Guía del nuevo desarrollador de Debian

COLABORADORES

	TÍTULO : Guía del nuevo desarrollador de Debian	

ACCIÓN	*NOMBRE*	*FECHA*	*FIRMA*
ESCRITO POR	Josip Rodin, Osamu Aoki, Javier Fernández-Sanguino Peña, David Martínez, Ana Beatriz Guerrero López, Francisco Javier Cuadrado, y Innocent De Marchi	25 de mayo de 2017	
		25 de mayo de 2017	
		25 de mayo de 2017	
		25 de mayo de 2017	
		25 de mayo de 2017	
		25 de mayo de 2017	

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE

Índice general

Capítulo 1

Empezando «de la forma correcta».

Este documento tratará de describir cómo se construye un paquete Debian GNU/Linux para el usuario común de Debian y para futuros desarrolladores en un lenguaje informal, y con multitud de ejemplos. Hay un antiguo dicho romano que dice, «*Longum iter est per preaecepta, breve et efficax per exempla!*» (¡Es un largo camino con las reglas, pero corto y eficiente con ejemplos!).

Este documento está actualizado a la versión `jessie` de Debian [1].

Una de las cosas que hace a Debian una de las distribuciones más importantes del mercado es su sistema de paquetes. Aunque hay una gran cantidad de programas disponibles en forma de paquetes de Debian, algunas veces necesitarás instalar programas que no están disponible en este formato. Puede que te preguntes cómo hacer tus propios paquetes y que pienses que quizás ésta es una tarea demasiado difícil. Bueno, si eres un principiante en GNU/Linux, sí es duro, pero si eres un novato, no deberías estar leyendo esto ahora mismo. :-) Necesitas saber algo sobre programación en Unix, pero, desde luego, no tienes que ser un maestro [2].

Sin embargo, hay una cosa que es verdad: para construir y mantener paquetes Debian adecuadamente, necesitarás muchas horas. Para que nuestro sistema trabaje sin errores, nuestros desarrolladores necesitan ser técnicamente competentes y concienzudos.

Si quieres información complementaria sobre cómo construir paquetes lee Sección 1.4.

Las nuevas versiones de este documento están en http://www.debian.org/doc/maint-guide/ y en el paquete `maint-guide`. Las traducciones pueden obtenerse en paquetes como `maint-guide-es`. Debe tenerse presente que las traducciones pueden estar desactualizadas.

Como se trata de un tutorial, he optado por explicar cada paso detalladamente en algunos temas importantes. Algunas explicaciones pueden parecerte irrelevantes. Te ruego que seas paciente. También he omitido intencionalmente algunos casos extremos y algunos aspectos sólo se mencionan con la intención de que el documento sea sencillo.

1.1. Dinamismo social en Debian

Aquí explico mis observaciones sobre la dinámica social en Debian. Espero que esto te preparará para la interacción con Debian:

- Todos somos voluntarios.
  - No puedes imponer a los demás lo que debe hacerse.
  - Debes estar motivado para hacer las cosas por ti mismo.
- La cooperación amistosa es la fuerza motriz.

[1] El documento asume que estás utilizando la versión `jessie`. Si quieres utilizar este documento con una versión anterior (incluidas versiones anteriores de Ubuntu u otras) debes instalar (como mínimo) la versión «backport» de los paquetes `dpkg` y `debhelper`.

[2] Puedes aprender las operaciones básicas del sistema Debian en Debian Reference (http://www.debian.org/doc/manuals/debian-reference/) . Ese documento también trata algunos aspectos sobre programación en UNIX.

  - Tu contribución no debe estresar a los demás.
  - Tu contribución es valiosa sólo cuando los demás te lo agradecen..
- Debian no es la escuela donde recibir atención automática de los profesores.
  - Debes ser capaz de aprender muchas cosas por ti mismo.
  - La atención por parte de otros voluntarios es un recurso muy escaso.
- Debian está mejorando constantemente.
  - Se espera que generes paquetes de alta calidad.
  - Debes adaptarte al cambio.

Se utilizan distintos nombres para referirse a los roles dentro de Debian:

- **autor original** («upstream author»): para referirse a la persona que ha escrito el código original del programa (o la documentación original en el caso de paquetes de documentación).
- **desarrollador original** («upstream maintainer»): la persona que se encarga de mantener el programa (el código fuente) en la actualidad.
- **empaquetador (desarrollador)** («maintainer»): la persona que se encarga de construir y mantener actualizados paquetes para Debian (N. del t.: hay una cierta ambigüedad en la traducción de «maintainer» por desarrollador puesto que se puede confundir con la traducción de «Debian Developer», desarrollador que pertenece al proyecto, y de «upstream maintainer», desarrollador del programa original).
- **patrocinador** («sponsor»): la persona (que debe ser un DD o DM, véase más adelante) que transfiere los paquetes elaborados por el desarrollador al archivo de paquetes de Debian (al repositorio de Debian) después de comprobar que el paquete cumple los requisitos exigidos.
- **mentor**: la persona que ayuda a los desarrolladores principiantes a iniciarse en la construcción y mantenimiento de paquetes.
- **desarrollador de Debian (DD)** («Debian Developer»): la persona que es miembro de Debian. Tiene permiso total para transferir paquetes al repositorio oficial de Debian.
- **empaquetador de Debian (DM)** («Debian Maintainer»): la persona que tiene permiso limitado para transferir paquetes al repositorio oficial de Debian.

No puedes convertirte en **desarrollador oficial de Debian** (DD) de la noche a la mañana porque hace falta algo más que habilidades técnicas. No te sientas desilusionado por esto. Aún puedes subir tu paquete, si es útil a otras personas, como **empaquetador** a través de un **patrocinador** o un **empaquetador de Debian**.

Ten en cuenta que no es necesario construir un paquete nuevo para poder convertirte en desarrollador oficial de Debian. Una opción para ser desarrollador oficial es contribuir al mantenimiento de los paquetes ya existentes en la distribución. Hay muchos paquetes esperando un buen responsable (véase Sección 2.2).

Dado que nos concentramos sólo en los aspectos técnicos del mantenimiento de paquetes en este documento, consulta las siguientes referencias para aprender cómo funciona Debian y cómo puedes participar:

- Debian: 17 años de software libre, ”meritocracia” y democracia (http://upsilon.cc/~zack/talks/2011/20110321-taipei.pdf) (diapositivas de introducción en inglés)
- ¿Cómo puedo ayudar a Debian? (http://www.debian.org/intro/help) (documentación oficial)
- Las Preguntas Frecuentes de Debian GNU/Linux FAQ, capítulo 13 - ’Contribuir al proyecto Debian’ (http://www.debian.org/doc/FAQ/ch-contributing) (documentación semi-oficial)
- Debian Wiki, HelpDebian (http://wiki.debian.org/HelpDebian) (documentación complementaria)
- Sitio web de nuevos miembros de Debian (https://nm.debian.org/) (documentación oficial)
- PUF de mentores en Debian (http://wiki.debian.org/DebianMentorsFaq) (documentación complementaria)

1.2. Programas necesarios para el desarrollo

Antes de empezar, debes asegurarte que tienes instalados algunos paquetes adicionales necesarios para el desarrollo. Observa que la lista no contiene paquetes marcados como `esencial` o `requerido` - al dar por supuesto que ya los tienes instalados.

Los siguientes paquetes vienen en una instalación estándar de Debian, así que probablemente ya los tengas (junto con los paquetes de los que dependen). Aún así, deberías comprobarlo ejecutando `aptitude show` *`nombre_del_paquete`* o con `dpkg -s` *`nombre_del_paquete`*.

El paquete imprescindible para el desarrollo es `build-essential`. Al instalarlo, *también se instalaran otros paquetes* requeridos, consiguiendo una instalación básica para la construcción de paquetes.

Para la construcción de algunos paquetes esto seria suficiente, pero hay otros paquetes que, no siendo esenciales en general para la construcción de nuevos paquetes, puede ser útil tener instalados o, incluso, necesarios para el paquete que estás construyendo:

- `autoconf`, `automake` y `autotools-dev` - muchos programas nuevos usan ficheros de configuración y ficheros `Makefile` que se procesan con la ayuda de programas como éstos (véase `info autoconf`, `info automake`). El paquete `autotools-dev` permite mantener versiones actualizadas de los archivos autoconf y automake y tiene documentación para usar eficientemente ese tipo de archivos.
- `dh-make` y `debhelper` - `dh-make` es necesario para construir el esqueleto de nuestro paquete ejemplo, y se usarán algunas de las herramientas de `debhelper` para construir los paquetes. Aunque no son imprescindibles para la construcción de paquetes se recomiendan *encarecidamente* para nuevos desarrolladores. Hacen el proceso mucho más fácil al principio, y más fácil de controlar en el futuro (véase dh_make(8), debhelper(1)) [3].

  The new `debmake` may be used as the alternative to the standard `dh-make`. It does more and comes with HTML documentation with extensive packaging examples in `debmake-doc`.
- `devscripts` - este paquete contiene algunos guiones útiles para los desarrolladores, pero no son necesarios para construir paquetes. Vale la pena mirar los paquetes recomendados y sugeridos por este paquete (véase `/usr/share/doc/devscripts/README.gz`).
- `fakeroot` - esta utilidad te permite emular al usuario administrador («root»), lo cual es necesario para ciertas partes del proceso de construcción (véase fakeroot(1)).
- `file` - este útil programa puede determinar de qué tipo es un fichero (véase file(1)).
- `gfortran` - el compilador GNU de Fortran 95, necesario si el programa está escrito en Fortran (véase gfortran(1)).
- `git` - este paquete instala el popular sistema de control de versiones, diseñado para hacer el seguimiento de proyectos grandes con eficacia y rapidez; se utiliza para proyectos de código libre importantes como es el caso del núcleo («kernel») de Linux (véase git(1) y git Manual (`/usr/share/doc/git-doc/index.html`)).
- `gnupg` - herramienta que te permite *firmar* digitalmente los paquetes. Esto es especialmente importante si quieres distribuir tu paquete a otras personas, y ciertamente, tendrás que hacerlo cuando tu trabajo vaya a incluirse en la distribución de Debian (véase gpg(1)).
- `gpc` - el compilador GNU de Pascal, necesario si el programa está escrito en Pascal. Merece la pena mencionar aquí `fp-compiler`, un compilador libre de Pascal, que también es bueno en esta tarea (véase gpc(1), ppc386(1)).
- `lintian` - este es el programa que comprueba los paquetes de Debian, puede indicarte muchos de los errores comunes después de construir un paquete, y explica los errores encontrados (véase lintian(1) y Manual de usuario de Lintian (https://lintian.debian.org/-manual/index.html))
- `patch` - esta utilidad es muy práctica para modificar el original de un archivo a partir de un archivo de diferencias (elaborado por el programa **diff**) produciendo un archivo parcheado (véase patch(1)).
- `patchutils` - este paquete contiene programas para trabajar con los «parches» como **lsdiff**, **interdiff** y **filterdiff**.
- `pbuilder` - este paquete contiene programas para generar y mantener entornos **chroot**. Al construir paquetes Debian en estos entornos **chroot** se verifica que las dependencias son las adecuadas y se evitan fallos al construir desde el código fuente (FTBFS de «Fails To Build From Source»). Véase pbuilder(8) y pdebuild(1).

[3] Hay varios paquetes similares pero más específicos como `dh-make-perl`, `dh-make-php`, etc.

- `perl` - Perl es uno de los lenguajes interpretados para hacer guiones (o «scripts») más usados en los sistemas Un*x de hoy en día, comúnmente se refiere a él como la «navaja suiza de Unix» (véase perl(1)).
- `python` - Python es otro de los lenguajes interpretados para hacer guiones (o «scripts») en Debian debido a la combinación de su poder y sintaxis clara (véase python(1)).
- `quilt` - este paquete ayuda a aplicar modificaciones («parches») y hacer el seguimiento de los cambios realizados. Aplica las modificaciones ordenadamente en pila, y es posible aplicar, deshacer y actualizar las modificaciones fácilmente recorriendo la pila (véase quilt(1) y `/usr/share/doc/quilt/quilt.pdf.gz`.)
- `xutils-dev` - algunos programas, normalmente aquellos hechos para X11, también usan programas para generar `Makefile` ejecutando un conjunto de funciones de macro (véase imake(1), xmkmf(1)).

Las breves descripciones dadas anteriormente sólo sirven para introducirte a lo que hace cada paquete. Antes de continuar, por favor, lee la documentación de cada programa, al menos para su uso normal. Puede parecerte algo duro ahora, pero más adelante estarás *muy* contento de haberla leído.

1.3. Documentos necesarios para el desarrollo

Por último, la documentación que se indica a continuación es de *gran importancia* y debería leerse junto con este documento:

- `debian-policy` - el manual de normas de Debian (http://www.debian.org/doc/devel-manuals#policy) incluye la descripción de la estructura y contenidos del archivo, ciertas notas sobre el diseño del sistema operativo, el Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html) («FHS», que explica dónde deberían estar cada fichero y directorio), y, lo más importante para ti, describe los requisitos que debe satisfacer cada paquete para ser incluido en la distribución (véase la copias locales en `/usr/share/doc/debian-policy/policy.pdf.gz` y `/usr/share/doc/debian-policy/fhs/fhs-2.3.pdf.gz`).
- `developers-reference` - la referencia del desarrollador de Debian (http://www.debian.org/doc/devel-manuals#devref) para todos los temas no específicamente relacionados con los detalles técnicos de cómo empaquetar, tales como la estructura del archivo (o repositorio de Debian), cómo renombrar, abandonar, adoptar paquetes, cómo hacer NMU («Non-Maintainer Uploads», o envíos por personas distintas del desarrollador), como gestionar los errores que los usuarios envían, buenas prácticas de empaquetado, cómo y cuando enviar los paquetes, etc. (véase la copia local en `/usr/share/doc/developers-referen developers-reference.pdf`).

Por último, la documentación que se indica a continuación es de *gran importancia* y debería leerse junto con este documento:

- Tutorial de «Autotools» (http://www.lrde.epita.fr/~adl/autotools.html) contiene un tutorial muy bueno sobre el sistema de compilación GNU conocido como «GNU Autotools» cuyos componentes más importantes son «Autoconf», «Automake», «Libtool», y «gettext»
- `gnu-standards` - este paquete contiene dos documentos del proyecto GNU: «GNU Coding Standards» (http://www.gnu.org/prep/standards/html_node/index.html) , y «Information for Maintainers of GNU Software» (http://www.gnu.org/prep/maintain/html_node/index.html) . Aunque no se exige su cumplimiento en Debian, son útiles como orientación y sentido común (véase las copias locales en `/usr/share/doc/gnu-standards/standards.pdf.gz` y `/usr/share/doc/gnu-standards/maintain.pdf.gz`).

Si este documento contradice en algún aspecto a los documentos mencionados anteriormente, prevalecen estos últimos. Por favor, envía un informe de error del paquete `maint-guide` utilizando la orden **reportbug**.

El siguiente documento es un tutorial alternativo que puedes leer a la vez que el presente documento.

- Debian Packaging Tutorial (http://www.debian.org/doc/packaging-manuals/packaging-tutorial/packaging-tutorial)

1.4. Dónde pedir ayuda

Antes de decidirte a plantear alguna cuestión en algún lugar público, lee la documentación:

- archivos en `/usr/share/doc/`*`package`* para cada uno de los paquetes
- contenidos de **`man`** *`command`* para todos los comandos pertinentes
- contenidos de **`info`** *`command`* para todos los comandos pertinentes
- contenidos de debian-mentors@lists.debian.org archivo de la lista de correo (http://lists.debian.org/debian-mentors/)
- contenidos de debian-devel@lists.debian.org mailing list archive (http://lists.debian.org/debian-devel/)

Considera el uso eficaz de los motores de búsqueda en la web incluyendo `site:lists.debian.org` en la cadena de búsqueda, para limitar el dominio.

Construir un paquete pequeño es una buena forma de aprender los detalles del mantenimiento de paquetes. Inspeccionar paquetes bien mantenidos es la mejor forma de aprender cómo otros mantienen paquetes.

Si todavía tienes preguntas acerca de la construcción de paquetes a las que no puedes encontrar respuesta en la documentación disponible y los recursos web, puedes formularlas interactivamente:

- debian-mentors@lists.debian.org mailing list (http://lists.debian.org/debian-mentors/) . (Esta lista de correo es para los principiantes.)
- debian-devel@lists.debian.org mailing list (http://lists.debian.org/debian-devel/) . (Esta lista de correo es para expertos.)
- IRC (http://www.debian.org/support#irc) tal como `#debian-mentors`.
- Los equipos se centran en un conjunto específico de paquetes.(La lista completa está en https://wiki.debian.org/Teams (https://wiki.debi Teams))
- Listas de correo en un idioma concreto como debian-devel-{french,italian,portuguese,spanish}@lists.debian.org o bien debian-devel@debian.or.jp. (La lista completa está disponible en https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) and https://lists.debian.org/users.html (https://lists.debian.org/users.html))

Los desarrolladores de Debian con más experiencia te ayudaran con gusto, si haces las preguntas después realizar el esfuerzo requerido.

Cuando recibas un aviso de fallo (sí, avisos de fallos, ¡de verdad!) sabrás que es el momento de indagar en el Sistema de seguimiento de fallos de Debian (http://www.debian.org/Bugs/) y leer su documentación para poder tratar los informes de forma eficiente. Te recomiendo la lectura de la Referencia del Desarrollador, en particular el capítulo de «Referencia del desarrollador sección 5.8 - Manejo de fallos (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling) » («Handling Bugs»).

Aunque todo funcione bien, es el momento de cruzar los dedos. ¿Por qué? Por que en sólo unas horas (o días) usuarios de todo el mundo empezarán a usar tu paquete, y si cometiste algún error crítico centenares de usuarios furiosos de Debian te bombardearán con correos... sólo bromeaba :-)

Relájate y prepárate para recibir informes de fallos, porque hay mucho más trabajo que realizar antes de que tu paquete cumpla las Normas de Debian (una vez más lee la *documentación real* para más detalles). ¡Buena suerte!

Capítulo 2

Primeros pasos

Vamos a construir nuestro propio paquete (o, mejor aún, adoptar un paquete ya existente).

2.1. Plan de trabajo para la construcción de paquetes Debian

Si estas construyendo un paquete Debian con las fuentes originales, el plan de trabajo típico implica varias etapas (en las cuales se generan archivos con nombres específicos) y que se explican a continuación:

- Conseguimos las fuentes originales del programa, normalmente en un archivo comprimido en formato «tar».
  - *nombre_del_paquete*`-`*versión*`.tar.gz`
- Añadir las modificaciones específicas de Debian al programa original en el directorio `debian`, y construir un paquete de fuentes no nativo (es decir, el conjunto de archivos de entrada utilizados para la construcción de paquetes Debian) en formato `3.0 (quilt)`).
  - *nombre_del_paquete_versión*`.orig.tar.gz`
  - *nombre_del_paquete_versión*`-`*revisión*`.debian.tar.gz`[1]
  - *nombre_del_paquete_versión*`-`*revisión*`.dsc`
- Construimos los paquetes binarios Debian, que son archivos instalables ordinarios en formato `.deb` (o en formato `.udeb`, utilizado por el instalador de Debian), desde el paquete fuente de Debian.
  - *nombre_del_paquete_versión*`-`*revisión*`_`*arquitectura*`.deb`

Fíjate que el carácter que separa *nombre_del_paquete* y *versión* se ha cambiado de `-` (guión) a _ (guión bajo).

En los nombres de archivo que siguen, *nombre_del_paquete* se substituye por el **nombre del paquete**, *versión* por la **versión del código fuente**, *revisión* por la **revisión Debian**, *arquitectura* por la **arquitectura del paquete** [2].

Cada paso de este esquema se explica con ejemplos detallados en secciones posteriores.

[1] Para paquetes no nativos Debian construidos en el formato anterior `1.0`, se utiliza *nombre_del_paquete_versión*`-`*revisión*`.diff.gz`

[2] Consulta 5.6.1 ”Source” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source) , 5.6.7 ”Package” (http://www.debian.org/doc/-debian-policy/ch-controlfields.html#s-f-Package) , and 5.6.12 ”Version” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version) . La **arquitectura del paquete** sigue el Debian Policy Manual, 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) y se asigna automáticamente en el proceso de compilación del paquete.

2.2. Elige el programa

Probablemente hayas escogido ya el paquete que deseas construir. Lo primero que debes hacer es comprobar si el paquete está ya en el archivo de la distribución utilizando:

- la orden **aptitude**
- la página web Debian packages (http://www.debian.org/distrib/packages)
- Página web del sistema de seguimiento de paquetes de Debian (http://packages.qa.debian.org/common/index.html) .

Si el paquete ya existe, ¡instálalo! :-) Si te encuentras con que el paquete es un paquete **huérfano** (cuando su desarrollador es el Debian QA Group (http://qa.debian.org/) , es decir, el grupo de calidad de Debian), puedes adoptarlo (convertirte en el responsable de empaquetarlo y mantenerlo) si está disponible. También puedes adoptar un paquete para el cual se ha emitido una «solicitud de adopción» (Request for Adoption o **RFA**) por su desarrollador o por un DD [3].

Hay varios recursos para conocer el estado de los paquetes:

- La orden **wnpp-alert** del paquete `devscripts`
- Paquetes pendientes y futuros nuevos paquetes (http://www.debian.org/devel/wnpp/)
- Registro de informes de fallos de Debian: errores en el paquete `wnpp` en la versión `unstable` (http://bugs.debian.org/wnpp)
- Paquetes Debian que necesitan cariño (http://wnpp.debian.net/)
- Búsqueda de errores `wnpp` basada en palabras clave (http://wnpp-by-tags.debian.net/)

A modo de nota al margen, es importante tener presente que Debian incorpora paquetes de un gran número de programas de todo tipo, y que la cantidad de paquetes disponibles en el repositorio de Debian es mucho mayor al de colaboradores con permiso para incorporar paquetes al repositorio. En consecuencia, la colaboración en el mantenimiento de paquetes que ya están en el repositorio se valora muy positivamente (y es más fácil conseguir patrocinador) por el resto de desarrolladores [4]. Puedes hacer esto de distintas formas:

- hacerte cargo de paquetes huérfanos pero que son utilizados frecuentemente por otros usuarios
- incorporándote a los equipos de desarrolladores (http://wiki.debian.org/Teams) .
- seleccionando errores de los paquetes más populares
- preparando paquetes QA o NMU (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)

Si puedes adoptar el paquete, descarga las fuentes (con algo como `apt-get source` *`nombre_del_paquete`*) y examínalas. Desgraciadamente este documento no incluye aún información exhaustiva sobre la adopción de paquetes. No debería ser difícil entender cómo funciona el paquete ya que alguien ha hecho el trabajo inicial por ti. Aún así es mejor que sigas leyendo, muchos de los consejos que se dan a continuación serán también aplicables en tu caso.

Si el paquete es nuevo y decides que te gustaría verlo en Debian debes seguir los pasos indicados a continuación:

- En primer lugar deberías saber cómo funciona, y haberlo utilizado durante algún tiempo (para confirmar su utilidad).
- Comprueba que no hay nadie más trabajando ya en el paquete consultando la lista de paquetes en los que se está trabajando (http://www.debian.org/devel/wnpp/) . Si nadie está ya trabajando en el empaquetado del programa, envía un informe de error de tipo ITP («Intent To Package», Intento de empaquetado) al meta-paquete `wnpp` utilizando el programa de comunicación de errores **reportbug** (accesible en el menú de herramientas del sistema). Si ya hay alguien trabajando en él, contacta con esa persona: es posible que podáis colaborar. En caso contrario, intenta encontrar otro programa interesante que nadie mantenga.

[3] Véase Debian Developer's Reference 5.9.5. "Adopting a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting) .
[4] Dicho esto, por supuesto, hay nuevos programas que vale la pena empaquetar para Debian.

- El programa **debe tener una licencia**.
  - Si el paquete debe pertenecer a la sección `main` el programa **debe cumplir con las Directrices de Debian para el software libre** (DFSG (http://www.debian.org/social_contract#guidelines)) y **no debe precisar la instalación de otro paquete que no pertenezca a la sección `main`** para su compilación o ejecución como requiere la directiva de Debian («Debian Policy»). Es la situación deseada.
  - Para paquetes de la sección `contrib` la licencia debe cumplir todos los requisitos de la DFSG pero puede precisar la instalación de otro paquete que no sea de la sección `main` para su compilación o ejecución.
  - Para paquetes de la sección `non-free`, no es necesario que la licencia cumpla todos los requisitos de la DFSG pero **debe permitir la distribución del programa**.
  - Si no estás seguro sobre en qué lugar debería ir, envía el texto de la licencia y pide consejo con un correo (en inglés) dirigido a debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) .
- El programa **no** debe ocasionar problemas de seguridad y/o mantenimiento en los sistemas Debian.
  - El programa debería tener una buena documentación o al menos un código fuente legible y claro.
  - Deberías contactar con el autor o autores del programa para comprobar su conformidad con el empaquetado. Es importante que el autor o autores sigan manteniendo el programa para que puedas en el futuro consultarle/s en caso de que haya problemas específicos. No deberías intentar empaquetar programas que no estén mantenidos.
  - El programa **no** debería ejecutarse con «setuid root», o aún mejor: no debería ser «setuid» ni «setgid».
  - El programa no debería ser un demonio, o algo que vaya en los directorios `*/sbin`, o abrir un puerto como usuario administrador.

Por supuesto, esta lista son sólo medidas de seguridad, y con la intención de salvarte de usuarios enfurecidos si haces algo mal con algún demonio «setuid»... Cuando tengas más experiencia en empaquetar, podrás hacer este tipo de paquetes.

Como nuevo desarrollador, es aconsejable que adquieras experiencia con la construcción de paquetes sencillos y se desaconseja construir paquetes complicados.

- Paquetes sencillos
  - archivo binario único, arquitectura = todas (colección de datos como gráficos de fondo de pantalla)
  - archivo binario único, arquitectura = todas (ejecutables escritos en lenguajes interpretados como POSIX)
- Paquetes de complejidad intermedia
  - paquete binario simple, arquitectura = todas (ejecutables ELF escritos en lenguajes compilados tales como C y C + +)
  - paquete binario múltiple, arquitectura = todas y cualquiera (paquetes de ejecutables ELF y documentación)
  - paquetes en los que el formato del archivo fuente no es `tar.gz` ni `tar.bz2`
  - paquetes cuyas fuentes contienen partes que no se pueden distribuir.
- Paquetes muy complejos
  - paquete de módulos de lenguaje interpretado utilizado por otros paquetes
  - paquetes de bibliotecas ELF genéricas utilizadas por otros paquetes
  - múltiples paquetes binarios incluyendo un paquete(s) de bibliotecas ELF
  - paquetes de fuentes con múltiples originales
  - paquetes de módulos del núcleo
  - paquetes de parches del núcleo,
  - cualquier paquete de guiones de desarrollador no triviales

Construir paquetes de alta complejidad no es demasiado difícil, pero requiere más conocimientos. Debes buscar las orientaciones específicas para cada caso según su complejidad. Por ejemplo, algunos lenguajes interpretados tienen sus normas específicas.

- Perl policy (http://www.debian.org/doc/packaging-manuals/perl-policy/)
- Normas para Python (http://www.debian.org/doc/packaging-manuals/python-policy/)
- Normas para Java (http://www.debian.org/doc/packaging-manuals/java-policy/)

Hay otro dicho en latín: *Fabricando fit fabe* (la práctica conduce a la perfección). Es *muy* recomendable practicar y experimentar todos los pasos de la construcción de paquetes Debian con paquetes simples mientras se lee este tutorial. Un simple archivo `hello-sh-1.0.tar.gz` generado como en el ejemplo será un buen punto de partida [5]:

```
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo "Hello!"
EOF
$ chmod 755 hello
$ cd ..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

2.3. Obtén el programa y pruébalo

Lo primero que debes hacer es encontrar y descargar el código fuente original. A partir de este punto se da por supuesto que ya tienes el código fuente que obtuviste de la página del autor. Las fuentes de los programas libres de Unix (y GNU/Linux) generalmente vienen en formato **tar**+**gzip**, con extensión `.tar.gz` o en formato **tar**+**bzip2** con extensión `.tar.bz2`, y generalmente contienen un subdirectorio llamado *`programa-versión`* con todas las fuentes en él.

Si la última versión del código fuente se puede obtener de un sistema VCS del tipo Git, Subversion o un repositorio CVS, puedes descargarlo ejecutando «`git clone`», «`cvs co`» o «`svn co`» y, a continuación comprimiéndolo en un archivo con formato **tar**+**gzip** ejecutando la opción «`--exclude-vcs`».

Si tu programa viene en otro tipo de archivo (por ejemplo, el fichero termina en `.Z` o `.zip`[6]), descomprímelo con las herramientas adecuadas y reconstrúyelo de nuevo.

Si el código fuente del programa viene con algunos contenidos que no cumplan con las DFSG, debes descomprimirlo para eliminar dichos contenidos y volver a comprimirlo modificando el código de versión original añadiendo `dfsg`.

Como ejemplo, usaré el programa conocido como **gentoo**, un gestor de ficheros de X11 en GTK+ [7].

Crea un subdirectorio en tu directorio personal llamado `debian` o `deb` o lo que creas apropiado (por ejemplo `~/gentoo/` estaría bien en este caso). Mueve a él el archivo que has descargado, y descomprímelo de la siguiente forma: `tar xzf gentoo-0.9.12.tar.gz`. Asegúrate de que no hay errores, incluso errores *irrelevantes*, porque es muy probable que haya problemas al desempaquetarlo en sistemas de otras personas, cuyas herramientas de desempaquetado puede que no ignoren estas anomalías. En el terminal de órdenes, deberías ver lo siguiente.

```
$ mkdir ~/gentoo ; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvzf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
```

Ahora tienes otro subdirectorio, con el nombre `gentoo-0.9.12`. Accede a ese directorio y lee *atentamente* la documentación. Habitualmente encontrarás archivos con el nombre `README*`, `INSTALL*`, `*.lsm` o `*.html`. Deberías encontrar instrucciones

[5] No debes preocuparte por perder el `Makefile`. Puedes instalar la orden **hello** simplemente utilizando **debhelper** como en Sección 5.11, o modificando las fuentes originales agregando un nuevo `Makefile` con el objetivo `install` como en Capítulo 3.

[6] Puedes identificar el formato de archivo utilizando la herramienta **file** cuando no es suficiente con la extensión de fichero.

[7] Ten en cuenta que el programa ya ha sido empaquetado previamente. La versión actual (http://packages.qa.debian.org/g/gentoo.html) utiliza las «Autotools» en su construcción y ha cambiado sustancialmente de la versión 0.9.12 que se utiliza en los ejemplos mostrados a continuación.

sobre cómo compilar e instalar el programa (probablemente se asumirá que la instalación será en el directorio `/usr/local/bin`, aunque tú no lo harás eso, como se explica más adelante en Sección 3.3).

Deberías empezar a construir tu paquete en un directorio de fuentes completamente limpio, o simplemente con las fuentes recién desempaquetadas.

2.4. Métodos de compilación simple

Los programas sencillos incluyen un fichero `Makefile` y pueden (generalmente) compilarse con «`make`»[8]. Algunos de ellos soportan `make check`, esta orden ejecuta las comprobaciones automáticas que estén definidas. Generalmente se instalarán en sus directorios de destino ejecutando `make install`.

Ahora intenta compilar y ejecutar el programa, para asegurarte que funciona bien y que no genera errores en el sistema mientras está instalándose o ejecutándose.

También, generalmente, puedes ejecutar `make clean` (o mejor `make distclean`) para limpiar el directorio donde se compila el programa. A veces hay incluso un `make uninstall` que se puede utilizar para borrar todos los archivos instalados.

2.5. Métodos de compilación portables populares

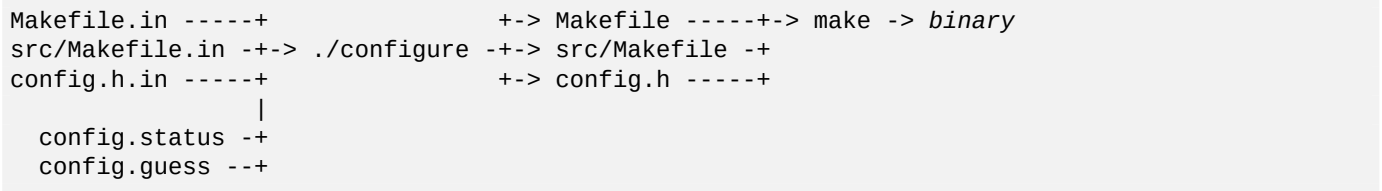
Buena parte de los programas libres están escritos en lenguaje C y C++. Muchos utilizan las «Autotools» y «CMake» para compilar en diferentes plataformas. Estas herramientas se utilizan para generar un archivo `Makefile` y otros archivos necesarios para la compilación. Así, muchos programas se compilan ejecutando «`make; make install`».

Las Autotools son el sistema de compilación GNU e incluyen Autoconf, Automake, Libtool y gettext. Confirmarás que el programa utiliza las autoools por la presencia de los archivos `configure.ac`, `Makefile.am`, y `Makefile.in` [9].

El primer paso en el uso de «Autotools» es la ejecución por parte del autor de la orden `autoreconf -i -f` la cual genera, a partir de los archivos fuente (a la izquierda del gráfico) los archivos que utilizará la orden «configure» (a la derecha del gráfico).

La edición de los archivos `configure.ac` y `Makefile.am` requiere conocer el funcionamiento de **autoconf** y **automake**. Véase «`info autoconf`» y «`info automake`» (ejecutando las órdenes en el terminal).

El segundo paso en el uso de «Autotools« es la ejecución de «`./configure && make`» en el directorio del código fuente para compilar el programa generando un archivo ***`binario`***.



Puedes hacer cambios en el archivo `Makefile`, por ejemplo cambiando el directorio de instalación predeterminado usando las opciones de la orden ejecutando: **./configure --prefix=/usr**.

[8] Muchos programas modernos vienen con un guión `configure` que genera un archivo `Makefile` personalizado para el sistema en uso.

[9] «Autotools» es demasiado complejo para explicarlo en este pequeño tutorial. Esta sección tiene por objeto proporcionar sólo los conceptos clave y referencias. Sin falta, lee el Tutorial de «Autotools» (http://www.lrde.epita.fr/~adl/autotools.html) y `/usr/share/doc/autotools-dev/README.Debian.gz` si debes utilizar «Autotools».

Aunque no es necesario, la actualización del archivo `configure` y de otros archivos con la orden `autoreconf -i -f` es la mejor manera para mejorar la compatibilidad del código fuente [10].

CMake es un sistema de compilación alternativo. La presencia del archivo `CMakeLists.txt` te indicará que se utiliza esta opción para compilar el programa.

2.6. Nombre del paquete y versión

Si el código original se presenta como `gentoo-0.9.12.tar.gz`, puedes poner como **nombre del paquete** `gentoo` y como **versión original** `0.9.12`. Estas referencias se utilizan en el archivo `debian/changelog` descrito más adelante en Sección 4.3.

Aunque este enfoque simple es útil la mayoría de las veces, es posible que debas cambiar el **nombre del paquete** y la **versión original** renombrando las fuentes originales para seguir las normas de Debian y las convenciones actuales.

Debes elegir el **nombre del paquete** de forma que contenga solo letras minúsculas (`a-z`), dígitos (`0-9`), el símbolo de suma (`+`) y resta (`-`) y puntos (`.`). Debe contener al menos dos caracteres, empezar con un carácter alfanumérico y no coincidir con alguno de los paquetes ya existentes. Es buena idea limitar su longitud a un máximo de 30 caracteres [11].

Si el código fuente original utiliza palabras genéricas tales como `prueba-privada` por nombre, es buena idea cambiarlo para no «contaminar» el espacio de nombres y para identificar su contenido en forma explícita [12].

Debes elegir la **versión original** para que tengo sólo caracteres alfanuméricos (`0-9 A-Z a-z`), signo de suma (`+`), tildes (`~`) y puntos (`.`). Se debe comenzar con un dígito (`0-9`) [13]. Es deseable mantener su longitud dentro de 8 caracteres, si es posible [14].

Si el código fuente no utiliza el sistema habitual para codificar la versión, como `2.30.32` sino que utiliza algún tipo de fecha como `09Oct23`, una cadena de nombre en clave al azar o un valor «hash» VCS como parte de la versión, elimínalo de la **versión del código fuente**. Esta información (eliminada) puede ser registrada en el archivo `debian/changelog`. Si debes inventarte una cadena de versión, utiliza el formato `AAAAMMDD` (por ejemplo `20110429`) para la versión del código fuente. Esto asegura que **dpkg** considerará correctamente las versiones posteriores como actualizaciones. Si debes garantizar una transición fluida al formato de versión más habitual, como `0.1` en el futuro, utiliza `0~AAMMDD` como `0~110429` para la versión principal.

El código de la versión [15] será comparado por dpkg(1) como sigue:

```
$ dpkg --compare-versions versión_1 op versión_2
```

La regla de comparación de versiones se pueden resumir de la siguiente manera.

- Las cadenas se comparan desde el inicio hasta el final.
- Los caracteres (letras y símbolos) son anteriores a los números.
- Los números se comparan como enteros.
- Los caracteres (letras y símbolos) se comparan en el orden del código ASCII.
- Hay algunas reglas especiales para los puntos (`.`), símbolo de suma (`+`) y tildes (`~`) como se explica a continuación:
  `0.0` < `0.5` < `0.10` < `0.99` < `1` < `1.0~rc1` < `1.0` < `1.0+b1` < `1.0+nmu1` < `1.1` < `2.0`

Uno de los casos difíciles sucede cuando la versions del código fuente `gentoo-0.9.12-ReleaseCandidate-99.tar.gz` se utiliza como el pre-lanzamiento de `gentoo-0.9.12.tar.gz`. Debes asegurarte que la actualización funciona correctamente cambiando el nombre a las fuentes originales por `gentoo-0.9.12~rc99.tar.gz`.

[10] Añade el paquete `dh-autoreconf` en el campo `Build-Depends`. Véase Sección 4.4.3.

[11] La longitud predeterminada del nombre del paquete en la orden **aptitude** es 30. En más del 90 % de los paquetes, la longitud del nombre del paquete es inferior a 24 caracteres.

[12] Si sigues las Debian Developer's Reference 5.1. "New packages" (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage) , el proceso de ITP (de «intento de empaquetado») abarcará este tipo de cuestiones.

[13] Esta norma más estricta debería ayudar a evitar confundir los nombres de archivo.

[14] El valor predeterminado para la longitud de la versión de la orden **aptitude** es 10. El código de la revisión Debian precedida por un guión consume 2. Para más del 80 % de los paquetes, el código de la versión original es de menos de 8 caracteres y el de la revisión Debian es de menos de 2 caracteres. Para más del 90 % de los paquetes, la versión original es de menos de 10 caracteres y la revisión Debian es de menos de 3 caracteres.

[15] Las cadenas de versión pueden ser **la versión del código fuente** (*`versión`*), **la revisión Debian** (*`revisión`*), o **versión** (*`versión-revisión`*). Consulta Sección 8.1 para saber cómo debe incrementarse **la revisión Debian**.

2.7. Configurar dh_make

Primero debes configurar las variables de entorno «shell» `$DEBEMAIL` y `$DEBFULLNAME` que son utilizadas por varias herramientas de mantenimiento de Debian para obtener tu nombre y correo electrónico como se indica a continuación [16].

```
$ cat >>~/.bashrc <<EOF
DEBEMAIL="tu.direccion.de.correo@ejemplo.org"
DEBFULLNAME="Nombre Apellido"
export DEBEMAIL DEBFULLNAME
EOF
$ . ~/.bashrc
```

2.8. Paquete no nativo Debian inicial

Los paquetes Debian normales son paquetes no nativos construidos a partir de los programas originales de los autores. Si deseas construir un paquete Debian no nativo a partir del código fuente original `gentoo-0.9.12.tar.gz`, puedes construir un primer paquete de Debian no nativo ejecutando la orden **dh_make** como sigue:

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

Deberás cambiar el nombre del archivo por el correspondiente a tus fuentes [17]. Véase dh_make(8) para una descripción más detallada.

Saldrá alguna información. Te preguntará qué tipo de paquete deseas construir. «gentoo» es un paquete de binario simple (crea sólo un binario) y, por tanto, sólo un fichero `.deb` - así que seleccionaremos la primera opción, con la tecla `s`. Comprueba la información que aparece en la pantalla y confirma pulsando la tecla *`ENTER`* [18].

Tras ejecutar **dh_make**, se genera una copia del código original con el nombre `gentoo_0.9.12.orig.tar.gz` en el directorio raíz para facilitar la construcción del paquete de fuentes no nativo de Debian con el archivo `debian.tar.gz`:

```
$ cd ~/gentoo ; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo_0.9.12.orig.tar.gz
```

Observa que hay dos cambios clave en el nombre del fichero `gentoo_0.9.12.orig.tar.gz`:

- El nombre del paquete y la versión están separados por «_».
- Hay un `.orig` antes de `.tar.gz`.

Observa que la ejecución de la orden ha generado varios archivos de plantilla en el directorio `debian`. Se tratará sobre ellos en Capítulo 4 y Capítulo 5. El proceso de empaquetado no está totalmente automatizado. Se tratará la modificación de los archivos Debian en Capítulo 3. A continuación se compilará el paquete Debian en el apartado Capítulo 6, la revisión del resultado en Capítulo 7 y el envío del paquete en Capítulo 9. Se explicará cada una de estas etapas a continuación.

[16] El texto mostrado a continuación da por supuesto que estás ejecutando «bash» como tu intérprete de línea de órdenes. Si utilizas otros intérpretes, como «Z shell», deberás utilizar sus correspondientes ficheros de configuración en lugar de `~/.bashrc`.

[17] Si el archivo del código fuente original ya contiene un directorio `debian` con su contenido, ejecuta la orden **dh_make** con la opción `--addmissing`. El nuevo formato `3.0 (quilt)` es lo bastante robusto para no romper ni esos paquetes. Así se actualizará el contenido aportado por el autor para tu paquete Debian.

[18] Se ofrecen varias opciones aquí: `s` para un binario,`i` para un paquete independiente de la arquitectura (sólo código fuente o bien documentación), `m` para más de un binario, `l` para una biblioteca, `k` para un módulo del núcleo («kernel»), `n` para un parche del núcleo y `b` para paquetes `cdbs`. Este documento se centra en el uso del paquete `debhelper` con la orden **dh** para la construcción de paquetes con un binario y trata solo parcialmente su uso en la construcción de paquetes independientes de la arquitectura y con más de un binario. El paquete `cdbs` ofrece guiones alternativos a la orden **dh** y su uso queda fuera de este documento.

Si accidentalmente has eliminado alguna de las plantillas mientras trabajabas en ellas, puedes regenerarlas ejecutando **dh_make** con la opción `--addmissing` desde el directorio con las fuentes del paquete Debian.

Actualizar un paquete existente puede complicarse debido a que puede estar construido con técnicas más antiguas. Para aprender lo básico, es mejor limitarse a la construcción de un paquete nuevo; se amplia la explicación en Capítulo 8.

Ten en cuenta que el archivo de las fuentes puede que no contenga ningún sistema de construcción discutido en Sección 2.4 y Sección 2.5. Podría ser simplemente una colección datos, etc. La instalación de archivos puede realizarse simplemente con los archivos de configuración de `debhelper` tales como `debian/install` (consulta Sección 5.11).

Capítulo 3

Modificar las fuentes

Ten en cuenta que no hay espacio aquí para entrar en *todos* los detalles respecto a los cambios que deben hacerse en las fuentes originales. Sin embargo, a continuación se detallan algunos de los problemas más frecuentes.

3.1. Configurar quilt

El programa **quilt** ofrece un método básico para realizar y guardar las modificaciones del código fuente para construir paquetes Debian. Para empaquetar, es preferible realizar algunos cambios en la configuración predeterminada del programa, vamos a generar un alias **dquilt** para la generación de paquetes Debian añadiendo la siguiente linea en el archivo `~/.bashrc`. La segunda línea garantiza que el alias tendrá los mismos beneficios que el programa «quilt» con la función de auto-completar del «shell»:

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion -o filenames dquilt
```

A continuación, vamos a generar el archivo `~/.quiltrc-dpkg` de la siguiente manera:

```
d=. ; while [ ! -d $d/debian -a `readlink -e $d` != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # if in Debian packaging tree with unset $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35: ↩
        diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

Véase quilt(1) y `/usr/share/doc/quilt/quilt.pdf.gz` para utilizar **quilt**.

3.2. Corregir un error en el código fuente

Vamos a suponer que has encontrado el siguiente error en el archivo `Makefile` original: donde pone «`install: gentoo`» debería poner «`install: gentoo-target`».

```
install: gentoo
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Vamos a arreglar este error con la orden **dquilt** y conservar las modificaciones a realizar en el archivo `fix-gentoo-target.patch` [1]:

```
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

Ahora cambia el archivo `Makefile` (con un editor) original como se muestra a continuación:

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

A continuación ejecuta **dquilt** para que actualice el parche generando el archivo `debian/patches/fix-gentoo-target.patch` y añade la descripción del parche como se describe en DEP-3: Patch Tagging Guidelines (http://dep.debian.net/deps/dep3/) :

```
$ dquilt refresh
$ dquilt header -e
... describe el parche
```

3.3. Instalación de los archivos en su destino

Por lo general, los programas se instalan a sí mismos en el subdirectorio `/usr/local`. Pero los paquetes Debian no pueden utilizar este directorio ya que está reservado para el uso privado del administrador (o de los usuarios), sino que deben utilizar los directorios del sistema como `/usr/bin` según lo establecido por la normativa de jerarquía del sistema de archivos («Filesystem Hierarchy Standard» FHS (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html)).

Es frecuente la utilización de make(1) para la construcción automatizada del programa y la ejecución de la orden `make install` instala directamente el programa en la ubicación deseada ejecutando la sección `install` del archivo `Makefile`. En la construcción de los paquetes binarios de Debian, el sistema de construcción instala los programas en una reconstrucción de la estructura de directorios del programa en un directorio temporal en lugar de hacerlo en su destino real.

Estas dos diferencias entre la instalación del programa y el proceso de construcción del paquete es manejado en Debian de forma transparente por `debhelper` con las órdenes **dh_auto_configure** y **dh_auto_install** siempre que se cumplan los requisitos indicados a continuación:

- El archivo `Makefile` debe seguir las convenciones GNU de forma que admita la variable `$(DESTDIR)` [2].
- El código fuente sigue el estándar de la jerarquía del sistema de ficheros. («Filesystem Hierarchy Standard» o FHS).

Los programas que usan **autoconf** de GNU cumplen *automáticamente* con las convenciones GNU y su empaquetamiento es casi *automático*. Con estos requisitos y la heurística que aplica el paquete `debhelper`, se estima que funciona sobre el 90 % de paquetes sin tener que realizar ningún cambio intrusivo en su sistema de construcción. El empaquetado no es tan complicado como puede parecer.

Si debes hacer cambios en el archivo `Makefile`, debes asegurarte que admite la variable `$(DESTDIR)`. La variable `$(DESTDIR)` no está definida en el archivo y se añadirá en todas las rutas de directorios usadas en la instalación del programa. El guión de empaquetamiento establece el valor de la variable `$(DESTDIR)` al valor del directorio temporal de instalación del programa en el proceso de construcción del paquete.

[1] El directorio `debian/patches` debería haberse creado en la anterior ejecución de **dh_make**. Sólo debes generarlo si no existe o bien si estás actualizando un paquete.

[2] Consulta GNU Coding Standards: 7.2.4 DESTDIR: Support for Staged Installs (http://www.gnu.org/prep/standards/html_node/DESTDIR.html#DESTDIR) .

El directorio temporal usado por la orden **dh_auto_install** es `debian/nombre_de_paquete` para paquetes con un binario [3]. El contenido completo del directorio temporal será instalado en el sistema del usuario cuando se instale el paquete, con la diferencia que con **dpkg** la instalación se realizará a partir del directorio raíz del sistema.

Aunque el programa se instale en el directorio `debian/nombre_del_paquete` será necesario que se comporte correctamente cuando se instale en el directorio raíz, esto es, cuando se instale usando el archivo `.deb`. Deberías evitar que el sistema de compilación incluya cadenas del tipo `/home/mi/deb/nombre_del_paquete-versión/usr/share/nombre_del_paquete` en los archivos del paquete.

Esta es la parte importante del archivo `Makefile` de `gentoo` [4]:

```
# ¿Dónde se colocaran los ejecutables con 'make install'?
BIN     = /usr/local/bin
# ¿Dónde se colocaran los iconos con 'make install'?
ICONS   = /usr/local/share/gentoo
```

Vemos que los ficheros están configurados para instalarse bajo `/usr/local`. Como se explicó anteriormente, esta rama de directorios esta reservada para uso privado en Debian, cambia estas rutas a:

```
# ¿Dónde se colocaran los ejecutables con 'make install'?
BIN     = $(DESTDIR)/usr/bin
# ¿Dónde se colocaran los iconos con 'make install'?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

La ubicación correcta de los binarios, iconos, documentación, etc, está especificada en el «Estándar de la jerarquía del sistema de ficheros». Te recomiendo que leas las secciones que podrían aplicarse a tu paquete.

Así pues, deberíamos instalar el binario en `/usr/bin` en lugar de `/usr/local/bin` y la página de manual en `/usr/share/man/man1` en lugar de `/usr/local/man/man1`. No hemos mencionado ninguna página de manual en el `Makefile` de `gentoo`, pero la normativa de Debian requiere que cada programa debe tener una, así que haremos una más tarde y la instalaremos en `/usr/share/man/man1`.

Algunos programas no usan variables en el `makefile` para definir rutas como éstas. Esto significa que tendrás que editar algunos de los ficheros de código C para arreglarlos y que usen las rutas correctas. Pero, ¿dónde buscar?, y exactamente, ¿el qué? Puedes probar a encontrarlos usando:

```
$ grep -nr --include='*.[c|h]' -e 'usr/local/lib' .
```

grep buscará recursivamente en los subdirectorios y te indicará el nombre del fichero y la línea cuando encuentre una concordancia con la cadena.

Ahora edita esos ficheros y cambia en esas líneas `usr/local/lib` por `usr/lib`. Esto se puede hacer de forma automática como sigue:

```
$ sed -i -e 's#usr/local/lib#usr/lib#g' \
       $(find . -type f -name '*.[c|h]')
```

Si deseas confirmar cada sustitución, puedes hacerlo de forma interactiva de la siguiente manera:

```
$ vim '+argdo %s#usr/local/lib#usr/lib#gce|update' +q \
       $(find . -type f -name '*.[c|h]')
```

Después de esto deberías encontrar (en el archivo `Makefile`) el objetivo «install» (busca una línea que comience por `install:`) y renombra todas las referencias a directorios distintos de los definidos al comienzo del `Makefile`.

En el original, el objetivo «install» de `gentoo` declaraba:

[3] Para paquetes con más de un archivo binario, la orden **dh_auto_install** utiliza como directorio temporal `debian/tmp` mientras que la orden **dh_install** con la ayuda de los archivos `debian/paquete-1.install` y `debian/paquete-2.install` distribuirá el contenido del directorio `debian/tmp` en los directorios temporales `debian/paquete-1` y `debian/paquete-2` para construir los paquetes binarios `package-1_*.deb` y `package-2_*.deb`.

[4] Se trata solo de un ejemplo del contenido del archivo `Makefile`. Si el archivo `Makefile` se construye con la orden **./configure**, la forma correcta de solucionar este tipo de errores en los archivos `Makefile` es ejecutando la orden **./configure** desde la orden **dh_auto_configure** con las opciones predeterminadas añadiendo `--prefix=/usr`.

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Ahora corregiremos el error, conservando las modificaciones en el archivo `debian/patches/install.patch` con la orden **dquilt**.

```
$ dquilt new install.patch
$ dquilt add Makefile
```

Y ahora escribe los cambios con el editor:

```
install: gentoo-target
        install -d $(BIN) $(ICONS) $(DESTDIR)/etc
        install ./gentoo $(BIN)
        install -m644 icons/* $(ICONS)
        install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

Seguramente has notado que ahora hay una orden `install -d` antes de las demás órdenes de la regla. El `Makefile` original no la tenía porque normalmente `/usr/local/bin` y otros directorios ya existen en el sistema donde se ejecuta `make install`. Sin embargo, dado que lo instalaremos en un directorio vacío (o incluso inexistente), tendremos que generar cada uno de estos directorios.

También podemos añadir otras cosas al final de la regla, como la instalación de documentación adicional que los desarrolladores originales a veces omiten:

```
       install -d $(DESTDIR)/usr/share/doc/gentoo/html
       cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Después de comprobar que sea todo correcto, ejecuta **dquilt** para actualizar la modificación en el archivo `debian/patches/install.patch` y añade la descripción en la cabecera del archivo:

```
$ dquilt refresh
$ dquilt header -e
... describe el parche
```

Ahora ya tienes un par de parches en el paquete.

1. Corrección de un error en el código fuente: `debian/patches/fix-gentoo-target.patch`
2. Una modificación específica del empaquetado Debian: `debian/patches/install.patch`

Siempre que hagas cambios que no estén específicamente relacionados con el paquete Debian, tales como `debian/patches/fix-gentoo-target.patch`, asegúrate de que los envías al desarrollador original para que éste los pueda incluir en la próxima revisión del programa y así le puedan ser útiles a alguien más. Además, recuerda hacer que tus cambios no sean específicos para Debian o GNU/Linux (¡ni siquiera para Unix!) antes de enviarlos, hazlo portable. Esto hará que tus arreglos sean más fáciles de aplicar.

Ten en cuenta que no tienes que enviar ninguno de los ficheros `debian/*` al desarrollador original.

3.4. Diferencias en las bibliotecas

Hay otro problema común: las bibliotecas son generalmente diferentes entre plataformas. Por ejemplo, un `Makefile` puede contener una referencia a una biblioteca que no exista en Debian o ni siquiera en GNU/Linux. En este caso, es necesario cambiarla por una biblioteca que sí exista en Debian y sirva para el mismo propósito.

Vamos a suponer que en el archivo `Makefile` (o `Makefile.in`) de tu programa se declara algo como lo siguiente.

```
LIBS = -lfoo -lbar
```

Si el programa no se compila debido a que ya no existe la biblioteca `nombre_biblioteca` y su equivalente es proporcionada por la biblioteca `nombre_biblioteca2` en el sistema Debian, puedes solucionar este problema de compilación, cambiando (en el archivo `debian/patches/parche2.patch`) `nombre_biblioteca` por `nombre_biblioteca2` [5]:

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e 's/-lfoo/-lfoo2/g' Makefile
$ dquilt refresh
$ dquilt header -e
... describe el cambio
```

[5] Si hay cambios en el API de la biblioteca `nombre_biblioteca` a `nombre_biblioteca2`, se deberá cambiar el código fuente para que se corresponda con la nueva API.

Capítulo 4

Archivos necesarios en el directorio `debian`

Ahora hay un nuevo subdirectorio bajo el directorio principal del programa («gentoo-0.9.12»), que se llama `debian`. Hay algunos ficheros en este directorio que debemos editar para adaptar el comportamiento del paquete. La parte más importante es modificar los ficheros `control`, `changelog`, `copyright` y `rules` que son necesarios en todos los paquetes. [1]

4.1. El archivo `control`

Este fichero contiene varios valores que **dpkg**, **dselect**, **apt-get**, **apt-cache**, **aptitude** y otras herramientas de gestión de paquetes usarán para gestionar el paquete. Su contenido está concretado en Debian Policy Manual, 5 'Control files and their fields' (http://www.debian.org/doc/debian-policy/ch-controlfields.html) .

Aquí está el fichero de `control` que **dh_make** construye para nosotros:

```
 1 Source: gentoo
 2 Section: unknown
 3 Priority: extra
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=9)
 6 Standards-Version: 3.9.4
 7 Homepage: <introduce aquí la URL del autor original  >
 8
 9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: <insertar hasta 60 caracteres de descripción>
13  <inserta una descripción larga, indentada con espacios.>
```

(He añadido los números de línea)

Las líneas 1 a 7 son la información de control para el paquete fuente. Las líneas 9 a 13 son la información de control para el paquete binario.

La línea 1 es el nombre del paquete fuente.

La línea 2 es la sección de la distribución dentro de la que estará este paquete.

Como puede que hayas notado, Debian está dividida en secciones: `main` (principal, los programas libres o de código abierto), `non-free` (no libre, los programas que no son libres, que son de propietario) y `contrib` (programas libres que dependen de programas no libre o de propietario). Bajo ellas hay subdivisiones lógicas que describen en una palabra qué paquetes hay dentro. Así que tenemos `admin` para programas que sólo usa un administrador, `base` para las herramientas básicas, `devel` para

[1] En este capitulo, los archivos del directorio `debian` se nombran sin el antecedente `debian/` para simplificar y siempre que no haya posibilidad de equívocos.

las herramientas de programación, `doc` para la documentación, `libs` para las bibliotecas, `mail` para lectores y demonios de correo-e, `net` para aplicaciones y demonios de red, `x11` para programas específicos de X11, y muchos más. [2]

Vamos a cambiarla para que ponga «x11». El prefijo `main/` ya va implícito, así que podemos omitirlo.

La línea 3 describe cómo de importante es para el usuario la instalación de este paquete [3].

- La prioridad `optional` se utiliza para paquetes nuevos que no entran en conflicto con otros de prioridad `required`, `important` o `standard`.
- La prioridad `extra` se utiliza para nuevos paquetes que entran en conflicto con otros que no tienen la prioridad `extra`.

«Section» y «Priority» se usan en las interfaces como **aptitude** cuando ordenan los paquetes y seleccionan los predeterminados. Una vez que envíes el paquete a Debian, el valor de estos dos campos puede no ser aceptado por los responsables del archivo, en cuyo caso te lo notificarán por correo electrónico.

Como es un paquete de prioridad normal y no tiene conflictos con ningún otro, lo dejaremos con prioridad `optional` (opcional).

La línea 4 es el nombre y correo electrónico del desarrollador. Asegúrate de que este campo incluye una cabecera válida `To` («A:»), para una dirección de correo electrónico, porque después de que envíes el paquete, el sistema de seguimiento de errores («Bug Tracking System») utilizará esta dirección para enviarte los mensajes de los errores. Evita usar comas, el signo «&» y paréntesis.

La línea 5 incluye la lista de paquetes requeridos para construir tu paquete (en el campo `Build-Depends`). Puedes tener una línea adicional con el campo `Build-Depends-Indep`[4]. Algunos paquetes como `gcc` y `make` están implícitos, consulta el paquete `build-essential` para más detalles. Si se necesita algún compilador no estándar u otra herramienta para construir tu paquete, deberías añadirla en la línea «Build-Depends». Las entradas múltiples se separan con comas, lee la explicación de las dependencias binarias para averiguar más sobre la sintaxis de este campo.

- Para todos los paquetes construidos con la orden **dh** en el archivo `debian/rules`, estará `debhelper (>=9)` en el campo `Build-Depends` para ajustarse a las normas de Debian respecto al objetivo `clean`.
- Los paquetes de fuentes que incluyen varios paquetes binarios con el campo `Architecture: any` se construyen con «autobuilder». Desde el procedimiento «autobuilder» se ejecuta el objetivo `debian/rules build` el cual, a su vez, instala los paquetes listados en el campo `Build-Depends` (véase Sección 6.2), que habitualmente son todos los necesarios de forma que el campo `Build-Depends-indep` se usa raramente.
- En el caso de los paquetes de fuentes que incluyen paquetes binarios únicamente del tipo `Architecture: all`, el campo `Build-Depends-Indep` debe listar todos los paquetes excepto los listados en el campo `Build-Depends` para satisfacer los requerimientos de las normas de Debian respecto al objetivo `clean`.

En caso de duda, utiliza el campo `Build-Depends` [5].

Para saber qué paquetes son necesarios para compilar el tuyo ejecuta esta orden:

```
$ dpkg-depcheck -d ./configure
```

Para buscar manualmente las dependencias de compilación para el paquete ***/usr/bin/nombre_del_paquete***, deberías ejecutar:

```
$ objdump -p /usr/bin/nombre_del_paquete | grep NEEDED
```

y para cada biblioteca listada por la orden anterior (en el ejemplo se hace para **libfoo.so.6**) ejecuta

[2] Consulta normas de Debian, 2.4 'Secciones' (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) y Lista de secciones en `sid` (http://packages.debian.org/unstable/) .

[3] Véase Debian Policy Manual, 2.5 'Priorities' (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) .

[4] Consulta Debian Policy Manual, 7.7 'Relationships between source and binary packages - Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep' (http://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps) .

[5] Este caso poco frecuente, está documentado en Debian Policy Manual, Footnotes 55 (http://www.debian.org/doc/debian-policy/footnotes.html#f55) . Esto se debe al funcionamiento de **dpkg-buildpackage**, no al uso de la orden **dh** en el archivo `debian/rules`. Esto también se aplica al «auto build system for Ubuntu» (https://bugs.launchpad.net/launchpad-buildd/+bug/238141) .

```
$ dpkg -S libfoo.so.6
```

Debes utilizar la versión `-dev` de cada uno de los paquetes dentro de la entrada `Build-Depends`. Si usas **ldd** para este propósito, también te informará de las dependencias de bibliotecas indirectas, lo que puede llevar a que se introduzcan demasiadas dependencias de construcción.

`gentoo` también depende de `xlibs-dev`, `libgtk1.2-dev` y `libglib1.2-dev` para su construcción, así que lo añadiremos junto a `debhelper`.

La línea 6 es la versión de los estándares definidos en las normas de Debian que sigue este paquete, es decir, la versión del manual de normas que has leído mientras haces tu paquete (véase «Debian Policy Manual» (http://www.debian.org/doc/devel-manuals#policy)).

En la línea 7 está la dirección URL de la página web del programa (donde has obtenido las fuentes).

La línea 9 es el nombre del paquete binario. Este suele ser el mismo que el del paquete fuente, aunque no es necesario que sea así siempre.

La linea 10 describe las arquitecturas en las que puede compilarse el paquete binario. Este valor suele ser uno de los listados a continuación dependiendo del tipo de paquete [6]:

- `Architecture: any`
  - El paquete binario generado depende de la arquitectura si consiste en un programa escrito en un lenguaje compilado.
- `Architecture: all`
  - El paquete binario generado es independiente de la arquitectura si consiste en archivos de texto, imágenes o guiones escritos en lenguajes interpretados.

Eliminamos la línea 10 debido a que el programa está escrito en C. dpkg-gencontrol(1) la rellenará con el valor apropiado cuando se compile este paquete en cualquier arquitectura para la cual pueda ser compilado.

Si tu paquete es independiente de la arquitectura (por ejemplo, un documento, un guión escrito en Perl o para el intérprete de órdenes), cambia esto a `all`, y consulta más adelante Sección 4.4 sobre cómo usar la regla `binary-indep` en lugar de `binary-arch` para construir el paquete.

La línea 11 muestra una de las más poderosas posibilidades del sistema de paquetes de Debian. Los paquetes se pueden relacionar unos con otros de diversas formas. Aparte de `Depends` (depende de) otros campos de relación son `Recommends` (recomienda), `Suggests` (sugiere), `Pre-Depends` (predepende de), `Breaks` (rompe a), `Conflicts` (entra en conflicto con), `Provides` (provee), `Replaces` (reemplaza a).

Las herramientas de gestión de paquetes se comportan habitualmente de la misma forma cuando tratan con esas relaciones entre paquetes; si no es así, se explicará en cada caso. (véase dpkg(8), dselect(8), apt(8), aptitude(1), etc.)

He aquí una descripción simplificada de relaciones entre paquetes: [7]

- `Depends`:

  No se instalará el programa a menos que los paquetes de los que depende estén ya instalados. Usa esto si tu programa no funcionará de ninguna forma (o se romperá fácilmente) a no ser que se haya instalado un paquete determinado.

- `Recommends`:

  Esta opción es para paquetes cuya instalación no es estrictamente necesaria para el funcionamiento de tu programa pero que suelen utilizarse junto con tu programa. Cuando los usuarios instalen tu paquete, todas las interfaces de instalación aconsejaran la instalación de los paquetes recomendados. **aptitude** y **apt-get** instalan los paquetes recomendados (pero el usuario puede decidir no hacerlo). **dpkg** ignora el contenido de este campo.

[6] Véase Debian Policy Manual 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) para más detalles.

[7] Consulta Debian Policy Manual, 7 ”Declaring relationships between packages” (http://www.debian.org/doc/debian-policy/ch-relationships.html) .

- `Suggests`:

  Utiliza esto para paquetes que funcionarán bien con tu programa pero que no son necesarios en absoluto. Es posible configurar **aptitude** para que instale los paquetes sugeridos, aunque no es la opción predeterminada. **dpkg** y **apt-get** ignorarán estas dependencias.

- `Pre-Depends`:

  Esto es más fuerte que `Depends`. El paquete no se instalará a menos que los paquetes de los que pre-dependa estén instalados y *correctamente configurados*. Utiliza esto *muy* poco y sólo después de haberlo discutido en la lista de distribución (debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)). En resumidas cuentas: no lo utilices en absoluto :-)

- `Conflicts`:

  El paquete no se instalará hasta que todos los paquetes con los que entra en conflicto hayan sido eliminados. Utiliza esto si tu programa no funcionará en absoluto (o fallará fácilmente) si un paquete en concreto está instalado.

- `Breaks`:

  Si el paquete se instala, todos los paquetes de la lista se romperán. Normalmente, los paquetes incluidos en la lista `Breaks` tienen una cláusula de versión anterior. La solución es actualizar los paquetes de la lista a la versión más actual.

- `Provides`:

  Se han definido nombres virtuales para algunos tipos determinados de paquetes que ofrecen múltiples alternativas para la misma función. Puedes obtener la lista completa en el archivo virtual-package-names-list.txt.gz (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) . Usa esto si tu programa ofrece las funciones de un paquete virtual que ya exista.

- `Replaces`:

  Usa esto si tu programa reemplaza ficheros de otro paquete o reemplaza totalmente otro paquete (generalmente se usa conjuntamente con `Conflicts`). Se sobrescribirán los ficheros de los paquetes indicados con los ficheros de tu paquete.

Todos estos campos tienen una sintaxis uniforme: se trata de una lista de nombres de paquetes separados por comas. Estos nombres de paquetes también puede ser listas de paquetes alternativos, separados por los símbolos de barra vertical « | » (símbolos tubería).

Los campos pueden restringir su aplicación a versiones determinadas de cada paquete nombrado. Esto se hace listando después de cada nombre de paquete individual las versiones entre paréntesis, e indicando antes del número de versión una relación de la siguiente lista. Las relaciones permitidas son: <<, <=, =, >= y >> para estrictamente anterior, anterior o igual, exactamente igual, posterior o igual o estrictamente posterior, respectivamente. Por ejemplo:

```
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (<< 5), quux-foo (<= 7.6)
```

La última funcionalidad que necesitas conocer es `${shlibs:Depends}`,`${perl:Depends}`, `${misc:Depends}`, etc.

Después de que tu paquete se compile y se instale en el directorio temporal, dh_shlibdeps(1) lo escaneará en busca de binarios y bibliotecas para determinar las dependencias de bibliotecas compartidas. Esta lista se utiliza para la sustitución de `${shlibs:Depends}`.

La orden dh_perl(1) genera las dependencias de Perl. Genera la lista de dependencias de `perl` o `perlapi` para cada paquete binario. Esta lista es utilizada para substituir a `${perl:Depends}`.

Algunas órdenes de `debhelper` determinan las dependencias de los paquetes listados anteriormente. Cada orden generar una lista de los paquetes necesarios para cada paquete binario. La lista de estos paquetes se usará para substituir a `${misc:Depends}`.

La orden dh_gencontrol(1) genera el archivo `DEBIAN/control` para cada paquete binario substituyendo `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}`, etc.

Después de decir todo esto, podemos dejar la línea de `Depends` exactamente como está ahora e insertar otra línea tras ésta con el texto `Suggests: file`, porque `gentoo` utiliza algunas funciones proporcionadas por el paquete/programa `file`.

La línea 9 es la dirección URL del programa. Supongamos que es http://www.obsession.se/gentoo/.

La línea 12 es una descripción corta. La mayor parte de los monitores (en realidad, de las terminales en modo de texto) de la gente son de 80 columnas de ancho, así que no debería tener más de 60 caracteres. Cambiaré esto a `fully GUI-configurable, two-pane X file manager`. («Gestor de ficheros GTK+ completamente configurable por GUI»).

La línea 13 es donde va la descripción larga del paquete. Debería ser al menos de un párrafo que dé más detalles del paquete. La primera columna de cada línea debería estar vacía. No puede haber líneas en blanco, pero puedes poner un `.` (punto) en una columna para simularlo. Tampoco debe haber más de una línea en blanco después de la descripción completa [8].

Vamos a añadir los campos `Vcs-*` para documentar la localización del sistema de control de versiones (VCS) entre las lineas 6 y 7 [9]. Se supone que el paquete `gentoo` está alojado en el servicio «Debian Alioth Git» en `git://git.debian.org/git/collab-maint/gentoo.git`.

Aquí está el archivo `control` actualizado:

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=9), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 3.9.4
 7 Vcs-Git: git://git.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: http://git.debian.org/?p=collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
18  program itself. If you still prefer to hand-edit configuration files,
19  they’re fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(He añadido los números de línea)

4.2. El archivo `copyright`

Este fichero contiene la información sobre los recursos, licencia y derechos de autoría de las fuentes originales del paquete. El formato no está definido en las normas, pero sí sus contenidos en «Debian Policy Manual, 12.5 ”Copyright information”» (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) y DEP-5: Machine-parseable `debian/copyright` (http://dep.debian.net/deps/dep5/) proporciona directrices sobre su formato.

dh_make proporciona una plantilla para el archivo `copyright`. Con la opción `--copyright gpl2` se consigue la plantilla para el paquete `gentoo` con la licencia GPL-2.

[8] Las descripciones deben redactarse en inglés. Las traducciones de estas descripciones son proporcionados por The Debian Description Translation Project - DDTP (http://www.debian.org/intl/l10n/ddtp) .

[9] Véase Developer’s Reference, 6.2.5. ”Version Control System location” (http://www.debian.org/doc/manuals/developers-reference/best-pkging-practices.html#bpp-vcs) .

Debes completar la información sobre el lugar donde se puede obtener el código fuente, la condiciones de derechos de autor y la licencia. Las licencias de código libre más comunes son GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0 o la «Artistic license» y hacer referencia al archivo correspondiente ubicado en el directorio `/usr/share/common-licenses/`, que existen en todos los sistemas Debian. De lo contrario, debe incluirse el texto de la licencia completo.

En resumen, el archivo `copyright` del paquete `gentoo` debería ser similar a esto:

```
 1 Format-Specification: http://svn.debian.org/wsvn/dep/web/deps/dep5.mdwn?op=file&rev=135
 2 Name: gentoo
 3 Maintainer: Josip Rodin <joy-mg@debian.org>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 7 License: GPL-2+
 8
 9 Files: icons/*
10 Copyright: 1998 Johan Hanson <johan@tiq.com>
11 License: GPL-2+
12
13 Files: debian/*
14 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
15 License: GPL-2+
16
17 License: GPL-2+
18  This program is free software; you can redistribute it and/or modify
19  it under the terms of the GNU General Public License as published by
20  the Free Software Foundation; either version 2 of the License, or
21  (at your option) any later version.
22  .
23  This program is distributed in the hope that it will be useful,
24  but WITHOUT ANY WARRANTY; without even the implied warranty of
25  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
26  GNU General Public License for more details.
27  .
28  You should have received a copy of the GNU General Public License along
29  with this program; if not, write to the Free Software Foundation, Inc.,
30  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
31  .
32  On Debian systems, the full text of the GNU General Public
33  License version 2 can be found in the file
34  '/usr/share/common-licenses/GPL-2'.
```

(He añadido los números de línea)

Por favor, sigue el COMO redactado por «ftpmasters» y enviado a «debian-devel-announce»: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3. El archivo `changelog`

Este es un fichero requerido, con un formato especial descrito en Debian Policy Manual, 4.4 "debian/changelog" (http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog) . Este es el formato utilizado por **dpkg** y otros programas para obtener el número de versión, revisión, distribución y urgencia de tu paquete.

Para ti es también importante, ya que es bueno tener documentados todos los cambios que hayas hecho. Esto ayudará a las personas que se descarguen tu paquete para ver si hay temas pendientes en el paquete que deberían conocer de forma inmediata. Se guardará como `/usr/share/doc/gentoo/changelog.Debian.gz` en el paquete binario.

dh_make genera uno predeterminado, el cual es como sigue:

```
1  gentoo (0.9.12-1) unstable; urgency=low
2
3   * Initial release (Closes: #nnnn)  <nnnn
    is the bug number of your ITP>
4
5  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
6
```

(He añadido los números de línea)

La línea 1 es el nombre del paquete, versión, distribución y urgencia. El nombre debe coincidir con el nombre del paquete fuente, la distribución debería ser, por ahora, `unstable` y la urgencia no debería cambiarse a algo mayor que `low`. :-)

Las líneas 3 a 5 son una entrada de registro, donde se documentan los cambios hechos en esta revisión del paquete (no los cambios en las fuentes originales - hay un fichero especial para este propósito, creado por los autores originales y que instalarás luego como `/usr/share/doc/gentoo/changelog.gz`). En el ejemplo se supone que el código del informe de error ITP («Intent To Package», intento de empaquetar) es `12345`. Las nuevas líneas deben insertarse justo antes de la línea que hay más arriba que comienza por un asterisco (`*`). Puedes hacerlo con dch(1), o manualmente con cualquier editor de texto.

Para evitar que un paquete sea accidentalmente subido al repositorio sin estar completado, es una buena idea cambiar el nombre de la distribución a valor `UNRELEASED` que es incorrecto.

Terminarás con algo así:

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2
3   * Initial Release. Closes: #12345
4   * This is my first Debian package.
5   * Adjusted the Makefile to fix $(DESTDIR) problems.
6
7  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
8
```

(He añadido los números de línea)

Cuando estés satisfecho con los cambios realizados y estén documentados en el fichero `changelog`, entonces cambie el nombre de la distribución de `UNRELEASED` a `unstable` (o bien a `experimental`). [10]

Puedes leer más sobre cómo actualizar el fichero `changelog` más adelante en Capítulo 8 .

4.4. El archivo `rules`

Ahora necesitamos mirar las reglas exactas que dpkg-buildpackage(1) utilizará para construir el paquete. Este fichero es en realidad otro `Makefile`, pero diferente al que viene en las fuentes originales. A diferencia de otros ficheros en `debian`, éste debe ser un fichero ejecutable.

4.4.1. Objetivos del archivo `rules`

Cada archivo `rules`, como cualquier otro archivo `Makefile`, se compone de varias reglas, cada una de ellas define el objetivo y cómo se ejecuta [11]. Cada nueva regla empieza con la declaración del objetivo en la primera columna. Las líneas siguientes empiezan con un código de tabulación (ASCII 9) y especifican cómo llevar a cabo ese objetivo. Las líneas en blanco o que empiezan con # se tratan como comentarios y se ignoran [12].

[10] Si utiliza la orden `dch -r` para realizar este último cambio, asegúrese que guarda el archivo `changelog` explícitamente con el editor.

[11] Puedes empezar a aprender a escribir archivos `Makefile` con Debian Reference, 12.2. ”Make” (http://www.debian.org/doc/manuals/debian-reference/-ch12#_make) . La documentación completa está disponible en http://www.gnu.org/software/make/manual/html_node/index.html y en el paquete `make-doc` de la sección `non-free` del repositorio.

[12] Debian Policy Manual, 4.9 ”Main building script: debian/rules” (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) explica los detalles.

Una regla que se desea ejecutar se invoca por el nombre de objetivo como un argumento de línea de comandos. Por ejemplo, `debian/rules` *`build`* y `fakeroot make -f debian/rules` *`binary`* ejecutan las reglas para los objetivos *`build`* y *`binary`* respectivamente.

A continuación se proporciona una explicación simplificada de los distintos objetivos.

- `clean` (obligatorio): elimina todos los archivos generados, compilados o innecesarios del árbol de directorios de las fuentes.
- `build` (obligatorio): para la construcción de archivos compilados a partir de los archivos fuente o la construcción de documentos formateados.
- objetivo `build-arch` (obligatorio): para la compilación de las fuentes en programas compilados (dependientes de la arquitectura) en el árbol de directorios de compilación.
- objetivo `build-indep` (obligatorio): para la compilación de las fuentes en documentos formateados (independientes de la arquitectura) en el árbol de directorios de compilación.
- `install` (opcional): para la instalación en la estructura de directorios temporal bajo el directorio `debian` de los archivos para cada uno de los paquetes binarios. Si existe el objetivo `binary*`, dependerá de este.
- `binary` (obligatorio): para la construcción de cada uno de los paquetes binarios (combinado con los objetivos `binary-arch` y `binary-indep`) [13].
- `binary-arch` (obligatorio): para la construcción de paquetes dependientes de la arquitectura (`Architecture: any`) [14].
- `binary-indep` (obligatorio): para la construcción de paquetes independientes de la arquitectura (`Architecture: all`) [15].
- `get-orig-source` (opcional): para obtener la versión más reciente de las fuentes originales desde el lugar de almacenaje del autor.

Probablemente ya te hayas perdido, pero todo quedará más claro después de ver el fichero `rules` que **dh_make** pone por omisión.

4.4.2. Archivo `rules` predeterminado

La nueva versión de **dh_make** genera un archivo `rules` muy simple pero poderoso utilizando la orden **dh**:

```
 1 #!/usr/bin/make -f
 2 # See debhelper(7) (uncomment to enable)
 3 # output every command that modifies files on the build system.
 4 #DH_VERBOSE = 1
 5
 6 # see EXAMPLES in dpkg-buildflags(1) and read /usr/share/dpkg/*
 7 DPKG_EXPORT_BUILDFLAGS = 1
 8 include /usr/share/dpkg/default.mk
 9
10 # see FEATURE AREAS in dpkg-buildflags(1)
11 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
12
13 # see ENVIRONMENT in dpkg-buildflags(1)
14 # package maintainers to append CFLAGS
15 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
16 # package maintainers to append LDFLAGS
17 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
18
19 # main packaging script based on dh7 syntax
20 %:
21         dh $@
```

[13] Este objetivo es utilizado por `dpkg-buildpackage` como en Sección 6.1.
[14] Este objetivo es utilizado por `dpkg-buildpackage -B` como en Sección 6.2.
[15] Este objetivo es utilizado por `dpkg-buildpackage -A`.

(He añadido los números de línea y añadido algunos comentarios. En el fichero `debian/rules` los espacios iniciales de las líneas son tabulaciones)

Probablemente estés familiarizado con líneas como la primera de guiones escritos en shell o Perl. Esta línea indica que el fichero debe ejecutarse con`/usr/bin/make`.

La linea 4 debe descomentarse para asignar el valor 1 a la variable `DH_VERBOSE`. Entonces, la orden **dh** mostrará (en el terminal) las órdenes **dh_*** ejecutadas por **dh**. Puedes añadir la linea `export DH_OPTIONS=-v` aquí. Así podrás ver la salida de la ejecución de cada orden **dh_*** y solucionar los problemas que se produzcan. Esto te ayudará a entender como funciona el archivo `rules` y a solucionar problemas. Esta nueva orden **dh** es parte fundamental de las herramientas `debhelper` y no te esconde nada.

Todo el trabajo del archivo se reduce a las líneas 20 y 21. El símbolo de porcentaje substituye a cualquier objetivo para a continuación ejecutar únicamente **dh** con el nombre del objetivo (como opción) [16]. La orden **dh** es un guión que ejecuta las secuencias necesarias de órdenes **dh_*** según sus parámetros, como se describe a continuación [17]:

- `debian/rules clean` ejecuta `dh clean`, que a su vez ejecuta lo siguiente:

```
dh_testdir
dh_auto_clean
dh_clean
```

- `debian/rules build` ejecuta `dh build`, que a su vez ejecuta lo siguiente:

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- `fakeroot debian/rules binary` ejecuta `fakeroot dh binary`, que a su vez ejecuta lo siguiente [18]:

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
dh_installchangelogs
dh_installexamples
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
dh_installmodules
dh_installlogcheck
dh_installlogrotate
```

[16] Aquí se utilizan las nuevas funciones de la versión 7+ de `debhelper` que se explican en «Not Your Grandpa's Debhelper» (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) presentadas en la Debconf9 por el autor de `debhelper`. En `lenny`, **dh_make** construía un archivo `rules` más complejo con el listado de todas las órdenes **dh_*** necesarias para cada objetivo y los mantenía en el estado del empaquetado inicial. La nueva orden **dh** es simple y nos libera de esta restricción. Aún así, es posible personalizar el archivo con objetivos `override_dh_*`. Véase Sección 4.4.3. Se basa únicamente en el paquete `debhelper` y no complica el proceso de construcción de paquetes como el paquete `cdbs`.

[17] Puedes comprobar las secuencias de órdenes **dh_*** invocadas por cada *`objetivo`* ejecutando `dh --no-act` *`objetivo`* o bien `debian/rules -- '--no-act` *`objetivo`*`'` sin que se ejecuten realmente.

[18] El siguiente ejemplo presupone que su fichero `debian/compat` tiene un valor igual o superior a 9 para evitar invocar cualquier orden «python» automáticamente.

```
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- `fakeroot debian/rules binary-arch` ejecuta `fakeroot dh binary-arch`; que a su vez ejecuta la misma secuencia que `fakeroot dh binary` pero con la opción `-a` para cada orden.
- `fakeroot debian/rules binary-indep` ejecuta `fakeroot dh binary-indep`; que a su vez ejecuta casi la misma secuencia que `fakeroot dh binary` puesto que excluye la ejecución de **dh_strip**, **dh_makeshlibs** y **dh_shlibdeps** a la vez que ejecuta el resto de órdenes añadiendo la opción `-i`.

La función de las órdenes **dh_*** puede deducirse de su nombre [19]. A continuación se resume las funciones de las órdenes más importantes asumiendo que se utiliza un sistema de compilación basado en un archivo `Makefile` [20]:

- **dh_auto_clean** normalmente ejecuta lo siguiente, siempre que exista un fichero `Makefile` con el objetivo `distclean`[21].

```
make distclean
```

- **dh_auto_configure** ejecuta lo siguiente si existe el archivo `./configure` (se han abreviado los argumentos para facilitar la lectura).

```
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
```

- **dh_auto_build** ejecuta lo siguiente para ejecutar el primer objetivo del archivo `Makefile` (supuesto que este existe).

```
make
```

- **dh_auto_test** ejecuta lo siguiente si existe el objetivo `test` en el archivo `Makefile` [22].

```
make test
```

- **dh_auto_install** ejecuta lo siguiente si en el archivo `Makefile` existe el objetivo `install` (se ha truncado la linea para permitir su lectura).

```
make install \
  DESTDIR=/ruta/a/paquete_versión-revisión/debian/paquete
```

[19] Para una descripción completa de la función de cada guión **dh_*** y de sus opciones, lee los manuales respectivos así como la documentación de `debhelper`.

[20] La orden permite otros sistemas de compilación como `setup.py` y que pueden ser listados con la ejecución de `dh_auto_build --list` desde el directorio de las fuentes del paquete.

[21] En realidad busca el primer objetivo `distclean`, `realclean` o `clean` disponible en el `Makefile` y lo ejecuta.

[22] En realidad busca el primero de los objetivos `test` o `check` en el archivo `Makefile` y lo ejecuta.

Los objetivos que deben ejecutarse con la orden **fakeroot** contienen **dh_testroot**. Si no utilizas la orden para simular la ejecución por el usuario «root», se producirá un error que detendrá la ejecución.

Es importante tener presente que el archivo `rules` que genera **dh_make** es sólo una sugerencia. Será suficiente para la mayoría de los paquetes simples, pero no dejes de adaptarlo a tus necesidades en paquetes más complejos.

A pesar de que `install` no es un objetivo obligatorio, se admite su uso. `fakeroot dh install` se comporta como `fakeroot dh binary` pero se detiene después de **dh_fixperms**.

4.4.3. Personalización del archivo `rules`

Puedes realizar muchos cambios para adaptar el archivo `rules` construido por la orden **dh**.

La orden `dh $@` permite las siguientes adaptaciones [23]:

- Añadir la orden **dh_python2** (la mejor opción para Python) [24].
  - Añade el paquete `python` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with python2` en su lugar.
  - Esto gestiona el módulo Python utilizando las funcionalidades de `python`.
- Añadir la orden **pysupport**. (obsoleto)
  - Añade el paquete `python-support` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with pysupport`.
  - Esto gestiona el módulo Python utilizando las funcionalidades de `python-support`.
- Añadir la orden **dh_pycentral**. (obsoleto)
  - Añade el paquete `python-central` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with python-central` en su lugar.
  - Esto desactiva la orden **dh_pysupport**.
  - Esto gestiona el módulo Python utilizando las funcionalidades de `python-central`.
- Añadir la orden **dh_installtex**.
  - Añade el paquete `tex-common` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with tex` en su lugar.
  - Esto registra el tipo de letra «Type 1», los patrones de separación de palabras («hyphenation patterns») o los formatos TeX.
- Añadir las órdenes **dh_quilt_patch** y **dh_quilt_unpatch**.
  - Añade el paquete `quilt` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with quilt` en su lugar.
  - Esto aplica o revierte los parches en los archivos de las fuentes originales, basándose en los ficheros del directorio `debian/patches` en los paquetes con el formato `1.0`.
  - Esta adaptación no es necesaria para los paquetes con el nuevo formato `3.0 (quilt)`.
- Añadir la orden **dh_dkms**.
  - Añade el paquete `dkms` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with dkms` en su lugar.

[23] Si un paquete instala el archivo `/usr/share/perl5/Debian/Debhelper/Sequence/`*`nombre_personalizado`*`.pm` puedes activar la función adaptada con `dh $@ --with` *`nombre_personalizado`*.

[24] Es preferible el uso de la orden **dh_python2** respecto a la orden **dh_pysupport** u **dh_pycentral**. No uses la orden **dh_python**.

  - Esto controla correctamente el uso de DKMS en la construcción de paquetes del núcleo.

- Añadir las ordenes **dh_autotools-dev_updateconfig** y **dh_autotools-dev_restoreconfig**.
  - Añade el paquete `autotools-dev` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with autotools-dev` en su lugar.
  - Esto actualiza y restaura `config.sub` y `config.guess`.
- Añadir la orden **dh_autoreconf** y **dh_autoreconf_clean**.
  - Añade el paquete `dh-autoreconf` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with autoreconf` en su lugar.
  - Así se actualiza los archivos del sistema de compilación GNU y los restaura después de la compilación.
- Añadir la orden **dh_girepository**.
  - Añade el paquete `gobject-introspection` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with gir` en su lugar.
  - Esto calcula las dependencias de paquetes de envío de datos de introspección de «GObject» y genera la substitución de la variable `${gir:Depends}` por las dependencias del paquete.
- Añadir la funcionalidad de autocompletar a **bash**.
  - Añade el paquete `bash-completion` en el campo `Build-Depends`.
  - Utiliza `dh $@ --with bash-completion` en su lugar.
  - Esto instala la función autocompletar de **bash** utilizando el archivo de configuración de `debian/`*`nombre_del_paquete`*`.bash-c`

Muchas de las órdenes **dh_*** invocadas por la nueva orden **dh** son personalizables mediante sus archivos de configuración en el directorio `debian`. Véase Capítulo 5 y los manuales (las «manpage») para cada orden.

Algunas órdenes **dh_*** invocadas por la nueva orden **dh** pueden precisar la adición de argumentos (opciones), la ejecución de órdenes adicionales u omitirlas del todo. Para estos casos, deberás añadir el objetivo `override_dh_`*`nombre_de_la_orden`* con las reglas a ejecutar en el archivo `rules` sólo para la orden **dh_*`nombre_de_la_orden`*** que vas a cambiar. Se trata de decir *ejecútame a mí en su lugar* [25].

Las ordenes **dh_auto_*** hacen más cosas que las expuestas aquí. El uso de ordenes equivalentes más sencillas en lugar de éstas en los objetivos `override_dh_*` (excepto el objetivo `override_dh_auto_clean`) es una mala idea ya que puede eliminar funciones inteligentes de `debhelper`.

Si vas a guardar los datos de configuración del paquete `gentoo` en el directorio `/etc/gentoo` en lugar del directorio habitual `/etc`, debes anular la ejecución del argumento predeterminado `--sysconfig=/etc` de la orden **dh_auto_configure** por **./configure** con lo siguiente:

```
override_dh_auto_configure:
        dh_auto_configure -- --sysconfig=/etc/gentoo
```

Los argumentos a continuación de `--` se añaden a los argumentos predeterminados, anulándolos, en la ejecución automática del programa. Es mejor utilizar la orden **dh_auto_configure** que el **./configure** ya que así sólo anulará el argumento `--sysconfig` manteniendo intactos otros argumentos de **./configure**.

Si el `Makefile` de las fuentes de `gentoo` requiere la especificación del objetivo `build` para compilarlo [26], puedes añadir un objetivo `override_dh_auto_build` para anularlo.

```
override_dh_auto_build:
        dh_auto_build -- build
```

[25] En `lenny`, cuando querías cambiar el comportamiento de un programa **dh_*** tenías que encontrar la línea adecuada en el archivo `rules` y cambiarla.
[26] **dh_auto_build** sin argumentos ejecutará el primer objetivo del archivo `Makefile`.

De esta forma se garantiza la ejecución de `$(MAKE)` con todos los argumentos predeterminados dados por la orden **dh_auto_build** y del argumento `build`.

Si el `Makefile` de las fuentes de `gentoo` requiere la especificación del objetivo `packageclean` para limpiarlo, en lugar de los objetivos `distclean` o `clean` en el archivo `Makefile`, puedes añadir un objetivo `override_dh_auto_clean` para habilitarlo.

```
override_dh_auto_clean:
        $(MAKE) packageclean
```

Si el `Makefile` de las fuentes de `gentoo` contiene un objetivo `test` que no deseas que se ejecute en la construcción del paquete Debian, puedes usar un objetivo `override_dh_auto_test` sin órdenes para ignorarlo.

```
override_dh_auto_test:
```

Si `gentoo` contiene el poco frecuente archivo de cambios del autor con el nombre `FIXES`, **dh_installchangelogs** no lo instalará por omisión. La orden **dh_installchangelogs** requiere como argumento `FIXES` para instalarlo [27].

```
override_dh_installchangelogs:
        dh_installchangelogs FIXES
```

Cuando utilizas el nuevo programa **dh**, la utilización explícita de objetivos como los listados en Sección 4.4.1 (excepto `get-orig-source`) puede dificultar la correcta comprensión de sus efectos exactos. Por favor, limita el uso de objetivos explícitos a objetivos del tipo `override_dh_*` y de forma que sean completamente independientes entre sí (siempre que sea posible).

[27] Los archivos `debian/changelog` y `debian/NEWS` siempre se instalan automáticamente. También se busca el archivo de cambios del autor para cambiar el nombre a minúsculas y por su coincidencia con `changelog`, `changes`, `changelog.txt`, y `changes.txt`.

Capítulo 5

Otros ficheros en el directorio `debian`.

Para controlar el trabajo de `debhelper` en la compilación del paquete, puedes añadir archivos de configuración en el directorio `debian`. En este capítulo se resumirá lo que puede hacerse con cada uno de ellos y su formato. Por favor, lee «Debian Policy Manual» (http://www.debian.org/doc/devel-manuals#policy) y «Debian Developer's Reference» (http://www.debian.org/doc/-devel-manuals#devref) para más información.

La orden **dh_make** construye varios archivos de configuración a modo de plantillas y los ubica en el directorio `debian`. Algunos de ellos tienen el sufijo `.ex` (de «example») en el nombre. Otros tienen como prefijo del nombre el nombre del paquete en la forma *`nombre_del_paquete`*. Mira el contenido de todos ellos. [1]

En otros casos, **dh_make** no puede construir plantillas de configuración para `debhelper`. En estos casos, deberás construir tu mismo los archivos con un editor.

Si quieres utilizar estos archivos en la construcción de tu paquete, haz lo siguiente.

- elimina los sufijos `.ex` o `.EX` de los archivos de plantilla que lo tengan.
- renombra los archivos de configuración utilizando el nombre del archivo del paquete binario en lugar de *`nombre_del_paquete`*.
- modifica el contenido de los archivos de plantilla para adaptarlos a tus necesidades.
- elimina aquellos archivos que no necesites.
- realiza las modificaciones necesarias en el archivo `control` (véase Sección 4.1), si es necesario.
- modifica el archivo `rules` (véase Sección 4.4), si es necesario.

Los archivos de configuración construidos por `debhelper` que no tienen el prefijo *`nombre_del_paquete`* tales como `install` se aplicaran al primer paquete binario. Si hay varios paquetes binarios, sus configuraciones se especificaran con el prefijo de paquete binario correspondiente en su nombre: así tendrás los archivos *`paquete-1`*`.install`, *`paquete-2`*`.install`, etc.

5.1. Archivo `README.Debian` (LÉEME.debian)

Cualquier detalle extra o discrepancias entre el programa original y su versión debianizada debería documentarse aquí.

dh_make construye uno predeterminado, y éste es su aspecto:

```
gentoo for Debian
-----------------
<possible notes regarding this package - if none, delete this file>
 -- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

Dado que no tenemos que poner nada aquí, elimínalo.Véase dh_installdocs(1).

[1] En este capitulo, los archivos del directorio `debian` se nombran sin el antecedente `debian/` para simplificar y siempre que no haya posibilidad de equívocos.

5.2. Archivo `compat`

El archivo `compat` define el nivel de compatibilidad de `debhelper`. Actualmente, establecerás la compatibilidad a la versión 9 de `debhelper` como se indica a continuación.

```
$ echo 9 > debian/compat
```

5.3. Archivo `conffiles`

Una de las cosas más molestas de los programas es cuando pasas mucho tiempo y esfuerzo adaptando un programa (como usuario) y una actualización destroza todos tus cambios. Debian resuelve este problema marcando los ficheros de configuración. [2] Así, cuando actualizas un paquete se te pregunta si deseas mantener la nueva configuración o no.

Desde la versión 3 de `debhelper`, dh_installdeb(1) considera *automáticamente* a todos los archivos ubicados en el directorio `/etc` como «conffiles» (archivos de configuración gestionados por el sistema de paquetes). Así, si todos los «conffiles» están en este directorio no es necesario que los incluyas en este archivo. Para la mayoría de paquetes, la única ubicación de los «conffiles» es `/etc` por lo que no es necesario generar este archivo.

En el caso de que tu programa utilice ficheros de configuración pero también los reescriba él mismo es mejor no marcarlos como «conffiles». Si lo haces, **dpkg** informará a los usuarios que verifiquen los cambios de estos ficheros cada vez que lo actualicen.

Si el programa que estás empaquetando requiere que cada usuario modifique los archivos de configuración del directorio `/etc`, hay dos formas para no marcarlos como archivos «conffiles» y que no sean manipulados por **dpkg**:

- Construir un enlace simbólico de los archivos ubicados en `/etc` que apunten a archivos ubicados en el directorio `/var` generados por *guiones del desarrollador* («maintainer scripts»).
- Poner los archivos generados por los *guiones del desarrollador* en el directorio `/etc`.

Para más información sobre los *guiones del desarrollador* véase Sección 5.18 .

5.4. Archivos *`nombre_del_paquete`*`.cron.*`

Si tu paquete requiere tareas periódicas para funcionar adecuadamente, puedes usar este fichero como patrón. Puedes establecer la realización de tareas que se ejecuten cada hora, día, semana, mes, o en cualquier otro período de tiempo. Los nombres de los archivos son:

- *`nombre_del_paquete`*`.cron.hourly` - instalados como `/etc/cron.hourly/`*`nombre_del_paquete`*: se ejecutan cada hora.
- *`nombre_del_paquete`*`.cron.daily` - instalados como `/etc/cron.daily/`*`nombre_del_paquete`*: se ejecutan cada día, habitualmente por la mañana temprano.
- *`nombre_del_paquete`*`.cron.weekly` - instalados como `/etc/cron.weekly/`*`nombre_del_paquete`*: se ejecutan cada semana, habitualmente en la mañana del domingo.
- *`nombre_del_paquete`*`.cron.hourly` - instalados como `/etc/cron.hourly/`*`nombre_del_paquete`*: se ejecutan cada hora.
- *`nombre_del_paquete`*`.cron.d` - instalados como `/etc/cron.d/`*`package`*: para cualquier otro período de tiempo.

Para los archivos mencionados, su formato es el de guiones «shell». La única excepción son los archivos *`nombre_del_paquete`*`.cron.d` que deben ajustarse al formato descrito en crontab(5).

No es necesario determinar un archivo `cron.*` para configurar la rotación de registros, para hacer eso consulta dh_installlogrotate(1) y logrotate(8).

[2] Véase dpkg(1) and Debian Policy Manual, ”D.2.5 Conffiles” (http://www.debian.org/doc/debian-policy/ap-pkg-controlfields.html#s-pkg-f-Conffiles) .

5.5. Archivo `dirs`

Este fichero especifica los directorios que se necesitan pero que por alguna razón no se crean en un proceso de instalación normal (`make install DESTDIR=...` invocado por `dh_auto_install`). Generalmente es debido a un problema con el archivo `Makefile`.

Los archivos listados en el archivo `install` no requieren la construcción previa de los directorios. Véase Sección 5.11 .

Es recomendable ejecutar en primer lugar la instalación y solo hacer uso de este archivo si se produce algún problema. No debe ponerse la barra inicial en los nombres de los directorios listados en el archivo `dirs`.

5.6. Archivo *`nombre_del_paquete`*`.doc-base`

Si tu paquete tiene documentación además de las páginas de manual y de información, puedes utilizar el archivo `doc-base` para registrarla de modo que el usuario pueda encontrar esta documentación suplementaria con dhelp(1), dwww(1) o doccentral(1).

La documentación incluirá archivos HTML, PS y PDF ubicados en `/usr/share/doc/`*`nombre_del_paquete/`*.

A continuación se muestra cómo es el fichero doc-base de `gentoo`, `gentoo.doc-base`:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: File Management
Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

Para más información sobre el formato del archivo véase install-docs(8) y el manual Debian `doc-base` en la copia local `/usr/share/doc/doc-base/doc-base.html/index.html` proporcionada por el paquete `doc-base`.

Para más detalles sobre la instalación de documentación adicional, lee Sección 3.3 .

5.7. Archivo `docs`

Este fichero especifica los nombres de los ficheros de documentación que dh_installdocs(1) instalará en el directorio temporal.

Por omisión, se incluirán todos los archivos existentes en los directorios de más alto nivel del código con los nombres `BUGS`, `README*`, `TODO` etc.

También incluiré algunos otros para `gentoo`:

```
  BUGS
  CONFIG-CHANGES
  CREDITS
  ONEWS
  README
  README.gtkrc
  TODO
```

5.8. Archivo `emacsen-*`

Si tu paquete proporciona ficheros Emacs que pueden ser compilados a bytes («bytescompile») en el momento de la instalación, puedes usar estos ficheros.

El programa dh_installemacsen(1) instalará estos archivos en el directorio temporal .

Elimínalos si no los necesitas.

5.9. Archivo *nombre_del_paquete*.examples

La orden dh_installexamples(1) instala los archivos y directorios listados en este archivo como archivos de ejemplos.

5.10. Archivos *nombre_del_paquete*.init y *nombre_del_paquete*.default

Si tu paquete es un demonio que necesita ejecutarse en el arranque del sistema, obviamente has desatendido mi recomendación inicial, ¿o no? :-)

El archivo *nombre_del_paquete*.init se instala como un guión en /etc/init.d/*nombre_del_paquete*. Se trata de una plantilla genérica construida por la orden **dh_make** como init.d.ex. Deberás renombrarlo y hacerle muchos cambios para asegurarte que cumpla con las cabeceras del estándar base de Linux (Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) , LSB). La orden dh_installinit(1) lo instalará en el directorio temporal.

El archivo *nombre_del_paquete*.default se instalará en el directorio /etc/default/*nombre_del_paquete*. Este archivo establece los valores predeterminados que utiliza el *guión init*. En la mayoría de casos, el archivo *nombre_del_paquete*.defaul se utiliza para desactivar la ejecución del demonio, estableciendo algunos indicadores predeterminados o tiempos de espera. Las características *configurables* establecidas en tu *guión init* deben incluirse en este archivo predeterminado, y no en el propio guión.

Si el programa original incluye un archivo *guión init*, tu puedes hacer uso de él o bien descartarlo. Si optas por no hacer uso del guión *guión init* original, deberás construir uno nuevo en debian/*nombre_del_paquete*.init. En cualquier caso deberás construir los enlaces simbólicos rc* aunque el *guión init* original parezca correcto y se instale en el lugar adecuado. Para ello, deberás reescribir el objetivo **dh_installinit** en el archivo rules con las siguientes líneas:

```
override_dh_installinit:
        dh_installinit --onlyscripts
```

Elimina el fichero si no lo necesitas.

5.11. Archivo install

Si hay archivos que deben ser instalados por el paquete pero no lo hace make install , puedes listar los archivos y sus destinos en el archivo install. Se encargará de la instalación la orden dh_install(1) [3]. Debes asegurarte que no hay un sistema más específico para hacer esta instalación. Por ejemplo, para la documentación debes utilizar el archivo docs, en lugar de este archivo.

El archivo install tendrá una línea para cada uno de los archivos a instalar, con el nombre del archivo (relativo al directorio superior de la compilación) seguido de un espacio y a continuación el directorio de instalación (relativo al directorio superior de instalación). Suponiendo que el archivo binario src/*archivo_binario* no se instalase, deberías utilizar el archivo install como sigue:

```
src/archivo_binario usr/bin
```

Al instalarse el paquete, se instalará el archivo binario /usr/bin/nombre_archivo_binario.

En el archivo install puedes escribir el nombre del archivo sin el directorio de instalación siempre que no cambie la ruta relativa de directorio. Este formato se usa en paquetes grandes que separan el resultado de la compilación en múltiples paquetes binarios haciendo uso de *nombre_del_paquete-1*.install, *nombre_del_paquete-2*.install, etc.

La orden **dh_install** retrocederá al directorio debian/tmp para buscar los archivos si no los encuentra en el directorio actual (o en la ubicación que hayas establecido para la búsqueda con --sourcedir).

5.12. Archivo *nombre_del_paquete*.info

Si tu paquete utiliza páginas de información («info pages»), podrás instalarlas utilizando dh_installinfo(1) que utilizará el listado del archivo *nombre_del_paquete*.info.

[3] Esto reemplaza la orden obsoleta dh_movefiles(1) que se configuraba con el archivo files.

5.13. Archivo *nombre_del_paquete*.links

Si es necesario generar enlaces simbólicos adicionales, como responsable del paquete, en los directorios de compilación del paquete, puedes instalarlos utilizando dh_link(1) haciendo una lista de las rutas completas de los ficheros de origen y de destino en un fichero *nombre_del_paquete*.links.

5.14. Archivos {*nombre_del_paquete*.source/} lintian-overrides

Si el programa lintian emite un informe de diagnóstico erróneo en algún caso en que las normas admitan excepciones, podrás utilizar los archivos *nombre_del_paquete*.lintian-overrides o source/lintian-overrides para evitar que se emita el error. Por favor, lee Lintian User's Manual (https://lintian.debian.org/manual/index.html) error y procura no abusar de esta opción (para ocultar errores auténticos).

nombre_del_paquete.lintian-overrides es para un paquete binario con el nombre *nombre_del_paquete* y es instalado en usr/share/lintian/overrides/*nombre_del_paquete* por la orden **dh_lintian**.

El archivo source/lintian-overrides es para los paquetes de fuentes y no se instala.

5.15. Archivos manpage.*

El/los programa/s debería/n tener una página de manual. Tendrás que escribirla si no existe. La orden **dh_make** construye varios archivos de plantilla para las páginas de manual. Deberás copiarlos y editarlos para cada programa que no tenga página de manual. Asegúrate de eliminar los que no utilices.

5.15.1. Archivo manpage.1.ex

Las páginas de manual se escriben normalmente con nroff(1). La plantilla manpage.1.ex está escrita con **nroff**. Consulta la página de manual man(7) para una breve descripción sobre como editar el archivo.

El nombre del archivo de manual debería incluir el nombre del programa que está documentando, así que lo renombraremos de manpage a gentoo. El nombre del fichero incluye también .1 como primer sufijo, lo que significa que es una página de manual para una programa de usuario. Asegúrate de verificar que esa sección es la correcta. Aquí tienes una pequeña lista de las secciones de las páginas de manual.

Sección	Descripción	Notas
1	Órdenes de usuario	Órdenes ejecutables o guiones
2	Llamadas del sistema	Funciones proporcionadas por el núcleo
3	Llamadas a bibliotecas	Funciones de las bibliotecas del sistema
4	Archivos especiales	Por lo general se encuentran en /dev
5	Formatos de archivo	p. ej. el formato de /etc/passwd
6	Juegos	Juegos y otros programas de entretenimiento
7	Paquetes macro	Tales como macros **man**
8	Administración del sistema	Programas que sólo suele ejecutar el superusuario
9	Rutinas del núcleo	Llamadas al sistema no estándar e internas

Así que la página de manual de gentoo debería nombrarse como gentoo.1. No había una página de manual gentoo.1 en el paquete fuente así que la escribí renombrando la plantilla manpage.1.ex como gentoo.1 y modificándola usando la información del ejemplo y de los documentos del programador original.

Utiliza la orden **help2man** para generar la página de manual a partir de la información dada por «--help» y --version» para cada programa [4].

[4] Ten presente que el marcador de posición de páginas de manual **help2man** reclamará que la documentación más detallada se encuentra disponible en el sistema de información. Si el comando no encuentra una página **info**, deberás editar manualmente la página de manual construida por **help2man**.

5.15.2. Archivo `manpage.sgml.ex`

Por otro lado, puede que prefieras escribir usando SGML en lugar de **nroff**. En este caso, puedes usar la plantilla `manpage.sgml.ex`. Si haces esto, tendrás que:

- renombrar el fichero a algo como `gentoo.sgml`.
- instalar el paquete `docbook-to-man`
- añadir `docbook-to-man` en el campo `Build-Depends` en el archivo `control`
- añadir el objetivo `override_dh_auto_build` en el fichero `rules`:

```
override_dh_auto_build:
        docbook-to-man debian/gentoo.sgml > debian/gentoo.1
        dh_auto_build
```

5.15.3. Archivo `manpage.xml.ex`

Si prefieres el formato XML en lugar de SGML, puedes utilizar la plantilla `manpage.xml.ex`. En este caso debes:

- renombrar el archivo a `gentoo.1.xml`
- instalar el paquete `docbook-xsl` y un procesador XSLT como `xsltproc` (recomendado)
- añadir los paquetes `docbook-xsl`, `docbook-xml` y `xsltproc` en la línea de `Build-Depends` en el fichero de `control`
- añadir el objetivo `override_dh_auto_build` en el fichero `rules`:

```
override_dh_auto_build:
        xsltproc --nonet \
         --param make.year.ranges 1 \
         --param make.single.year.ranges 1 \
         --param man.charmap.use.subset 0 \
         -o debian/ \
http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\
        debian/gentoo.1.xml
        dh_auto_build
```

5.16. Archivo *`nombre_del_paquete`*`.manpages`

Si tu paquete tiene páginas de manual, deberías instalarlas con dh_installman(1) listando los archivos correspondientes en el archivo *`nombre_del_paquete`*`.manpages`.

Para instalar el archivo `docs/gentoo.1` del paquete `gentoo` como su manual, construirás el archivo `gentoo.manpages` con el contenido:

```
docs/gentoo.1
```

5.17. Archivo `NEWS`

La orden dh_installchangelogs(1) instala este archivo.

5.18. Archivos `{pre,post}{inst,rm}`

Los archivos `postinst`, `preinst`, `postrm` y `prerm` [5] se denominan *guiones del desarrollador*. Son guiones que se colocan en el área de control del paquete y que **dpkg** ejecuta cuando tu paquete se instala, se actualiza o se elimina.

Por ahora, deberías intentar evitar editar manualmente estos *guiones del desarrollador* si puedes porque suelen hacerse muy complejos. Para más información lee «Debian Policy Manual, 6 "Package maintainer scripts and installation procedure"» (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) , y echa un vistazo a los ejemplos proporcionados por **dh_make**.

Si a pesar de mis advertencias, adaptas los *guiones del desarrollador* para el paquete, asegúrate de comprobar su funcionamiento no sólo para la **instalación** y **actualización** del paquete, sino también para su **desinstalación** y **eliminación completa**.

Las actualizaciones a una nueva versión deben ser «silenciosas» y no intrusivas. Los usuarios no deberían darse cuenta de la actualización, salvo quizás para descubrir que se han arreglado errores antiguos y porque hay alguna nueva funcionalidad.

Cuando la actualización es necesariamente intrusiva (p.e. archivos de configuración dispersos en varios directorios con una estructura totalmente modificada), se deberían establecer los valores predeterminados seguros (p.e. desactivar los servicios) y facilitar la documentación apropiada establecida por las normas (archivos `README.Debian` y `NEWS.Debian`) como último recurso. Hay que evitar molestar al usuario con notas **debconf** invocadas por los *guiones del desarrollador* en las actualizaciones.

El paquete `ucf` facilita el sistema *conffile-like* para preservar los cambios de configuración realizados por el usuario y por ello no deben nombrarse como *conffiles* los archivos manejados por los *guiones del desarrollador*. Así se minimizan las incidencias asociadas con ellos.

Estos *guiones del desarrollador* son un ejemplo de las características de Debian que explican **por qué la gente elige Debian**. Debes ser cuidadoso con no molestarles con ellos.

5.19. Archivo *`nombre_del_paquete`*`.symbols`

El mantenimiento de paquetes de bibliotecas no es fácil para los principiantes y se debe evitar. Aún así, si el paquete tiene bibliotecas, debes tener los ficheros `debian/`*`nombre_del_paquete`*`.symbols`. Consulte Sección A.2.

5.20. Archivo `TODO`

La orden dh_installdocs(1) instala este archivo.

5.21. Archivo `watch`

El formato del archivo `watch` se documenta en la página de manual de uscan(1). El archivo `watch` se usa para configurar el programa **uscan** (del paquete `devscripts`) que vigila el servidor de donde obtuviste las fuentes originales. También lo utiliza «Debian External Health Status (DEHS)» (http://wiki.debian.org/DEHS) .

Este es su contenido:

```
# watch control file for uscan
version=3
http://sf.net/gentoo/gentoo-(.+)\.tar\.gz debian uupdate
```

Con el archivo `watch`, la URL `http://sf.net/gentoo` se descarga y se buscan los enlaces del tipo `<a href=...>`. El nombre base (justo la parte después del `/` final) de las direcciones URL enlazadas se comparan con la expresión regular de Perl (véase perlre(1)) con el patrón `gentoo-(.+)\.tar\.gz`. Se descarga el archivo de versión más reciente de entre todos los archivos encontrados cuyo nombre se ajuste al patrón, y se ejecutará el programa **uupdate** para construir el árbol de código fuente actualizado en base a este fichero.

[5] Aunque el texto utilice la expresión abreviada **bash** para el nombre de los archivos `{pre,post}{inst,rm}`, debes utilizar la sintaxis POSIX pura para estos guiones del desarrollador para mantener la compatibilidad con el «shell» del sistema **dash**.

Aunque es posible hacer esto con otros portales, el servicio de descarga de «SourceForge» en http://sf.net (http://sf.net) es una excepción. Cuando el archivo `watch` tiene una URL que concuerda con el patrón Perl `^http://sf\.net/`, el programa **uscan** lo substituye por `http://qa.debian.org/watch/sf.php/` y después aplica esta regla. El servicio de redireccionamiento URL a la dirección http://qa.debian.org/ (http://qa.debian.org/) está diseñado para ofrecer un servicio estable de redireccionamiento al archivo presente en `watch` y que concuerda con `http://sf.net/`*`proyecto/nombre_archivo_tar`*`-(.+)\.tar\.gz`. Esto resuelve los problemas relacionados con los cambios periódicos en la dirección URL.

Si el autor publica la firma criptográfica del fichero tarball, se recomienda comprobar su autenticidad utilizando la opción `pgpsigurlmangle` descrita en uscan(1).

5.22. Archivo `source/format`

El archivo `debian/source/format`, solo contendrá una línea indicando el formato deseado para el paquete (véase dpkg-source(1) para consultar la lista completa). Después de `squeeze` debería ser:

- `3.0 (native)` para paquetes nativos de Debian o
- `3.0 (quilt)` para el resto de paquetes.

El nuevo formato `3.0 (quilt)` registra los cambios en series de archivos de parches **quilt** en el directorio `debian/patches`. Estos cambios se aplican automáticamente en la extracción de las fuentes del paquete [6]. Las modificaciones se guardan en el archivo `debian.tar.gz` que contiene todos los archivos del directorio `debian` utilizado en la construcción del paquete. El nuevo formato permite la inclusión de archivos como los iconos PNG sin necesidad de trucos [7].

Cuando **dpkg-source** extrae un paquete fuente con el formato `3.0 (quilt)`, automáticamente aplica todos los parches listados en el archivo `debian/patches/series`. Puedes evitar la ejecución de los parches al final de la extracción con la opción `--skip-patches`.

5.23. Archivo `source/local-options`

Si deseas gestionar los trabajos de empaquetado en un entorno VCS, seguramente tendrás una rama (p.ej. `upstream`) que sigue las fuentes originales y otra rama (p.ej. normalmente `master` en Git) para el paquete Debian. Para este último caso, generalmente tendrás las fuentes originales sin modificar (sin aplicarles los parches) con los archivos `debian/*` para el empaquetamiento Debian para facilitar la fusión con las nuevas versiones de las fuentes.

Una vez compilado el paquete, las fuentes estarán parcheadas. Deberás deshacer los parches manualmente ejecutando `dquilt pop -a` antes de sincronizar con la rama `master`. Puedes automatizar esto añadiendo el archivo opcional `debian/source/local-options` cuyo contenido será «`unapply-patches`». Este archivo no se incluye en el paquete fuente generado y sólo cambia el entorno local de compilación. Este archivo también puede contener la línea «`abort-on-upstream-changes`» (véase dpkg-source(1)).

```
unapply-patches
abort-on-upstream-changes
```

5.24. Archivo `source/options`

Los archivos generados automáticamente en el árbol del código fuente pueden ser bastante molestos en la construcción del paquete debido a que generan archivos de parche grandes. Hay módulos personalizados, tales como **dh_autoreconf** para aliviar este problema como se describe en Sección 4.4.3.

[6] Véase DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) para un resumen informativo sobre los formatos `3.0 (quilt)` y `3.0 (native)`.

[7] Actualmente, este nuevo formato también permite trabajar con múltiples archivos «tar» fuente y otros sistemas de compresión. Estas funciones están fuera del objetivo de este documento.

Puedes proporcionar una expresión regular en Perl para la opción `--extend-diff-ignore` del argumento de dpkg-source(1) para hacer caso omiso de los cambios realizados en los archivos generados automáticamente al crear el paquete fuente.

Puedes conservar las opciones de la orden **dpkg-source** en el archivo `source/options` de las fuentes del paquete como solución genérica al problema de los archivos autogenerados. En el ejemplo siguiente, se evita la generación de archivos de parche para `config.sub` `config.guess` y `Makefile`.

```
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

5.25. Archivos `patches/*`

El antiguo formato `1.0` construía un archivo `diff.gz` cuyo contenido era el de los archivos de construcción del paquete del directorio `debian` así como los cambios a realizar en las fuentes. Este formato para conservar los cambios resultaba engorroso cuando se trataba de inspeccionar y entender cada modificación de las fuentes. Ya no resulta eficaz.

El nuevo formato `3.0 (quilt)` de las fuentes, guarda las modificaciones (los parches) en archivos en el directorio `debian/patches/*` utilizando la orden **quilt**. Estos parches y otros datos del paquete que están en el directorio `debian` se conservan en el archivo `debian.tar.gz`. Desde que la orden **dpkg-source** puede aplicar los parches en las fuentes con el nuevo formato tipo **quilt** sin el paquete `quilt`, no es necesario añadir el paquete `quilt` en el campo `Build-Depends` del fichero `control` [8].

El funcionamiento de la orden **quilt** se explica en quilt(1). Conserva las modificaciones de las fuentes en una colección de archivos de parches `-p1` en el directorio `debian/patches` y las fuentes originales permanecen sin modificar fuera del directorio `debian`. El orden de aplicación de las modificaciones se conserva en el archivo `debian/patches/series`. Puedes aplicar («push»), deshacer («pop») y actualizar las modificaciones fácilmente [9].

En Capítulo 3, se han construido tres archivos de parches en `debian/patches`.

Puesto que los parches se ubican en `debian/patches`, asegúrate que has configurado adecuadamente la orden **dquilt** como se describe en Sección 3.1 .

Cuando alguien (incluido tú mismo), facilita un parche *`nombre_parche`*`.patch` para las fuentes, cuando ya está construido el paquete, la modificación de un paquete con formato `3.0 (quilt)` es así de simple:

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../nombre_parche.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
.. descripción de la modificación
```

Los parches conservados con el nuevo formato de fuentes `3.0 (quilt)` deben estar exentos de *cosas innecesarias*. Debes asegurarte ejecutando `dquilt pop -a; while dquilt push; do dquilt refresh; done`.

[8] Se han propuesto y se están utilizando otros métodos de aplicación de los parches en Debian. El sistema **quilt** es el preferido. Otros sistemas son **dpatch**, **dbs**, **cdbs**, etc. La mayoría de ellos conservan los parches en archivos en el directorio `debian/patches/*`.

[9] Si has solicitado a un patrocinador que transfiera el paquete al repositorio, este sistema de separación y documentación de los cambios es muy importante para facilitar la revisión del paquete por el patrocinador.

Capítulo 6

Construcción del paquete

Ahora deberíamos estar preparados para construir el paquete.

6.1. (Re)construcción completa

Para realizar correctamente la (re)compilación completa de un paquete, debes asegurarte que tienes instalados:

- el paquete `build-essential`.
- los paquetes listados en el campo `Build-Depends` del archivo «control» (Sección 4.1), y
- los paquetes listados en el campo `Build-Depends-indep` (también del archivo «control», Sección 4.1).

Después ejecuta la siguiente orden en el directorio raíz del código fuente del programa:

```
$ dpkg-buildpackage -us -uc
```

Esto hará todo lo necesario para construir los paquetes binarios y de fuentes para ti. Para ello:

- limpia el árbol del código fuente («`debian/rules clean`»)
- construye el paquete de código («`dpkg-source -b`»)
- construye el programa («`debian/rules build`»)
- construye el paquete binario (`fakeroot debian/rules binary`)
- genera el fichero `.dsc`
- genera el fichero `.changes`, utilizando **dpkg-genchanges**

Si el resultado de la compilación es satisfactorio, firma los ficheros de extensión `.dsc` y `.changes` con tu clave privada GPG utilizando la orden **debsign**. Deberás escribir la contraseña de tu clave dos veces. [1]

Para los paquetes no nativos Debian, p.ej, `gentoo`, podrás ver los siguientes archivos en el directorio padre (`~/gentoo`) después de construir el paquete:

[1] Esta clave GPG debe ser firmado por un desarrollador de Debian para conectarse a la red de confianza y debe registrarse en el anillo de claves de Debian (http://keyring.debian.org) . Esto permite que los paquetes sean aceptados en los repositorios de Debian. Consulta Cómo crear un clave GPG nueva (http://keyring.debian.org/creating-key.html) y Wiki de Debian sobre la firma de claves (http://wiki.debian.org/Keysigning) .

- `gentoo_0.9.12.orig.tar.gz`

  Este es el código fuente original comprimido, simplemente se ha renombrado para seguir los estándares de Debian. Debe tenerse en cuenta que fue generado usando la opción «-f» de `dh_make -f ../gentoo-0.9.12.tar.gz` ejecutada en el inicio.

- `gentoo_0.9.12-1.dsc`

  Este es un sumario de los contenidos del código fuente. Este fichero se genera a partir del fichero de `control` y se usa cuando se descomprimen las fuentes con dpkg-source(1).

- `gentoo_0.9.12-1.debian.tar.gz`

  Este fichero comprimido contiene el directorio `debian` completo. Todas las modificaciones de las fuentes originales se conservan en los archivos de parches **quilt** en el directorio `debian/patches`.

  Si alguien quiere volver a construir tu paquete desde cero, puede hacerlo fácilmente usando los tres ficheros de arriba. El proceso de extracción es trivial: sólo se debe copiar los tres ficheros en algún lado y ejecutar `dpkg-source -x gentoo_0.9.12-1.dsc` [2].

- `gentoo_0.9.12-1_i386.deb`

  Este es el paquete binario completo. Puedes usar **dpkg** para instalar o eliminar tanto este paquete como cualquier otro.

- `gentoo_0.9.12-1_i386.changes`

  Este fichero describe todos los cambios hechos en la revisión actual del paquete, y lo utilizan los programas de mantenimiento del archivo FTP de Debian para instalar los paquetes binarios y fuentes. Se genera parcialmente a partir del fichero `changelog` y el fichero `.dsc`.

  Mientras sigues trabajando en el paquete, éste cambiará su comportamiento y se le añadirán nuevas funciones. Las personas que descarguen tu paquete pueden leer este fichero y ver qué ha cambiado. Los programas de mantenimiento del archivo de Debian, también enviarán el contenido de este fichero a la lista de correo debian-changes-announce@lists.debian.org (http://lists.debian.org/debian-devel-changes/) .

Sera necesario firmar los ficheros `gentoo_0.9.12-1.dsc` y `gentoo_0.9.12-1_i386.changes` utilizando la orden **debsign** con la clave privada GPG almacenada en el directorio `~/.gnupg/`, antes de subirlos al archivo FTP de Debian. La firma GPG proporciona la prueba de que estos ficheros son realmente tuyos usando tu clave pública GPG.

La orden **debsign** puede ejecutarse para firmar utilizando la ID de tu clave GPG privada (útil para patrocinar paquetes) ,especificándola en el siguiente `~/.devscripts` como sigue:

```
DEBSIGN_KEYID=Tu_GPG_keyID
```

Las largas listas de números en los ficheros `.dsc` y `.changes` son las sumas MD5/SHA1/SHA256 de los ficheros incluidos en el paquete. Las personas que descarguen estos ficheros pueden comprobarlos con md5sum(1), sha1sum(1) o sha256sum(1) y si los números no coinciden, sabrán que el fichero está corrupto o ha sido modificado.

6.2. Autobuilder

Debian mantiene diversas adaptaciones «ports» (http://www.debian.org/ports/) con la red de servidores de compilación automática (http://www.debian.org/devel/buildd/) que ejecuta demonios **buildd** en ordenadores con diferentes arquitecturas. Aunque no deberás hacer todo esto tú mismo, debes conocer el proceso al que se verá sometido tu paquete. Veamos cómo se procesa el paquete para compilarlo en diversas arquitecturas [3].

Los paquetes del tipo `Architecture: any`, son construidos por el sistema de compilación automática. El sistema garantiza la instalación de:

- el paquete `build-essential`, y

[2] Puedes evitar la aplicación automática de las modificaciones por **dquilt** en los paquetes con formato `3.0 (quilt)` al final de la extracción con la opción `--skip-patches`. También puedes deshacer las modificaciones al finalizar la extracción con la ejecución de `dquilt pop -a`.

[3] El funcionamiento del sistema actual de compilación automática es más complicado que lo expuesto en este documento. Muchos detalles quedan fuera de los objetivos de este documento.

- los paquetes listados en el campo `Build-Depends` (véase Sección 4.1).

A continuación se ejecuta la siguiente orden en el directorio de las fuentes:

```
$ dpkg-buildpackage -B
```

De esta manera, se ejecuta todo lo necesario para construir el paquete binario dependiente de la arquitectura en cada arquitectura. Hace lo siguiente:

- limpia el árbol del código fuente («`debian/rules clean`»)
- construye el programa («`debian/rules build`»)
- construye el paquete binario para la arquitectura (`fakeroot debian/rules binary-arch`)
- firma el fichero fuente `.dsc`, usando **gpg**
- genera y firma el fichero de envío `.changes`, usando **dpkg-genchanges** y **gpg**

Esta es la razón por la que el paquete estará disponible para otras arquitecturas.

Aunque es necesario instalar los paquetes listados en el campo `Build-Depends-indep` para el empaquetamiento normal (véase Sección 6.1), el sistema automático de construcción no lo requiere puesto que solo construye paquetes binarios dependientes de la arquitectura [4]. Estas diferencias entre el empaquetamiento normal y el automático determinan si los paquetes requeridos para la compilación se deben listar en el campo `Build-Depends` o bien en el campo `Build-Depends-indep` del archivo `debian/control` (véase Sección 4.1).

6.3. La orden debuild

Puedes automatizar aún más el proceso de construcción de paquetes de la orden **dpkg-buildpackage** con la orden **debuild**. Véase debuild(1).

La orden **debuild** ejecuta a su vez la orden **lintian** para comprobar el paquete Debian compilado. La orden **lintian** puede configurarse con el siguiente `~/.devscripts`:

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

Por ejemplo, limpiar el código y reconstruir el paquete desde una cuenta de usuario es tan simple como:

```
$ debuild
```

Puedes eliminar los archivos generados en la compilación ejecutando:

```
$ debuild clean
```

6.4. El paquete `pbuilder`

El paquete `pbuilder` es muy útil para conseguir un entorno de compilación limpio (**chroot**) donde verificar las dependencias de compilación [5]. Esto asegura una construcción limpia desde el código en los auto-compiladores de `sid` para las distintas arquitecturas y evita fallos de nivel de severidad serios del tipo FTBFS («Fail to Build From Source», o «fallo al construir desde la fuente »), que son siempre de categoría RC (fallos críticos o «release critical»). [6]

Para personalizar el funcionamiento del paquete `pbuilder` haz lo siguiente:

[4] Contrariamente al modo de funcionamiento del paquete `pbuilder` (que se tratará más adelante), el entorno **chroot** del paquete `sbuild` utilizado por el sistema automático no tiene una instalación mínima de paquetes del sistema y puede dejar muchos paquetes instalados.

[5] Dado que el paquete `pbuilder` está evolucionando, deberías comprobar la situación de la configuración consultando la última documentación oficial disponible.

[6] Consulta http://buildd.debian.org/ para saber más sobre la construcción automática de paquetes Debian.

- permite que tu usuario del sistema tenga permiso de escritura en el directorio `/var/cache/pbuilder/result`.
- crea un directorio, p.e. */var/cache/pbuilder/hooks*, con permiso de escritura para tu usuario. En este directorio pondrás los guiones «hook»
- establece los archivos `~/.pbuilderrc` o `/etc/pbuilderrc` con el siguiente contenido:

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

Ahora puedes inicializar el sistema local `pbuilder` **chroot** por primera vez ejecutando:

```
$ sudo pbuilder create
```

Si ya dispones de un paquete fuente completo, ejecuta las siguientes órdenes en el directorio donde tengas los archivos *nombre_del_paquete*`.orig.tar.gz`, *nombre_del_paquete*`.debian.tar.gz` y *nombre_del_paquete*`.dsc` para actualizar el sistema local `pbuilder` **chroot** y, a continuación, compilar el paquete binario en él:

```
$ sudo pbuilder --update
$ sudo pbuilder --build nombre_del_paquete.dsc
```

Una vez finalizada la compilación, el paquete compilado estará en el directorio `/var/cache/pbuilder/result/` cuyo propietario será tu usuario en lugar del usuario administrador.

Las firmas GPG en los archivos `.dsc` y `.changes` se puede generar como sigue

```
$ cd /var/cache/pbuilder/result/
$ debsign nombre_del_paquete_versión_arquitectura.changes
```

Si en lugar de iniciar la construcción del paquete a partir de un paquete ya construido, dispones de un directorio con las fuentes originales actualizadas, deberás ejecutar las siguientes órdenes desde el directorio de las fuentes originales (y en el que deberá estar presente el directorio `debian` con todo su contenido):

```
$ sudo pbuilder --update
$ pdebuild
```

Puedes «conectarte» al entorno **chroot** ejecutando la orden `pbuilder --login --save-after-login` y configurarlo para adaptarlo a tus necesidades. Este entorno puede guardarse saliendo del «shell» con `^D` (Control-D).

La última versión de la orden **lintian** puede ejecutarse automáticamente en el entorno `chroot` utilizando el guión «hook» disponible en */var/cache/pbuilder/hooks/*`B90lintian` y configurado como sigue [7]:

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --force-yes install "$@"
        }
install_packages lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes" - pbuilder
# use esta versión si quiere evitar que lintian pare la compilación
#su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" - pbuilder
echo "+++ final del informe de lintian +++"
```

Debes tener un entorno `sid` actualizado para construir correctamente paquetes para `sid`. En realidad, al versión `sid` puede contener errores que no hacen recomendable la migración de tu sistema a esta versión. El paquete `pbuilder` te ayuda a hacer frente a esta situación.

[7] Se supone que la variable de entorno `HOOKDIR=/var/cache/pbuilder/hooks` ya está configurada. Puedes consultar otros ejemplos en `/usr/share/doc/pbuilder/examples`.

Es posible que debas actualizar tu paquete `stable` después de distribuirlo para `stable-proposed-updates`, `stable/updates`, etc [8]. En algunas ocasiones, «Yo trabajo con la versión `sid`» puede no ser una excusa para dejar de actualizar el paquete. El paquete `pbuilder` te ayuda trabajar en entornos para todas las distribuciones derivadas Debian para una arquitectura determinada.

Consulta http://www.netfort.gr.jp/~dancer/software/pbuilder.html (http://www.netfort.gr.jp/~dancer/software/pbuilder.html) , pdebuild(1), pbuilderrc(5), y pbuilder(8).

6.5. La orden git-buildpackage y similares

Si el autor original utiliza un sistema de gestión de versiones para el código [9] puedes considerar utilizarlo. Así se facilita la coordinación y la selección de los parches en las fuentes. Debian dispone de varios paquetes de guiones especializados en cada tipos de VCS:

- `git-buildpackage`: conjunto para la compilación de paquetes en repositorios «Git».
- `svn-buildpackage`: programas de ayuda para el mantenimiento de paquetes Debian con «Subversion».
- `cvs-buildpackage`: conjunto de paquete de guiones Debian para estructuras de directorios CVS.

El uso de `git-buildpackage` se está convirtiendo en muy popular para los desarrolladores de Debian para el mantenimiento de los paquetes Debian con el servidor Git en alioth.debian.org (http://alioth.debian.org/) . [10] Este paquete ofrece muchas órdenes para *automatizar* el mantenimiento de paquetes:

- git-import-dsc(1): importa la versión anterior del paquete Debian al repositorio «Git».
- git-import-orig(1): importa el nuevo fichero «tar» del autor al repositorio «Git».
- git-dch(1): genera el fichero de cambios de Debian desde los mensajes de cambios de «Git».
- git-buildpackage(1): compila los paquetes Debian del repositorio «Git».
- git-pbuilder(1): compila los paquetes Debian del repositorio «Git» utilizando **pbuilder**/**cowbuilder**.

Estas órdenes utilizan 3 ramas en la gestión del proceso de empaquetado:

- `main` (principal) del árbol de directorios del paquete Debian.
- `upstream` (autor) para el árbol de las fuentes del autor.
- `pristine-tar` por el fichero comprimido «.tar» del autor generado por la opción `--pristine-tar`.[11]

Puedes configurar `git-buildpackage` con `~/.gbp.conf`. Lee gbp.conf(5). [12]

[8] Hay algunas restricciones para estas actualizaciones de tu paquete `stable`.

[9] Consulta Version control systems (http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) para saber más.

[10] Debian wiki Alioth (http://wiki.debian.org/Alioth) documenta cómo utilizar el servicio alioth.debian.org (http://alioth.debian.org/) .

[11] La opción `--pristine-tar` invoca la orden **pristine-tar** la cual puede regenerar una copia exacta del fichero comprimido de fuentes impoluto utilizando solo un pequeño fichero binario «delta» y el contenido del fichero comprimido tipo «.tar», habitualmente ubicado en la rama `upstream` del VCS.

[12] Aquí tienes algunos recursos disponibles en la red, dirigidos a usuarios expertos.

- Construcción paquetes Debian con «git-buildpackage» (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`)
- debian packages in git (https://honk.sigxcpu.org/piki/development/debian_packages_in_git/)
- Using Git for Debian Packaging (http://www.eyrie.org/~eagle/notes/debian/git.html)
- git-dpm: Debian packages in Git manager (http://git-dpm.alioth.debian.org/)
- Using TopGit to generate quilt series for Debian packaging (http://git.debian.org/?p=collab-maint/topgit.git;a=blob_plain;f=debian/HOWTO-tg2quilt;hb=HEAD)

6.6. Reconstrucción rápida

Con un paquete grande, puede que no quieras recompilar desde cero cada vez que tocas un detalle en el fichero `debian/rules`. Para propósitos de prueba, puedes hacer un fichero `.deb` sin necesidad de recompilar las fuentes originales de esta forma[13]:

```
  fakeroot debian/rules binary
```

O simplemente puedes comprobar si el paquete se compila con:

```
$ fakeroot debian/rules build
```

Una vez terminada la puesta a punto, recuerda reconstruir el paquete siguiendo el procedimiento adecuado explicado anteriormente. Puede que no seas capaz de enviar correctamente el paquete si intentas enviar los archivos `.deb` construidos de esta forma.

6.7. Jerarquía de órdenes

A continuación hay un breve resumen de cómo las órdenes de construcción de paquetes encajan jerárquicamente. Hay muchas maneras de hacer lo mismo.

- `debian/rules` = guión del desarrollador para la construcción del paquete
- **dpkg-buildpackage** = orden principal de las herramientas de compilación
- **debuild** = **dpkg-buildpackage** + **lintian** (construcción con la configuración estándar de las variables de entorno)
- **pbuilder** = núcleo de la herramienta de entorno «chroot» de Debian
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (construcción en el entorno «chroot»”
- **cowbuilder** = ejecución acelerada de **pbuilder**
- **git-pbuilder** = la versión de sintaxis fácil para **pdebuild** (utilizado por **gbp buildpackge**)
- **gbp** = gestor de los ficheros Debian del paquete en un repositorio «git»
- **gbp buildpackge** = **pbuilder** + **dpkg-buildpackage** + **gbp**

Aunque el uso de las órdenes de nivel superior como **gbp buildpackge** y **pbuilder** asegura el entorno perfecto de construcción de paquetes, es esencial para comprender cómo órdenes de menor nivel, tales como `debian/rules` y **dpkg-buildpackage** se ejecutan subordinadas.

[13] Las variables de entorno que normalmente están configuradas con los valores adecuados, no se configuran con este método. Nunca cree paquetes reales para enviarlos utilizando este método **rápido**.

Capítulo 7

Comprobación del paquete en busca de fallos

Debes conocer varios métodos para comprobar el paquete y localizar errores antes de transferirlo a repositorios públicos.

Probar el paquete en una máquina distinta a la usada en su construcción es una magnífica idea. Debes poner atención en todos los avisos y errores que se produzcan en las pruebas explicadas a continuación.

7.1. Cambios sospechosos

Si encuentras un nuevo archivo de parche autogenerado con el nombre `debian-changes-*` en el directorio `debian/patches` después de la construcción de tu paquete de Debian no nativo en formato `3.0 (quilt)`, es probable que hayas cambiado algún archivo accidentalmente o bien que el guión de compilación haya modificado las fuentes originales. Si el error es tuyo, corrígelo. Si es causado por el guión de compilación, corrige el origen del error con **dh-autoreconf** como en Sección 4.4.3 o bien inténtalo con los archivos `source/options` como en Sección 5.24.

7.2. Comprobación de la instalación del paquete

Instala el paquete para probarlo tú mismo, por ejemplo, usando la orden debi(1) como superusuario. Intenta instalarlo y ejecutarlo en otras máquinas distintas de la tuya, y presta atención para detectar errores o avisos tanto en la instalación como en la ejecución del programa.

```
$ sudo debi gentoo_0.9.12-1_i386.changes
```

Debes asegurarte que no haya archivos en conflicto con otros paquetes utilizando el archivo `Contents-`*`i386`* (descargándolo del repositorio Debian) para prevenir problemas de instalación en distintos sistemas. La orden **apt-file** será útil para realizar esta tarea. Si hay archivos en conflicto, deberás hacer lo necesario para evitar problemas utilizando el mecanismo de alternativas (véase update-alternatives(1)) coordinándote con los responsables de los otros paquetes o bien estableciendo correctamente el campo `Conflicts` del archivo `debian/control`.

7.3. Comprobación de los *guiones del desarrollador* («maintainer scripts»)

Ya se ha comentado que los *guiones del desarrollador* (los archivos `preinst`, `prerm`, `postinst` y `postrm`) son complicados, excepto si se utilizan los generados por el paquete `debhelper`. No se recomienda su utilización a los desarrolladores principiantes (véase Sección 5.18).

Si el paquete utiliza estos *guiones del desarrollador* modificados o no triviales, debes comprobar su funcionamiento en la instalación, desinstalación, eliminación y actualización. Algunos errores en estos *guiones del desarrollador* sólo se producen cuando los paquetes se eliminan o purgan. Utiliza la orden **dpkg** como se indica a continuación para probarlos:

```
$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_versión-revisión_i386.deb
```

Sigue esta secuencia para la comprobación:

- Instala la versión anterior del paquete (requerido).
- Actualiza ahora a la versión actual.
- Vuelve a la versión anterior (opcional).
- Elimínalo.
- Instala la nueva versión del paquete.
- Elimínalo.
- Instálalo de nuevo.
- Elimínalo.

Si estás trabajando en la construcción de la primera versión del paquete, construye versiones «fantasma» anteriores (será suficiente cambiar el número de la versión) para realizar las pruebas y así prevenir problemas.

Recuerda que si ya hay versiones anteriores del paquete en el repositorio Debian, los usuarios actualizarán el paquete desde la versión anterior disponible (y esta versión puede ser distinta en la versión estable y de pruebas). Realiza las comprobaciones también con estas versiones.

Aunque no se garantiza la reinstalación de una versión anterior, es preferible asegurarse que es posible sin generar problemas.

7.4. El paquete `lintian`

Ejecuta lintian(1) en tu archivo `.changes`. La orden **lintian** ejecutará varios guiones para revisar los errores más frecuentes de los paquetes [1].

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

Por supuesto, cambia el nombre del fichero por el nombre del fichero `.changes` generado por tu paquete. Los mensajes de error de **lintian** se codifican con una letra al inicio de la línea del mensaje:

- `E:` para los errores; indica violaciones de las normas o un error en el paquete.
- `W:` para las advertencias; indica una posible violación de las normas o error en el paquete (pero pudiendo ser una falsa alarma).
- `I:` para información; proporciona algo de información sobre algún aspecto del paquete (que tal vez sea mejorable).
- `N:` para las notas o anotaciones; proporciona mensajes detallados que pueden ayudarte en la depuración del paquete.
- `O:` para mensajes ignorados; un mensaje ignorado (según lo configurado en los archivos `lintian-overrides` pero que se emite debido a la opción `--show-overrides`.

En el caso de los errores (líneas que comienzan por E:), lee la explicación (líneas N:), cambia el paquete para eliminarlos o verifica que los avisos son falsos. En este caso, genera los archivos `lintian-overrides` como se ha descrito en Sección 5.14.

Observa que puedes construir el paquete con **dpkg-buildpackage** y ejecutar **lintian** todo con sólo una orden si utilizas debuild(1) o pdebuild(1).

[1] No es necesario añadir la opción `-i -I --show-overrides` a la orden **lintian** si la has incluido en la configuración en `/etc/devscripts.conf` o `~/.devscripts` según se explicó en Sección 6.3 .

7.5. La orden debc

Puedes ver la lista de archivos del paquete binario Debian ejecutando la orden debc(1) como sigue:

```
$ debc nombre_del_paquete.changes
```

7.6. La orden debdiff

Puedes comparar el contenido de dos paquetes de fuentes Debian ejecutando la orden debdiff(1) como sigue:

```
$ debdiff versión_anterior.dsc nueva_versión.dsc
```

Puedes comparar la lista de ficheros de dos paquetes binarios de Debian con la orden debdiff(1) ejecutando la orden como sigue:

```
$ debdiff versión_anterior.changes nueva_versión.changes
```

Este programa es útil para verificar que no hay ficheros que se hayan cambiado de sitio o eliminado por error, y que no se ha realizado ningún otro cambio no deseado al actualizar el paquete.

7.7. La orden interdiff

Puedes comparar dos ficheros `diff.gz` con la orden interdiff(1). Esto es muy útil para verificar que no se han realizado cambios inadvertidos por el responsable del paquete al actualizar el paquetes que se han construido con el formato `1.0`. Ejecuta lo siguiente:

```
$ interdiff -z versión_anterior.diff.gz nueva_versión.diff.gz
```

El nuevo formato de fuentes `3.0` conserva los cambios en varios archivos de parches («.patch») como se describe en Sección 5.25. Puedes seguir los cambios realizados por cada archivo `debian/patches/*` utilizando la orden **interdiff**.

7.8. La orden mc

Algunas de las operaciones de comprobación del paquete descritas puede realizarse de forma muy intuitiva si empleamos un gestor de ficheros como mc(1), que permite visionar tanto el contenido del paquete `*.deb`, como el de los ficheros `*.udeb`, `*.debian.tar.gz`, `*.diff.gz`, `*.orig.tar.gz`.

Vigila que no haya ficheros innecesarios extra o de tamaño cero, tanto en el binario como en el paquete fuente. A veces, hay cosas que no se limpiaron adecuadamente, debes ajustar tu fichero `rules` para arreglar esto.

Capítulo 8

Actualizar el paquete

Después del lanzamiento del paquete, es posible que debas actualizarlo pronto.

8.1. Nueva revisión Debian del paquete

Supongamos que se ha creado un informe de fallo en tu paquete con el número `#654321`, y que describe un problema que puedes solucionar. Para construir una nueva revisión del paquete, necesitas:

- Si debes aplicar una modificación nueva, ejecuta:
  - `dquilt new` *`nombre_modificación.patch`* para establecer el nombre de la modificación;
  - `dquilt add` *`archivo_a_modificar`* para establecer el fichero al cual se aplicará la modificación.
  - Corregir el problema en el archivo original.
  - `dquilt refresh` para guardar los cambios realizados en el archivo del parche *`nombre_modificación.patch`*.
  - `dquilt header -e` para añadir la descripción (breve) del cambio realizado;
- Si debes actualizar una modificación ya existente, ejecuta:
  - `dquilt pop` *`nombre_modificación.patch`* para deshacer el parche *`nombre_modificación.patch`* que debes actualizar (puesto que se habrá ejecutado y se supone que es necesario modificarlo).
  - Corregir el problema existente en la versión incorrecta del archivo de parche *`nombre_modificación.patch`*.
  - `dquilt refresh` para actualizar *`nombre_modificación.patch`*.
  - `dquilt header -e` para actualizar la descripción en la cabecera del archivo del parche.
  - `while dquilt push; do dquilt refresh; done` para aplicar todos los parches eliminando *cosas innecesarias*;
- Añadir la información de la revisión en el inicio del archivo `changelog` (del directorio «Debian»), por ejemplo ejecutando `dch -i` o explícitamente indicando el número de versión y revisión ejecutando `dch -v` *`versión-revisión`*, y a continuación detallar los cambios realizados utilizando un editor [1].
- Incluye la descripción (breve) del error y la solución, seguida de la referencia de la notificación del error con `(Closes: #654321)`. De esta manera, el informe de error sera «cerrado» *automáticamente* por el sistema de mantenimiento del repositorio Debian cuando el paquete sea aceptado en el repositorio.
- Deberás repetir los pasos anteriores para cada una de las modificaciones realizadas en la actualización del paquete, a la par que actualizas el fichero `changelog` de Debian mediante `dch`.

[1] Para escribir la fecha y hora en el formato requerido, debes utilizar `LANG=C date -R`.

- Repite lo que hiciste en Sección 6.1 y Capítulo 7.
- Una vez que estes satisfecho, cambia el nombre de la distribución en el archivo `changelog` de `UNRELEASED` a `unstable` (o bien `experimental`).[2]
- Sube el paquete como se ha descrito en Capítulo 9. La diferencia es que esta vez, el fichero fuente original no se incluye, ya que no se ha cambiado y ya esta en el archivo de Debian.

Uno de los casos difíciles sucede cuando haces una copia local del paquete para realizar pruebas antes de subir la versión definitiva al archivo oficial, por ejemplo,*`1.0.1-1`*. Para una actualización menor, es una buena idea documentar una entrada en el archivo `changelog` con una cadena de código de versión como *`1.0.1-1~rc1`*. Es posible ordenar el archivo `changelog` mediante la consolidación de tales entradas «de pruebas» en una única entrada para el paquete oficial. Véase Sección 2.6 para el orden de las cadenas de versión.

8.2. Inspección de una nueva versión del autor

Para la actualización de un paquete cuando el autor original libera una nueva versión de las fuentes, debes empezar por revisar la nueva versión original.

Empieza por leer los archivos `changelog`, `NEWS` y cualquier otra documentación donde el autor original describa los cambios de la nueva versión.

Puedes comprobar los cambios entre las fuentes originales de la nueva versión y de la anterior para detectar cualquier cambio sospechoso de producir errores ejecutando:

```
$ diff -urN nombre_archivo-versión_anterior nombre_archivo-nueva_versión
```

Las modificaciones realizadas en los archivos generados por «Autotools» (`missing`, `aclocal.m4`, `config.guess`, `config.h.in`, `config.sub`, `configure`, `depcomp`, `install-sh`, `ltmain.sh` y `Makefile.in`) puedes ignorarlas. Puedes eliminarlos antes de ejecutar **diff** en las fuentes para inspeccionarlas.

8.3. Nueva versión del programa fuente

Si el paquete *`nombre_del_paquete`* que examinas está correctamente empaquetado utilizando los nuevos formatos `3.0 (native)` o `3.0 (quilt)` para empaquetar una nueva versión del autor es esencial copiar el directorio `debian` de la versión anterior a la nueva, para a continuación, realizar las adaptaciones necesarias. Puedes copiar el directorio `debian` de la versión anterior a la nueva versión ejecutando `tar xvzf /ruta/a/nombre_del_paquete_versión_anterior.debian.tar.gz` desde el directorio de las fuentes de la nueva versión [3]. A continuación deberás realizar algunos tareas obvias:

- Comprimir las fuentes originales en el archivo `nombre_del_paquete_`*`nueva_versión`*`.orig.tar.gz`.
- Actualizar el archivo `changelog` Debian ejecutando `dch -v` *`nueva_versión-1`* .
  - Añade una nueva linea con el texto «New upstream release» para indicar que se trata de una nueva versión de las fuentes originales.
  - Describe sucintamente los cambios realizados *en las fuentes originales por el autor* que solucionan errores informados y cerrar los informes añadiendo `Closes:` *`#numero_del_error`*.
  - Describe sucintamente los cambios de la *nueva versión* del desarrollador que solucionan errores previamente reportados y cierra dichos errores añadiendo `Closes:` *`#número_del_error`*.
- `while dquilt push; do dquilt refresh; done` para aplicar todos los parches eliminando *cosas innecesarias*;

[2] Si utiliza la orden `dch -r` para realizar este último cambio, asegúrese que guarda el archivo `changelog` explícitamente con el editor.

[3] Si el paquete *`nombre_del_paquete`* está construido con el anterior formato `1.0`, esto se puede hacer ejecutando `zcat /ruta/a/nombre_del_paquete_numero_de_versión_anterior.diff.gz|patch -p1` en la nueva versión de las fuentes.

Si las modificaciones no se ejecutan correctamente, inspecciona la situación (mira la información de los archivos `.rej`) como se muestra a continuación.

- Si uno de los parches aplicados está integrado en las fuentes originales,
  - ejecuta `dquilt delete` para eliminarlo.
- Si uno de los parches entra en conflicto con los cambios realizados por el autor en las fuentes originales,
  - ejecuta `dquilt push -f` para aplicar los parches de la versión anterior para forzar los rechazos (tendrás la información de los rechazos en los archivos *`rechazo`*`.rej`).
  - Edita los archivos *`rechazo`*`.rej` manualmente para saber el efecto que se pretende con *`rechazo`*`.rej`.
  - Ejecuta `dquilt refresh` para actualizar el parche.
- Continua hasta la ejecución de `while dquilt push; do dquilt refresh; done`.

Puedes automatizar este proceso utilizando la orden uupdate(1) como sigue:

```
$ apt-get source nombre_del_paquete
...
dpkg-source: info: extracting nombre_del_paquete in nombre_del_paquete-versión_anterior
dpkg-source: info: unpacking nombre_del_paquete_versión_anterior.orig.tar.gz
dpkg-source: info: applying nombre_del_paquete_versión_anterior-1.debian.tar.gz
$ ls -F
nombre_del_paquete-versión_anterior/
nombre_del_paquete_versión_anterior-1.debian.tar.gz
nombre_del_paquete_versión_anterior-1.dsc
nombre_del_paquete_versión_anterior.orig.tar.gz
$ wget http://ejemplo.org/nombre_del_paquete/nombre_del_paquete-nueva_versión.tar.gz
$ cd nombre_del_paquete-versión_anterior
$ uupdate -v nueva_versión ../nombre_del_paquete-nueva_versión.tar.gz
$ cd ../nombre_del_paquete-nueva_versión
$ while dquilt push; do dquilt refresh; done
$ dch
... documenta las modificaciones realizadas
```

Si has configurado el archivo «`debian/watch`» como se ha descrito en Sección 5.21 ,puedes saltarte la orden **wget**. Simplemente, ejecuta uscan(1) en el directorio *`nombre_del_paquete-antigua_versión`* en lugar de la orden **uupdate**. Así, se buscará *automáticamente* el archivo de las fuentes, se descargará en tu ordenador y se ejecutará la orden **uupdate** [4].

Puedes liberar la nueva versión del paquete repitiendo lo expuesto en Sección 6.1 , Capítulo 7 y Capítulo 9 .

8.4. Actualizar el formato del paquete

Para actualizar un paquete no es necesario actualizar el formato del paquete. Aún así, puedes aprovechar completamente la funcionalidad de `debhelper` y del formato `3.0` haciendo lo siguiente [5]:

- Si necesitas de nuevo algunos de los archivos de plantilla eliminados, puedes regenerarlos ejecutando otra vez **dh_make** con la opción `--addmissing` en el directorio de las fuentes. A continuación modifícalos correctamente.
- Si el paquete no está actualizado para utilizar la nueva sintaxis de la versión 7+ de la orden **dh** de `debhelper` en el archivo `debian/rules`, actualízalo para usar **dh**. También deberás actualizar `debian/control`.

[4] Si la orden **uscan** descargar las fuentes pero no ejecuta la orden **uupdate**, debes corregir el archivo `debian/watch` añadiendo `debian uupdate` al final de la URL del archivo.

[5] Si quien patrocina tu paquete u otros desarrolladores hacen objeciones a la actualización del formato del paquete, no vale la pena empeñarse en argumentar a favor. Hay otras cosas más importantes que atender.

- Si deseas actualizar el archivo `rules` construido por el mecanismo de inclusión `Makefile` del sistema de compilación Debian (`cdbs`) a la nueva sintaxis **dh**, lee el siguiente documento para comprender las variables de configuración `DEB_*`.
  - copia local de `/usr/share/doc/cdbs/cdbs-doc.pdf.gz`
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)
- Si estás trabajando con un paquete construido con el formato `1.0` sin el archivo *`nombre_del_paquete`*`.diff.gz`, puedes actualizarlo a la nueva versión `3.0 (native)` añadiendo el archivo `debian/source/format` con la linea `3.0 (native)`. Copia los otros archivos del directorio `debian/*`.
- Si estás trabajando con un paquete construido con el formato `1.0` con el archivo *`nombre_del_paquete`*`.diff.gz`, puedes actualizarlo a la nueva versión `3.0 (native)` añadiendo el archivo `debian/source/format` con la linea `3.0 (native)`. Copia los otros archivos del directorio `debian/*`. Importa el archivo `nombre_del_paquete.diff` generado por la orden `filterdiff -z -x '*/debian/*'` *`nombre_del_paquete`*`.diff.gz > nombre_del_paquete.diff` al sistema **quilt** [6].
- Si el paquete se ha construido utilizando un sistema de parches distinto como `dpatch`, `dbs` o `cdbs` utilizando las opciones `-p0`, `-p1` o `-p2`, puedes convertirlo al formato `quilt` utilizando el guión `deb3` explicado en http://bugs.debian.org/581186.
- Si el paquete se ha construido ejecutando la orden **dh** con la opción `--with quilt` o bien con **dh_quilt_patch** y **dh_quilt_unpatch**, elimina todo esto y utiliza el nuevo formato fuente `3.0 (quilt)`.

Consulte DEP - Debian Enhancement Proposals (http://dep.debian.net/) y adoptar las propuestas ACEPTADAS.

Repasa la sección Sección 8.3 por si debes repetir algunos de los pasos indicados en ella.

8.5. Conversión a UTF-8

Si los documentos originales utilizan una codificación antigua, es una buen práctica actualizarlos a UTF-8.

- Utiliza iconv(1) para hacer conversiones de codificación en ficheros de texto sin formato.

```
iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

- Utiliza w3m(1) para conversiones desde ficheros HTML a ficheros de texto sin formato con codificación UTF-8. Al hacerlo, el sistema debe estar configurado para usar la codificación UTF-8.

```
LC_ALL=C.UTF-8 w3m -o display_charset=UTF-8 \
        -cols 70 -dump -no-graph -T text/html \
        < nombre_del_archivo_a_modificar.html > nombre_del_archivo_modificado.txt
```

8.6. Recordatorio para actualizar paquetes

A continuación se listan algunas cosas a tener en cuenta al actualizar paquetes:

- Conserva las entradas anteriores del archivo `changelog` (suena a obviedad, pero se han dado casos de ejecutar `dch` en lugar de `dch -i`).
- Los cambios en la construcción del paquete Debian deben ser reconsiderados; elimina las modificaciones anteriores (sea lo que sea) y recuerda de añadir todo lo necesario, a no ser que haya una buena razón para no hacerlo.

[6] Puedes fragmentar el archivo `nombre_del_paquete.diff` en varios archivos de parches utilizando la orden **splitdiff**.

- Si se ha realizado alguna modificación en la compilación (te enterarás al inspeccionar los cambios en las fuentes originales) puede que sea necesario actualizar el archivo `debian/rules` y las dependencias de compilación en el archivo `debian/control`.
- Debes comprobar si hay alguna comunicación de parches del paquete en el sistema de gestión de errores (puede darse el caso que algún usuario envíe un parche ya construido y que te sea de utilidad) en Debian Bug Tracking System (BTS) (http://www.debian.org/Bugs/) .
- Comprueba el contenido del archivo `.changes` para asegurarte que envías el paquete a la distribución correcta, que los informes de errores que se cierran con la nueva versión del paquete están listados en el campo `Closes` del archivo, que el contenido de los campos `Maintainer` y `Changed-By` son correctos, que has firmado el archivo con tu clave GPG, etc.

Capítulo 9

Enviar el paquete

Ahora que has probado tu nuevo paquete en profundidad, podrás transferirlo a un archivo público para compartirlo.

9.1. Enviar al repositorio de Debian

Cuando seas desarrollador oficial Debian [1], podrás transferir el paquete al repositorio de Debian [2]. Puedes hacer esto manualmente, pero es más fácil hacerlo con las herramientas automáticas ya disponibles como dupload(1) o dput(1). A continuación describiremos cómo hacerlo con **dupload** [3].

En primer lugar, debes generar un fichero de configuración de **dupload**. Puedes hacerlo editando el fichero general del sistema `/etc/dupload.conf`, o generando tu propio fichero `~/.dupload.conf` con lo que tu quieras cambiar.

Consulta el manual de dupload.conf(5) para saber el significado de cada opción.

La opción `$default_host` es la más problemática; determina cuál de las colas de envíos se usará por omisión. «`anonymous-ftp-master`» es la primaria, pero es posible que quieras usar otra más rápida [4].

Si tienes conexión a Internet, puedes subir el paquete de la siguiente manera:

```
  dupload gentoo_0.9.12-1_i386.changes
```

dupload comprueba que las sumas MD5/SHA1/SHA256 de los archivos transferidos coinciden con las listadas en el fichero `.changes`, si no coinciden te avisará para que reconstruyas el paquete como se describe en Sección 6.1 para poder enviarlo correctamente.

Si encuentras algún problema con la subida del paquete a ftp://ftp.upload.debian.org/pub/UploadQueue/, puedes arreglarlo subiendo manualmente un fichero `*.commands` firmado con GPG con la orden **ftp** [5]. Por ejemplo, utiliza `hello.commands`:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
 rm hello_1.0-1_i386.deb
 mv hello_1.0-1.dsx hello_1.0-1.dsc
```

[1] Consulta Sección 1.1.

[2] Hay archivos públicamente accesibles, como http://mentors.debian.net/ que funcionan de forma similar al archivo Debian y facilitan un sistema de transferencia de paquetes para personas que no son desarrolladores oficiales («DD»). Tu puedes construirte un archivo personal utilizando las herramientas listadas en http://wiki.debian.org/HowToSetupADebianRepository. Así, esta sección también es útil si no eres «DD».

[3] Consulta Sección 1.1.

[4] Véase Debian Developer's Reference 5.6. "Uploading a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#upload) .

[5] Consulta ftp://ftp.upload.debian.org/pub/UploadQueue/README. Como alternativa, puedes usar la orden **dcut** del paquete `dput`.

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

[...]
-----END PGP SIGNATURE-----
```

9.2. Incluir `orig.tar.gz` para la transferencia del paquete al repositorio.

Cuando se envía por primera vez al archivo, se debe incluir el archivo `orig.tar.gz` con las fuentes originales. Si el número de revisión Debian del paquete no es ni `1` ni `0`, debes ejecutar la orden **dpkg-buildpackage** con la opción `-sa`. Por otra parte, es posible excluir la inclusión del archivo fuente original `orig.tar.gz` con la opción `-sd`.

Para la orden **dpkg-buildpackage**:

```
$ dpkg-buildpackage -sa
```

Para la orden **debuild**:

```
$ debuild -sa
```

Para la orden **pdebuild**:

```
$ pdebuild --debbuildopts -sa
```

Por otra parte, con la opción `-sd` se evita la inclusión del archivo `orig.tar.gz` con las fuentes originales.

9.3. Envíos discontinuados

Si en el fichero `debian/changelog` hay diversas entradas correspondientes a diferentes versiones del paquete, pero alguna de ellas no se ha enviado al repositorio, deberás generar correctamente un fichero `*_.changes` que incluya todas las modificaciones desde la última versión presente en el repositorio. Puedes hacerlo ejecutando la orden **dpkg-buildpackage** con la opción `-v` seguida de la versión (p. ej. *`1.2`*).

Para la orden **dpkg-buildpackage**:

```
$ dpkg-buildpackage -v1.2
```

Para la orden **debuild**:

```
$ debuild -v1.2
```

Para la orden **pdebuild**:

```
$ pdebuild --debbuildopts "-v1.2"
```

Apéndice A

Técnicas avanzadas

Éstos son algunos consejos e indicaciones sobre aspectos avanzados del mantenimiento de paquetes que pueden ser útiles. Se recomienda encarecidamente que leas todas las referencias sugeridas aquí.

Puede ser necesario editar manualmente los ficheros de las plantillas generadas con la orden **dh_make** para abordar los temas tratados en este capítulo. La nueva orden **debmake** maneja mejor estos aspectos de la construcción de paquetes.

A.1. Bibliotecas compartidas

Antes de empaquetar bibliotecas compartidas, debes leer atentamente la siguientes referencias básicas:

- Debian Policy Manual, 8 ”Shared libraries” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html)
- Debian Policy Manual, 9.1.1 ”File System Structure” (http://www.debian.org/doc/debian-policy/ch-opersys.html#s-fhs)
- Debian Policy Manual, 10.2 ”Libraries” (http://www.debian.org/doc/debian-policy/ch-files.html#s-libraries)

Éstos son algunos consejos básicos para empezar:

- Las bibliotecas compartidas son archivos objeto en formato ELF que contienen código compilado.
- Las bibliotecas compartidas se distribuyen como ficheros `*.so` (no como ficheros `*.a` o `*.la`).
- Las bibliotecas compartidas se utilizan principalmente para compartir código común entre varios ejecutables con la orden **ld**.
- Las bibliotecas compartidas se utilizan, a veces, para proveer varios complementos («plugins») a un ejecutable mediante el procedimiento **dlopen**.
- Las bibliotecas compartidas exportan símbolos que representan objetos compilados como variables, funciones y clases; y permite acceder a ellos desde los ejecutables enlazados.
- El SONAME (el nombre lógico) de la biblioteca compartida `lib`*`nombre_biblioteca`*`.so.`*`1`*: `objdump -p lib`*`nombre_biblioteca`*`.so.`*`1`*` | grep SONAME` [1]
- El «SONAME» (el nombre lógico) de una biblioteca compartida generalmente coincide con el nombre del archivo de biblioteca (pero no siempre).
- El «SONAME» (el nombre lógico) de las bibliotecas compartidas enlazadas a */usr/bin/foo*: `objdump -p` */usr/bin/foo* `| grep NEEDED` [2]

[1] Como alternativa: `readelf -d lib`*`nombre_biblioteca`*`.so.`*`1`*` | grep SONAME`
[2] Como alternativa: `readelf -d lib`*`foo`*`.so.`*`1`*` | grep NEEDED`

- `lib`*`foo1`*: el paquete de biblioteca de la biblioteca compartida `lib`*`foo`*`.so.`*`1`* con la versión ABI del nombre lógico («SONAME») *`1`*.[3]
- Los guiones de desarrollador de un paquete de biblioteca deben ejecutar **ldconfig** cuando sea necesario para generar los enlaces simbólicos para el «SONAME» (el nombre lógico).[4]
- `lib`*`foo1`*`-dbg`: el paquete de símbolos de depuración que contiene los símbolos de depuración del paquete de la biblioteca compartida `lib`*`foo1`*.
- `lib`*`foo`*`-dev`: el paquete de desarrollo que contiene los ficheros de cabeceras y otros de la biblioteca compartida `lib`*`foo`*`.so.`*`1`*.[5]
- En general, los paquetes Debian no deben contener ficheros «Libtool» `*.la`.[6]
- En general, en los paquetes Debian no debe utilizarse «RPATH».[7]
- Aunque está un poco anticuado y es sólo una referencia secundaria, Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) aún puede ser útil.

A.2. Gestionando `debian/`*`package`*`.symbols`

Cuando empaquetes bibliotecas compartidas, debes generar el fichero `debian/`*`nombre_del_paquete`*`.symbols` para gestionar la versión mínima asociada a cada símbolo para cambios ABI compatibles con versiones anteriores con el mismo «SONAME» (nombre lógico) de la biblioteca para el mismo nombre de paquete de biblioteca compartida.[8] Se aconseja la lectura de las siguientes referencias básicas:

- Debian Policy Manual, 8.6.3 "The symbols system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-symbols) [9]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

Aquí hay un ejemplo para generar el paquete `libfoo1` de la versión `1.3` del autor original con el fichero `debian/libfoo1.symbols` adecuado:

- Preparar el esqueleto de directorios fuente Debian utilizando el fichero con las fuentes `libfoo-1.3.tar.gz`.
  - Si es la primera versión del paquete `libfoo1`, genera un fichero `debian/libfoo1.symbols` en blanco.
  - Si la versión anterior del autor (la `1.2`) fue empaquetada en el paquete `libfoo1` con el fichero `debian/libfoo1.symbols` adecuado en el paquete fuente, utilízalo otra vez.
  - Si la versión anterior del autor (la `1.2`) no se empaquetó con el fichero `debian/libfoo1.symbols`, genera el fichero `symbols` a partir de todos los paquetes binarios disponibles con el mismo nombre de paquete de biblioteca compartida que compartan el mismo «SONAME» (el nombre lógico) de la biblioteca; por ejemplo las versiones `1.1-1` y `1.2-1`. [10]

[3] Véase Debian Policy Manual, 8.1 "Run-time shared libraries" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) .
[4] Véase Debian Policy Manual, 8.1.1 "ldconfig" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) .
[5] Consulta Debian Policy Manual, 8.3 "Static libraries" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-static) y Debian Policy Manual, 8.4 "Development files" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-dev) .
[6] Consulta Debian wiki ReleaseGoals/LAFileRemoval (http://wiki.debian.org/ReleaseGoals/LAFileRemoval) .
[7] Consulta Debian wiki RpathIssue (http://wiki.debian.org/RpathIssue) .
[8] Los cambios ABI incompatibles con versiones anteriores, normalmente requieren actualizar el «SONAME» (el nombre lógico) de la biblioteca y el de la biblioteca compartida a otros nuevos.
[9] Para bibliotecas C++ y otros casos en los que el seguimiento individual de símbolos es difícil, es mejor consultar Debian Policy Manual, 8.6.4 "The shlibs system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) .
[10] Las versiones previas de los paquetes Debian están disponibles en http://snapshot.debian.org/ (http://snapshot.debian.org/) . La revisión Debian del paquete sigue a la versión para facilitar mantenimiento de versiones anteriores («backport») del paquete: `1.1` << `1.1-1~bpo70+1` << `1.1-1` y `1.2` << `1.2-1~bpo70+1` << `1.2-1`

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

- Ejecuta compilaciones de prueba del árbol de directorios de las fuentes con órdenes como **debuild** y **pdebuild**. Si fallan debido a símbolos perdidos u otras causas, busca cambios ABI incompatibles con versiones anteriores que requieran cambios en el nombre del paquete de biblioteca compartida a algo del tipo `libnombrebiblioteca1a` y vuelta a empezar.

```
$ cd libfoo-1.3
$ debuild
...
dpkg-gensymbols: advertencia: hay símbolos nuevos en el fichero de símbolos: ...
 mire las diferencias a continuación
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx        2012-11-11 02:24:53.609667389 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
  foo_get_type@Base 1.1
+ foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...
```

- Si miras el informe de cambios generado a continuación por la orden **dpkg-gensymbols**, lista el fichero `symbols` actualizado adecuadamente para el paquete binario generado de la biblioteca compartida. [11]

```
$ cd ..
$ dpkg-deb -R  libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1\.3-1/1\.3/' libfoo1-tmp/DEBIAN/symbols \
       >libfoo-1.3/debian/libfoo1.symbols
```

- Compilar paquetes para publicarlos con herramientas como **debuild** y **pdebuild**.

```
$ cd libfoo-1.3
$ debuild clean
$ debuild
...
```

Además de los ejemplos anteriores, también debes comprobar la compatibilidad ABI con más atención y actualizar manualmente las versiones de los símbolos (si es necesario). [12]

Aunque es sólo una referencia secundaria, Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) y sus enlaces a otras páginas web puede ser útil.

A.3. Varias arquitecturas

La función de varias arquitecturas introducida en Debian «wheezy» integra la instalación en más de una arquitectura de los paquetes binarios (en particular `i386`<->`amd64`, pero también con otras combinaciones) en `dpkg` y `apt`. Se recomienda leer atentamente las siguientes referencias:

[11] La revisión de Debian se deriva de la versión para hacer más fácil el mantenimiento de versiones anteriores («backport») del paquete: `1.3` << `1.3-1~bpo70+1` << `1.3-1`

[12] Véase Debian Policy Manual, 8.6.2 "Shared library ABI changes" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-updates) .

- Ubuntu wiki MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) (desarrollador original)
- Debian wiki Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) (Debian)

Se utilizan tripletes del tipo `i386-linux-gnu` y `x86_64-linux-gnu` para el directorio de instalación de bibliotecas compartidas. El triplete de trabajo se establece dinámicamente al valor `$(DEB_HOST_MULTIARCH)` por dpkg-architecture(1) para cada compilación. Por ejemplo, el directorio de instalación de bibliotecas para varias arquitecturas se cambia como sigue:[13]

Directorio antiguo	directorio multi-arquitectura i386	directorio multi-arquitectura amd64
`/lib/`	`/lib/i386-linux-gnu/`	`/lib/x86_64-linux-gnu/`
`/usr/lib/`	`/usr/lib/i386-linux-gnu/`	`/usr/lib/x86_64-linux-gnu/`

Estos son algunos ejemplos típicos de casos posibles de paquetes para varias arquitecturas para los siguientes paquetes:

- el código fuente de la biblioteca `lib`*`foo`*`-1.tar.gz`
- el código fuente de una orden *`bar`*`-1.tar.gz` escrito en un lenguaje compilado
- el código fuente de una orden *`baz`*`-1.tar.gz` escrito en un lenguaje interpretado

Paquete	Arquitectura:	Multi-arquitectura:	Contenido del paquete
`lib`*`foo1`*	any	same	la biblioteca compartida, coinstalable
`lib`*`foo1`*`-dbg`	any	same	los símbolos de depuración de la biblioteca compartida, coinstalable
`lib`*`foo`*`-dev`	any	same	los ficheros de cabeceras y otros de una biblioteca compartida, coinstalable
`lib`*`foo`*`-tools`	any	foreign	los programas de soporte en tiempo de ejecución no son co-instalables.
`lib`*`foo`*`-doc`	all	foreign	los ficheros de documentación de la biblioteca compartida
`bar`	any	foreign	los ficheros compilados del programa, no son coinstalables
`bar``-doc`	all	foreign	los ficheros de documentación del programa
`baz`	all	foreign	los ficheros de programa interpretados

Hay que tener en cuenta que el paquete de desarrollo debe contener un enlace simbólico a la biblioteca compartida asociada **sin el número de versión**. P. ej.: `/usr/lib/x86_64-linux-gnu/libfoo.so` -> `libfoo.so.1`

A.4. Construcción de un paquete de biblioteca compartida

Puedes construir un paquete de biblioteca Debian con la opción multi-arquitectura activada utilizando dh(1) como sigue:

- Actualizar `debian/control`.
  - Añadir `Build-Depends: debhelper (>=9)` en la sección del paquete fuente.
  - Añadir `Pre-Depends: ${misc:Pre-Depends}` por cada paquete binario de biblioteca compartida.
  - Añadir el campo `Multi-Arch:` en cada sección de paquete binario.
- Establecer `debian/compat` a « 9 ».
- Cambiar el directorio habitual `/usr/lib/` por el directorio multi-arquitectura `/usr/lib/$(DEB_HOST_MULTIARCH)/` para todos los guiones de empaquetado.

[13] Viejas ubicaciones especiales para bibliotecas como `/lib32/` y `/lib64/` ya no se utilizaran más.

- Añadir (primero) `DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` en `debian/rules` para establecer la variable `DEB_HOST_MULTIARCH`.
- Reemplazar `/usr/lib/` por `/usr/lib/$(DEB_HOST_MULTIARCH)/` en `debian/rules`.
- Si se utiliza `./configure` como parte del objetivo `override_dh_auto_configure` en el fichero `debian/rules`, hay que reemplazarlo por `dh_auto_configure -- .`[14]
- Cambiar todas referencias a `/usr/lib/` por `/usr/lib/*/` en los ficheros `debian/`*`nombre_del_paquete`*`.install`
- Generar ficheros como `debian/`*`nombre_del_paquete`*`.links` desde `debian/`*`nombre_del_paquete`*`.links.in` dinámicamente añadiendo un guión al objetivo `override_dh_auto_configure` en `debian/rules`.

```
override_dh_auto_configure:
        dh_auto_configure
        sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
                debian/nombre_del_paquete.links.in > debian/nombre_del_paquete.links
```

Debes comprobar que el paquete de biblioteca compartida solo contiene los ficheros esperados y que el paquete «-dev» sigue funcionando correctamente.

Todos los ficheros instalados simultáneamente por el paquete multi-arquitectura en el mismo directorio deben contener los mismos ficheros. Debes tener cuidado en las diferencias generadas por el orden de los bits de datos y por el algoritmo de compresión.

A.5. Paquete nativo Debian

Si un paquete se mantiene exclusivamente para Debian o para uso local, su paquete fuente puede contener todos los ficheros `debian/*`. Hay dos maneras para empaquetarlos.

Puedes excluir los ficheros de `debian/*` al generar el fichero comprimido con los ficheros fuente y construir el paquete como un paquete Debian no nativo como se explica en Sección 2.1. Algunas personas animan a utilizar este esquema de trabajo.

La alternativa es el esquema de trabajo para paquetes nativos Debian.

- Generaremos un paquete fuente Debian en el formato `3.0 (native)`, utilizando un archivo comprimido en formato «tar» que incluirá todos los archivos.
  - *`nombre_del_paquete_versión`*`.tar.gz`
  - *`nombre_del_paquete_versión`*`.dsc`
- Construiremos un paquete binario Debian del paquete de fuentes nativo Debian.
  - *`nombre_del_paquete_versión_arquitectura`*`.deb`

Por ejemplo, si los ficheros fuente están en `~/mi_paquete-1.0` sin los ficheros del directorio `debian/*`, puedes construir un paquete Debian nativo ejecutando la orden **dh_make** como sigue:

```
$ cd ~/mi_paquete-1.0
$ dh_make --native
```

Entonces el directorio `debian` y su contenido son generados como en Sección 2.8. De esta manera no se genera un archivo «tarball» ya que este es un paquete Debian nativo. Pero esa es la única diferencia. El resto de las actividades de empaquetado son prácticamente las mismas.

Después de la ejecución de la orden **dpkg-buildpackage**, encontrarás los siguientes ficheros en el directorio superior:

[14] Como alternativa, puedes añadir los argumentos `--libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` y `--libexecdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` en `./configure`. Fíjate que `--libexecdir` especifica el directorio predeterminado para la instalación de programas ejecutables que son ejecutados por otros programas en lugar de por los usuarios. El valor predeterminado por «Autotools» es `/usr/libexec/` pero en Debian es `/usr/lib/`.

- `mi_paquete_1.0.tar.gz`

  Este es el archivo del código fuente generado a partir del directorio `mi_paquete-1.0` por la orden **dpkg-source** (su sufijo no es `orig.tar.gz`.).

- `mi_paquete_1.0.dsc`

  Este es el sumario del contenido del código fuente en el caso de los paquetes Debian no nativos (no tiene código de revisión Debian).

- `mi_paquete_1.0_i386.deb`

  Este es el paquete binario completo en el caso de los paquetes Debian no nativos (no tiene código de revisión Debian).

- `mi_paquete_1.0_i386.changes`

  Este archivo describe todos los cambios realizados en el versión actual del paquete en el caso de los paquetes Debian no nativos (no tiene código de revisión Debian).