Building Debian Packages

Package your software

the quick and easy way

with CDBS

The Contestants

POSIX Makefile

GNU Autotools

Perl
MakeMaker

Python
Distutils

Ruby
setup.rb
extconf.rb

Qt
Qmake

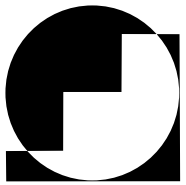
KDE 3
GNU Autotools

Haskell
HBuild

GNOME
GNU Autotools

Java
Apache Ant

PHP
PEAR
PECL



GNUstep
Makefile

OCaml
Makefile

Why Bother?

- Integration
  - Paths
- Maintenance
  - Upgrading
  - Clean removal
  - Multiple installations
- Security
  - Package signatures
  - `debsums`

What is a Debian Package?

- Not just a bunch of files to unpack
  - Meta-data
  - Tools & mechanisms
- Source packages
  - `.orig.tar.gz` *Unmodified upstream source*
  - `.diff.gz` *Debian-specific changes*
  - `.dsc` *Debian source control info*
- Binary packages
  - `.deb` *Debian Package*

Package-Building Tools

- **build-essential**
  - **make** *GNU Make*
  - **dpkg-dev** *Debian Package Building Tools*
  - **gcc & g++** *C & C++ compilers*
  - **libc6-dev** *C Standard Library headers*
- **cdbs** *Common Debian Build System*
- **debhelper** *Automates usual packaging tasks*
- **dh_make** *Debianizes a source directory*
- **devscripts** *Convenience scripts for packagers*

Laying the Foundation

- Get your source, put it in a directory named

  sourcename-*version*/

- Set environment variables for name & email

```
export DEBFULLNAME=”Deborah Netch”
export DEBEMAIL=”debbienetch@hacker.net”
```

- Debianize!

```
dh_make --cdbs --createorig
```

Package Data in debian/

- `changelog` *History of the package*
- `control` *Meta-data about packages*
- `copyright` *License information*
- `rules` *Makefile to build binary packages*
- `watch` *Location to check for updates*
- Maintainer scripts
  - `preinst`, `postinst`, `prerm`, `postrm`, and so on...

Debhelper Files in debian/

- `cron.d` — *A job to register with* `cron`
- `dirs` — *List of directories to create*
- `docs` — *List of documentation files*
- `emacsen-*` — *EMACS site-lisp files*
- *packagename*`-default` — *Variables for* `init` *script*
- *packagename*`.doc-base` — *Online help page*
- `init.d` — `init` *script*
- `manpage.*` — *Manual page*
- `menu` — *Entry for the Debian menu*
- `README.Debian` — *Debian-specific ReadMe*

Describing the Source

- Source: *Name of the source package*
- Maintainer: *Your name <email address>*
- Build-Depends: *Packages needed for building*
  - `devscripts` includes `dpkg-genbuilddeps`

```
fakeroot dpkg-genbuilddeps
```

Describing the Packages

- Package: *Name of a package*
- Architecture: *Compatible platforms*
- Description: *Brief synopsis (< 80 characters)*

  An extended description of the package which gives the user an impression of the package's purpose. It should describe the package's capabilities, intended use, and relation to other parts of the system.

  .

  The description should be word-wrapped to 80 characters and every line must begin with a space. A line with only a space then a period will be shown as a blank line.

Declaring Dependencies

- Depends: *Absolutely needed to run me*
  - `devscripts` includes `dpkg-depcheck`

- Recommends: *Usually needed for me to function*
- Suggests: *Would enhance my functionality*
- Pre-Depends: *Needed during my installation*
- Conflicts: *Cannot be installed alongside me*
- Replaces: *Remove this when I am installed*
- Provides: *I can perform this capability*

Keeping Up-to-Date

- Track your changes with `changelog`
  - Add information about the current version:

```
debchange --append
```

  - Make changes to your package:

```
debchange --increment
```

- Upgrade to a new version of the source:

```
fakeroot uupdate ../sourcename-newversion/
```

Building the Packages

- `rules` Makefile used to create the packages
  - Build the source:

```
debian/rules build
```

  - Make the binary packages:

```
fakeroot debian/rules binary
```

  - Clean up:

```
fakeroot debian/rules clean
```

  - All-in-one

```
dpkg-buildpackage -rfakeroot
```

Tweaking the Result

- Root directory for each binary package is **debian/*packagename*/**
  - `DEB_DESTDIR` variable in `rules`
- What if the build system doesn't install files in the right place?
  - `binary-post-install/`*packagename* runs after build system's install step (ie. “`make install`”)

```
binary-post-install/packagename::
    mv wrong_place/file $(DEB_DESTDIR)/right_place/file
    rm $(DEB_DESTDIR)/this/does/not/belong/here
```

rules for Makefile Packages

- Default Makefile rules

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/makefile.mk
```

- Define targets for make:

```
DEB_MAKE_BUILD_TARGET   = all
DEB_MAKE_CLEAN_TARGET   = clean

DEB_MAKE_CHECK_TARGET   = check
DEB_MAKE_INSTALL_TARGET = install DESTDIR=$(DEB_DESTDIR)
```

- If needed, set environment variables

```
DEB_MAKE_ENVVARS        = CFLAGS="-fpic"
```

`control` for Autotools Packages

Build-Dependencies

- `autotools-dev` *Support files for Autotools*

and possibly...

- `autoconf` *`configure` script generator*
- `automake` *Makefile generator*
- `libtool` *Library support script*

rules for Autotools Packages

- Many projects can use the default rules:

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

- To specify flags for the configure script:

```
DEB_CONFIGURE_EXTRA_FLAGS = --enable-foo --with-bar
```

- In rare cases it may be necessary to update autoconf, automake, and/or libtool

```
DEB_AUTO_UPDATE_AUTOCONF
DEB_AUTO_UPDATE_AUTOMAKE
DEB_AUTO_UPDATE_LIBTOOL
```

control for Perl Module Packages

- Architecture: all
- Section: perl
- Package: lib*something*-perl
  - Foo::Bar ⇨ libfoo-bar-perl

Build-Dependencies

- perl *Interpreter & core modules*

Dependencies

- ${perl:Depends} *Automatic Perl dependencies*

rules for Perl Module Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with Perl Module rules

```
include /usr/share/cdbs/1/class/perlmodule.mk
```

- To specify flags for Makefile.PL script:

```
DEB_MAKEMAKER_USER_FLAGS = --with-something
```

control for Python Packages

- Architecture: all
- Section: python
- Package: python-*something*
  - Foo ⇨ python-foo
  - PyBar ⇨ python-pybar

Build-Dependencies

- python-all-dev *Build files for all Python versions*
- python-central *Compiles modules automatically*

Continued...

control for Python Packages

- For the source package, add field:

XS-Python-Version: all

  - “current“ or specific version(s) can also be used

- For each Python binary package, add field:

XB-Python-Version: ${python:Versions}

Dependencies

- Depends: ${python:Depends}
- Provides: ${python:Provides}

rules for Python Packages

- Before include lines, set DEB_PYTHON_SYSTEM

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

DEB_PYTHON_SYSTEM = pycentral
```

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with Python rules

```
include /usr/share/cdbs/1/class/python-distutils.mk
```

- If build script isn't setup.py, specify it

```
DEB_PYTHON_SETUP_CMD = install.py
```

Continued...

`rules` for Python Packages

- If needed, set arguments for the build script:

```
DEB_PYTHON_CLEAN_ARGS = -all
DEB_PYTHON_BUILD_ARGS = --build-base="$(DEB_BUILDDIR)/there"
DEB_PYTHON_INSTALL_ARGS_ALL = --no-compile
DEB_PYTHON_INSTALL_ARGS_foo = --force
```

- Exclude packages that should **not** be compiled:

```
DEB_PYTHON_PACKAGES_EXCLUDE = python-something-doc
```

control for Ruby Packages

- Architecture: all
- Section: interpreters 
- Package: lib*something*-ruby
  - Foo ⇨ libfoo-ruby
  - RubyBar ⇨ libbar-ruby

Build-Dependencies

- ruby-pkg-tools *Ruby Packaging Tools*

`control` for Ruby Packages

- Make a package for each version of Ruby:
  - Package: lib*something*-ruby1.8
    - Depends: libruby1.8
  - Package: lib*something*-ruby1.9
    - Depends: libruby 1.9
- Make a “dummy” package
  - Package: lib*something*-ruby
    - Depends: lib*something*-ruby1.8
    - Add a blank line and this text to the Description:

      This is a dummy package depending on the library for the current default version of Ruby.

rules for Ruby Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with either setup.rb rules...

```
include /usr/share/ruby-pkg-tools/1/class/ruby-setup-rb.mk
```

...or extconf.rb rules

```
include /usr/share/ruby-pkg-tools/1/class/ruby-extconf-rb.mk
```

- If needed, specify a different build script

```
DEB_RUBY_SETUP_CMD := install.rb
```

- If needed, set arguments for the build script

```
DEB_RUBY_CONFIG_ARGS = --site-ruby=$(DEB_RUBY_LIBDIR)
```

`control` for Qmake Packages

Build-Dependencies

- Either `libqt4-dev`... *Headers for Qt 4*
- ...or `libqt3-mt-dev` *Headers for Qt 3*

- Set alternatives for the correct Qt version

```
update-alternatives --config application
```

  - `lrelease`, `lupdate`, `moc`, `qmake`, `uic`

rules for Qmake Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with Qmake rules

```
include /usr/share/cdbs/1/class/qmake.mk
```

`control` for KDE Packages

- `Section: kde`

Build-Dependencies

- Same build-dependencies of Autotools rules
- `kdelibs4-dev` *Headers for KDE Libraries*
- Possibly `-dev` packages for KDE modules

Dependencies

- If possible, depend on individual KDE applications instead of whole KDE modules

rules for KDE Packages

- Replace default Autotools rules...

```
include /usr/share/cdbs/1/class/autotools.mk
```

...with KDE rules

```
include /usr/share/cdbs/1/class/kde.mk
```

  - If there are errors about “unsermake”, remove the unsermake package

Local Repository

- Keep your packages in a local repository so they can be managed with APT
  - Make a directory & put the the packages there

```
mkdir /usr/local/debian
mv *.deb /usr/local/debian/
```

  - Generate an APT index

```
dpkg-scanpackages /usr/local/debian /dev/null | \
gzip - > /usr/local/debian/Packages.gz
```

  - Edit `/etc/apt/sources.list`

```
deb file:/usr/local/debian ./
```

More Information

- Debian Documentation
  - General & User Information
    - Debian Project Documentation `doc-debian`
      - The Debian Free Software Guidelines are a part of the Debian GNU/Linux Social Contract
    - Debian Reference `debian-reference`
  - Developer Information
    - CDBS documentation found in `/usr/share/doc/cdbs`
    - Debian New Maintainers' Guide `maint-guide`
    - Debian Policy Manual `debian-policy`
    - Debian Developer's Reference `developers-reference`

More Information

- Documentation about other examples
  - Perl
    - Debian Perl Policy `debian-policy`
  - Python
    - Debian Python Policy
    - Debian Wiki: DebianPython/NewPolicy
    - python-central HOWTO
  - Ruby
    - Debian/Ruby Extras Team: Ruby Package Tools
    - Ruby Policy (Draft) `ruby`

Building Debian Packages

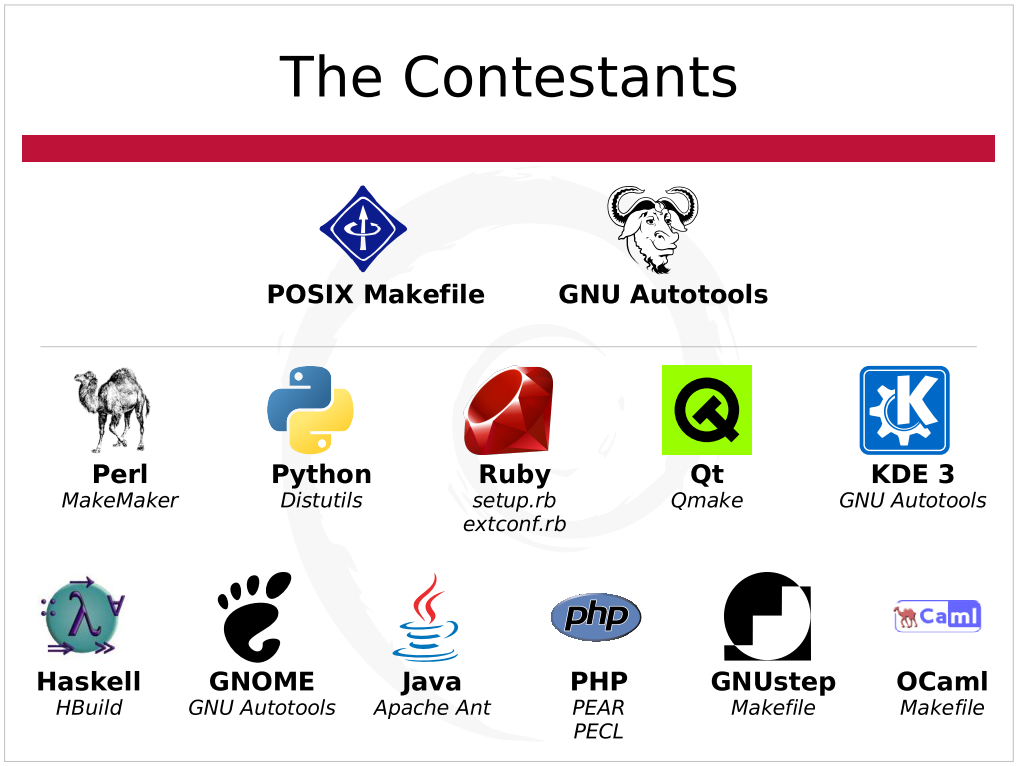
Package your software

the quick and easy way

with CDBS

- Aim: Walk though the creation of Policy-compliant Debian packages in the easiest possible way using the Common Debian Build System (CDBS)

- Audience: Primarily for people who don't want to become Debian Maintainers
  - Administrators wanting to ease maintenance of custom software
  - Hackers wanting to distribute their works
  - Debian enthusiasts who want to learn more



- CDBS has pre-made rules to build a Debian package from many different build systems
  - For binary-only installers, use `checkinstall`

- Makefile & Autotools package types will be covered in this presentation
- If time permits, any package type from the middle row may be covered
- Package types from the bottom row can be built, but won't be covered

- If requested, an example can be walked-through for any package types covered

Why Bother?

- Integration
  - Paths
- Maintenance
  - Upgrading
  - Clean removal
  - Multiple installations
- Security
  - Package signatures
  - `debsums`

- Why use dpkg instead of “make install”?
  - Integrates with distribution
    - Otherwise: /usr/local, /opt/, /usr/foo, $HOME?
  - Easier to maintain
    - Provides easy way to remove & upgrade
      - No Cruft Left Behind
    - Install on many machines without extra effort
  - Easier to audit & secure
    - APT signatures prevent corruption / tampering
    - debsums verify integrity of installed files
- Why not just use `checkinstall`?
  - Crude packages, often doesn't work
- Why not use alien?
  - RPMs for other distributions won't work reliably
    - Different paths & versions
    - Distribution-specific scripts
    - In worst case, can wreck your system!

What is a Debian Package?

- Not just a bunch of files to unpack
  - Meta-data
  - Tools & mechanisms
- Source packages
  - `.orig.tar.gz` *Unmodified upstream source*
  - `.diff.gz` *Debian-specific changes*
  - `.dsc` *Debian source control info*
- Binary packages
  - `.deb` *Debian Package*

- It's not a big truck!
  - Meta-data for
    - tools: dependencies, signatures
    - users: classification, description, notices
    - DDs: maintainer info, changes, bugs
  - Tools & mechanisms to facilitate use & development

- Source packages for developers
  - .orig.tar.gz: “Upstream” source
  - .diff.gz: Changes for Debian packaging
  - .dsc: Source package meta-data & signature

- Binary packages for users
  - “ar” archive containing files & meta-data
  - Built procedurally from source, never by hand

Package-Building Tools

- **build-essential**
  - **make** *GNU Make*
  - **dpkg-dev** *Debian Package Building Tools*
  - **gcc & g++** *C & C++ compilers*
  - **libc6-dev** *C Standard Library headers*
- **cdbs** *Common Debian Build System*
- **debhelper** *Automates usual packaging tasks*
- **dh_make** *Debianizes a source directory*
- **devscripts** *Convenience scripts for packagers*

- Some basic tools are needed to build any package
  - build-essential
    - make: Package built from source with Makefile
    - dpkg-dev: Basic tools for handling packages
    - gcc, g++, libc
- Debian archive is huge (4.0 “etch” > 18k pkgs!)
  - DDs need tools to ease packaging
  - Common Debian Build System (CDBS)
    - Packaging rules for typical build systems
  - debhelper
    - Scripts called by the package build rules to automate typical tasks
  - dh_make
    - Makes a Debian source package skeleton out of “upstream” source
  - devscripts
    - Useful little scripts to perform typical actions

Laying the Foundation

- Get your source, put it in a directory named

  sourcename-*version*/

- Set environment variables for name & email

```
export DEBFULLNAME="Deborah Netch"
export DEBEMAIL="debbienetch@hacker.net"
```

- Debianize!

```
dh_make --cdbs --createorig
```

- Get your source
  - Must be in directory named $srcname-$version
  - This will be referred to as the source directory
- Enter the source directory
  - All commands & paths will be relative from here
- Set name & email environment variables in shell
- Run dh_make
  - "--createorig" makes a copy of the source directory, which will become the .orig.tar.gz
  - License can be specified with "-c gpl" or "--copyright gpl"
  - dh_make will create debian/ subdirectory & example files
    - Delete example files if not needed

Package Data in debian/

- `changelog` *History of the package*
- `control` *Meta-data about packages*
- `copyright` *License information*
- `rules` *Makefile to build binary packages*
- `watch` *Location to check for updates*
- Maintainer scripts
  - `preinst`, `postinst`, `prerm`, `postrm`, and so on...

- All packaging info lives in debian/ subdirectory
  - Files for dpkg
    - changelog: packaging-specific changes
      - Not same as upstream changelog!
    - control: Meta-data for all packages to be built, may have multiple package entries
    - copyright: License (same as “upstream”)
    - rules: Makefile used to build binary packages
    - watch: URL to track updates by “upstream”
    - Maintainer Scripts:
      - Actions to perform during install/remove
      - Their name determines when they are run

Debhelper Files in debian/

• `cron.d`	*A job to register with* `cron`
• `dirs`	*List of directories to create*
• `docs`	*List of documentation files*
• `emacsen-*`	*EMACS site-lisp files*
• *packagename*`-default`	*Variables for* `init` *script*
• *packagename*`.doc-base`	*Online help page*
• `init.d`	`init` *script*
• `manpage.*`	*Manual page*
• `menu`	*Entry for the Debian menu*
• `README.Debian`	*Debian-specific ReadMe*

- Debhelper provides mechanisms to handle typical aspects of packaging
  - compat: Debhelper version, ignore for now
  - cron.d: Registers a job for cron
  - dirs: Directories to create before install step
  - docs: put in /usr/share/doc/$PKGNAME/
    - Files here over certain size are compressed
  - emacsen-*: site-lisp files to be compiled for various emacs flavors during install
  - $PKGNAME-default: Configuration variables for init script, renamed to /etc/default/$PKGNAME
  - $PKGNAME.doc-base: Page for Debian's centralized documentation framework
  - init.d: “init” script to be installed in /etc/init.d/
  - manpage: “man” page to install
    - nroff, or Docbook (auto-converted to nroff)
  - menu: Registers entry in Debian menu
  - README.Debian: readme for Debian users
    - put in /usr/share/doc/$PKGNAME

Describing the Source

- Source: *Name of the source package*
- Maintainer: *Your name <email address>*
- Build-Depends: *Packages needed for building*

– devscripts includes dpkg-genbuilddeps

```
fakeroot dpkg-genbuilddeps
```

- control file has several sections
- first section describes the source package
  - Also sets default values for binary packages
  - Source: name of the source package
    - Same name as source directory
  - Maintainer: You!
  - Build-Depends: Any packages needed to build the binary packages from the source package
    - build-essential is assumed, don't include

1. What software is necessary can often be found in the README or INSTALL files
2. Try building, see if it complains about missing
3. dpkg-genbuilddeps from devscripts
   - Builds package, watching with strace
   - Lists packages containing any files used in the build process
     - List is usually overkill, some packages are obviously unnecessary

Describing the Packages

- Package: *Name of a package*
- Architecture: *Compatible platforms*
- Description: *Brief synopsis (< 80 characters)*

  An extended description of the package which gives the user an impression of the package's purpose. It should describe the package's capabilities, intended use, and relation to other parts of the system.
  .
  The description should be word-wrapped to 80 characters and every line must begin with a space. A line with only a space then a period will be shown as a blank line.

- All following sections describe binary packages
  - Package: name of the package
    - For single package, same as source package
  - Architecture: package works on CPU or OS
    - Use “all” for arch-independant packages
    - Use “any” for arch-dependant packages
  - Description:
    - < 80 character description of the package
    - Extended description underneath
      - Describe package's purpose to user
        - Should be obvious even to unfamiliar users
      - Word wrap at 80 characters
      - Each line begins with a space
      - Make a blank line with only a period
- There are other fields, but these are required ones

Declaring Dependencies

- Depends: *Absolutely needed to run me*
  - `devscripts` includes `dpkg-depcheck`
- Recommends: *Usually needed for me to function*
- Suggests: *Would enhance my functionality*
- Pre-Depends: *Needed during my installation*
- Conflicts: *Cannot be installed alongside me*
- Replaces: *Remove this when I am installed*
- Provides: *I can perform this capability*

- Each package entry also has dependencies
  - Depends: Must always be installed with package
    - Usually the only kind you'll need
    - debhelper auto-adds any shared libraries used
    - Use dpkg-depcheck for a hint (see manpage)
  - Recommends: Usually required for package
  - Suggests: Makes package more useful
  - Pre-Depends: (avoid if possible) Used by package's install process, install this before
  - Conflicts: Cannot be installed on the same system as package (user asked to resolve)
  - Replaces: This package will be auto-removed if package is installed
    - Often used with Conflicts so user doesn't have to solve the conflict manually
  - Provides: Package provides functionality of this virtual package

Keeping Up-to-Date

- Track your changes with `changelog`
  - Add information about the current version:

```
debchange --append
```

  - Make changes to your package:

```
debchange --increment
```

- Upgrade to a new version of the source:

```
fakeroot uupdate ../sourcename-newversion/
```

- Track changes to packaging with changelog
  - Makes it easier to work on in future
- Two kinds of version for package
  - “upstream” version (changes to source)
  - packaging version (changes to packaging)
- debchange (devscripts): helper for changelog
  - Gives correct changelog format
  - shell env vars:DEBFULLNAME,DEBEMAIL,EDITOR
  - --append: add more info about current version
  - --increment: You released your package, now you want to make some changes to the packaging and release an update
- uupdate (devscripts): helper for new version
  - Argument is location of new version of source (tarball or directory)
  - Tries to guess version, or specify with “-v”
  - Updates version in control & adds new changelog entry

Building the Packages

- `rules` Makefile used to create the packages
  - Build the source:

```
debian/rules build
```

  - Make the binary packages:

```
fakeroot debian/rules binary
```

  - Clean up:

```
fakeroot debian/rules clean
```

  - All-in-one

```
dpkg-buildpackage -rfakeroot
```

- All packages built from source using Makefile called rules
- Various actions in rules
  - build: runs build system's “build” step to compile from source, etc.
  - binary: runs build system's “install” step & assembles the result into a package
  - clean: cleans up files left over from other actions

- dpkg-buildpackage does everything
  - Builds
  - Creates the binary packages
  - Creates .orig.tar.gz, .diff.gz, .dsc
  - Signs everything with your GPG key

Tweaking the Result

- Root directory for each binary package is **debian/*packagename*/**
  - DEB_DESTDIR variable in `rules`
- What if the build system doesn't install files in the right place?
  - `binary-post-install/`*packagename* runs after build system's install step (ie. "`make install`")

```
binary-post-install/packagename::
   mv wrong_place/file $(DEB_DESTDIR)/right_place/file
   rm $(DEB_DESTDIR)/this/does/not/belong/here
```

- When build system does install step, files installed to a fake root directory at debian/$PKGNAME
  - When making multiple packages from one source, each package has its own fake root
  - Each package is generated from the files there
  - `rules` has DEB_DESTDIR variable that points to root directory of package being built/installed

- Sometimes the build system doesn't install files into the right place
  - "binary-post-install" rule runs specified commands after build system's install step

rules for Makefile Packages

- Default Makefile rules

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/makefile.mk
```

- Define targets for make:

```
DEB_MAKE_BUILD_TARGET   = all
DEB_MAKE_CLEAN_TARGET   = clean

DEB_MAKE_CHECK_TARGET   = check
DEB_MAKE_INSTALL_TARGET = install DESTDIR=$(DEB_DESTDIR)
```

- If needed, set environment variables

```
DEB_MAKE_ENVVARS        = CFLAGS="-fpic"
```

- “dh_make --cdbs” defaults to CDBS Makefile rules
  - Projects built using only a makefile
    - “make”, “make install” ONLY
    - Not often used, only for very simple projects
    - Makefile is simplest build system, many other CDBS rules are derived from it

- Targets can be specified for build system steps
  - clean & build are optional, defaults used if unset
  - install/check steps won't be run if target is unset
    - If Makefile has no install rule, you need to move the files to DEB_DESTDIR manually

- If needed, set environment variables for make

`control` for Autotools Packages

Build-Dependencies

- `autotools-dev` *Support files for Autotools*

and possibly...

- `autoconf` *`configure` script generator*
- `automake` *Makefile generator*
- `libtool` *Library support script*

- GNU Autotools is the most common build system
  - “./configure”, “make”, “make install”
    - “configure” generates Makefile, which is then used to build
  - CDBS can make a package from these with almost no configuration

- autotools-dev: contains latest versions of files used by “configure” script
- If source doesn't come with a configure script
  - autoconf, automake, & libtool are the 3 main parts of autotools
    - Used to generate “configure” script & various other files

rules for Autotools Packages

- Many projects can use the default rules:

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

- To specify flags for the configure script:

```
DEB_CONFIGURE_EXTRA_FLAGS = --enable-foo --with-bar
```

- In rare cases it may be necessary to update autoconf, automake, and/or libtool

```
DEB_AUTO_UPDATE_AUTOCONF
DEB_AUTO_UPDATE_AUTOMAKE
DEB_AUTO_UPDATE_LIBTOOL
```

- If configure script is found, default is Autotools
  - Based on Makefile rules, options from Makefile rules example can be used
    - Don't need to specify “install” target
  - For many projects, the 2-line default rules are perfectly adequate

- If necessary, use DEB_CONFIGURE_EXTRA_FLAGS to pass flags to the configure script

- A few projects need to run autoconf, automake, or libtool first to generate the configure script
  - Most projects do not, and running these can break the build system, so don't add those lines unless you know you absolutely have to!

- There are more options, but they're rarely used

control for Perl Module Packages

- Architecture: all
- Section: perl
- Package: lib*something*-perl
  - Foo::Bar ⇨ libfoo-bar-perl

Build-Dependencies

- perl *Interpreter & core modules*

Dependencies

- ${perl:Depends} *Automatic Perl dependencies*

- Build Perl modules, like from CPAN
  - Build system is Makemaker (“Makefile.PL”)
- Perl modules are interpreted, so arch is “all”
- Should probably go in the “perl” section
- The name of a package containing a Perl module should be “lib”, followed by the module name. followed by “-perl”, colons replaced with dash
  - Important, needed by the packaging system!

- Build-depend on perl for interpreter and core modules

- Debhelper can auto-determine dependency on correct Perl interpreter package, depend on ${perl:Depends}

rules for Perl Module Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with Perl Module rules

```
include /usr/share/cdbs/1/class/perlmodule.mk
```

- To specify flags for `Makefile.PL` script:

```
DEB_MAKEMAKER_USER_FLAGS = --with-something
```

- Based on Makefile rules
  - Runs `Makefile.PL` to generate Makefile, then `make` & `make install`
- Replace default Makefile rules with Perl module rules

- If necessary, use `DEB_MAKER_USER_FLAGS` to pass flags to the `Makefile.PL` script

`control` for Python Packages

- `Architecture: all`
- `Section: python`
- `Package: python-`*`something`*
  - `Foo` ⇨ `python-foo`
  - `PyBar` ⇨ `python-pybar`

`Build-Dependencies`

- `python-all-dev` *Build files for all Python versions*
- `python-central` *Compiles modules automatically*

Continued...

- Build system is Distutils

- Python modules are interpreted, so arch is “`all`”
- Should probably go in the “`python`” section
- To name package, add “`python-`” before the module's name
  - Important, used by packaging system!

- Build depend on “`python-all-dev`“ & “`python-central`“
  - python-central auto-magically compiles & manages Python modules

`control` for Python Packages

- For the source package, add field:

`XS-Python-Version: all`

  - "`current`" or specific version(s) can also be used

- For each Python binary package, add field:

`XB-Python-Version: ${python:Versions}`

Dependencies

- `Depends: ${python:Depends}`
- `Provides: ${python:Provides}`

- python-central automatically makes a package of the module for each version of Python & one virtual package that installs appropriate version
  - Needs extra fields for source & binaries
- Add field to source
  `XS-Python-Version: all`
  - If module doesn't work on all Python versions, use "`current`" or specify versions instead
- Add field
  `XB-Python-Version: ${python:Versions}` to to any package that needs to compile Python code

- Depend on "`${python:Depends}`"
  - Python versions from python-central
- Provide "`${python:Provides}`"
  - Makes virtual package

rules for Python Packages

- Before include lines, set DEB_PYTHON_SYSTEM

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

DEB_PYTHON_SYSTEM = pycentral
```

- Replace default Makefile rules…

```
include /usr/share/cdbs/1/class/makefile.mk
```

…with Python rules

```
include /usr/share/cdbs/1/class/python-distutils.mk
```

- If build script isn't setup.py, specify it

```
DEB_PYTHON_SETUP_CMD = install.py
```

Continued...

- Uses Distutils build script (usually setup.py) to build and install
  - Modules compiled automatically at install-time
- Before any include lines, set DEB_PYTHON_SYSTEM to “pycentral”
- Replace default Makefile rules with Python Module rules
  - Python module rules must be after debhelper rules!
- If the build script isn't named setup.py specify it with DEB_PYTHON_SETUP_CMD

rules for Python Packages

- If needed, set arguments for the build script:

```
DEB_PYTHON_CLEAN_ARGS = -all
DEB_PYTHON_BUILD_ARGS = --build-base="$(DEB_BUILDDIR)/there"
DEB_PYTHON_INSTALL_ARGS_ALL = --no-compile
DEB_PYTHON_INSTALL_ARGS_foo = --force
```

- Exclude packages that should **not** be compiled:

```
DEB_PYTHON_PACKAGES_EXCLUDE = python-something-doc
```

- If you need to pass arguments to the build script, you can use DEB_PYTHON_***ACTION***_ARGS
  - Can specify "`install`" arguments for all packages and/or individual packages

- The "`python-`" prefix in a package's name tells CDBS to treat it as a Python package (compile modules, etc.)
  - However, if you need to make a package that begins with "`python-`" but shouldn't be compiled (like a "`-doc`" package), use DEB_PYTHON_PACKAGES_EXCLUDE

control for Ruby Packages

- Architecture: all
- Section: interpreters
- Package: lib*something*-ruby
  - Foo ⇨ libfoo-ruby
  - RubyBar ⇨ libbar-ruby

Build-Dependencies

- ruby-pkg-tools *Ruby Packaging Tools*

- Ruby modules are interpreted, so arch is “all”
- Unlike Python & Perl, Ruby doesn't have its own section, but many packages are in “interpreters” section
- To name packages, add “lib” before the module name and “-ruby” after
  - Important, needed by the packaging system!

- Build-depend on ruby-pkg-tools
  - Provides Ruby rules
  - Depends on ruby & rdoc, no need to add those

`control` for Ruby Packages

- Make a package for each version of Ruby:
  - Package: lib*something*-ruby1.8
    - Depends: libruby1.8
  - Package: lib*something*-ruby1.9
    - Depends: libruby 1.9
- Make a “dummy” package
  - Package: lib*something*-ruby
    - Depends: lib*something*-ruby1.8
    - Add a blank line and this text to the Description:

      This is a dummy package depending on the library for the current default version of Ruby.

- Unlike Python modules, Ruby modules currently have no helper to automatically generate packages for each version of Ruby
  - Must be done manually, but it's not difficult

- Write a package entry for each version of Ruby
  - Add Ruby version after the name
  - Depend on appropriate version of `libruby` & any modules used
  - All should have the same description
- Write a package entry for a “dummy” package
  - Makes upgrading much easier
  - Depends on package for default Ruby version
    - Currently 1.8
  - Same description as other packages, but with blurb about being a dummy package

rules for Ruby Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with either `setup.rb` rules...

```
include /usr/share/ruby-pkg-tools/1/class/ruby-setup-rb.mk
```

...or `extconf.rb` rules

```
include /usr/share/ruby-pkg-tools/1/class/ruby-extconf-rb.mk
```

- If needed, specify a different build script

```
DEB_RUBY_SETUP_CMD := install.rb
```

- If needed, set arguments for the build script

```
DEB_RUBY_CONFIG_ARGS = --site-ruby=$(DEB_RUBY_LIBDIR)
```

- If using `setup.rb`, replace default Makefile rules with Ruby `setup.rb` rules
- If using `extconf.rb`, replace default Makefile rules with Ruby `extconf.rb` rules
  - Ruby rules must be after debhelper rules!

- If the build script isn't named `setup.rb` or `extconf.rb`, specify it with `DEB_RUBY_SETUP_CMD`
  - Predecessor of `setup.rb` was `install.rb`, can be used with `setup.rb` rules

- If you need to pass arguments to the build script's configuration step, you can use `DEB_RUBY_CONFIG_ARGS`
  - Older `install.rb` used `--site-ruby` instead of `--siteruby`

`control` for Qmake Packages

Build-Dependencies

- Either `libqt4-dev`… *Headers for Qt 4*
- …or `libqt3-mt-dev` *Headers for Qt 3*

- Set alternatives for the correct Qt version

```
update-alternatives --config application
```

  - `lrelease`, `lupdate`, `moc`, `qmake`, `uic`

- Qmake is build tool for Qt (but not KDE) projects
  - `qmake` & other tools found in `libqt*-dev`
- Qt 3 & Qt 4 are incompatible & need different packages
  - Debian uses the “alternatives” system to allow both to be installed at the same time
    - Use `update-alternatives` to make sure all Qt tools are set to the appropriate version

- Binary packages should depend on corresponding Qt libraries

rules for Qmake Packages

- Replace default Makefile rules...

```
include /usr/share/cdbs/1/class/makefile.mk
```

...with Qmake rules

```
include /usr/share/cdbs/1/class/qmake.mk
```

- Based on Makefile rules
  - Runs `qmake` to generate Makefile, then `make` & `make install`
- Replace default Makefile rules with qmake rules

- Qmake projects often don't have “`install`” rule, in which case file would need to be moved to `DEB_DESTDIR` manually

`control` for KDE Packages

- `Section: kde`

Build-Dependencies

- Same build-dependencies of Autotools rules
- `kdelibs4-dev` *Headers for KDE Libraries*
- Possibly `-dev` packages for KDE modules

Dependencies

- If possible, depend on individual KDE applications instead of whole KDE modules

- Should probably go in the “kde” section

- Uses Autotools, similar build-depends from Autotools rules are needed
- Always build-depend on `kdelibs4-dev`
- If your package needs specific features from a KDE module (ie. `kdebase`, `kdepim`, etc.), build-depend on corresponding `-dev` package (ie. `kdebase-dev`, `kdepim-dev`, etc.)

- Unless you need the whole KDE module, depend only on the applications you need
  - “`konsole`” instead of “`kdebase`”

rules for KDE Packages

- Replace default Autotools rules…

```
include /usr/share/cdbs/1/class/autotools.mk
```

…with KDE rules

```
include /usr/share/cdbs/1/class/kde.mk
```

  - If there are errors about “unsermake”, remove the unsermake package

- Extends Autotools rules with various settings & actions needed by KDE applications
- Replace default Autotools rules with KDE

- unsermake is used by some KDE applications but doesn't work with CDBS, so if you see errors about it during build, remove the unsermake package & try again

Local Repository

- Keep your packages in a local repository so they can be managed with APT
  - Make a directory & put the the packages there

```
mkdir /usr/local/debian
mv *.deb /usr/local/debian/
```

  - Generate an APT index

```
dpkg-scanpackages /usr/local/debian /dev/null | \
gzip - > /usr/local/debian/Packages.gz
```

  - Edit `/etc/apt/sources.list`

```
deb file:/usr/local/debian ./
```

- Use a very simple repository on your machine to keep your packages organized using APT
  - Make a directory, preferably under `/srv` or `/usr/local`, so it won't be modified by Debian
  - Copy all packages there
  - Run `dpkg-scanpackages` like above, pipe its output through `gzip`, and into `Packages.gz`
    - Re-run `dpkg-scanpackages` whenever the contents of the repository change
  - Add the repository to `sources.list`
  - Update APT
- Can be shared with other machines using HTTP, FTP, SSH, or any mountable filesystem
  - For more advanced setups, try `debarchiver`, `mini-dinstall`, or `reprepro`

More Information

- Debian Documentation
  - General & User Information
    - Debian Project Documentation `doc-debian`
      - The Debian Free Software Guidelines are a part of the Debian GNU/Linux Social Contract
    - Debian Reference `debian-reference`
  - Developer Information
    - CDBS documentation found in `/usr/share/doc/cdbs`
    - Debian New Maintainers' Guide `maint-guide`
    - Debian Policy Manual `debian-policy`
    - Debian Developer's Reference `developers-reference`

- DDP: www.debian.org/doc/
  - `doc-debian`: info about The Debian Project
    - The Debian Linux Manifesto
    - Constitution for the Debian Project
    - "Social Contract" with the Free Software Community
      - Contains Debian Free Software Guidelines
    - The Debian GNU/Linux FAQ
    - Debian Bug Tracking System documentation
    - Introduction to the Debian mailing lists
  - `debian-reference`: “broad overview of the Debian system” (They really mean broad!)
  - CDBS docs in `cdbs` package
  - `maint-guide`: in-depth guide to creating packages & getting them into Debian archive
  - `debian-policy`: design & policies of Debian's OS, package standard, archive, etc.
  - `developers-reference`: compendium of procedures & resources for Debian Developers

More Information

- Documentation about other examples
  - Perl
    - Debian Perl Policy `debian-policy`
  - Python
    - Debian Python Policy
    - Debian Wiki: DebianPython/NewPolicy
    - python-central HOWTO
  - Ruby
    - Debian/Ruby Extras Team: Ruby Package Tools
    - Ruby Policy (Draft) `ruby`