https://code.launchpad.net/~subgenius/ubuntu/natty/tomboy/bug-12345
```

where most of the URL matches what you used for *bzr push*. On this page, you'll see a link that says *Propose for merging into another branch*. Type in an explanation of your change in the *Initial Comment* box. Lastly, click *Propose Merge* to complete the process.

Merge proposals to package source branches will automatically subscribe the *~ubuntu-branches* team, which should be enough to reach an Ubuntu developer who can review and sponsor your package change.

2.6.2 Generating a debdiff

As noted above, some sponsors still prefer reviewing a *debdiff* attached to bug reports instead of a merge proposal. If you're requested to include a debdiff, you can generate one like this (from inside your *bug-12345* branch):

```
$ bzr diff -rbranch:../tomboy.dev
```

Another way is to is to open the merge proposal and download the diff.

You should ensure that diff has the changes you expect, no more and no less. Name the diff appropriately, e.g. `foobar-12345.debdiff` and attach it to the bug report.

2.6.3 Dealing with feedback from sponsors

If a sponsor reviews your branch and asks you to change something, you can do this fairly easily. Simply go to the branch that you were working in before, make the changes requested, and then commit:

```
$ bzr commit
```

Now when you push your branch to Launchpad, Bazaar will remembered where you pushed to, and will update the branch on Launchpad with your latest commits. All you need to do is:

```
$ bzr push
```

You can then reply to the merge proposal review email explaining what you changed, and asking for re-review, or you can reply on the merge proposal page in Launchpad.

Note that if you are sponsored via a debdiff attached to a bug report you need to manually update by generating a new diff and attaching that to the bug report.

2.6.4 Expectations

The Ubuntu developers have set up a schedule of "patch pilots", who regularly review the sponsoring queue and give feedback on branches and patches. Even though this measure has been put in place it might still take several days until you hear back. This depends on how busy everybody is, if the development release is currently frozen, or other factors.

If you haven't heard back in a while, feel free to join *#ubuntu-devel* on *irc.freenode.net* and find out if somebody can help you there.

For more information on the generall sponsorship process, review the documentation on our wiki as well: https://wiki.ubuntu.com/SponsorshipProcess

2.7 Uploading a package

Once your merge proposal is reviewed and approved, you will want to upload your package, either to the archive (if you have permission) or to your Personal Package Archive (PPA). You might also want to do an upload if you are

sponsoring someone else's changes.

2.7.1 Uploading a change made by you

When you have a branch with a change that you would like to upload you need to get that change back on to the main source branch, build a source package, and then upload it.

First, you need to check that you have the latest version of the package in your checkout of the development package trunk:

```
$ cd tomboy/tomboy.dev
$ bzr pull
```

This pulls in any changes that may have been committed while you were working on your fix. From here, you have several options. If the changes on the trunk are large and you feel should be tested along with your change you can merge them into your bug fix branch and test there. If not, then you can carry on merging your bug fix branch into the development trunk branch. As of bzr 2.5 and bzr-builddeb 2.8.1, this works with just the standard `merge` command:

```
$ bzr merge ../bug-12345
```

For older versions of bzr, you can use the `merge-package` command instead:

```
$ bzr merge-package ../bug-12345
```

This will merge the two trees, possibly producing conflicts, which you'll need to resolve manually.

Next you should make sure the `debian/changelog` is as you would like, with the correct distribution, version number, etc.

Once that is done you should review the change you are about to commit with `bzr diff`. This should show you the same changes as a debdiff would before you upload the source package.

The next step is to build and test the modified source package as you normally would:

```
$ bzr builddeb -S
```

When you're finally happy with your branch, make sure you've committed all your changes, then tag the branch with the changelog's version number. The `bzr tag` command will do this for you automatically when given no arguments:

```
$ bzr tag
```

This tag will tell the package importer that what is in the Bazaar branch is the same as in the archive.

Now you can push the changes back to Launchpad:

```
$ bzr push ubuntu:tomboy
```

(Change the destination if you are uploading an SRU or similar.)

You need one last step to get your changes uploaded into Ubuntu or your PPA; you need to `dput` the source package to the appropriate location. For example, if you want to upload your changes to your PPA, you'd do:

```
$ dput ppa:imasponsor/myppa tomboy_1.5.2-1ubuntu5_source.changes
```

or, if you have permission to upload to the primary archive:

```
$ dput tomboy_1.5.2-1ubuntu5_source.changes
```

You are now free to delete your feature branch, as it is merged, and can be re-downloaded from Launchpad if needed.

2.7.2 Sponsoring a change

Sponsoring someone else's change is just like the above procedure, but instead of merging from a branch you created, you merge from the branch in the merge proposal:

```
$ bzr merge lp:~subgenius/ubuntu/natty/tomboy/bug-12345
```

If there are lots of merge conflicts you would probably want to ask the contributor to fix them up. See the next section to learn how to cancel a pending merge.

But if the changes look good, commit and then follow the rest of the uploading process:

```
$ bzr commit --author "Bob Dobbs <subgenius@example.com>"
```

2.7.3 Canceling an upload

At any time before you *dput* the source package you can decide to cancel an upload and revert the changes:

```
$ bzr revert
```

You can do this if you notice something needs more work, or if you would like to ask the contributor to fix up conflicts when sponsoring something.

2.7.4 Sponsoring something and making your own changes

If you are going to sponsor someone's work, but you would like to roll it up with some changes of your own then you can merge their work in to a separate branch first.

If you already have a branch where you are working on the package and you would like to include their changes, then simply run the `bzr merge` from that branch, instead of the checkout of the development package. You can then make the changes and commit, and then carry on with your changes to the package.

If you don't have an existing branch, but you know you would like to make changes based on what the contributor provides then you should start by grabbing their branch:

```
$ bzr branch lp:~subgenius/ubuntu/natty/tomboy/bug-12345
```

then work in this new branch, and then merge it in to the main one and upload as if it was your own work. The contributor will still be mentioned in the changelog, and Bazaar will correctly attribute the changes they made to them.

2.8 Getting The Latest

If someone else has landed changes on a package, you will want to pull those changes in your own copies of the package branches.

2.8.1 Updating your main branch

Updating your copy of a branch that corresponds to the package in a particular release is very simple, simply use *bzr pull* from the appropriate directory:

```
$ cd tomboy/tomboy.dev
$ bzr pull
```

This works wherever you have a checkout of a branch, so it will work for things like branches of *maverick*, *hardy-proposed*, etc.

2.8.2 Getting the latest in to your working branches

Once you have updated your copy of a distroseries branch, then you may want to merge this in to your working branches as well, so that they are based on the latest code.

You don't have to do this all the time though. You can work on slightly older code with no problems. The disadvantage would come if you were working on some code that someone else changed. If you are not working on the latest version then your changes may not be correct, and may even produce conflicts.

The merge does have to be done at some point though. The longer it is left, the harder may be, so doing it regularly should keep each merge simple. Even if there are many merges the total effort would hopefully be less.

To merge the changes you just need to use `bzr merge`, but you must have committed your current work first:

```
$ cd tomboy/bug-12345
$ bzr merge ../tomboy.dev
```

Any conflicts will be reported, and you can fix them up. To review the changes that you just merged use `bzr diff`. To undo the merge use `bzr revert`. Once you are happy with the changes then use `bzr commit`.

2.8.3 Referring to versions of a package

You will often think in terms of versions of a package, rather than the underlying Bazaar revision numbers. *bzr-builddeb* provides a revision specifier that makes this convenient. Any command that takes a `-r` argument to specify a revision or revision range will work with this specifier, e.g. `bzr log`, `bzr diff`, and so on. To view the versions of a package, use the `package:` specifier:

```
$ bzr diff -r package:0.1-1..package:0.1-2
```

This shows you the difference between package version 0.1-1 and 0.1-2.

2.9 Merging — Updating from Debian and Upstream

Merging is one of the strengths of Bazaar, and something we do often in Ubuntu development. Updates can be merged from Debian, from a new upstream release, and from other Ubuntu developers. Doing it in Bazaar is pretty simple, and all based around the `bzr merge` command [1].

While you are in any branch's working directory, you can merge in a branch from a different location. First check that you have no uncommitted changes:

```
$ bzr status
```

If that reports anything then you will either have to commit the changes, revert them, or shelve them to come back to later.

[1] You will need newer versions of `bzr` and the `bzr-builddeb` for the `merge` command to work. Use the versions from Ubuntu 12.04 (Precise) or the development versions from the `bzr` PPA. Specifically, you need `bzr` version 2.5 beta 5 or newer, and `bzr-builddeb` version 2.8.1 or newer. For older versions, use the `bzr merge-package` command instead.

2.9.1 Merging from Debian

Next run `bzr merge` passing the URL of the branch to merge from. For example, to merge from the version of the package in Debian Squeeze run:

```
$ bzr merge debianlp:squeeze/tomboy
```

This will merge the changes since the last merge point and leave you with changes to review. This may cause some conflicts. You can see everything that the `merge` command did by running:

```
$ bzr status
$ bzr diff
```

If conflicts are reported then you need to edit those files to make them look how they should, removing the *conflict markers*. Once you have done this, run:

```
$ bzr resolve
$ bzr conflicts
```

This will resolve any conflicted files that you fixed, and then tell you what else you have to deal with.

Once any conflicts are resolved, and you have made any other changes that you need, you will add a new changelog entry, and commit:

```
$ dch -i
$ bzr commit
```

as described earlier.

However, before you commit, it is always a good thing to check all the Ubuntu changes by running:

```
$ bzr diff -r tag:0.6.10-5
```

which will show the differences between the Debian (0.6.10-5) and Ubuntu versions (0.6.10-5ubuntu1). In similar way you can compare to any other versions. To see all available versions run:

```
$ bzr tags
```

After testing and committing the merge, you will need to seek sponsorship or upload to the archive in the normal way.

If you are going to build the source package from this merged branch, you would use the `-S` option to the `bd` command. One other thing you'll want to consider is also using the `--package-merge` option. This will add the appropriate `-v` and `-sa` options to the source package so that all the changelog entries since the last Ubuntu change will be included in your `_source.changes` file. For example:

```
$ bzr builddeb -S --package-merge
```

2.9.2 Merging a new upstream version

When upstream releases a new version (or you want to package a snapshot), you have to merge a tarball into your branch.

This is done using the `bzr merge-upstream` command. If your package has a valid `debian/watch` file, from inside the branch that you want to merge to, just type this:

```
$ bzr merge-upstream
```

This will download the tarball and merge it into your branch, automatically adding a `debian/changelog` entry for you. `bzr-builddeb` looks at the `debian/watch` file for the upstream tarball location.

If you do *not* have a `debian/watch` file, you'll need to specify the location of the upstream tarball, and the version manually:

```
$ bzr merge-upstream --version 1.2 http://example.org/releases/foo-1.2.tar.gz
```

The `--version` option is used to specify the upstream version that is being merged in, as the command isn't able to infer that (yet).

The last parameter is the location of the tarball that you are upgrading to; this can either be a local filesystem path, or a http, ftp, sftp, etc. URI as shown. The command will automatically download the tarball for you. The tarball will be renamed appropriately and, if required, converted to `.gz`.

The *merge-upstream* command will either tell you that it completed successfully, or that there were conflicts. Either way you will be able to review the changes before committing as normal.

If you are merging an upstream release into an existing Bazaar branch that has not previously used the UDD layout, `bzr merge-upstream` will fail with an error that the tag for the previous upstream version is not available; the merge can't be completed without knowing what base version to merge against. To work around this, create a tag in your existing repository for the last upstream version present there; e.g., if the last Ubuntu release was *1.1-0ubuntu3*, create the tag *upstream-1.1* pointing to the bzr revision you want to use as the tip of the upstream branch.

2.10 Using Chroots

If you are running one version of Ubuntu but working on packages for another versions you can create the environment of the other version with a `chroot`.

A `chroot` allows you to have a full filesystem from another distribution which you can work in quite normally. It avoids the overhead of running a full virtual machine.

2.10.1 Creating a Chroot

Use the command `debootstrap` to create a new chroot:

```
$ sudo debootstrap oneiric oneiric/
```

This will create a directory `oneiric` and install a minimal oneiric system into it.

If your version of `debootstrap` does not know about oneiric you can try upgrading to the version in `backports`.

You can then work inside the chroot:

```
$ sudo chroot oneiric
```

Where you can install or remove any package you wish without affecting your main system.

You might want to copy your GPG/ssh keys and Bazaar configuration into the chroot so you can access and sign packages directly:

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

To stop apt and other programs complaining about missing locales you can install your relevant language pack:

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

If you want to run X programs you will need to bind the /tmp directory into the chroot, from outside the chroot run:

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

Some programs may need you to bind /dev or /proc.

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

2.10.2 Alternatives

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.

You can also set up pbuilder to pause when it comes across a build failure. Copy C10shell from /usr/share/doc/pbuilder/examples into a directory and use the `--hookdir=` argument to point to it.

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.

2.11 Traditional Packaging

The majority of this guide deals with *Ubuntu Distributed Development* (UDD) which utilizes the distributed version control system (DVCS) Bazaar for *retrieving package sources* and submitting fixes with *merge proposals.* This article will discuss what we will call traditional packaging methods for lack of a better word. Before Bazaar was adopted for Ubuntu development, these were the typical methods for contributing to Ubuntu.

In some cases, you may need to use these tools instead of UDD. So it is good to be familiar with them. Before you begin, you should already have read the article *Getting Set Up.*

2.11.1 Getting the Source

In order to get a source package, you can simply run:

```
$ apt-get source <package_name>
```

This method has some draw backs though. It downloads the version of the source that is available on **your system.** You are likely running the current stable release, but you want to contribute your change against the package in the development release. Luckily, the `ubuntu-dev-tools` package provides a helper script:

```
$ pull-lp-source <package_name>
```

By default, the latest version in the development release will be downloaded. You can also specify a version or Ubuntu release like:

```
$ pull-lp-source <package_name> precise
```

to pull the source from the `precise` release, or:

```
$ pull-lp-source <package_name> 1.0-1ubuntu1
```

to download version `1.0-1ubuntu1` of the package. For more information on the command, see `man pull-lp-source`.

For our example, let's pretend we got a bug report saying that "colour" in the description of `xicc` should be "color," and we want to fix it. *(Note: This is just an example of something to change and not really a bug.)* To get the source, run:

```
$ pull-lp-source xicc 0.2-3
```

2.11.2 Creating a Debdiff

A `debdiff` 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
```

To continue with our example, let's edit the `debian/control` and "fix" our "bug":

```
$ cd xicc-0.2
$ sed -i 's/colour/color/g' debian/control
```

We also must adhere to the Debian Maintainer Field Spec and edit `debian/control` to replace:

```
Maintainer: Ross Burton <ross@debian.org>
```

with:

```
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Ross Burton <ross@debian.org>
```

You can use the `update-maintainer` tool (in the `ubuntu-dev-tools` package) to do that.

Remember to document your changes in `debian/changelog` using `dch -i` and then we can generate a new source package:

```
$ debuild -S
```

Now we can examine our changes using `debdiff`:

```
$ cd ..
$ debdiff xicc_0.2-3.dsc xicc_0.2-3ubuntu1.dsc | less
```

To create a patch file that you can send to others or attach to a bug report for sponsorship, run:

```
$ debdiff xicc_0.2-3.dsc xicc_0.2-3ubuntu1.dsc > xicc_0.2-3ubuntu1.debdiff
```

2.11.3 Applying a Debdiff

In order to apply a debdiff, first make sure you have the source code of the version that it was created against:

```
$ pull-lp-source xicc 0.2-3
```

Then in a terminal, change the to the directory where the source was uncompressed:

```
$ cd xicc-0.2
```

A debdiff is just like a normal patch file. Apply it as usual with:

```
$ patch -p1 < ../xicc_0.2.2ubuntu1.debdiff
```

2.12 Packaging Python modules and applications

Our packaging follows Debian's Python policy. We will use the python-markdown package as an example, which can be downloaded from PyPI. You can look at its packaging at its Subversion repository.

There are two types of Python packages — *modules* and *apps*.

At the time of writing, Ubuntu has two incompatible versions of Python — *2.x* and *3.x*. `/usr/bin/python` is a symbolic link to a default Python 2.x version, and `/usr/bin/python3` — to a default Python 3.x version. Python modules should be built against all supported Python versions.

If you are going to package a new Python module, you might find the `py2dsc` tool useful (available in python-stdeb package).

2.12.1 debian/control

Python 2.x and 3.x versions of the package should be in separate binary packages. Names should have `python{,3}-modulename` format (like: `python3-dbus.mainloop.qt`). Here, we will use `python-markdown` and `python3-markdown` for module packages and `python-markdown-doc` for the documentation package.

Things in `debian/control` that are specific for a Python package:

- The section of module packages should be `python`, and `doc` for the documentation package. For an application, a single binary package will be enough.
- We should add build dependencies on `python-all (>= 2.6.6-3~)` and `python3-all (>= 3.1.2-7~)` to make sure Python helpers are available (see the next section for details).
- It's recommended to add `X-Python-Version` and `X-Python3-Version` fields — see "Specifying Supported Versions" section of the Policy for details. For example:

  ```
  X-Python-Version: >= 2.6
  X-Python3-Version: >= 3.1
  ```

  If your package works only with Python 2.x or 3.x, build depend only on one `-all` package and use only one `-Version` field.
- Module packages should have `{python:Depends}` and `{python3:Depends}` substitution variables (respectively) in their dependency lists.

2.12.2 debian/rules

The recommended helpers for python modules are `dh_python2` and `dh_python3`. Unfortunately, `debhelper` doesn't yet build Python 3.x packages automatically (see bug 597105 in Debian BTS), so we'll need to do that manually in override sections (skip this if your package doesn't support Python 3.x).

Here's our `debian/rules` file (with annotations):

```
# These commands build the list of supported Python 3 versions
# The last version should be just "python3" so that the scripts
# get a correct shebang.
# Use just "PYTHON3 := $(shell py3versions -r)" if your package
# doesn't contain scripts
```

```
PY3REQUESTED := $(shell py3versions -r)
PY3DEFAULT := $(shell py3versions -d)
PYTHON3 := $(filter-out $(PY3DEFAULT),$(PY3REQUESTED)) python3

%:
    # Adding the required helpers
    dh $@ --with python2,python3

override_dh_auto_clean:
    dh_auto_clean
    rm -rf build/

override_dh_auto_build:
    # Build for each Python 3 version
    set -ex; for python in $(PYTHON3); do \
        $$python setup.py build; \
    done
    dh_auto_build

override_dh_auto_install:
    # The same for install; note the --install-layout=deb option
    set -ex; for python in $(PYTHON3); do \
        $$python setup.py install --install-layout=deb --root=debian/tmp; \
    done
    dh_auto_install
```

It is also a good practice to run tests during the build, if they are shipped by upstream. Usually tests can be invoked using `setup.py test` or `setup.py check`.

2.12.3 debian/*.install

Python 2.x modules are installed into `/usr/share/pyshared/` directory, and symbolic links are created in `/usr/lib/python2.x/dist-packages/` for every interpreter version, while Python 3.x ones are all installed into `/usr/lib/python3/dist-packages/`.

If your package is an application and has private Python modules, they should be installed in `/usr/share/module`, or `/usr/lib/module` if the modules are architecture-dependent (e.g. extensions) (see "Programs Shipping Private Modules" section of the Policy).

So, our `python-markdown.install` file will look like this (we'll also want to install a `markdown_py` executable):

```
usr/lib/python2.*/
usr/bin/
```

and `python3-markdown.install` will only have one line:

```
usr/lib/python3/
```

2.12.4 The `-doc` package

The tool most commonly used for building Python docs is Sphinx. To add Sphinx documentation to your package (using `dh_sphinxdoc` helper), you should:

- Add a build-dependency on `python-sphinx` or `python3-sphinx` (depending on what Python version do you want to use);

- Append `sphinxdoc` to the `dh --with` line;
- Run `setup.py build_sphinx` in `override_dh_auto_build` (sometimes not needed);
- Add `{sphinxdoc:Depends}` to the dependency list of your `-doc` package;
- Add the path of the built docs directory (usually `build/sphinx/html`) to your `.docs` file.

In our case, the docs are automatically built in `build/docs/` directory when we run `setup.py build`, so we can simply put this in the `python-markdown-doc.docs` file:

```
build/docs/
```

Because docs also contain source `.txt` files, we'll also tell `dh_compress` to not compress them — by adding this to `debian/rules`:

```
override_dh_compress:
    dh_compress -X.txt
```

2.12.5 Checking for packaging mistakes

Along with `lintian`, there is a special tool for checking Python packages — `lintian4py`. It is available in the lintian4python package. For example, these two commands invoke both versions of `lintian` and check source and binary packages:

```
lintian -EI --pedantic *.dsc *.deb
lintian4py -EI --pedantic *.dsc *.deb
```

Here, `-EI` option is used to enable experimental and informational tags.

2.12.6 See also

- The Python policy;
- Python/Packaging article on Debian wiki;
- Python/LibraryStyleGuide and Python/AppStyleGuide articles on Debian wiki;
- Debian python-modules and python-apps teams.

2.13 KDE Packaging

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.

2.13.1 Patching Policy

Kubuntu does not add patches to KDE programs unless they come from the upstream authors or submitted upstream with the expectation they will be merged soon or we have consulted the issue with the upstream authors.

Kubuntu does not change the branding of packages except where upstream expects this (such as the top left logo of the Kickoff menu) or to simplify (such as removing splash screens).

2.13.2 debian/rules

Debian packages include some additions to the basic Debhelper usage. These are kept in the `pkg-kde-tools` package.

Packages which use Debhelper 7 should add the `--with=kde` option. This will ensure the correct build flags are used and add options such as handling kdeinit stubs and translations:

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

Some newer KDE packages use the `dhmk` system, an alternative to `dh` made by the Debian Qt/KDE team. You can read about it in /usr/share/pkg-kde-tools/qt-kde-team/2/README. Packages using this will `include /usr/share/pkg-kde-tools/qt-kde-team/2/debian-qt-kde.mk` instead of running `dh`.

2.13.3 Translations

Packages in main have their translations imported into Launchpad and exported from Launchpad into Ubuntu's language-packs.

So any KDE package in main must generate translation templates, include or make available upstream translations and handle `.desktop` file translations.

To generate translation templates the package must include a `Messages.sh` file; complain to the upstream if it does not. You can check it works by running `extract-messages.sh` which should produce one or more `.pot` files in `po/`. This will be done automatically during build if you use the `--with=kde` option to `dh`.

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 dpm to do this.

If a package is moved from universe to main it will need to be re-uploaded before the translations get imported into Launchpad.

`.desktop` files also need translations. We patch KDELibs to read translations out of `.po` files which are pointed to by a line `X-Ubuntu-Gettext-Domain=` added to `.desktop` files at package build time. A .pot file for each package is be generated at build time and .po files need to be downloaded from upstream and included in the package or in our language packs. The list of .po files to be downloaded from KDE's repositories is in `/usr/lib/kubuntu-desktop-i18n/desktop-template-list`.

2.13.4 Library Symbols

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.