Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

UBUNTU DEVELOPMENT SPECIAL EDITION

UBUNTU DEVELOPMENT SERIES SPECIAL EDITION

SINGLE TOPIC SERIES FROM MAGAZINE ISSUES #49 - #52

About Full Circle

Full Circle is a free, independent, magazine dedicated to the Ubuntu family of Linux operating systems. Each month, it contains helpful how-to articles and reader-submitted stories.

Full Circle also features a companion podcast, the Full Circle Podcast which covers the magazine, along with other news of interest.

Please note: this Special Edition is provided with absolutely no warranty whatsoever; neither the contributors nor Full Circle Magazine accept any responsibility or liability for loss or damage resulting from readers choosing to apply this content to theirs or others computers and equipment.

Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

Welcome to another 'single-topic special'

In response to reader requests, we are assembling the content of some of our serialised articles into dedicated editions.

For now, this is a straight reprint of the series **'Ubuntu Development', Parts 1-4, by Daniel Holbach** from issues #49 through #52; nothing fancy, just the facts.

Please bear in mind the original publication date; current versions of hardware and software may differ from those illustrated, so check your hardware and software versions before attempting to emulate the tutorials in these special editions. You may have later versions of software installed or available in your distributions' repositories.

Enjoy!

Find Us

Website:
http://www.fullcirclemagazine.org/

Forums:
http://ubuntuforums.org/forumdisplay.php?f=270

IRC: #fullcirclemagazine on chat.freenode.net

Editorial Team

Editor: Ronnie Tucker
(aka: RonnieTucker)
ronnie@fullcirclemagazine.org

Webmaster: Rob Kerfia
(aka: admin / linuxgeekery-
admin@fullcirclemagazine.org

Podcaster: Robin Catling
(aka RobinCatling)
podcast@fullcirclemagazine.org

Communications Manager:
Robert Clipsham
(aka: mrmonday) -
mrmonday@fullcirclemagazine.org

Intro To Ubuntu Development

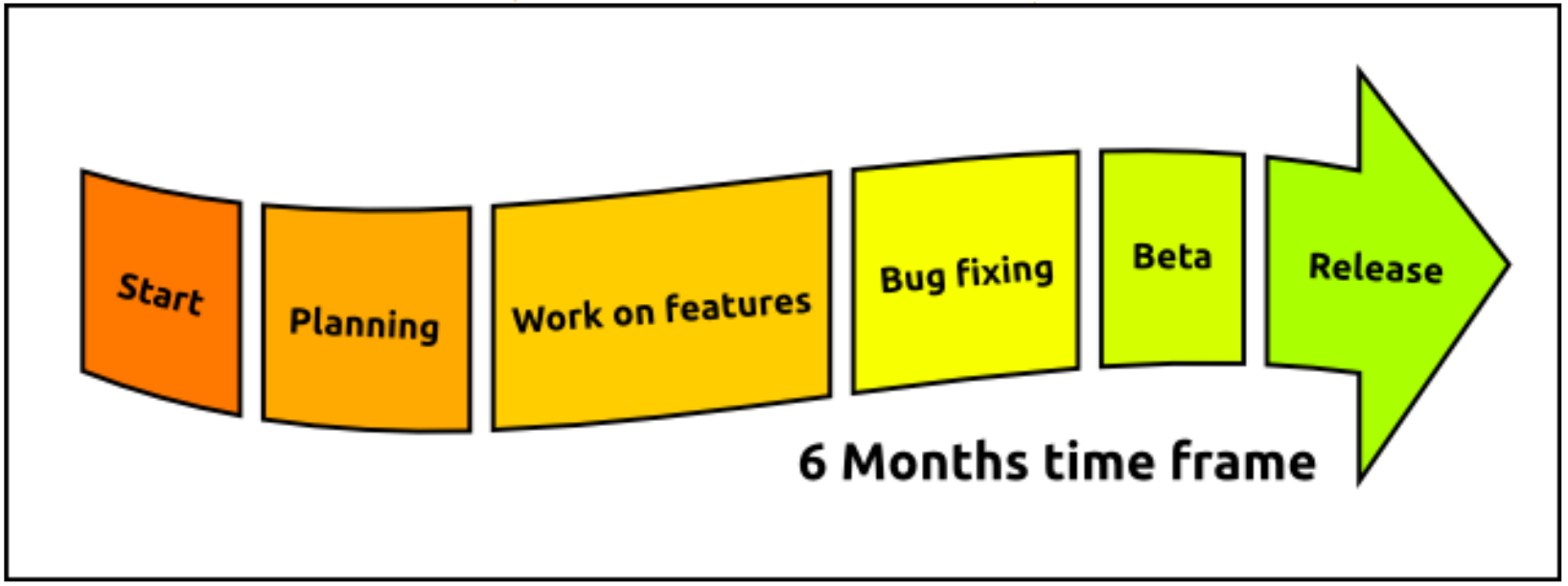
Written by Daniel Holbach

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 the 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 possible only 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.

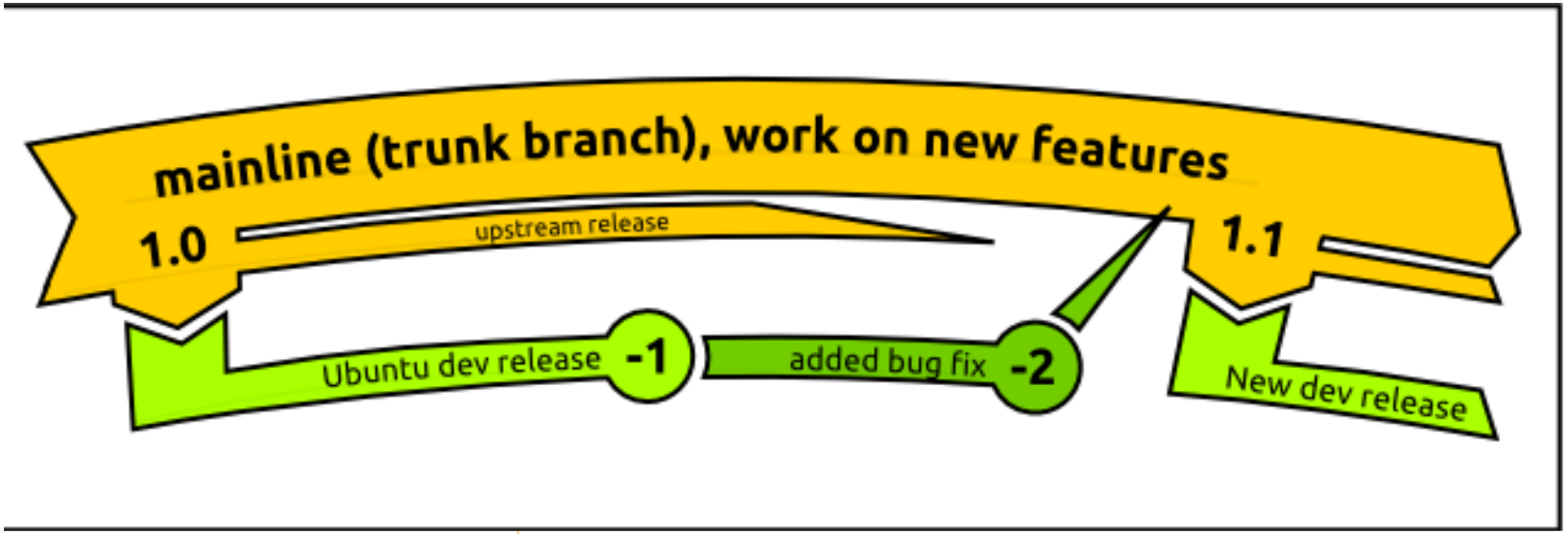
Thousands of source packages, billions of lines of code, and hundreds of contributors, require a lot of communication and planning to maintain high standards of quality. At the beginning of each release cycle, we have the Ubuntu Developer Summit where developers and contributors come together to plan 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 contributers 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 might be interested in working on a new big feature, while Ubuntu, because of the tight release schedule, might be

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.

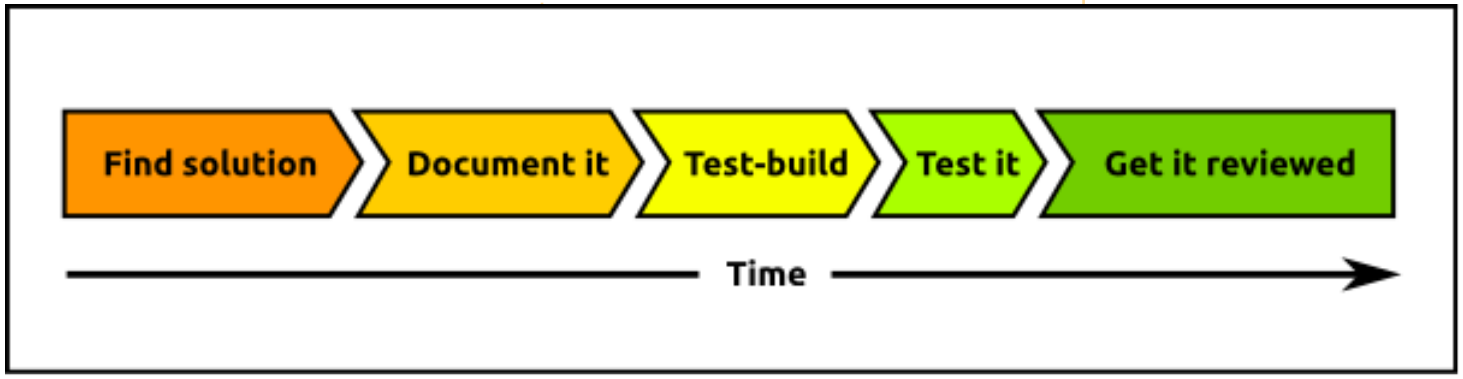
In the example mentioned above, it would make sense to ship Ubuntu with the existing version of the project, add the bugfix, get it into Upstream for their next release, and ship that (if suitable) in the next Ubuntu release. It would be the best possible compromise and a situation where everybody wins.

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-mentors@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.



A PLEA ON BEHALF OF THE PODCAST PARTY

As you heard in episode #15 of the podcast, we're calling for opinion topics for that section of the show.

Instead of us having a rant about whatever strikes us, why not prompt us with a topic and watch for the mushroom clouds over the horizon! It's highly unlikely that the three of us will agree.

Or, an even more radical thought, send us an opinion by way of a contribution!

You can post comments and opinions on the podcast page at fullcirclemagazine.org, in our Ubuntu Forums section, or email podcast@fullcirclemagazine.org. You can also send us a comment by recording an audio clip of no more than 30 seconds and sending it to the same address.

Comments and audio may be edited for length. Please remember this is a family-friendly show.

It would be great to have contributors come on the show and express an opinion in person.

Ubuntu Development Pt. 2- Set Up

Written by Daniel Holbach

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 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 (https://wiki.ubuntu.com/UsingDevelopmentReleases) shows a variety of ways to safely use the development release.

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 required tools, run this command:

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

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 tools that 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.
apt-cache (part of the apt package) provides even more information about packages on Ubuntu.

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 accept the package only 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.

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!

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 communicate with Launchpad.

To generate a 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. Edit ~/.pbuilderrc and add the following line to it:

```
COMPONENTS="main universe multiverse restricted"
```

This will ensure that build dependencies are satisfied using all components. Then 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.

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.

Get a Launchpad account

If you don't already have a Launchpad account, you can easily create one (at: https://launchpad.net/+login). If you have a Launchpad account but cannot remember your Launchpad id, you can find this out by going to https://launchpad.net/people/+me, 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 (https://help.launchpad.net/YourAccount/NewAccount) on Launchpad has more information about the process, and additional settings you can change.

Upload your GPG key to Launchpad

To find out 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/people/+me/+editpgpkeys and copy the part about your "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 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/people/+me/+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 (https://help.launchpad.net/YourAccount/CreatingAnSSHKeyPair) 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. 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/people/+me 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 use a shell different from the default (which is bash), please edit the configuration file for that shell accordingly.)

NEXT MONTH: Fixing a bug

A PLEA ON BEHALF OF THE PODCAST PARTY

As you heard in episode #15 of the podcast, we're calling for opinion topics for that section of the show.

Instead of us having a rant about whatever strikes us, why not prompt us with a topic and watch for the mushroom clouds over the horizon! It's highly unlikely that the three of us will agree.

Or, an even more radical thought, send us an opinion by way of a contribution!

You can post comments and opinions on the podcast page at fullcirclemagazine.org, in our Ubuntu Forums section, or email podcast@fullcirclemagazine.org. You can also send us a comment by recording an audio clip of no more than 30 seconds and sending it to the same address. **Comments and audio may be edited for length. Please remember this is a family-friendly show.**

It would be great to have contributors come on the show and express an opinion in person.

Robin

Ubuntu Development Pt. 3 - Bug Fixing

Written by Daniel Holbach

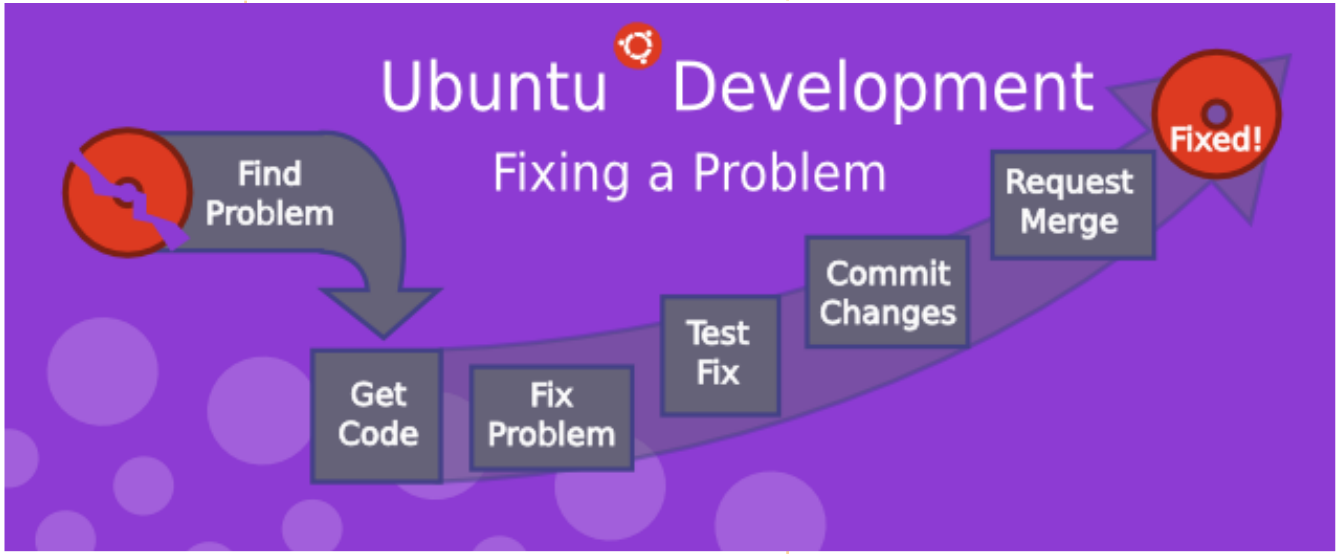
If you followed the instructions to get set up with Ubuntu Development, you should be all set and ready to go.

As you can see in the image shiwn right, there are 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.

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.



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 show tomboy | grep ^Source:
```

In this case, nothing is printed, meaning that tomboy is also the name of the binary package. An example where the source and binary package names differ is python-vigra. While that is the binary package name, the source package is actually libvigraimpex and can be found with this command (and its output):

```
apt-cache show python-vigra | grep ^Source:

Source: libvigraimpex
```

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. 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.

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.

If you find a patch to fix the problem, say, attached to a bug report, running this command in the source directory should apply the patch:

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

Refer to the patch(1) manpage for options and arguments such as --dry-run, -p<num>, etc.

Testing the fix

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

```
bzr bd -- -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:

- where the change was done
- what was changed
- 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-open
```

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, and click it to get the change reviewed by somebody and included in Ubuntu.

Next month: an overview of the Debian directory structure.

Ubuntu Development Pt. 4 - debian/

Written by Daniel Holbach

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 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.

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.

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:

```
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/h
ello/

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.

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://svn.debian.org/wsvn/dep/web/deps/dep5.mdwn?op=file&rev=166
Upstream-Name: Hello
Source:
ftp://ftp.example.com/pub/games

Files: *
Copyright: Copyright 1998
John Doe <jdoe@example.com>
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'.

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

This example follows the DEP-5: Machine-parseable debian/copyright proposal. You are encouraged to use this format as well.

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 work only on binary independent packages, and commands in the binary-arch sequences are passed the “-a” option to ensure they work only 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.

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/hell
o/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
```

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. Currently, the package source format defaults to 1.0 if this file does not exist. You are encouraged to use the newer 3.0 source format. In this case, the file should contain a single line indicating the desired format:

- 3.0 (native) for Debian native packages (no upstream version) or
- 3.0 (quilt) for packages with a separate upstream tarball

If, for some reason, you wish to keep using the old format, please create this file and put 1.0 in it to be explicit about the source package version. This allows for the future removal of the 1.0 default for the package source format.

http://wiki.debian.org/Projects/DebSrc3.0 summarizes information concerning, and the benefits of 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.

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.