About this tutorial

- Goal: **tell you what you really need to know about Debian packaging**
  - Modify existing packages
  - Create your own packages
  - Interact with the Debian community
  - Become a Debian power-user

- Covers the most important points, but is not complete
  - You will need to read more documentation

- Most of the content also applies to Debian derivative distributions
  - That includes Ubuntu
Outline

1. Introduction
2. Creating source packages
3. Building and testing packages
4. Practical session 1: modifying the grep package
5. Advanced packaging topics
6. Maintaining packages in Debian
7. Conclusions
8. Additional practical sessions
9. Answers to practical sessions
Outline

1 Introduction
2 Creating source packages
3 Building and testing packages
4 Practical session 1: modifying the grep package
5 Advanced packaging topics
6 Maintaining packages in Debian
7 Conclusions
8 Additional practical sessions
9 Answers to practical sessions
Debian

➤ GNU/Linux distribution

➤ 1st major distro developed “openly in the spirit of GNU”

➤ Non-commercial, built collaboratively by over 1,000 volunteers

➤ 3 main features:
  ➤ Quality – culture of technical excellence
    *We release when it’s ready*
  
  ➤ Freedom – devs and users bound by the *Social Contract*
    Promoting the culture of Free Software since 1993
  
  ➤ Independence – no (single) company babysitting Debian
    And open decision-making process (*doocracy* + *democracy*)

➤ Amateur in the best sense: done for the love of it
Debian packages

- .deb files (binary packages)
- A very powerful and convenient way to distribute software to users
- One of the two most common package formats (with RPM)

Universal:
- 30,000 binary packages in Debian
  → most of the available free software is packaged in Debian!
- For 12 ports (architectures), including 2 non-Linux (Hurd; KFreeBSD)
- Also used by 120 Debian derivative distributions
The Deb package format

▶ .deb file: an ar archive

```
$ ar tv wget_1.12-2.1_i386.deb
rw-r--r-- 0/0  4 Sep 5 15:43 2010 debian-binary
rw-r--r-- 0/0 2403 Sep 5 15:43 2010 control.tar.gz
rw-r--r-- 0/0 751613 Sep 5 15:43 2010 data.tar.gz
```

▶ debian-binary: version of the deb file format, "2.0\n"
▶ control.tar.gz: metadata about the package
  control, md5sums, (pre|post)(rm|inst), triggers, shlibs,...
▶ data.tar.gz: data files of the package

▶ You could create your .deb files manually

▶ But most people don’t do it that way

This tutorial: create Debian packages, the Debian way
Tools you will need

- A Debian (or Ubuntu) system (with root access)

- Some packages:
  - **build-essential**: has dependencies on the packages that will be assumed to be available on the developer’s machine (no need to specify them in the Build-Depends: control field of your package)
    - includes a dependency on **dpkg-dev**, which contains basic Debian-specific tools to create packages
  - **devscripts**: contains many useful scripts for Debian maintainers

Many other tools will also be mentioned later, such as **debhelper**, **cdfs**, **quilt**, **pbuilder**, **sbuild**, **lintian**, **svn-buildpackage**, **git-buildpackage**, ... Install them when you need them.
General packaging workflow

Debian mirror

apt-get source
dget

Web

upstream source
dh_make

source package

where most of the manual work is done
debuild (build and test with lintian)
or dpkg-buildpackage

one or several binary packages

upload (dput)

install (debi)

one or several binary packages

deb
Example: rebuilding dash

1. Install packages needed to build dash, and devscripts
   sudo apt-get build-dep dash  
   (requires deb-src lines in /etc/apt/sources.list)
   sudo apt-get install --no-install-recommends devscripts fakeroot

2. Create a working directory, and get in it:
   mkdir /tmp/debian-tutorial ; cd /tmp/debian-tutorial

3. Grab the dash source package
   apt-get source dash  
   (This needs you to have deb-src lines in your /etc/apt/sources.list)

4. Build the package
   cd dash-*
   debuild -us -uc  (-us -uc disables signing the package with GPG)

5. Check that it worked
   ▶ There are some new .deb files in the parent directory

6. Look at the debian/ directory
   ▶ That's where the packaging work is done
Outline

1 Introduction

2 Creating source packages

3 Building and testing packages

4 Practical session 1: modifying the grep package

5 Advanced packaging topics

6 Maintaining packages in Debian

7 Conclusions

8 Additional practical sessions

9 Answers to practical sessions
Source package

- One source package can generate several binary packages
e.g. the libtar source generates the libtar0 and libtar-dev binary packages

- Two kinds of packages: (if unsure, use non-native)
  - Native packages: normally for Debian specific software (dpkg, apt)
  - Non-native packages: software developed outside Debian

- Main file: .dsc (meta-data)

- Other files depending on the version of the source format
  - 1.0 or 3.0 (native): package_version.tar.gz
  - 1.0 (non-native):
    - pkg_ver.orig.tar.gz: upstream source
    - pkg_debver.diff.gz: patch to add Debian-specific changes
  - 3.0 (quilt):
    - pkg_ver.orig.tar.gz: upstream source
    - pkg_debver.debian.tar.gz: tarball with the Debian changes

(See dpkg-source(1) for exact details)
Source package example (wget_1.12-2.1.dsc)

Format: 3.0 (quilt)
Source: wget
Binary: wget
Architecture: any
Version: 1.12-2.1
Maintainer: Noel Kothe <noel@debian.org>
Homepage: http://www.gnu.org/software/wget/
Standards-Version: 3.8.4
Build-Depends: debhelper (>> 5.0.0), gettext, texinfo, libssl-dev (>= 0.9.8), dpatch, info2man
Checksums-Sha1:
  50d4ed2441e67 [...]1ee0e94248 2464747 wget_1.12.orig.tar.gz
d4c1c8bbe431d [...]dd7cef3611 48308 wget_1.12-2.1.debian.tar.gz
Checksums-Sha256:
  7578ed0974e12 [...]dcba65b572 2464747 wget_1.12.orig.tar.gz
  1e9b0c4c00eae [...]89c402ad78 48308 wget_1.12-2.1.debian.tar.gz
Files:
  141461b9c04e4 [...]9d1f2abf83 2464747 wget_1.12.orig.tar.gz
e93123c934e3c [...]2f380278c2 48308 wget_1.12-2.1.debian.tar.gz
Retrieving an existing source package

▶ From the Debian archive:
  ▶ apt-get source package
  ▶ apt-get source package=version
  ▶ apt-get source package/release

(You need deb-src lines in sources.list)

▶ From the Internet:
  ▶ dget url-to.dsc
  ▶ dget http://snapshot.debian.org/archive/debian-archive/20090802T004153Z/debian/dists/bo/main/source/web/wget_1.4.4-6.dsc

(snapshot.d.o provides all packages from Debian since 2005)

▶ From the (declared) version control system:
  ▶ debcheckout package

▶ Once downloaded, extract with dpkg-source -x file.dsc
Creating a basic source package

- Download the upstream source
  (*upstream source* = the one from the software’s original developers)
- Rename to `<source_package>_<upstream_version>.orig.tar.gz`
  (example: `simgrid_3.6.orig.tar.gz`)
- Untar it
- Rename the directory to `<source_package>-<upstream_version>`
  (example: `simgrid-3.6`)
- `cd <source_package>-<upstream_version> && dh_make`
  (from the `dh-make` package)
- There are some alternatives to `dh_make` for specific sets of packages:
  `dh-make-perl, dh-make-php, ...`
- `debian/` directory created, with a lot of files in it
Files in debian/

All the packaging work should be made by modifying files in debian/

▸ Main files:
  ▸ **control** – meta-data about the package (dependencies, etc.)
  ▸ **rules** – specifies how to build the package
  ▸ **copyright** – copyright information for the package
  ▸ **changelog** – history of the Debian package

▸ Other files:
  ▸ compat
  ▸ watch
  ▸ dh_install* targets
    *dirs, *docs, *manpages, ...
  ▸ maintainer scripts
    *postinst, *prerm, ...
  ▸ source/format
  ▸ patches/ – if you need to modify the upstream sources

▸ Several files use a format based on RFC 822 (mail headers)
debian/changelog

- Lists the Debian packaging changes
- Gives the current version of the package

1.2.1.1-5

Upstream Debian version revision

- Edited manually or with dch
  - Create a changelog entry for a new release: dch -i
- Special format to automatically close Debian or Ubuntu bugs
  Debian: Closes: #595268; Ubuntu: LP: #616929
- Installed as /usr/share/doc/package/changelog.Debian.gz

mpich2 (1.2.1.1-5) unstable; urgency=low

* Use /usr/bin/python instead of /usr/bin/python2.5. Allow to drop dependency on python2.5. Closes: #595268
* Make /usr/bin/mpdroot setuid. This is the default after the installation of mpich2 from source, too. LP: #616929
  + Add corresponding lintian override.

-- Lucas Nussbaum <lucas@debian.org> Wed, 15 Sep 2010 18:13:44 +0200
debian/control

- Package metadata
  - For the source package itself
  - For each binary package built from this source

- Package name, section, priority, maintainer, uploaders, build-dependencies, dependencies, description, homepage, ...

- Documentation: Debian Policy chapter 5

---

Source: wget
Section: web
Priority: important
Maintainer: Noel Kothe <noel@debian.org>
Build-Depends: debhelper (>> 5.0.0), gettext, texinfo,
  libssl-dev (>= 0.9.8), dpatch, info2man
Standards-Version: 3.8.4
Homepage: http://www.gnu.org/software/wget/

Package: wget
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: retrieves files from the web
  Wget is a network utility to retrieve files from the Web
Architecture: all or any

Two kinds of binary packages:

- Packages with different contents on each Debian architecture
  - Example: C program
  - Architecture: any in debian/control
    - Or, if it only works on a subset of architectures:
      Architecture: amd64 i386 ia64 hurd-i386
      - buildd.debian.org: builds all the other architectures for you on upload
      - Named package_version_architecture.deb

- Packages with the same content on all architectures
  - Example: Perl library
  - Architecture: all in debian/control
  - Named package_version_all.deb

A source package can generate a mix of Architecture: any and Architecture: all binary packages
debian/rules

- Makefile
- Interface used to build Debian packages
- Documented in Debian Policy, chapter 4.8
  https://www.debian.org/doc/debian-policy/ch-source#s-debianrules
- Required targets:
  - build, build-arch, build-indep: should perform all the configuration and compilation
  - binary, binary-arch, binary-indep: build the binary packages
    - dpkg-buildpackage will call binary to build all the packages, or binary-arch to build only the Architecture: any packages
  - clean: clean up the source directory
Packaging helpers – debhelper

- You could write shell code in `debian/rules` directly
- Better practice (used by most packages): use a Packaging helper
- Most popular one: `debhelper` (used by 98% of packages)
- Goals:
  - Factor the common tasks in standard tools used by all packages
  - Fix some packaging bugs once for all packages
  - `dh_installdirs`, `dh_installchangelogs`, `dh_installdocs`, `dh_install`, `dh_installdebconf`, `dh_installinit`, `dh_link`, `dh_strip`, `dh_compress`, `dh_fixperms`, `dh_perl`, `dh_makeshlibs`, `dh_installdeb`, `dh_shlibdeps`, `dh_gencontrol`, `dh_md5sums`, `dh_builddeb`, ...
  - Called from `debian/rules`
  - Configurable using command parameters or files in `debian/package.docs`, `package.examples`, `package.install`, `package.manpages`, ...
- Third-party helpers for sets of packages: `python-support`, `dh_ocaml`, ...
- `debian/compat`: Debhelper compatibility version
  - Defines precise behaviour of `dh_*`
  - New syntax: Build-Depends: `debhelper-compat (= 13)`
#!/usr/bin/make -f

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

build:
  $(MAKE)
  #docbook-to-man debian/packagename.sgml > packagename.1

clean:
  dh_testdir
dh_testroot
rm -f build-stamp configure-stamp
$(MAKE) clean
dh_clean

install: build
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs
  # Add here commands to install the package into debian/packagename
  $(MAKE) DESTDIR=$(CURDIR)/debian/packagename install
debian/rules using debhelper (2/2)

# Build architecture-independent files here.
binary-indep: build install

# Build architecture-dependent files here.
binary-arch: build install
dh_testdir
dh_testroot
dh_installchangelogs
dh_installdocs
dh_installexamples
dh_install
dh_installman
dh_link
dh_strip
dh_compress
dh_fixperms
dh_installldeb
dh_shlibdeps
dh_gencontrol
dh_md5sums
dh_builddeb

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary-arch binary install configure
With debhelper, still a lot of redundancy between packages

Second-level helpers that factor common functionality
  - E.g. building with `./configure && make && make install` or `CMake`

CDBS:
  - Introduced in 2005, based on advanced GNU make magic
  - Documentation: `/usr/share/doc/cdbs/`
  - Support for Perl, Python, Ruby, GNOME, KDE, Java, Haskell, ...
  - But some people hate it:
    - Sometimes difficult to customize package builds:
      "twisty maze of makefiles and environment variables"
    - Slower than plain debhelper (many useless calls to `dh_*`)

```bash
#!/usr/bin/make -f
include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk

# add an action after the build
build/mypackage::
  /bin/bash debian/scripts/foo.sh
```
Dh (aka Debhelper 7, or dh7)

- Introduced in 2008 as a *CDBS killer*
- `dh` command that calls `dh_*`
- Simple *debian/rules*, listing only overrides
- Easier to customize than CDBS
- Doc: manpages (*debhelper*(7), *dh*(1)) + slides from DebConf9 talk
  

```bash
#!/usr/bin/make -f
%
  dh $@

override_dh_auto_configure:
  dh_auto_configure -- --with-kitchen-sink

override_dh_auto_build:
  make world
```
Classic debhelper vs CDBS vs dh

Mind shares:
Classic debhelper: 15%  CDBS: 15%  dh: 68%

Which one should I learn?
- Probably a bit of all of them
- You need to know debhelper to use dh and CDBS
- You might have to modify CDBS packages

Which one should I use for a new package?
- dh (only solution with an increasing mind share)
- See https://trends.debian.net/#build-systems
Outline

1. Introduction
2. Creating source packages
3. Building and testing packages
4. Practical session 1: modifying the grep package
5. Advanced packaging topics
6. Maintaining packages in Debian
7. Conclusions
8. Additional practical sessions
9. Answers to practical sessions
Building packages

- `apt-get build-dep mypackage`
  Installs the *build-dependencies* (for a package already in Debian)
  Or `mk-build-deps -ir` (for a package not uploaded yet)

- `debuild`: build, test with *lintian*, sign with GPG

- Also possible to call `dpkg-buildpackage` directly
  - Usually with `dpkg-buildpackage -us -uc`

- It is better to build packages in a clean & minimal environment
  - `pbuilder` – helper to build packages in a *chroot*
    Good documentation: https://wiki.ubuntu.com/PbuilderHowto
    (optimization: `cowbuilder ccache distcc`)
  - `schroot` and `sbuild`: used on the Debian build daemons
    (not as simple as `pbuilder`, but allows LVM snapshots
    see: https://help.ubuntu.com/community/SbuildLVMHowto)

- Generates `.deb` files and a `.changes` file
  - `.changes`: describes what was built; used to upload the package
Installing and testing packages

- Install the package locally: `debi` (will use `.changes` to know what to install)

- List the content of the package: `debc ..mypackage<TAB>.changes`

- Compare the package with a previous version:
  - `debdiff ..mypackage_1_*.changes ..mypackage_2_*.changes`
  - or to compare the sources:
  - `debdiff ..mypackage_1_*.dsc ..mypackage_2_*.dsc`

- Check the package with `lintian` (static analyzer):
  - `lintian ..mypackage<TAB>.changes`
  - `lintian -i`: gives more information about the errors
  - `lintian -Evil --pedantic`: shows more problems

- Upload the package to Debian (`dput`) (needs configuration)

- Manage a private Debian archive with `reprepro` or `aptly`
  
  Documentation: https://wiki.debian.org/HowToSetupADebianRepository
Outline

1 Introduction
2 Creating source packages
3 Building and testing packages
4 Practical session 1: modifying the grep package
5 Advanced packaging topics
6 Maintaining packages in Debian
7 Conclusions
8 Additional practical sessions
9 Answers to practical sessions
Practical session 1: modifying the grep package

   - If the source package is not unpacked automatically, unpack it with dpkg-source -x grep_* .dsc

2. Look at the files in debian/.
   - How many binary packages are generated by this source package?
   - Which packaging helper does this package use?

3. Build the package

4. We are now going to modify the package. Add a changelog entry and increase the version number.

5. Now disable perl-regexp support (it is a ./configure option)

6. Rebuild the package

7. Compare the original and the new package with debdiff

8. Install the newly built package
Outline

1 Introduction

2 Creating source packages

3 Building and testing packages

4 Practical session 1: modifying the grep package

5 Advanced packaging topics

6 Maintaining packages in Debian

7 Conclusions

8 Additional practical sessions

9 Answers to practical sessions
Copyright and license information for the source and the packaging
Traditionally written as a text file
New machine-readable format:

https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/

Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: X Solitaire

Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+
This program is free software; you can redistribute it
[...]

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 Smith <jsmith@example.net>
License: [LICENSE TEXT]
Modifying the upstream source

Often needed:
- Fix bugs or add customizations that are specific to Debian
- Backport fixes from a newer upstream release

Several methods to do it:
- Modifying the files directly
  - Simple
  - But no way to track and document the changes
- Using patch systems
  - Eases contributing your changes to upstream
  - Helps sharing the fixes with derivatives
  - Gives more exposure to the changes
  
  http://patch-tracker.debian.org/ (down currently)
Patch systems

- Principle: changes are stored as patches in `debian/patches/`
- Applied and unapplied during build
- Past: several implementations – *simple-patchsys (cdfs)*, *dpatch*, *quilt*
  - Each supports two `debian/rules` targets:
    - `debian/rules patch`: apply all patches
    - `debian/rules unpatch`: de-apply all patches
  - More documentation: [https://wiki.debian.org/debian/patches](https://wiki.debian.org/debian/patches)

- New source package format with built-in patch system: 3.0 (*quilt*)
  - Recommended solution
  - You need to learn *quilt*
    - [https://perl-team.pages.debian.net/howto/quilt.html](https://perl-team.pages.debian.net/howto/quilt.html)
  - Patch-system-agnostic tool in `devscripts`: `edit-patch`
Description: Fix widget frobnication speeds
Frobnicating widgets too quickly tended to cause explosions.
Forwarded: http://lists.example.com/2010/03/1234.html
Author: John Doe <johndoe-guest@users.alioth.debian.org>
Last-Update: 2010-03-29

--- a/src/widgets.c
+++ b/src/widgets.c
@@ -101,9 +101,6 @@ struct {

Doing things during installation and removal

- Decompressing the package is sometimes not enough
- Create/remove system users, start/stop services, manage *alternatives*
- Done in *maintainer scripts*
  - preinst, postinst, prerm, postrm
  - Snippets for common actions can be generated by debhelper
- Documentation:
  - Debian Policy Manual, chapter 6
    https://www.debian.org/doc/debian-policy/ch-maintainerscripts
  - Debian Developer’s Reference, chapter 6.4
  - https://people.debian.org/~srivasta/MaintainerScripts.html
- Prompting the user
  - Must be done with *debconf*
  - Documentation: debconf-devel(7) (debconf-doc package)
Monitoring upstream versions

▶ Specify where to look in debian/watch (see uscan(1))

version=3

http://tmrc.mit.edu/mirror/twisted/Twisted/([^d\./d]+)/
  Twisted-([^d\./]*)\..tar\.bz2

▶ There are automated trackers of new upstream versions, that notify the maintainer on various dashboards including
https://tracker.debian.org/ and https://udd.debian.org/dmd/

▶ uscan: run a manual check

▶ uupdate: try to update your package to the latest upstream version
Packaging with a Version Control System

- Several tools to help manage branches and tags for your packaging work:
  - `svn-buildpackage`, `git-buildpackage`

- **Example**: `git-buildpackage`
  - upstream branch to track upstream with `upstream/version` tags
  - master branch tracks the Debian package
  - `debian/version` tags for each upload
  - pristine-tar branch to be able to rebuild the upstream tarball


- **Vcs-* fields in debian/control to locate the repository**
  - [https://wiki.debian.org/Salsa](https://wiki.debian.org/Salsa)

Vcs-Browser: [https://salsa.debian.org/debian/devscripts](https://salsa.debian.org/debian/devscripts)
Vcs-Git: [https://salsa.debian.org/debian/devscripts.git](https://salsa.debian.org/debian/devscripts.git)

Vcs-Browser: [https://salsa.debian.org/perl-team/modules/packages/libwww-perl](https://salsa.debian.org/perl-team/modules/packages/libwww-perl)
Vcs-Git: [https://salsa.debian.org/perl-team/modules/packages/libwww-perl.git](https://salsa.debian.org/perl-team/modules/packages/libwww-perl.git)

- **VCS-agnostic interface**: `debcheckout`, `debcommit`, `debrelease`
  - `debcheckout grep` → checks out the source package from Git
Backporting packages

- Goal: use a newer version of a package on an older system
e.g. use *mutt* from Debian *unstable* on Debian *stable*

- General idea:
  - Take the source package from Debian unstable
  - Modify it so that it builds and works fine on Debian stable
    - Sometimes trivial (no changes needed)
    - Sometimes difficult
    - Sometimes impossible (many unavailable dependencies)

- Some backports are provided and supported by the Debian project
  http://backports.debian.org/
Outline

1 Introduction
2 Creating source packages
3 Building and testing packages
4 Practical session 1: modifying the grep package
5 Advanced packaging topics
6 Maintaining packages in Debian
7 Conclusions
8 Additional practical sessions
9 Answers to practical sessions
Debian archive and suites

- Debian archive and suites
  - security
  - stable-updates
  - stable
  - experimental
  - backports
  - developer
  - unstable
  - testing
  - testing-proposed-updates
  - stable-proposed-updates
  - stable-new
  - oldstable
  - archive.d.o
  - preparation of next release
  - release management

Based on graph by Antoine Beaupré. https://salsa.debian.org/debian/package-cycle
Suites for development

- New versions of packages are uploaded to **unstable** (sid)
- Packages migrate from **unstable** to **testing** based on several criterias (e.g. has been in unstable for 10 days, and no regressions)
- New packages can also be uploaded to:
  - **experimental** (for more experimental packages, such as when the new version is not ready to replace the one currently in unstable)
  - **testing-proposed-updates**, to update the version in **testing** without going through **unstable** (this is rarely used)
Freezing and releasing

- At some point during the release cycle, the release team decides to freeze testing: automatic migrations from unstable to testing are stopped, and replaced by manual review.

- When the release team considers testing to be ready for release:
  - The testing suite becomes the new stable suite.
  - Similarly, the old stable becomes oldstable.
  - Unsupported releases are moved to archive.debian.org.

- See https://release.debian.org/
Stable release suites and management

- Several suites are used to provide stable release packages:
  - **stable**: the main suite
  - **security** updates suite provided on security.debian.org, used by the security team. Updates are announced on the debian-security-announce mailing list
  - **stable-updates**: updates that are not security related, but that should urgently be installed (without waiting for the next point release): antivirus databases, timezone-related packages, etc. Announced on the debian-stable-announce mailing list
  - **backports**: new upstream versions, based on the version in testing

- The **stable** suite is updated every few months by *stable point releases* (that include only bug fixes)
  - Packages targetting the next stable point release are uploaded to **stable-proposed-updates** and reviewed by the release team

- The **oldstable** release has the same set of suites
Several ways to contribute to Debian

- **Worst** way to contribute:
  1. Package your own application
  2. Get it into Debian
  3. Disappear

- **Better** ways to contribute:
  - Get involved in packaging teams
    - Many teams that focus on set of packages, and need help
    - List available at [https://wiki.debian.org/Teams](https://wiki.debian.org/Teams)
    - An excellent way to learn from more experienced contributors
  - Adopt existing unmaintained packages (*orphaned packages*)
  - Bring new software to Debian
    - Only if it’s interesting/useful enough, please
    - Are there alternatives already packaged in Debian?
Adopting orphaned packages

- Many unmaintained packages in Debian
- Full list + process: [https://www.debian.org/devel/wnpp/](https://www.debian.org/devel/wnpp/)
- Installed on your machine: wnpp-alert
  Or better: how-can-i-help

- Different states:
  - **Orphaned**: the package is unmaintained
    Feel free to adopt it
  - **RFA**: Request For Adopter
    Maintainer looking for adopter, but continues work in the meantime
    Feel free to adopt it. A mail to the current maintainer is polite
  - **ITA**: Intent To Adopt
    Someone intends to adopt the package
    You could propose your help!
  - **RFH**: Request For Help
    The maintainer is looking for help

- Some unmaintained packages not detected → not orphaned yet
- When in doubt, ask debian-qa@lists.debian.org
Adopting a package: example

From: You <you@yourdomain>
To: 640454@bugs.debian.org, control@bugs.debian.org
Cc: Francois Marier <francois@debian.org>
Subject: ITA: verbiste -- French conjugator

retitle 640454 ITA: verbiste -- French conjugator
owner 640454 !
thanks

Hi,

I am using verbiste and I am willing to take care of the package.

Cheers,

You

- Polite to contact the previous maintainer (especially if the package was RFAed, not orphaned)
- Very good idea to contact the upstream project
Getting your package in Debian

- You do not need any official status to get your package into Debian
  1. Submit an ITP bug (Intent To Package) using reportbug wnpp
  2. Prepare a source package
  3. Find a Debian Developer that will sponsor your package

- Official status (when you are an experienced package maintainer):
  - **Debian Maintainer (DM):**
    Permission to upload your own packages
    See https://wiki.debian.org/DebianMaintainer
  - **Debian Developer (DD):**
    Debian project member; can vote and upload any package
Things to check before asking for sponsorship

- Debian puts **a lot of focus on quality**
- Generally, **sponsors are hard to find and busy**
  - Make sure your package is ready before asking for sponsorship
- Things to check:
  - Avoid missing build-dependencies: make sure that your package builds fine in a clean *sid chroot*
    - Using `pbuilder` is recommended
  - Run `lintian -EvIIL +pedantic` on your package
    - Errors must be fixed, all other problems should be fixed
  - Do extensive testing of your package, of course
- In doubt, ask for help
Where to find help?

Help you will need:

▶ Advice and answers to your questions, code reviews
▶ Sponsorship for your uploads, once your package is ready

You can get help from:

▶ **Other members of a packaging team**
  ▶ List of teams: https://wiki.debian.org/Teams

▶ The **Debian Mentors group** (if your package does not fit in a team)
  ▶ https://wiki.debian.org/DebianMentorsFaq
  ▶ Mailing list: debian-mentors@lists.debian.org
    (also a good way to learn by accident)
  ▶ IRC: #debian-mentors on irc.debian.org
  ▶ http://mentors.debian.net/
  ▶ Documentation: http://mentors.debian.net/intro-maintainers

▶ **Localized mailing lists** (get help in your language)
  ▶ debian-devel-{french,italian,portuguese,spanish}@lists.d.o
  ▶ Full list: https://lists.debian.org/devel.html
  ▶ Or users lists: https://lists.debian.org/users.html
More documentation

- Debian Developers’ Corner
  https://www.debian.org/devel/
  Links to many resources about Debian development

- Guide for Debian Maintainers
  https://www.debian.org/doc/manuals/debmake-doc/

- Debian Developer’s Reference
  https://www.debian.org/doc/developers-reference/
  Mostly about Debian procedures, but also some best packaging practices (part 6)

- Debian Policy
  https://www.debian.org/doc/debian-policy/
  - All the requirements that every package must satisfy
  - Specific policies for Perl, Java, Python, ...

- Ubuntu Packaging Guide
  https://packaging.ubuntu.com/html/
Debian dashboards for maintainers

- **Source package centric:**
  https://tracker.debian.org/dpkg

- **Maintainer/team centric:** Developer’s Packages Overview (DDPO)
  pkg-ruby-extras-maintainers@lists.alioth.debian.org

- **TODO-list oriented:** Debian Maintainer Dashboard (DMD)
  https://udd.debian.org/dmd/
Using the Debian Bug Tracking System (BTS)

- A quite unique way to manage bugs
  - Web interface to view bugs
  - Email interface to make changes to bugs

- Adding information to bugs:
  - Write to 123456@bugs.debian.org (does not include the submitter, you need to add 123456-submitter@bugs.debian.org)

- Changing bug status:
  - Send commands to control@bugs.debian.org
  - Command-line interface: bts command in devscripts
  - Documentation: https://www.debian.org/Bugs/server-control

- Reporting bugs: use reportbug
  - Normally used with a local mail server: install ssmtp or nullmailer
  - Or use reportbug --template, then send (manually) to submit@bugs.debian.org
Using the BTS: examples

- Sending an email to the bug and the submitter: 
  https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680822#10

- Tagging and changing the severity: 
  https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680227#10

- Reassigning, changing the severity, retitling ...: 
  https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680822#93
  - notfound, found, notfixed, fixed are for **version-tracking**
    See https://wiki.debian.org/HowtoUseBTS#Version_tracking

- Using usertags: https://bugs.debian.org/cgi-bin/bugreport.cgi?msg=42;bug=642267
  See https://wiki.debian.org/bugs.debian.org/usertags

- BTS Documentation:
  - https://www.debian.org/Bugs/
  - https://wiki.debian.org/HowtoUseBTS
More interested in Ubuntu?

- Ubuntu mainly manages the divergence with Debian
- No real focus on specific packages
  Instead, collaboration with Debian teams
- Usually recommend uploading new packages to Debian first
- Possibly a better plan:
  - Get involved in a Debian team and act as a bridge with Ubuntu
  - Help reduce divergence, triage bugs in Launchpad
  - Many Debian tools can help:
    - Ubuntu column on the Developer’s packages overview
    - Ubuntu box on the Package Tracking System
    - Receive launchpad bugmail via the PTS
Outline

1 Introduction

2 Creating source packages

3 Building and testing packages

4 Practical session 1: modifying the grep package

5 Advanced packaging topics

6 Maintaining packages in Debian

7 Conclusions

8 Additional practical sessions

9 Answers to practical sessions
Conclusions

▶ You now have a full overview of Debian packaging
▶ But you will need to read more documentation
▶ Best practices have evolved over the years
  ▶ If not sure, use the dh packaging helper, and the 3.0 (quilt) format

Feedback: packaging-tutorial@packages.debian.org
Copyright ©2011–2019 Lucas Nussbaum – lucas@debian.org

This document is free software: you can redistribute it and/or modify it under either (at your option):

▶ The terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. http://www.gnu.org/licenses/gpl.html

▶ The terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License. http://creativecommons.org/licenses/by-sa/3.0/
Contribute to this tutorial

► Contribute:
  ▶ apt-get source packaging-tutorial
  ▶ debcheckout packaging-tutorial
  ▶ git clone
    https://salsa.debian.org/debian/packaging-tutorial.git
  ▶ https://salsa.debian.org/debian/packaging-tutorial
  ▶ Open bugs: bugs.debian.org/src:packaging-tutorial

► Provide feedback:
  ▶ mailto:packaging-tutorial@packages.debian.org
    ▶ What should be added to this tutorial?
    ▶ What should be improved?
  ▶ reportbug packaging-tutorial
Outline

1 Introduction
2 Creating source packages
3 Building and testing packages
4 Practical session 1: modifying the grep package
5 Advanced packaging topics
6 Maintaining packages in Debian
7 Conclusions
8 Additional practical sessions
9 Answers to practical sessions
Practical session 2: packaging GNUjump

1. Download GNUjump 1.0.8 from
   http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz

2. Create a Debian package for it
   ▶ Install build-dependencies so that you can build the package
   ▶ Fix bugs
   ▶ Get a basic working package
   ▶ Finish filling debian/control and other files

3. Enjoy
Practical session 2: packaging GNUjump (tips)

- To get a basic working package, use dh_make
- To start with, creating a 1.0 source package is easier than 3.0 (quilt) (change that in debian/source/format)
- To search for missing build-dependencies, find a missing file, and use apt-file to find the missing package
- If you encounter that error:

```
/usr/bin/ld: SDL_rotozoom.o: undefined reference to symbol 'ceil@@GLIBC_2.2.5'
//lib/x86_64-linux-gnu/libm.so.6: error adding symbols: DSO missing from command line
collect2: error: ld returned 1 exit status
Makefile:376: recipe for target 'gnujump' failed
```

You need to add -lm to the linker command line:
Edit src/Makefile.am and replace

```
gnujump_LDFLAGS = $(all_libraries)
```

by

```
gnujump_LDFLAGS = -Wl,--as-needed
gnujump_LDADD = $(all_libraries) -lm
```

Then run autoreconf -i
Practical session 3: packaging a Java library

1. Take a quick look at some documentation about Java packaging:
   - https://wiki.debian.org/Java
   - https://wiki.debian.org/Java/Packaging
   - /usr/share/doc/javahelper/tutorial.txt.gz

2. Download IRClib from http://moepii.sourceforge.net/

3. Package it
Practical session 4: packaging a Ruby gem

1. Take a quick look at some documentation about Ruby packaging:
   - https://wiki.debian.org/Ruby
   - https://wiki.debian.org/Teams/Ruby
   - https://wiki.debian.org/Teams/Ruby/Packaging
   - `gem2deb(1), dh_ruby(1)` (in the `gem2deb` package)

2. Create a basic Debian source package from the `peach` gem:
   ```
   gem2deb peach
   ```

3. Improve it so that it becomes a proper Debian package
Practical session 5: packaging a Perl module

1. Take a quick look at some documentation about Perl packaging:
   - https://perl-team.pages.debian.net
   - https://wiki.debian.org/Teams/DebianPerlGroup
   - `dh-make-perl(1), dpt(1)` (in the `pkg-perl-tools` package)

2. Create a basic Debian source package from the `Acme` CPAN distribution:
   - `dh-make-perl --cpan Acme`

3. Improve it so that it becomes a proper Debian package
Outline

1 Introduction

2 Creating source packages

3 Building and testing packages

4 Practical session 1: modifying the grep package

5 Advanced packaging topics

6 Maintaining packages in Debian

7 Conclusions

8 Additional practical sessions

9 Answers to practical sessions
Answers to practical sessions
Practical session 1: modifying the grep package

2. Look at the files in debian/.
   - How many binary packages are generated by this source package?
   - Which packaging helper does this package use?
3. Build the package.
4. We are now going to modify the package. Add a changelog entry and increase the version number.
5. Now disable perl-regexp support (it is a ./configure option).
6. Rebuild the package.
7. Compare the original and the new package with debdiff.
8. Install the newly built package.
Fetching the source


- Use dget to download the .dsc file:
  dget http://cdn.debian.net/debian/pool/main/g/grep/grep_2.12-2.dsc

- If you have deb-src for a Debian release that has grep version 2.12-2 (find out on https://tracker.debian.org/grep), you can use: apt-get source grep=2.12-2
  or apt-get source grep/release (e.g. grep/stable)
  or, if you feel lucky: apt-get source grep

- The grep source package is composed of three files:
  - grep_2.12-2.dsc
  - grep_2.12-2.debian.tar.bz2
  - grep_2.12.orig.tar.bz2

  This is typical of the "3.0 (quilt)" format.

- If needed, uncompress the source with dpkg-source -x grep_2.12-2.dsc
Looking around and building the package

2 Look at the files in `debian/`
   ▶ How many binary packages are generated by this source package?
   ▶ Which packaging helper does this package use?

   ▶ According to `debian/control`, this package only generates one binary package, named `grep`.

   ▶ According to `debian/rules`, this package is typical of `classic` debhelper packaging, without using `CDBS` or `dh`. One can see the various calls to `dh_`*` commands in `debian/rules`.

3 Build the package

   ▶ Use `apt-get build-dep grep` to fetch the build-dependencies
   ▶ Then `debuild` or `dpkg-buildpackage -us -uc` (Takes about 1 min)
We are now going to modify the package. Add a changelog entry and increase the version number.

- *debian/changelog* is a text file. You could edit it and add a new entry manually.
- Or you can use `dch -i`, which will add an entry and open the editor.
- The name and email can be defined using the `DEBFULLNAME` and `DEBEMAIL` environment variables.
- After that, rebuild the package: a new version of the package is built.
Disabling Perl regexp support and rebuilding

5 Now disable perl-regexp support (it is a ./configure option)
6 Rebuild the package

- Check with ./configure --help: the option to disable Perl regexp is --disable-perl-regexp
- Edit debian/rules and find the ./configure line
- Add --disable-perl-regexp
- Rebuild with debuild or dpkg-buildpackage -us -uc
Comparing and testing the packages

7. Compare the original and the new package with debdiff
8. Install the newly built package

▶ Compare the binary packages: debdiff ../*changes
▶ Compare the source packages: debdiff ../*dsc
▶ Install the newly built package: debi
  Or dpkg -i ../grep_<TAB>
▶ grep -P foo no longer works!

Reinstall the previous version of the package:
▶ apt-get install --reinstall grep=2.6.3-3 (= previous version)
Practical session 2: packaging GNUjump

1. Download GNUjump 1.0.8 from
   http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz

2. Create a Debian package for it
   - Install build-dependencies so that you can build the package
   - Get a basic working package
   - Finish filling `debian/control` and other files

3. Enjoy
Step by step…

- wget http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz
- mv gnujump-1.0.8.tar.gz gnujump_1.0.8.orig.tar.gz
- tar xf gnujump_1.0.8.orig.tar.gz
- cd gnujump-1.0.8/
- dh_make -f ../gnujump-1.0.8.tar.gz

Type of package: single binary (for now)

gnujump-1.0.8$ ls debian/
changelog
gnujump.default.ex
compat
gnujump.doc-base.EX
control
init.d.ex
README.Debian
copyright
manpage.1.ex
README.source
docs
manpage.sgml.ex
rules
deps
emacsen-install.ex
manpage.xml.ex
source
deps
emacsen-remove.ex
menu.ex
watch.ex
deps
emacsen-startup.ex
postinst.ex
emacsen-cron.d.ex
postrm.ex
Step by step… (2)

- Look at debian/changelog, debian/rules, debian/control (auto-filled by dh_make)

- In debian/control:
  Build-Depends: debhelper (>= 7.0.50 ), autotools-dev
  Lists the *build-dependencies* = packages needed to build the package

- Try to build the package as-is with *debuild* (thanks to *dh* magic)
  - And add build-dependencies, until it builds
  - Hint: use *apt-cache search* and *apt-file* to find the packages
  - Example:

    ```
    checking for sdl-config... no
    checking for SDL - version >= 1.2.0... no
    [...] configure: error: *** SDL version 1.2.0 not found!
    ```

    → Add *libsdl1.2-dev* to Build-Depends and install it.

- Better: use *pbuilder* to build in a clean environment
Step by step...(3)

- Required build-dependencies are `libsdl1.2-dev`, `libsdl-image1.2-dev`, `libsdl-mixer1.2-dev`

- Then, you will probably run into another error:

  ```
  /usr/bin/ld: SDL_rotozoom.o: undefined reference to symbol 'ceil@@GLIBC_2.2.5'
  /lib/x86_64-linux-gnu/libm.so.6: error adding symbols: DSO missing from command line
  collect2: error: ld returned 1 exit status
  Makefile:376: recipe for target 'gnujump' failed
  ```

- This problem is caused by bitrot: `gnujump` has not been adjusted following linker changes.

- If you are using source format version 1.0, you can directly change upstream sources.

  - Edit `src/Makefile.am` and replace

      ```
      gnujump_LDFLAGS = $(all_libraries)
      ```

    by

      ```
      gnujump_LDFLAGS = -Wl,--as-needed
      gnujump_LDADD = $(all_libraries) -lm
      ```

  - Then run `autoreconf -i`
If you are using source format version 3.0 (quilt), use quilt to prepare a patch. (see https://wiki.debian.org/UsingQuilt)

- export QUILT_PATCHES=debian/patches
- mkdir debian/patches
- quilt new linker-fixes.patch
- quilt add src/Makefile.am

Edit src/Makefile.am and replace

gnujump_LDFLAGS = $(all_libraries)

by

gnujump_LDFLAGS = -Wl,--as-needed
gnujump_LDADD = $(all_libraries) -lm

- quilt refresh

Since src/Makefile.am was changed, autoreconf must be called during the build. To do that automatically with dh, change the dh call in debian/rules from: dh $ --with autotools-dev to: dh $ --with autotools-dev --with autoreconf
Step by step...(5)

- The package should now build fine.
- Use `debc` to list the content of the generated package, and `debi` to install it and test it.
- Test the package with `lintian`
  - While not a strict requirement, it is recommended that packages uploaded to Debian are `lintian-clean`
  - More problems can be listed using `lintian --Evil +pedantic`
- Some hints:
  - Remove the files that you don’t need in `debian/`
  - Fill in `debian/control`
  - Install the executable to `/usr/games` by overriding `dh_auto_configure`
  - Use `hardening` compiler flags to increase security.
    See [https://wiki.debian.org/Hardening](https://wiki.debian.org/Hardening)
Step by step… (6)

- Compare your package with the one already packaged in Debian:
  - It splits the data files to a second package, that is the same across all architectures (→ saves space in the Debian archive)
  - It installs a .desktop file (for the GNOME/KDE menus) and also integrates into the Debian menu
  - It fixes a few minor problems using patches
Practical session 3: packaging a Java library

1. Take a quick look at some documentation about Java packaging:
   - https://wiki.debian.org/Java
   - https://wiki.debian.org/Java/Packaging
   - /usr/share/doc/javahelper/tutorial.txt.gz

2. Download IRClib from http://moepii.sourceforge.net/

3. Package it
Step by step...

- `apt-get install javahelper`
- Create a basic source package: `jh_makepkg`
  - Library
  - None
  - Default Free compiler/runtime
- Look at and fix `debian/*`
- `dpkg-buildpackage -us -uc` or `debuild`
- `lintian`, `debc`, etc.
- Compare your result with the `libirclib-java` source package
Practical session 4: packaging a Ruby gem

1. Take a quick look at some documentation about Ruby packaging:
   - [https://wiki.debian.org/Ruby](https://wiki.debian.org/Ruby)
   - [https://wiki.debian.org/Teams/Ruby](https://wiki.debian.org/Teams/Ruby)
   - [https://wiki.debian.org/Teams/Ruby/Packaging](https://wiki.debian.org/Teams/Ruby/Packaging)
   - `gem2deb(1), dh_ruby(1)` (in the `gem2deb` package)

2. Create a basic Debian source package from the `peach` gem:
   ```bash
   gem2deb peach
   ```

3. Improve it so that it becomes a proper Debian package
Step by step...

gem2deb peach:
- Downloads the gem from rubygems.org
- Creates a suitable .orig.tar.gz archive, and untar it
- Initializes a Debian source package based on the gem’s metadata
  - Named ruby-gemname
- Tries to build the Debian binary package (this might fail)

dh_ruby (included in gem2deb) does the Ruby-specific tasks:
- Build C extensions for each Ruby version
- Copy files to their destination directory
- Update shebangs in executable scripts
- Run tests defined in debian/ruby-tests.rb, debian/ruby-tests.rake, or debian/ruby-test-files.yaml, as well as various other checks
Step by step…(2)

Improve the generated package:

- Run `debclean` to clean the source tree. Look at `debian/`.
- `changelog` and `compat` should be correct
- Edit `debian/control`: improve `Description`
- Write a proper `copyright` file based on the upstream files
- Build the package
- Compare your package with the `ruby-peach` package in the Debian archive
Practical session 5: packaging a Perl module

1. Take a quick look at some documentation about Perl packaging:
   - https://perl-team.pages.debian.net
   - https://wiki.debian.org/Teams/DebianPerlGroup
   - dh-make-perl(1), dpt(1) (in the pkg-perl-tools package)

2. Create a basic Debian source package from the Acme CPAN distribution:
   dh-make-perl --cpan Acme

3. Improve it so that it becomes a proper Debian package
dh-make-perl --cpan Acme:

▶ Downloads the tarball from the CPAN
▶ Creates a suitable .orig.tar.gz archive, and untars it
▶ Initializes a Debian source package based on the distribution’s metadata
  ▶ Named libdistname-perl
Improve the generated package:

- `debian/changelog`, `debian/compat`, `debian/libacme-perl.docs`, and `debian/watch` should be correct.

- Edit `debian/control`: improve Description, and remove boilerplate at the bottom.

- Edit `debian/copyright`: remove boilerplate paragraph at the top, add years of copyright to the `Files: *` stanza.