

Desarrollo y distribución de Software Libre en Ubuntu GNU/Linux

Denis Fuenzalida

Desarrollo y distribución de Software Libre en Ubuntu GNU/Linux

por Denis Fuenzalida

Resumen

Este documento intentará ilustrar y resumir una gran cantidad de conocimientos más o menos dispersos que encontré mientras aprendía a desarrollar Software Libre en Linux, y en especial mientras desarrollaba software utilizando Ubuntu GNU/Linux, que es una distribución muy popular basada en Debian.

El texto pretende servir como una colección de ejemplos, recetas y atajos a una serie de problemas que un programador puede encontrar al hacer un esfuerzo adicional para empaquetar y distribuir la aplicación para alcanzar un universo potencialmente mayor de usuarios.

Finalmente, este documento también me sirve para agradecer a una enorme cantidad de personas, muchas de las cuales han trabajado durante años en uno de los esfuerzos que más admiro: crear una enorme colección de Software Libre que puede ser utilizado universalmente y sin restricciones. Mis respetos para todos ustedes.

Tabla de contenidos

1. Introducción	1
2. Desarrollo con Python y Glade	3
Introducción	3
Diseño de una interfaz de usuario con Glade	3
Soporte multilinguaje en la interfaz de usuario	7
3. Empaquetado y distribución de software para Ubuntu	11
Entendiendo los paquetes de software	11
Versiones de un elemento de software	11
Dependencias de un elemento de software	11
Creación de un paquete de software Debian	11
Creación de una firma digital con GnuPG	12
Agregar la llave pública a tus proveedores de software de confianza	13
Creación de un paquete Debian	14
Creación de un paquete Debian con Java, Ant y CDBS	19
Creación de un repositorio sencillo	22
Soporte para Internacionalización en los repositorios	24

Capítulo 1. Introducción

De acuerdo a la literatura, “todo buen trabajo de software comienza a partir de las necesidades personales del programador”, así que vamos a comenzar con una declaración de los supuestos con los que vamos a trabajar en esta misma línea:

- Tenemos un problema a resolver,
- Tenemos el convencimiento de poder resolver el problema mediante herramientas de Software Libre,
- Vamos a liberar nuestra solución a nuestro problema, pensando que puede servir a otros y pensando que eventualmente podemos recibir colaboraciones de otros.

Para efectos de este documento, el problema que nos interesa resolver será crear una aplicación sencilla que permita dividir archivos de gran tamaño en trozos para copiar mediante algún medio entre dos computadores.

Mi problema original surgió debido a que uso un equipo para bajar archivos grandes desde Internet, y me interesa copiar los archivos a otro PC que no tiene conexión. Generalmente son archivos ISO de distribuciones de Linux, pero también pueden ser archivos MP3 de gran tamaño, etc. Tengo pendrives USB de 128 y 256 megabytes para transportar archivos, pero un archivo ISO tiene sobre 700 megabytes y aunque podría grabarlo en un CD-RW, hacerlo es muy lento.

En la actualidad existen programas que realizan esto mismo, como el programa `split` que funciona en la línea de comandos, pero me interesa crear una aplicación de escritorio que me permita hacer esto mismo de una manera intuitiva y que me permitiera validar de alguna manera sencilla que al cortar el archivo en trozos y volverlo a unir se obtenga el mismo archivo original (sin corrupción de los datos).

Entre las alternativas para construir el software en mi sistema operativo consideré los siguientes lenguajes de programación: C, Perl, Python, C# y Java.

Descarté el lenguaje C desde el principio porque llevo demasiado tiempo sin programar en C y recientemente tuve que ayudar a un par de familiares que estudian informática con sus tareas y no iba a avanzar rápido si ni siquiera podía recordar bien la notación para hacer una lista enlazada. Quizás vale la pena reconsiderar volver a programar en C si el rendimiento con cualquier otra alternativa es muy malo, pero quizás sólo alguna sección crítica.

Descarté programar en Perl porque no lo uso hace mucho tiempo y su sintaxis es desagradable. Cuesta demasiado seguir un programa después de una semana de haberlo escrito. Fin del asunto.

Descarté C# por falta de conocimientos, aunque se ve muy interesante y cercano a Java, pero cuando comencé a desarrollar la aplicación, el Runtime de Mono (la implementación de C# para Linux) no venía en la instalación por omisión de Ubutu (en el paquete `ubuntu-desktop`), así que me parecía un poco pretencioso que alguien bajase varios megas de paquetes de dependencias para un programa de sólo cien kilobytes o menos.

Finalmente, descarté Java porque para desarrollar una aplicación con look nativo en el escritorio Ubuntu requería algo mejor que Swing y AWT. La alternativa era usar bibliotecas Java para GTK+, pero no estaban muy maduras cuando partí. Y se repetía el problema de usar varios megas de dependencias para una aplicación muy pequeña. Y ni siquiera sabía si el GNU Classpath iba a soportar mi aplicación, así que si quería usar una máquina Java que fuera libre, tenía mis dudas.

Para mí, Python me ofrecía un lenguaje robusto y orientado a objetos, sintaxis clara para poder mantener el código, viene en la instalación base del escritorio (de hecho, las herramientas de escritorio para

la administración están escritas en Python y GTK+) así que me dispuse a refrescar mis escasos conocimientos de Python.

Alguna vez hice un script que permitía recuperar una página web del diario La Tercera y generar feeds RSS cuando el diario todavía no lo hacía, e hice una aplicación similar para notificar cuando me había llegado correo nuevo usando la interfaz web de una versión vieja de Microsoft Outlook. Pero no había hecho algo así como una aplicación de escritorio.

Capítulo 2. Desarrollo con Python y Glade

Introducción

Hasta ahora, la combinación que me parece más atractiva para desarrollar aplicaciones de escritorio en Linux es la del lenguaje Python con Glade (como biblioteca y utilidad para crear la interfaz de usuario).

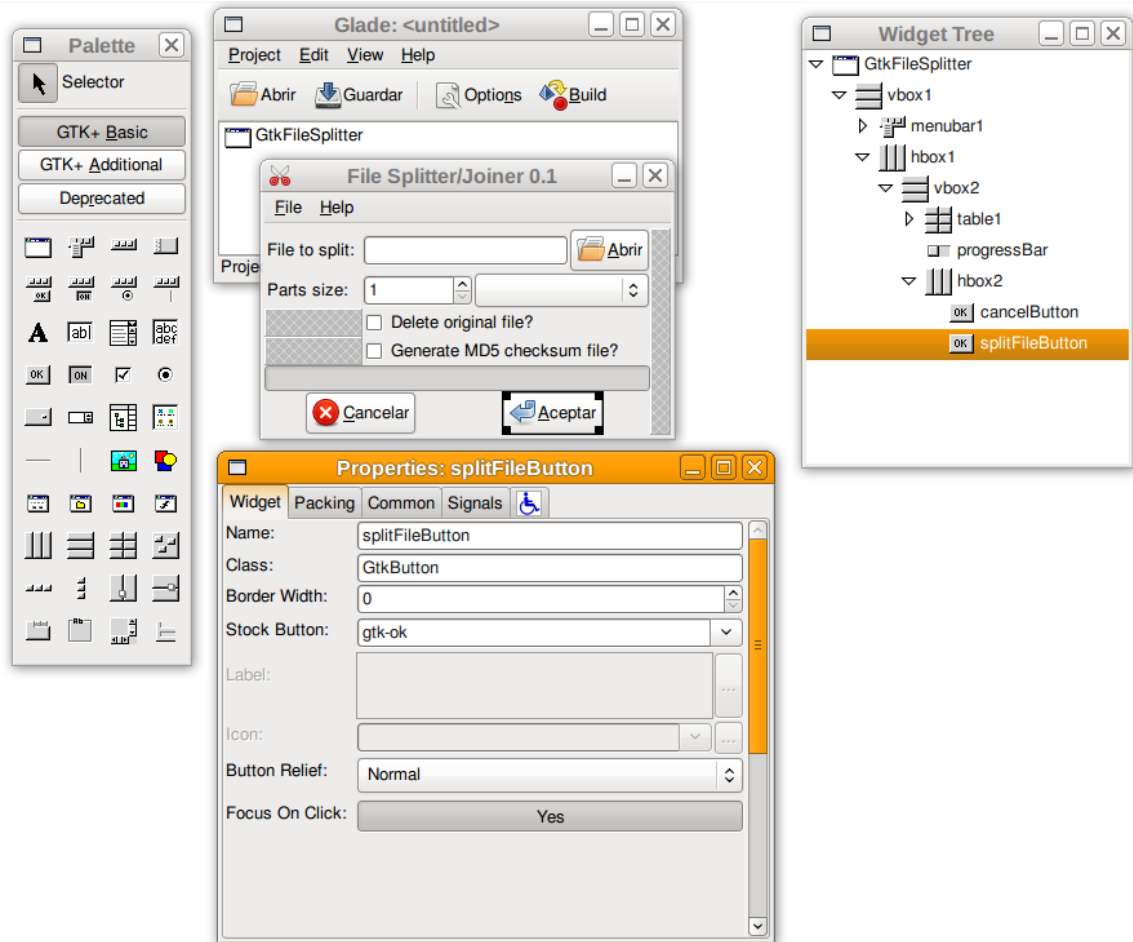
En Python tenemos un lenguaje de programación maduro, con características de orientación a objetos pero liviano en sintaxis y que deja una buena cantidad de opciones a los desarrolladores responsables. De hecho, Sean Kelly, en un destacado videocast sobre desarrollo de aplicaciones Web habla de la excesiva mistificación de la encapsulación en el mundo de la orientación a objetos. El se refiere a los desarrolladores Python como "consenting adults" ("adultos con consentimiento") porque si se necesita, se puede romper la encapsulación con consentimiento y "tocar las partes privadas" :-)

El desarrollo de una aplicación con Python y Glade es bastante sencillo: Primero se crean una o más clases en Python con los métodos que resuelven la lógica del problema que nos interesa resolver. Idealmente, deberíamos dejar solo un esqueleto de los métodos para tener una idea de como vamos a abordar el problema, para luego crear pruebas unitarias y recién ahí comenzar a implementar la lógica de cada método de forma de pasar sin problemas las pruebas unitarias.

Luego, se crea una o más clases que representan la interfaz de usuario de la aplicación. Esta clase de interfaz de usuario utiliza un archivo XML que contiene la definición de la interfaz de usuario: ventanas, barras de menú, botones, íconos, etc. Esta clase tiene métodos que están conectados con los eventos que ocurren en la interfaz de usuario, y ejecutan cierta lógica que programamos en las clases que conocen la lógica de nuestro problema.

Diseño de una interfaz de usuario con Glade

A quienes hayan utilizado entornos de desarrollo como Delphi o Visual Basic les será bastante familiar la idea de arrastrar controles hacia una ventalla con una grilla donde se pueden colocar botones, menús y otros.



Con Glade puedes crear de forma bastante rápida un prototipo de interaz de usuario. Comienzas seleccionando una Ventana desde la Paleta de elementos y comienzas a agregarle contenedores donde puedes colocar otros elementos como menús, cajas de texto y botones.

Cada uno de estos elementos emite eventos cada vez que el usuario interactúa con ellos. Algunos ejemplos son: cuando el usuario pulsa sobre un botón o cuando una tecla se presiona mientras el usuario llena un campo de texto.

Al editar la interfaz de usuario en Glade, cada uno de los eventos que nos interese manejar deberá tener asociado un manejador, es decir, un valor que lo identifique de forma que nuestro programa identificará cada evento que se haya producido.

Un ejemplo mínimo en Python para cargar un archivo de intefaz de usuario creado con Glade es uno como el siguiente, copiado desde mi proyecto GtkFileSplitter. Un archivo de interfaz de usuario creado con Glade puede encontrarse en esta dirección: [gtkfilesplitter.glade](http://gtkfilesplitter.googlecode.com/svn/trunk/gtkfilesplitter.glade) [http://gtkfilesplitter.googlecode.com/svn/trunk/gtkfilesplitter.glade]

```
#!/usr/bin/env python
import pygtk
pygtk.require("2.0")
import gtk
import gtk.glade
```

```
class GtkFileSplitter:
    """GTK/Glade User interface to FileSplitter"""

    def __init__(self):

        # Cargar interfaz de usuario
        self.gladefile = "gtkfilesplitter.glade"
        self.wTree = gtk.glade.XML(self.gladefile, "GtkFileSplitter",
            'gtkfilesplitter')

        # Conectar eventos con metodos en la clase
        dic = {
            "on_splitFileButton_clicked" : self.on_splitFileButton_clicked,
            "on_FileSplitGui_destroy"    : gtk.main_quit }
        self.wTree.signal_autoconnect(dic)

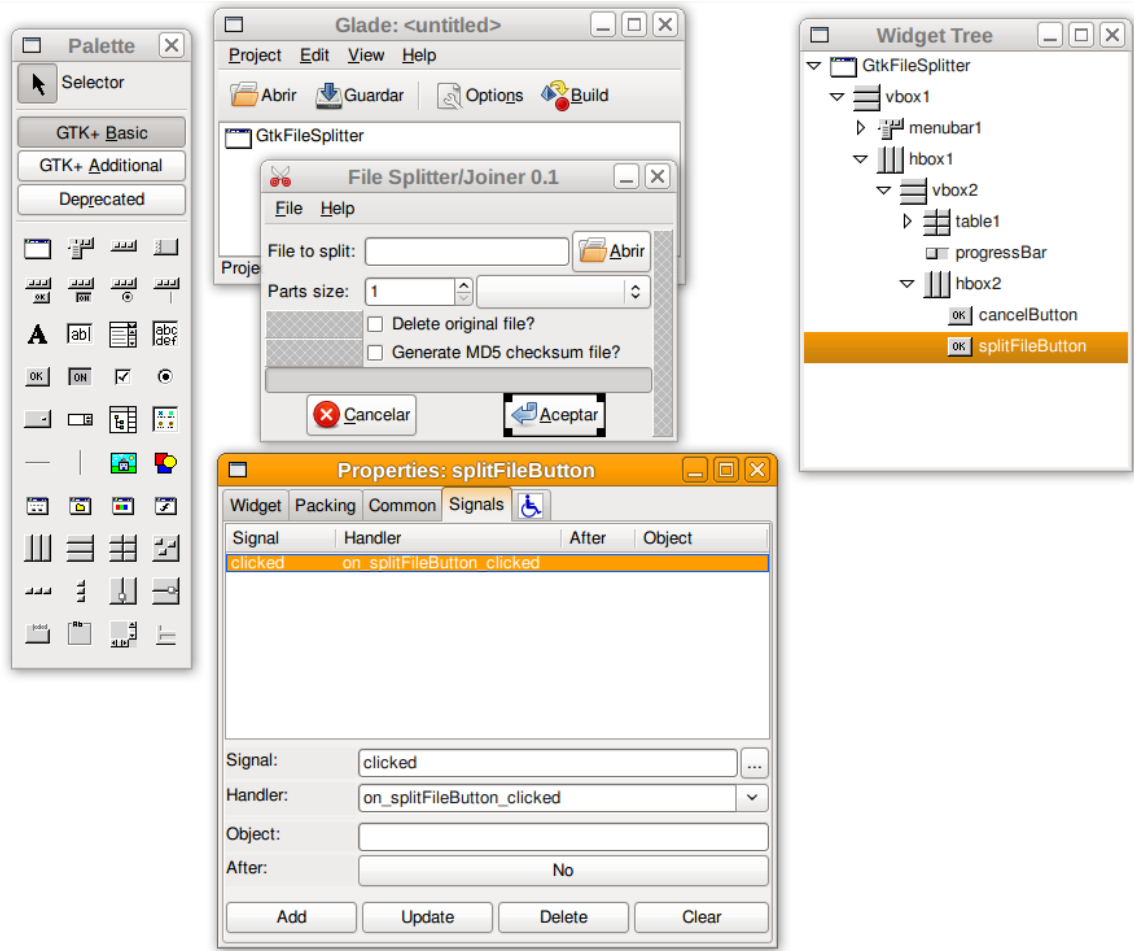
        # El metodo invocado al hacer click en el Boton
        def on_splitFileButton_clicked(self, widget):
            print "Click"

if __name__ == "__main__":
    gtkfilesplitter = GtkFileSplitter()
    gtk.main()
```

En el ejemplo, se carga el archivo `gtkfilesplitter.glade` y se crea un árbol de objetos con todos los elementos de la interfaz de usuario. El objeto `Ventana` será el padre de otros objetos contenedores que a su vez serán padres de botones, menús y a su vez otros contenedores...

Luego, se crea un diccionario que relaciona los eventos definidos en la interfaz de usuario con métodos que deben estar definidos en la clase Python, de lo contrario, se producirá una excepción en tiempo de ejecución (`AttributeError: GtkFileSplitter instance has no attribute 'on_splitFileButton_clicked'`).

En este caso, el evento `on_splitFileButton_clicked`, definido en la interfaz de usuario para el caso en que el usuario hace click en un botón, queda vinculado al método `on_splitFileButton_clicked()` que está definido en la clase `GtkFileSplitter`.



El siguiente es el fragmento del archivo XML generado por Glade que contiene la definición de usuario de un botón que al ser pulsado envía el evento **on_splitFileButton_clicked**

```

...
  <child>
    <widget class="GtkButton" id="splitFileButton">
      <property name="visible">True</property>
      <property name="can_focus">True</property>
      <property name="label">gtk-ok</property>
      <property name="use_stock">True</property>
      <property name="relief">GTK_RELIEF_NORMAL</property>
      <property name="focus_on_click">True</property>
      <signal name="clicked" handler="on_splitFileButton_clicked"
        last_modification_time="Tue, 05 Dec 2006 13:35:04 GMT"/>
    </widget>
    <packing>
      <property name="padding">0</property>
      <property name="expand">False</property>
      <property name="fill">False</property>
    </packing>
  </child>
...

```

Soporte multilinguaje en la interfaz de usuario

Si te has fijado en las capturas de pantalla anteriores, notarás que la interfaz de usuario tiene casi todos sus elementos en idioma inglés. Los únicos elementos en español que aparecen son los botones. Esto no es casualidad, sino una convención utilizada proveer soporte multilinguaje en las aplicaciones.

Lo que se hace es utilizar un conjunto de archivos que proveen las traducciones de todos los mensajes de la interfaz de usuario de nuestra aplicación. Dependiendo de los archivos de traducciones que se distribuyan con el instalador, la aplicación podrá estar disponible para más idiomas. En este caso, utilizamos el soporte de la biblioteca GNU Gettext.

En nuestro programa Python, utilizamos la función `_(' ')` sobre todos los mensajes que se despliegan al usuario, de forma que

```
print ("Hola Mundo")
s = "Error: %s (%s)" % (a, b)
```

se convierten en

```
print _("Hola Mundo")
s = _("Error: %s (%s)") % (a, b)
```

Luego, para diferenciar las traducciones de nuestra aplicación de las del resto del sistema, utilizamos el nombre de nuestra aplicación, que debe ser diferente a las otras aplicaciones que se encuentren en nuestro sistema (no debe ser un nombre ambiguo).

Para usar la biblioteca **gettext** desde Python, importaremos las bibliotecas `locale` y `gettext` al inicio de nuestra aplicación. El inicio de nuestro programa quedará como el siguiente:

```
import locale, gettext

APP = 'gtkfilesplitter'
DIR = 'locale'

# Mapeo la funcion gettext.gettext como "_"
_ = gettext.gettext

try:
    import pygtk
    # Para usar la versión 2.0 de pygtk
    pygtk.require("2.0")
except:
    pass
try:
    import gtk
    import gtk.glade
except:
    sys.exit(1)

# Enlazo las traducciones de la aplicación e interfaz de usuario
gettext.bindtextdomain(APP, DIR)
gettext.textdomain(APP)
gtk.glade.bindtextdomain(APP, DIR)
```

```
gtk.glade.textdomain(APP)
```

Para crear las traducciones utilizamos el programa `xgettext`, que es parte del paquete `gettext` por lo que lo instalamos con:

```
$ sudo apt-get install gettext
```

Este programa nos permite extraer todos los mensajes de nuestro código fuente a un archivo con extensión POT (Portable Object Template), que utilizaremos como base para las traducciones de nuestro software.

```
$ xgettext -k_ -kN_ -o messages.pot *.py *.glade
```

Este archivo `messages.pot` contiene todos los mensajes que es posible traducir, y tiene una apariencia como la siguiente:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2006-12-13 22:47-0300\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: gtkfilesplitter.py:551
#, python-format
msgid "Working ... %d%c done"
msgstr ""

#: gtkfilesplitter.py:385 gtkfilesplitter.py:511
msgid "Do you want to cancel?"
msgstr ""

#: gtkfilesplitter.glade.h:7
msgid "Delete original file?"
msgstr ""

...
```

Cada mensaje identificado como "traducible" aparece en un ítem `msgid` junto con una traducción vacía en la línea siguiente, después de `msgstr`.

Para comenzar a traducir, utilizamos el siguiente comando:

```
$ msginit --input messages.pot
```

```
created es.po
```

El idioma se obtiene del valor de la variable de ambiente `LANG`, y en el caso de un sistema con idioma español, será `es.po`. Editamos el archivo para agregar nuestras traducciones y modificamos el preámbulo del mismo con información de contacto del traductor:

```
# Messages Translation File for GtkFileSplitter
# Copyright (C) 2006 Denis Fuenzalida
# This file is distributed under the GPL license
# Denis Fuenzalida <denis.fuenzalida@gmail.com>, 2006
#
msgid ""
msgstr ""
"Project-Id-Version: gtkfilesplitter 0.1\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2006-12-13 22:47-0300\n"
"PO-Revision-Date: 2006-12-07 10:32-0300\n"
"Last-Translator: <denis.fuenzalida@gmail.com>\n"
"Language-Team: Spanish\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CP1252\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=(n != 1);\n"

#: gtkfilesplitter.py:551
#, python-format
msgid "Working ... %d%c done"
msgstr "Trabajando ... %d%c listo"

#: gtkfilesplitter.py:385 gtkfilesplitter.py:511
msgid "Do you want to cancel?"
msgstr "¿Desea cancelar la operación?"

#: gtkfilesplitter.glade.h:7
msgid "Delete original file?"
msgstr "¿Borrar el archivo original?"

...
```

Al inicio del código fuente del programa se definió la constante `DIR` que indica el directorio en el que se encuentran los archivos con las traducciones. Los archivos con las traducciones no se utilizan directamente como texto plano sino que se convierten a un formato binario con extensión `.mo`. Para colocar la traducción al español creamos una estructura de directorios predefinida:

```
$ mkdir locale
$ mkdir locale/es
$ mkdir locale/es/LC_MESSAGES
$ msgfmt es.po -o locale/es/LC_MESSAGES/gtkfilesplitter.mo
```

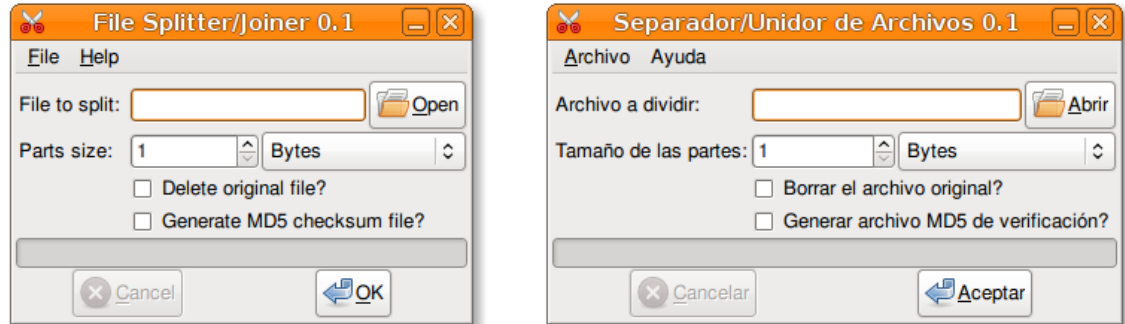
Es importante ser consistente en el uso de los nombres de archivos, de lo contrario, la traducción no se encontrará y la interfaz de usuario aparecerá en inglés en lugar de la esperada. En este caso, el archivo `.mo` esperado es el mismo que aparece en el código fuente en la constante `APP`.

Si todo ha salido bien, podrás iniciar tu aplicación en el idioma deseado, e incluso indicar otros idiomas diferentes al que tengas configurado, reemplazando el valor de la variable de ambiente `LANG` al momento

de iniciar la aplicación. Por ejemplo, para iniciar mi aplicación **gtkfilesplitter** en idioma inglés basta con iniciarla en un Terminal con:

```
$ LANG=en gtkfilesplitter
```

En la siguiente imagen se muestra la misma aplicación ejecutándose con dos idiomas distintos:



Para un mayor detalle sobre el uso de Python, Glade y GTK, junto con instrucciones detalladas para problemas que puedes encontrar al desarrollar, revisa este enlace a las Preguntas Frecuentes de PyGTK [<http://faq.pygtk.org/>] y en particular a la pregunta "How do I internationalize a PyGTK and libglade program?" (en inglés).

Capítulo 3. Empaquetado y distribución de software para Ubuntu

Hemos llegado al punto en que tenemos software con una calidad y robustez suficiente como para comenzar a distribuirlo con nuestros usuarios. En este capítulo revisaremos el proceso de empaquetar un proyecto de software como un paquete instalable en un sistema y de como este paquete se puede colocar en un repositorio de software de forma que los usuarios de nuestra aplicación puedan obtener de forma automática las nuevas versiones mejoradas y ampliadas de nuestro software ya publicado.

Entendiendo los paquetes de software

En sistemas basados en Debian (como Ubuntu, Knoppix y otros) el software que se instala en nuestros equipos se encuentra agrupado lógicamente en paquetes, que se obtienen desde los discos de instalación e Internet. Estos paquetes contienen los archivos que se instalan en nuestros sistemas, más cierta información de contexto (metadatos) que indican que cada uno de estos componentes de software obedece ciertas reglas y cuales son los recursos de los que provee a nuestro sistema.

En general, entonces, diremos que un paquete de software es un elemento que se compone de un conjunto de archivos a instalar, y que posee información sobre su versión, los otros paquetes de software que requiere para funcionar, los recursos que este paquete provee para nuestro sistema y eventualmente, con cuales componentes no se lleva bien (porque puede haber incompatibilidades entre 2 componentes diferentes).

Versiones de un elemento de software

Los paquetes de software que instalamos y producimos poseen un número de versión que sirve para identificar un conjunto de características que ese software posee, en términos de funcionalidades, errores corregidos y por corregir, entre otros. Hay una regla más o menos genérica que indica que se usa un número mayor de versión para indicar grandes hitos en el desarrollo de un programa, luego un número menor de versión para indicar el grado de avance en el desarrollo de funcionalidades entre estos grandes hitos, y finalmente, un tercer número que indica correcciones de seguridad hechas a un programa.

De esta forma, un programa en la versión 1 . 5 . 3 indica que ya se alcanzó el primer gran hito de desarrollo del programa (**1**), en camino hacia una versión 2. El número de versión menor **5** indica que es la quinta entrega de funcionalidades (desde la versión 1.0), y además el tercer dígito de versión (**3**) indica que se han hecho 3 correcciones importantes de seguridad a la versión 1.5 de este programa.

Dependencias de un elemento de software

En general, el software de nuestros equipos funciona sobre una cantidad bastante de otros componentes previamente instalados, de la misma forma en que el techo de una casa descansa sobre las murallas, que a su vez, descansan en los cimientos. Así, nuestro software funcionará sobre la base de que otros componentes de software ya se encuentren instalados.

Creación de un paquete de software Debian

Al ser una distribución basada en Debian, la creación de paquetes en Ubuntu utiliza el mismo conjunto de herramientas de ayuda para el proceso de empaquetamiento, entre las que se encuentran `dh_make`, `dch`, `fakeroot` y otros.

A diferencia de otros textos publicados en Internet, vamos a cubrir el escenario más completo posible, lo que implica que vamos a crear paquetes fuente y binarios que utilizarán un esquema de firma digital basada en GNU Privacy Guard (que es una implementación libre del estándar de firma digital PGP), para que los usuarios de los paquetes que empaquetemos puedan obtener las versiones más recientes del software sin advertencias que puedan intimidarlos. Si no usamos firmas en el software que empaquetamos y en los repositorios de software a crear, el sistema indica una serie de advertencias al usuario (sobre el riesgo de instalar software que no ha sido firmado por ningún responsable) que pueden despertar suspicacias o rechazo, cada vez que publiquemos una nueva actualización de nuestro software.

Creación de una firma digital con GnuPG

En primer lugar, se requiere instalar y configurar el software requerido para firmar digitalmente los archivos. Las herramientas para empaquetamiento y distribución de software en Debian y Ubuntu se encuentran bien integradas con GnuPG y es sencillo de utilizar.

Procedemos a instalar GnuPG:

```
$ sudo apt-get install gnupg
```

Una buena guía sobre la instalación de GnuPG se encuentra en la siguiente dirección: [http://www.gnupg.org/\(en\)/documentation/howtos.html](http://www.gnupg.org/(en)/documentation/howtos.html)

Luego de instalado, debemos generar un conjunto de llaves pública y privada con las que funciona el mecanismo de firma digital. Para ello, ejecutamos en un terminal:

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

El programa nos preguntará primero sobre el algoritmo a utilizar para la generación de las claves, para lo cual dejamos el algoritmo sugerido por omisión. La siguiente pregunta es la longitud de la clave, para lo cual también dejamos al largo sugerido. Luego se nos pregunta el tiempo durante el cual estas claves serán vigentes. Para efectos prácticos, 6 meses o un año pueden estar bien, pero es posible que otras organizaciones o proyectos usen períodos diferentes por motivos de seguridad. Luego, el programa pide algunos datos del usuario para registrar la identificación de quien firma: El nombre, un comentario o alias y una dirección de correo electrónico.

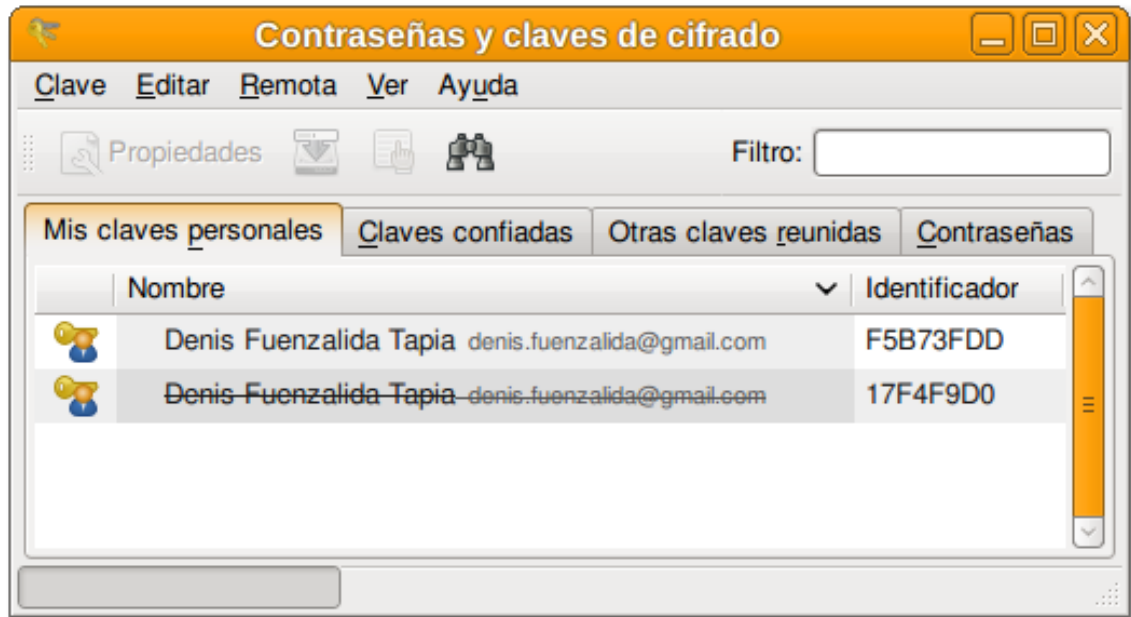
Finalmente, se pide una contraseña que se utiliza cada vez que se va a firmar. La contraseña idealmente debe ser larga, que combine mayúsculas y minúsculas, etc.

A continuación, el programa realiza un trabajo que puede demorar algunos momentos y que consiste en la generación de la llave pública y privada, que se basa en la generación de números primos mediante la recolección de datos más o menos aleatorios como la fecha y hora, carga del procesador, memoria utilizada y un largo etcétera. Puede demorar un poco. Para comprobar la firma recién generada, se pueden listar las claves junto a un identificador corto que le corresponde a cada una, con el siguiente comando:

```
$ gpg --list-keys  
/users/alice/.gnupg/pubring.gpg  
-----  
pub 1024D/BB7576AC 1999-06-04 Alice (Judge) <alice@cyb.org>  
sub 1024g/78E9A8FA 1999-06-04
```

Este comando produce un listado, del cual nos interesa el identificador hexadecimal junto a la llave pública. En el listado anterior, es el código BB7576AC. Conviene tener este identificador a mano a la hora de firmar paquetes y hacer cambios en el repositorio que vamos a crear.

Un programa que hace todas estas tareas más sencillas, con una interfaz gráfica, es Seahorse [<http://www.gnome.org/projects/seahorse/>], y que puede verse en la siguiente captura de pantalla.

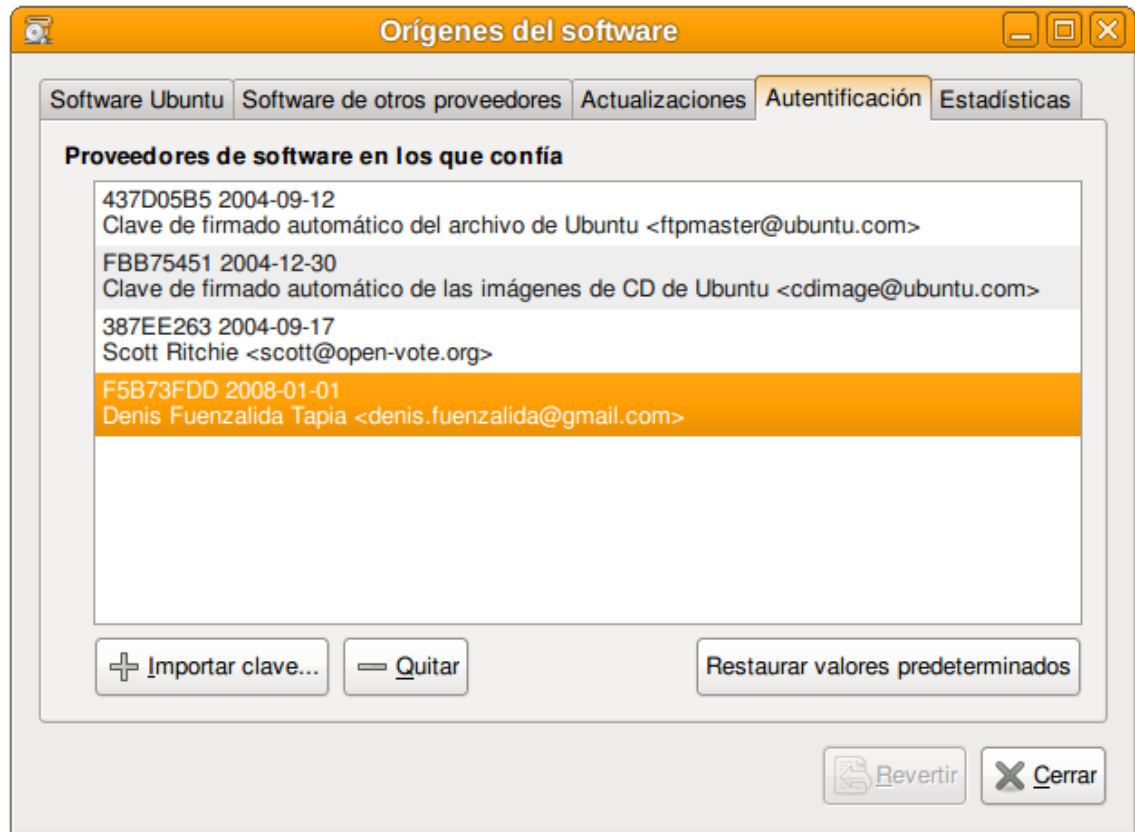


Seahorse queda instalado en **Aplicaciones - Accesorios - Contraseñas y claves de cifrado**

Agregar la llave pública a tus proveedores de software de confianza

Para probar la firma digital sobre los paquetes de software que desarrolles, deberás agregar tu llave pública al listado de proveedores de software a los que tienes confianza. De lo contrario, cada vez que instales software producido por tí, aparecerán advertencias indicando que sólo debes instalar programas que vienen de fuentes de confianza.

Lo que debes hacer es agregar el archivo con tu llave pública (`pubring.gpg`) a la aplicación que se encuentra en **Sistema - Administración - Orígenes del Software**. En la lengüeta **Autenticación** encontrarás el listado de firmas digitales que tu sistema acepta para validar paquetes de software firmados. Para agregar tu llave pública, usa el botón **Importar Clave** para importar el archivo `/home/USUARIO/.gnupg/pubring.gpg`. Tu firma debería aparecer en el listado, como en la siguiente pantalla:



El mismo proceso lo puedes realizar usando el comando **apt-key add** en una shell:

```
$ sudo apt-key add /home/usuario/.gnupg/pubring.gpg  
$ sudo apt-key list
```

Creación de un paquete Debian

Para crear un paquete Debian, el proceso consta de copiar el código fuente de nuestro proyecto y agregar unos archivos de datos extras utilizados por las herramientas que forman los paquetes. Cuando el código se obtiene desde un tarball o desde un repositorio de código fuente, este proceso de agregar archivos de Debian se llama "Debianización". La referencia oficial para la creación de paquetes en Debian y Ubuntu es el documento Debian New Maintainers' Guide [<http://www.debian.org/doc/maint-guide/>] (en inglés).

Procederemos a instalar los siguientes paquetes para instalar las utilidades mínimas para construir paquetes Debian:

```
$ sudo apt-get install dh-make devscripts fakeroot
```

(en caso de construir programas a partir de código fuente en C, también se requiere instalar el paquete `build-essential`):

Para producir el paquete, obtenemos una copia del código fuente de nuestro proyecto en un directorio temporal (por ejemplo, `/tmp/debianizando/gtkfilesplitter-0.2`). El directorio debe seguir la siguiente convención: `[nombre-del-proyecto]-[versión]`. En el ejemplo, el nombre del proyecto es `gtkfilesplitter` y la versión es la 0.2.

En el directorio donde se encuentra el código fuente de nuestro proyecto, ejecutamos

```
$ dh_make -c gpl -e usuario@miempresa.com -s --createorig
```

La herramienta `dh_make` agregará los archivos con la información extra para producir el paquete en un directorio `debian` en el directorio actual. El parámetro `-c` indica el tipo de licencia a utilizar (las opciones con GPL, LGPL, entre otras). La dirección de correo del usuario que produce el paquete se especifica con la opción `-e`. Para firmar el paquete hemos utilizado la opción `-s` (por "sign") y finalmente, es posible que no hayamos generado el directorio con el código fuente a partir de un tarball, así que la opción `--createorig` le indica al programa que la genere por nosotros.

La ejecución del programa deja tras de sí un directorio `debian` con un conjunto de archivos con información del paquete que deberíamos modificar, y en el directorio superior, un archivo comprimido (con `tar` y `gzip`) con el contenido del directorio antes de la Debianización, un archivo de cambios y un archivo extra.

El archivo `debian/changelog`

Si es necesario indicar que este paquete tiene algunas correcciones producto de una mejora o debido a que se aplicó algún parche, se puede indicar en un archivo de `ChangeLog`. Existe un utilitario que sirve para esto:

```
$ dch -i
```

Este programa abrirá el archivo `debian/changelog` del proyecto y agregará unas líneas de contexto para poder agregar la lista de cambios de esta versión de nuestro programa. Guardamos los cambios al archivo y salimos del editor.

El archivo `debian/control`

El siguiente archivo relevante que debemos modificar es el archivo `debian/control`, que es el que provee información sobre el mantenedor del paquete de software, los paquetes que son necesarios al momento de construir el paquete, y las dependencias al momento de ejecutar el software. Vale la pena destacar que no necesariamente son iguales. En el software desarrollado en lenguaje C es típico que se necesitan bibliotecas para desarrollos (ej. `libXXXYYYZZZ-dev`) que luego no son necesarias en tiempo de ejecución.

El campo **Description** sigue una convención: la primera línea, justo después de `Description:` contiene una descripción corta (de hasta 60 caracteres), y luego una descripción que puede tener varias líneas de hasta 80 caracteres, que comiencen con un espacio en blanco. Para hacer un punto aparte se deja una línea con un espacio y un punto.

En el caso de nuestro programa de ejemplo, el contenido del archivo `debian/control` es el siguiente:

```
Source: gtkfilesplitter
Section: admin
Priority: optional
Maintainer: Denis Fuenzalida <denis.fuenzalida@gmail.com>
Build-Depends: debhelper (>= 5), python-support (>= 0.3)
Standards-Version: 3.7.2.2

Package: gtkfilesplitter
Architecture: any
Depends: ${misc:Depends}, ${shlibs:Depends}, ${python:Depends}, python, python-gla
Description: Simple file splitter/joiner with checksum support
 A desktop application to split and join files
.
For more information, visit http://code.google.com/p/gtkfilesplitter
```

El archivo `debian/copyright`

En general, este archivo debería venir pre-llenado con una plantilla de la licencia que se haya seleccionado al momento de invocar a `dh_make` y sólo será necesario cambiar la información de contacto del creador del software y eventualmente revisar que los términos de la licencia sean los que el desarrollador ha acordado.

El archivo `debian/rules`

El archivo `debian/rules` en realidad es un **makefile** (un archivo con instrucciones para GNU Make) que sigue un cierto patrón de comandos que corresponden con una gran cantidad de tareas que se realizan al instalar un paquete de software en nuestro sistema: desde instalar la documentación (páginas man, etc.), instalar íconos en el menú de aplicaciones, instalar bibliotecas, etc.

Las tareas a realizar durante la instalación dependen muchísimo de la naturaleza del software que desarrollemos y de los componentes formen parte del mismo. Así, es posible que no incluyamos archivo de ejemplo, etc. El archivo `rules` creado contiene una gran cantidad de tareas que pueden comentarse en caso de que no apliquen. En el caso de mi programa, utilicé como plantilla una versión antigua de otro software (Apt On CD), que también está escrito en Python. En realidad lo único especial que hace es invocar a un Makefile en la raíz del proyecto, indicando que los archivos creados deben almacenarse en el directorio `debian/gtkfilesplitter`.

```
# Archivo debian/rules del proyecto gtkfilesplitter
# ver archivo completo en:
# http://gtkfilesplitter.googlecode.com/svn/trunk/debian/rules

...
install: build
    dh_testdir
    dh_testroot
    dh_clean -k
    dh_installdirs

$(MAKE) install DESTDIR=$(CURDIR)/debian/gtkfilesplitter
...
```

Al crear nuestro paquete, una copia de los archivos agregados al sistema se agrega dentro del directorio `debian` en un directorio con el mismo nombre del paquete que estamos creando. Por ejemplo: si al instalar mi software, copio un ícono en formato PNG en `/usr/share/icons/MiIcono.png`, lo que debe hacer el Makefile en realidad es copiarlo al directorio `debian/MiPaquete/usr/share/icons/MiIcono.png`.

El contenido (comentado) del Makefile en la raíz del proyecto es:

```
# Traducciones disponibles:
PO = es pl

PREFIX ?= /usr

all: check po-data
    @echo "Done"
    @echo "Type: make install now"

check:
    @/bin/echo -n "Checking for Python... "
```

```
@which python || ( echo "Not found." && /bin/false )
@./check.py

# borrar archivos de respaldos, etc.
clean:
    find . -type f -iregex '.*~$$' -print | xargs rm -rf
    find . -type d -iregex '.*\.svn$$' -print | xargs rm -rf
    find . -type f -iregex '.*\.pyc$$' -print | xargs rm -rf
    find . -type f -iregex '.*\.gladep$$' -print | xargs rm -rf
    find . -type f -iregex '.*\.bak$$' -print | xargs rm -rf

make-install-dirs: make-install-dirs-po

    # Crear directorios para copiar los archivos
    mkdir -p $(DESTDIR)$(PREFIX)/bin
    mkdir -p $(DESTDIR)$(PREFIX)/share/applications
    mkdir -p $(DESTDIR)$(PREFIX)/share/gtkfilesplitter
    mkdir -p $(DESTDIR)$(PREFIX)/share/pixmaps
    mkdir -p $(DESTDIR)$(PREFIX)/share/gnome/help/gtkfilesplitter/C
    mkdir -p $(DESTDIR)$(PREFIX)/share/locale

make-install-dirs-po:

    # Para cada idioma:
    # Crear un directorio para colocar los archivos de traducción
    for lang in $(PO); \
        do mkdir -p $(DESTDIR)$(PREFIX)/share/locale/$$lang/LC_MESSAGES; \
        done

install: make-install-dirs install-po

    # Copiar el ejecutable Python, con permiso de ejecución y lectura para todos
    install -m 755 gtkfilesplitter.py $(DESTDIR)$(PREFIX)/share/gtkfilesplitter

    # Copiar el archivo Glade de interfaz de usuario
    install -m 644 gtkfilesplitter.glade $(DESTDIR)$(PREFIX)/share/gtkfilesplitter

    # Copiar los íconos de la aplicación
    install -m 644 gtkfilesplitter*.png $(DESTDIR)$(PREFIX)/share/gtkfilesplitter

    # Copiar el archivo .desktop (que crea el ícono en el menú)
    install -m 644 gtkfilesplitter.desktop $(DESTDIR)$(PREFIX)/share/applications/

    # Creo un link simbólico al script en /usr/bin
    cd $(DESTDIR)$(PREFIX)/bin && \
    ln -sf ../share/gtkfilesplitter/gtkfilesplitter.py gtkfilesplitter && \
    chmod 755 gtkfilesplitter

install-po:

    # Instalo los archivos de traducciones (.mo)
    for lang in $(PO); \
        do install -m 644 \
            locale/$$lang/LC_MESSAGES/* \
```

```
$(DESTDIR)$(PREFIX)/share/locale/$$lang/LC_MESSAGES/; \  
done  
  
po-dir:  
  for lang in $(PO); do mkdir -p locale/$$lang/LC_MESSAGES/ ; done  
  
po-data: po-dir  
  for lang in $(PO); \  
  do msgfmt locale/$$lang.po -o \  
  locale/$$lang/LC_MESSAGES/gtkfilesplitter.mo; \  
  done  
  
po-gen:  
  
  # Generación de archivos .pot y .po a partir del código  
  intltool-extract --type=gettext/glade gtkfilesplitter.glade  
  xgettext -k_ -kN_ -o locale/messages.pot *.py *.h  
  for lang in $(PO); \  
  do msgmerge -U locale/$$lang.po locale/gtkfilesplitter.pot; \  
  done
```

Lo fundamental es entender las tareas que realiza el target **install**: los directorios en que se copian los archivos, etc.

Debido a la complejidad que supone crear uno de estos archivos y a que muchas de estas tareas se repiten de forma bastante rutinaria, se creó un proyecto llamado el Common Debian Build System [<https://perso.duckcorp.org/duck/cdbS-doc/cdbS-doc.xhtml>], que consiste en un conjunto de Makefiles ya creados para un conjunto de proyectos diferentes, de forma que bastaría con incluir estas recetas en tus archivos `debian/rules` para hacerlos de forma mucho más sencilla. Lamentablemente la documentación sobre CDBS es bastante escasa, pero la idea es que para un proyecto Python con una convención de directorios estándar, el archivo podría quedar tan sencillo como:

```
#!/usr/bin/make -f  
include /usr/share/cdbS/1/rules/debhelper.mk  
include /usr/share/cdbS/1/class/python-distutils.mk
```

De hecho, pude empaquetar con muy poco código una aplicación desarrollada en Java y compilada con Ant, como se verá en la siguiente sección.

Construir el paquete con `dpkg-buildpackage`

Para producir el paquete firmado, necesitamos el identificador corto de la firma digital (en el ejemplo en el punto anterior, era `BB7576AC`). Para producir el paquete firmado ejecutamos el siguiente comando:

```
$ dpkg-buildpackage -sgpg -kBB7576AC -rfakeroot
```

Las opciones utilizadas son `-sgpg` para utilizar firma GnuPG. La opción `-k` es el identificador de la llave a utilizar. Para crear los paquetes se requiere utilizar un pseudoambiente de root, para lo que se usa la herramienta `fakeroot`.

El proceso para generar el archivo de paquetes puede demorar un poco y se escribe una gran cantidad de texto en pantalla a medida que se ejecuta un gran conjunto de tareas. Finalmente, si no hay errores en la construcción, se nos pedirá la contraseña utilizada al crear nuestras llaves GnuPG para poder firmar unos archivos que forman parte del paquete.

Felicitaciones! En este momento deberías contar con tu programa empaquetado apropiadamente como un archivo Deb.

Validación de paquetes con Lintian y Linda

Luego de tanto esfuerzo, es posible que hayamos omitido algún paso, u omitido alguna información que venía por completar en alguna plantilla de documentos, o en general, errores menores o no conformidad con las políticas y convenciones de Debian para la instalación de Software (existen restricciones a donde se pueden colocar archivos, bibliotecas y con qué permisos, además de las políticas para Python y Java).

Para revisar la conformidad de nuestros paquetes de software con estas restricciones y convenciones, existe un par de programas, llamados **lintian** y **linda** que realizan estas tareas y reportan advertencias y errores que contienen nuestros paquetes.

```
$ lintian gtkfilesplitter_0.1.3-0ubuntu3_i386.deb
W: gtkfilesplitter: binary-without-manpage usr/bin/gtkfilesplitter
W: gtkfilesplitter: package-contains-empty-directory usr/share/pixmaps
W: gtkfilesplitter: package-contains-empty-directory usr/share/gnome/help/gtkfiles
...
E: gtkfilesplitter: debian-changelog-file-contains-invalid-address denis@laptop
...
```

En este caso, las advertencias son: El paquete no contiene página de manual, el paquete contiene directorios vacíos. El error es que en las entradas del archivo de ChangeLog aparecen direcciones de correo no válidas.

Creación de un paquete Debian con Java, Ant y CDBS

Durante varios años, Java era una alternativa atractiva (sobretudo a nivel de empresa) para desarrollo de aplicaciones, pero no había atraído de manera importante la atención de los desarrolladores de aplicaciones de escritorio. Al momento de escribir estas líneas, me parece que esto va a cambiar de forma significativa.

Java se está convirtiendo rápidamente en una plataforma Open Source. Sun Microsystems ya ha liberado la mayor parte del código fuente de la máquina virtual y de las bibliotecas de clases mediante el proyecto OpenJDK, de forma que está a la par con otras alternativas que deberían ser más o menos equivalentes, como el proyecto Mono.

De hecho, si bien existen paquetes que permiten instalar Java en Ubuntu (`sun-java5` y `sun-java6`), estos requieren aprobar una licencia por separado en el momento de instalarse. Hoy ya existe una implementación basada sólo en Software Libre (en el paquete `icedtea-java7`).

Si bien existen diferentes alternativas de ambientes de desarrollo libres para programar en Java (Eclipse y Netbeans, por ejemplo, cada una con sus partidarios y detractores), algo que se ha establecido como un estándar de facto es el uso de la herramienta Ant para la construcción de proyectos Java.

Ant partió como un reemplazo de GNU Make, porque al autor le parecía que la sintaxis de los Makefiles era horrible y era muy fácil caer en errores difíciles de detectar, como reemplazar tabuladores por espacios en un Makefile. En su lugar, Ant utiliza un archivo XML (`build.xml`) donde se establecen diferentes objetivos (targets) para un proyecto: desde limpiar los archivos de una compilación o respaldo anterior, hasta compilarlo, generar un una biblioteca Jar y una enorme cantidad de tareas que se han agregado con el tiempo.

Finalmente Ant se ganó el corazón de los equipos que deseaban una forma de construir sus proyectos Java de forma independiente de la herramienta de desarrollo y plataforma, y así Ant llega a tener soporte incluso en Eclipse y Netbeans.

Empaquetar software ya desarrollado en Java y Ant es bastante sencillo como veremos. En primer lugar, una aplicación Java como la mencionada suele tener su código fuente contenido en una carpeta (`source` o

src o similar) y un archivo build.xml en la raíz del proyecto. Para construir el proyecto, debería bastar con tener instalado Ant y un kit de desarrollo Java (JDK) y simplemente ejecutar en la línea de comandos:

```
$ ant
```

Dependiendo de la naturaleza del proyecto pueden ejecutarse varios pasos aparte de la simple compilación del código fuente: es posible que se ejecuten pruebas unitarias creadas con JUnit o algún tipo de generación de código, JavaDocs o similares.

Para comenzar a Debianizar este proyecto Java, ejecutamos:

```
$ dh_make -c gpl -e usuario@miempresa.com -s --createorig
```

Archivo debian/control de un paquete creado con Java

En el caso de que nuestro paquete de software dependa de una versión específica de la plataforma Java (ej. si utiliza "Generics" se requiere al menos de un entorno de Java 5) o de otras bibliotecas desarrolladas con Java (ej. Jakarta Commons), se deberá especificar en el archivo debian/control de forma precisa en las líneas **Build-Depends** y **Depends**. En el caso de los entornos de ejecución (JRE) existen paquetes virtuales que permiten que especifiques una versión de Java sin necesidad de limitarlo a una implementación en particular (ej. es mejor usar java5-runtime en lugar de sun-java5-jre).

Afortunadamente, ya hay muchas bibliotecas Java empaquetadas en Debian y Ubuntu, todas tienen un nombre de paquete con la convención libXXXX-java (ej. libcommons-httpclient-java) y la política para paquetes de software Java indica que todos los archivos de bibliotecas Jar quedarán instalados en /usr/share/java/.

Archivo debian/rules de un paquete creado con Java

Esto es algo realmente afortunado. Los archivos debian/rules de este tipo de paquetes pueden quedar realmente sencillos con ayuda de CDBS, como veremos en el ejemplo, un paquete hecho de un software llamado **Privatewiki** (es una aplicación que escribí hace tiempo, la idea es más o menos similar a Tomboy).

El archivo es el siguiente:

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

include /usr/share/cdb/1/rules/debhelper.mk
include /usr/share/cdb/1/class/ant.mk

# Estoy usando esta versión sólo para empaquetar!
JAVA_HOME := /usr/lib/jvm/java-1.5.0-sun
ANT_HOME := /usr/share/ant

install/privatewiki:: DEB_FINALDIR=$(CURDIR)/debian/privatewiki
install/privatewiki::

    # Creo varios directorios
    install -d $(DEB_FINALDIR)/usr/bin
    install -d $(DEB_FINALDIR)/usr/share/privatewiki
    install -d $(DEB_FINALDIR)/usr/share/java
    install -d $(DEB_FINALDIR)/usr/share/pixmaps
    install -d $(DEB_FINALDIR)/usr/share/applications

    # Instalo un shell que sirve para lanzar mi programa
    install -m 755 $(CURDIR)/debian/bin/privatewiki \
```

```
$(DEB_FINALDIR)/usr/bin/

# Un archivo con datos usados por mi programa, iconos, etc.
install -m 755 $(CURDIR)/bin/data.zip \
  $(DEB_FINALDIR)/usr/share/privatewiki/data.zip
install -m 755 $(CURDIR)/resources/icons/private.png \
  $(DEB_FINALDIR)/usr/share/pixmaps/privatewiki.png
install -m 755 $(CURDIR)/debian/bin/privatewiki.desktop \
  $(DEB_FINALDIR)/usr/share/applications/privatewiki.desktop
install -m 644 $(CURDIR)/lib/privatewiki.jar \
  $(DEB_FINALDIR)/usr/share/java/privatewiki.jar
```

Sí, eso es todo. El truco es la línea donde se incluye el archivo `ant.mk` lo que da instrucciones para realizar ciertas tareas con Ant y el archivo `build.xml` que debería tener mi proyecto Java.

El archivo `build.xml` de la raíz de mi proyecto es el siguiente:

```
<?xml version="1.0"?>
<project name="privatewiki" default="jar">

  <property name="src" value="src"/>

  <!-- limpiar proyecto -->
  <target name="clean">
    <delete failonerror="false">
      <fileset dir="lib" includes="**.*jar" />
    </delete>
    <delete dir="lib" failonerror="false" />
    <delete dir="classes" failonerror="false" />
  </target>

  <!-- compilar clases Java -->
  <target name="compile">
    <mkdir dir="classes" />
    <javac srcdir="src" destdir="classes" />
  </target>

  <!-- crear archivo Jar, requiere limpiar y compilar -->
  <target name="jar" depends="clean,compile">
    <mkdir dir="lib" />
    <jar jarfile="lib/privatewiki.jar"
      basedir="classes"
      compress="true"
    >
      <fileset dir="classes" />
      <fileset dir="resources" />

      <!-- indico la clase que inicia la ejecucion -->
      <manifest>
        <attribute name='Main-Class' value='privatewiki.Main' />
      </manifest>
    </jar>
  </target>

</project>
```

Lo importante es que construcción del proyecto con Ant realice todas las tareas necesarias para tener una aplicación lista para ejecutar. El archivo `debian/rules` toma el resto de los artefactos para producir el paquete y listo.

Finalmente, el documento de Java Policy en Debian indica que no se puede depender de un valor determinado para la variable de ambiente `CLASSPATH` (que se utiliza para indicar la ruta de búsqueda de bibliotecas Java para un programa en ejecución). Para modificar el `CLASSPATH` para los requerimientos de un programa, lo adecuado es crear un shell script que haga las modificaciones necesarias para el entorno de nuestro programa y luego invoque el programa en cuestión, por ejemplo:

```
#!/bin/sh

# CLASSPATH
export CLASSPATH=/usr/share/java/xxx1.jar:/usr/share/java/yyy2.jar

# Ejecuto mi programa
java org.miproyecto.MiAplicacion -Dflag1=valor -Dflag2=valor
```

En mi proyecto **privatewiki**, uso un script para detectar un archivo de datos (con las notas que llena el usuario en su wiki personal) y ejecuto el programa en un directorio determinado:

```
#!/bin/sh

set -e

# Test if ${HOME}/.privatewiki exists
DATADIR="${HOME}/.privatewiki

if [ -d $DATADIR ]; then
    echo "data dir found"
else
    mkdir ${HOME}/.privatewiki
fi

# Test if ${HOME}/.privatewiki/data.zip exists
DATAFILE="${HOME}/.privatewiki/data.zip

if [ -e $DATAFILE ]; then
    echo "datafile ok"
else
    echo "no datafile"
    cp /usr/share/privatewiki/data.zip ${HOME}/.privatewiki/data.zip
fi

cd "${HOME}/.privatewiki
java -jar /usr/share/java/privatewiki.jar
```

Para terminar, el código fuente completo y debianizado de **privatewiki** está disponible en <http://desarrollo-ubuntu.googlecode.com/svn/trunk/privatewiki-0.1/>

Creación de un repositorio sencillo

Un repositorio consiste en una estructura de carpetas publicadas en un servidor web con una estructura y contenidos tales que se pueden obtener paquetes y actualizaciones de software periódicas, integradas con nuestro sistema Ubuntu o Debian.

Un repositorio Debian utiliza una estructura de directorios para colocar paquetes Debian (archivos `.deb`) y añade ciertos archivos extras con información sobre los paquetes que se encuentran disponibles.

Para crear el repositorio, se debe crear una estructura de directorios (en este ejemplo, es para Ubuntu 7.10, "Gutsy Gibbon") en la raíz de un directorio publicado en un servidor web (puede ser Apache, pero generalmente yo uso **lighttpd** que es mucho más sencillo de configurar para pruebas). También estoy asumiendo que los paquetes de software son para arquitectura i386 (procesadores compatibles con Intel x86). Otras arquitecturas válidas que puedes ver son `amd64`, `powerpc` y `sparc`.

```
$ mkdir dists
$ mkdir dists/gutsy
$ mkdir dists/gutsy/main
$ mkdir dists/gutsy/main/binary-i386
```

Copiamos los archivos Deb (que queremos colocar en el repositorio) al directorio de destino:

```
$ cp /tmp/source/*.deb dists/gutsy/main/binary-i386
```

Creamos los archivos de descripción de los paquetes (llamado Packages) y lo comprimimos:

```
$ apt-ftparchive packages dists/gutsy/main/binary-i386/ \
> dists/gutsy/main/binary-i386/Packages
$ cat dists/gutsy/main/binary-i386/Packages | gzip -9c \
> dists/gutsy/main/binary-i386/Packages.gz
```

Creamos un archivo de descripción de nuestro repositorio, llamado `apt.conf`. Aquí reproduzco el que utilizan los desarrolladores de Automatix:

```
APT::FTPArchive::Release::Origin "automatix";
APT::FTPArchive::Release::Label "automatix updates";
APT::FTPArchive::Release::Suite "stable";
APT::FTPArchive::Release::Codename "gutsy";
APT::FTPArchive::Release::Architectures "i386 source";
APT::FTPArchive::Release::Components "main testing";
APT::FTPArchive::Release::Description "This repository contains ...";
```

con lo cual creamos un archivo de Release. Este archivo permite agregar más metadatos sobre el propósito del repositorio, las arquitecturas soportadas y la versión de Ubuntu o Debian para la cual queremos crear el repositorio.

```
$ apt-ftparchive -c apt.conf release dists/gutsy/ \
> dists/gutsy/Release
```

Finalmente, utilizamos firma digital sobre este archivo de Release. Esto permitirá que, si hemos agregado la llave pública a nuestro depósito de firmas, el actualizador de paquetes valide las firmas digitales de forma automática y sin alertas de seguridad para nuestros usuarios.

```
$ gpg --output dists/gutsy/Release.gpg -ba dists/gutsy/Release
```

Con esto, nuestro software queda disponible para que los usuarios agreguen la dirección web en la que hemos colocado el repositorio a su archivo `/etc/apt/sources.list` o mediante el menú **Sistema - Administración - Orígenes de Software - Software de Otros Proveedores - Agregar** y puedan instalar los paquetes de software del repositorio usando `apt-get install` o `synaptic`, etc.

Si la carpeta raíz (donde hemos creado las carpetas `dists`) y las otras están visibles en la dirección web `http://127.0.0.1/ubuntu`, entonces esta dirección web será la dirección que deberás utilizar para la línea de APT de tu repositorio.

Una línea de APT tiene la siguiente apariencia:

```
deb http://www.servidor.com/ruta/ release component1 componente2 ...
```

donde **release** es la versión de Ubuntu o Debian para que estamos creando el repositorio (en el ejemplo anterior, era `gutsy`). Los componentes 1, 2, etc., son los diferentes componentes del repositorio de acuerdo a si son software libre parte de la distribución oficial (**main**) o son software con una licencia no libre (**restricted**), software libre no mantenido por Ubuntu (**universe**) o software ni libre ni mantenido por Ubuntu (**multiverse**). Generalmente nuestros repositorios caerán en una de las 2 últimas categorías, por lo que nuestros repositorios tendrán una línea de APT como la siguiente:

```
deb http://www.miservidor.com/ubuntu/ gutsy universe
```

NOTA: En el ejemplo de estructura de directorios anterior hemos usado el componente `main` para colocar nuestros paquetes de software, por lo que la línea de APT correcta será

```
deb http://www.miservidor.com/ubuntu/ gutsy main
```

Esta es una receta para crear un repositorio de forma más o menos sencilla. Existen otras maneras que permiten optimizar espacio en disco cuando existen paquetes binarios con el mismo contenido, y varios releases y arquitecturas distintas en un mismo servidor y repositorio. La fuente oficial de documentación para estos casos es el Debian Repository HOWTO [www.debian.org/doc/manuals/repository-howto/repository-howto]

Soporte para Internacionalización en los repositorios

Ubuntu posee un amplio soporte para hacer que el sistema sea más accesible para usuarios muchos idiomas y eso incluye la capacidad para usar descripciones de los paquetes de software instalables en diferentes idiomas. La especificación del soporte para descripciones multilinguaje en los repositorios se encuentra en el documento Translated Package Descriptions Spec [<https://wiki.ubuntu.com/TranslatedPackageDescriptionsSpec>].

Para utilizar traducciones de los paquetes de software, es necesario crear un directorio en el repositorio que contenga los archivos con las traducciones de la información de paquetes:

```
$ mkdir dists/gutsy/main/i18n
```

En este directorio se colocan diferentes archivos comprimidos que siguen la siguiente convención de nombres: `Translation-XX.bz2` donde `XX` es el código ISO de 2 letras que identifica el idioma. En el caso de las traducciones al español, será el archivo comprimido `Translation-es.bz2`

El contenido del archivo será un conjunto de entradas que contienen un campo `Package:`, un campo con la suma MD5 de la descripción original del paquete en la versión en inglés (`Description-md5:`) y la descripción en el idioma de la traducción (`Description-es:` para la versión en español).

Para la traducción de mi repositorio en el que tengo el paquete `gtkfilesplitter`, el contenido del archivo `dists/gutsy/main/i18n/Translation-es` es la siguiente:

```
Package: gtkfilesplitter
Description-md5: bc3667f71f65f0ffa387067d48c2a620
Description-es: Aplicación sencilla que troza y junta archivos con comprobación
Una aplicación de escritorio para dividir archivos en trozos y volver a unirlos.
```

Para más información, visite <http://code.google.com/p/gtkfilesplitter/>

... luego un salto de línea, y se repite para tantos paquetes como tengamos en el repositorio.

Unos detalles adicionales para calcular la suma MD5 que se requiere colocar: La forma más sencilla de realizar el cálculo es crear un archivo temporal y colocar la descripción completa del paquete de software como viene en el archivo `dists/gutsy/main/binary-i386/Packages`, sin incluir el comienzo de la línea "Description: " (espacio luego de los dos puntos inclusive) de modo que quede así:

```
$ cat /tmp/descripcion.txt
Simple file splitter/joiner with checksum support
A desktop application to split and join files
.
For more information, visit http://code.google.com/p/gtkfilesplitter
```

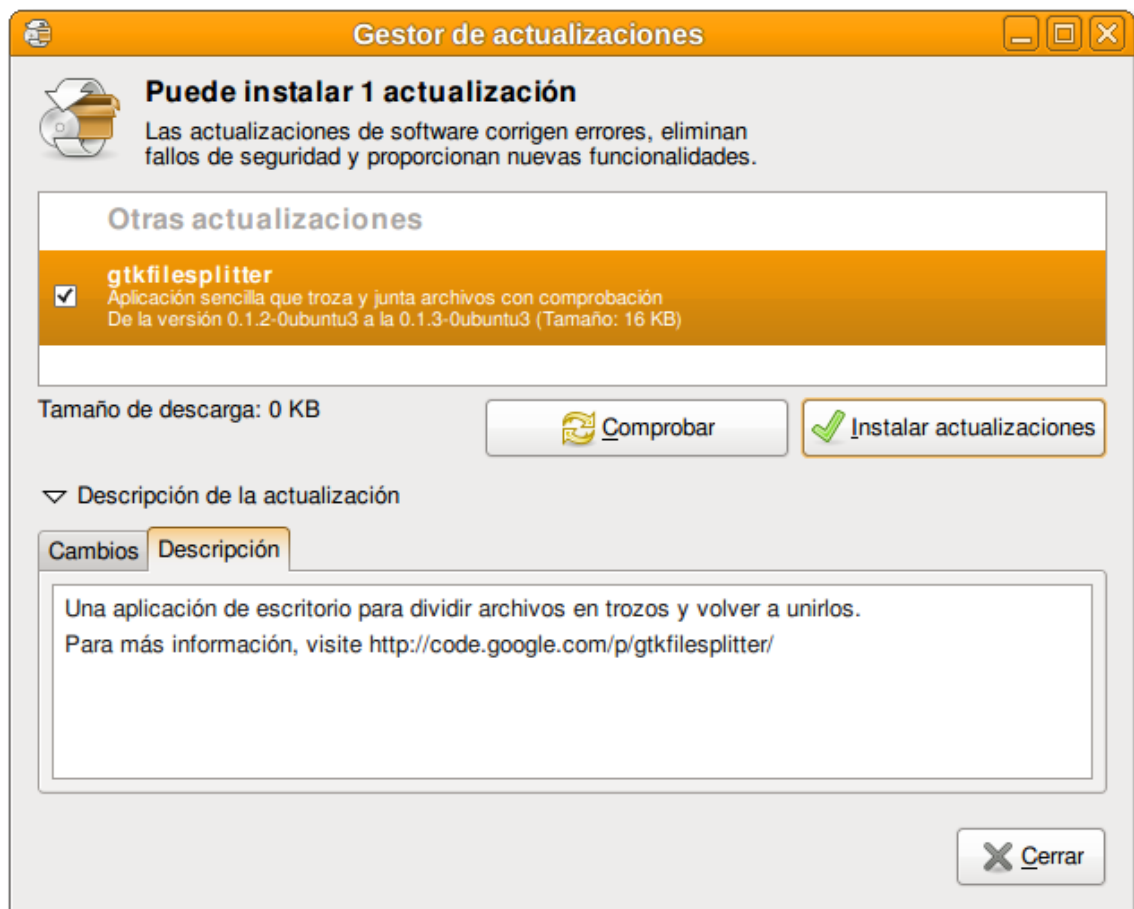
Para calcular la suma MD5 que necesitamos para la traducción, basta con usar:

```
$ md5sum /tmp/descripcion.txt
bc3667f71f65f0ffa387067d48c2a620 */tmp/descripcion.txt
```

y copiamos el valor de la suma MD5 en el campo que corresponde en nuestro archivo con traducciones. Finalmente, el archivo debe ir comprimido con **bzip2** de la siguiente forma:

```
$ bzip2 dists/gutsy/main/i18n/Translation-es
```

De esta manera, las traducciones aparecen al actualizar la información de las actualizaciones posibles en el Gestor de Actualizaciones (Update Manager):



Finalmente, la lista de cambios que debería aparecer en la lengüeta **Cambios** sólo aparece para paquetes que se encuentren oficialmente en la distribución. Revisé el código y en el caso de Ubuntu, el listado de cambios se descarga desde el servidor `changelogs.ubuntu.com` siguiendo una convención de directorios, que en caso de un paquete como `abiword` sería `http://changelogs.ubuntu.com/changelogs/pool/main/a/abiword/abiword_2.4.6-3ubuntu2/changelog`. En el caso de Debian, el listado de cambios se obtiene desde su propio servidor, y en el ejemplo anterior, la dirección es `http://changelogs.debian.net/abiword`