HOWTO Gestión de proyectos de software libre.

Benjamin Mako Hill

COMENZANDO UN PROYECTO.

Sin lugar a dudas, el comienzo es el periodo más difícil de la vida de la gestión de un proyecto libre de éxito. Establecer una buena base condiciona si tu proyecto florece o se marchita y muere. También es el principal tema de interés para cualquier lector de este documento como tutorial.

Comenzar un proyecto supone un dilema con el que, como desarrollador, deberás tratar: ningún usuario potencial de tu programa está interesado en un programa que no funciona, mientras que el proceso de desarrollo que quieres emplear necesita la imprescindible participación de los usuarios.

Es en estos peligrosos momentos iniciales en los que cualquier interesado en comenzar un proyecto de software libre debe buscar el equilibrio en lo que se expone en las siguientes líneas. Una de las más importantes maneras que se pueden hacer para encontrar este equilibrio consiste en establecer un marco de trabajo sólido a partir de algunas de las sugerencias mencionadas en esta sección.

ELIGIENDO UN PROYECTO.

Si estás leyendo este documento, lo más probable es que ya tengas en mente una idea de un proyecto. Probablemente son buenas ya que llenan un vacío haciendo algo que ningún otro software hace o haciendo algo de una manera que implica un nuevo componente de software.

Identifica y expresa tu idea.

Eric S. Raymond escribe sobre cómo empiezan los proyectos de software libre en su ensayo, “La Catedral y el Bazar”, cuya lectura es imprescindible para todo desarrollador de software. Este ensayo es accesible online.

En “La Catedral y el Bazar”, Raymond nos dice: “todo buen trabajo en software empieza cuando a un desarrollador “le pica algo”. La actualmente aceptada hipótesis de Raymond consiste en que un nuevo programa de software se ha escrito, antes de nada, para resolver un problema específico del desarrollador.

Si tienes en mente una idea de un programa, probablemente su objetivo es el de solucionar un problema específico o “quitarse la picor”. ***Esta idea es el proyecto***. Exprésala claramente. Escríbela. Describe en detalle el problema que vas a atacar. El éxito de tu proyecto al abordar un problema particular está relacionado con tu habilidad para identificar ese problema de forma clara y lo antes posible. Descubre de forma exacta qué es lo que quieres que haga tu proyecto.

Monty Manley expresa la importancia de este paso inicial en un ensayo, “Managing Projects The Open Source Way”. Como verás más adelante, hay una gran cantidad de trabajo a hacer antes incluso que el software esté listo para codificar. Manley dice, “Comenzar un proyecto de software libre de forma correcta significa que el desarrollador debe, ante todo, evitar escribir código demasiado pronto”.

Valora tu idea.

Para valorar tu idea, debes primera preguntarte a ti mismo unas cuantas preguntas. Además, debes hacerlo antes de seguir adelante con este HOWTO. Pregúntate: ¿Seguro que el modelo de desarrollo de software libre es el adecuado para tu proyecto?.

Obviamente, desde el momento que el programa "rasca tu picor", estás totalmente decidido a verlo implementado en código. Pero, debido a que un hacker codificando en soledad se equivoca al calificar su esfuerzo como desarrollo de software libre, debes preguntarte una segunda pregunta: ¿Hay alguien más interesado?.

A veces la respuesta es un simple "no". Si quieres escribir un conjunto de scripts para ordenar tu colección de MP3s de tu máquina, quizás el modelo de desarrollo de software libre no es la mejor opción. Sin embargo, si quieres escribir un conjunto de scripts para ordenar los MP3s de cualquier persona, un proyecto de software libre podría ser útil.

Por suerte, Internet es un lugar tan grande y diverso que, probablemente, hay alguien, en algún lugar, que comparte tus intereses y experimenta la misma "picor". Al ser un hecho que hay tantas personas con necesidades y deseos similares llegamos a la tercera pregunta: ¿Alguien ha tenido ya una idea igual o bastante parecida a la tuya?.

Encontrando proyectos similares.

Existen sitios en la web donde intentar contestar esta pregunta. Si tienes experiencia en la comunidad del software libre, es posible que te sean familiares muchos de esos sitios. Todos los recursos enumerados a continuación ofrecen la posibilidad de buscar en sus bases de datos.

freshmeat.net

> freshmeat.net se describe como, "El catálogo más grande de Linux y Software Open Source de la Web" y su reputación a lo largo de estas líneas es del todo incomparable e incuestionable. Si no lo encuentras en freshmeat, dudo que tú (o cualquier otro) lo encuentre.

Slashdot

> Slashdot proporciona "Noticias para Nerds. Cosas que importan", que normalmente incluyen debate sobre el software libre, código abierto, tecnología, y noticias y eventos sobre la cultura geek. No es poco corriente para un esfuerzo de desarrollo interesante el ser dado a conocer en este sitio, por lo que vale la pena comprobar nuestra búsqueda en él.

SourceForge

> SourceForge aloja y facilita un creciente número de proyectos de software libre y Open Source. Se está convirtiendo rápidamente en un nexo y una necesaria parada para desarrolladores de software libre. El mapa de software de SourceForge y las páginas de nuevas liberaciones de software deberían ser paradas necesarias antes de embarcarse en un nuevo proyecto de software libre. SourceForge proporciona también una Biblioteca de fragmentos de Código que contiene pedazos de código útiles y reutilizables en una selección de lenguajes que pueden ser útiles en algún proyecto.

Google y Google's Linuz Search proporcionan potentes utilidades de búsqueda que pueden mostrar personas trabajando en proyectos similares. No es un catálogo de software como freshmeat o Slashdot, pero vale la utilizarlas para estar seguro de que no estás dedicando tu esfuerzo en un proyecto redundante.

Decidiendo continuar.

Una vez que has tanteado el terreno de forma satisfactoria y tienes una idea sobre qué tipo de software similar existe, todo desarrollador necesita decidir continuar con su propio proyecto. Es raro que un nuevo proyecto busque lograr un objetivo que no sea totalmente similar o no esté relacionado de ninguna manera al el objetivo de otro proyecto. Cualquiera que arranque un nuevo proyecto debe preguntarse: ¿Estará el nuevo proyecto duplicando trabajo hecho ya por otro proyecto? ¿Estará el nuevo proyecto compitiendo por desarrolladores de un proyecto existente? ¿Es posible conseguir el objetivo del nuevo proyecto añadiendo funcionalidad a un proyecto existente?.

Si la respuesta a alguna de estas preguntas es “si”, intenta contactar con el desarrollador del proyecto o proyectos existentes para saber si estaría dispuesto a colaborar contigo.

Para muchos desarrolladores este puede ser la parte más difícil de la gestión de un proyecto de software libre, pero se trata de una parte esencial. Es fácil que se te encienda la bombilla y te dejes llevar por el momento y el entusiasmo de un nuevo proyecto. A menudo hacerlo es dificilísimo, pero es importante que todo desarrollador de software libre recuerde que a menudo no arrancar un esfuerzo de nuevo desarrollo es más provechoso para la comunidad de software libre y es la manera más rápida de lograr los objetivos de tu propio proyecto así como los objetivos de proyectos similares.

Poniendo nombre a tu proyecto.

Aunque existen muchos proyectos que fracasan con nombres descriptivos y muchos otros que han tenido éxito sin, creo que es bueno pensar un poco a la hora de ponerle nombre a tu proyecto. Leslie Orchard trata esta cuestión en un Advogato article. Su artículo es pequeño y decididamente es bueno examinarlo deprisa.

La sinopsis es que Orchard recomienda que elijas un nombre que, después de escucharlo, muchos usuarios o desarrolladores:

- Sepan lo que hace el proyecto.
- Recuerden el nombre mañana.

Curiosamente, el proyecto de Orchard, “Iajitsu”, no cumple ninguna de las dos características. Probablemente no hay ninguna relación con que el desarrollo haya sido parado desde que este artículo fue escrito.

Sin embargo, tiene un punto interesante. Hay empresas cuya única función es poner nombre a componentes de software. Hacen cantidades ridículas de dinero haciéndolo y supuestamente lo vale. Aunque puedas pagar una empresa de esas, puedes también saber de su existencia y pensar un poco sobre el nombre que le estás dando a tu proyecto porque vale la pena.

Si realmente quieres un nombre pero no se ajusta al criterio de Orchard, puedes seguir adelante. Pienso que “gnubile” fue uno de los mejores nombres de proyectos de software libre que he

conocido y sigo hablando de él mucho tiempo después de haber dejado de utilizar el programa. No obstante, si puedes ser flexible sobre este asunto, haz caso del consejo de Orchard. Podría ayudarte.

Licenciando tu Software.

En un (simple) nivel, la diferencia entre un componente de software libre y un componente de software propietario es la licencia. Una licencia te ayuda como desarrollador a proteger tus derechos legales para distribuir tu software bajo tus términos y ayuda a demostrar a aquellos que quieren ayudarte a ti o a tu proyecto que están invitados a unirse.

Eligiendo una licencia.

Cualquier debate sobre licencias generará seguramente al menos una pequeña "guerra"[1] ya que existen fuertes opiniones sobre si algunas licencias de software libre son mejores que otras. De este debate surge también la cuestión de "Software de código abierto" y el debate sobre los términos "Software de código abierto" y "Software libre". Sin embargo, como este HOWTO que escribo se refiere a la gestión de proyectos de software libre y no a la gestión de proyectos de software de código abierto, mis propias inclinaciones sobre este tema son evidentes.

Intentando buscar el punto medio diplomáticamente sin sacrificar mi propia filosofía, recomiendo escoger cualquier licencia que se ajuste a las pautas de la Debian Free Software. Originariamente recopilado por el proyecto Debian bajo Bruce Perens, las pautas forman la primera versión de la definición Open Source. Ejemplos de licencias libres creadas a partir de estas pautas son la GPL, la BSD, y la Artistic License. Así como ESR (Eric S. Raymond) menciona en su HOWTO [ESRHOWTO], no escribas tu propia licencia en cualquier caso. Las tres licencias que he mencionado tienen una larga tradición de interpretación. Indudablemente también hay software libre (y puede por consiguiente ser distribuido como parte de Debian y en otros lugares que permitan transferir software libre).

De acuerdo con la definición de software libre propuesta por Richard Stallman en "The Free Software Definition", cualquier de estas licencias conservarán, a los usuarios libres de ejecutar, copiar, distribuir. Estudiar, cambiar y mejorar el software. Existen muchas otras licencias que también se ajustan a las pautas de Debian Free Software pero adheriéndose a una licencia más conocida proveerá la ventaja de ser reconocido y comprendido de forma inmediata. Mucha gente escriben tres o cuatro frases en una documento y asumen que han escrito una licencia de software libre—mi larga experiencia con el mailing de manifiestos legales de debian, por lo general no es el caso.

Tratando de hacer un análisis más profundo, estoy de acuerdo con la descripción que Karl Fogel hace sobre las licencias diferenciándolas en dos grupos: las que son GPL y las que no son GPL.

Personalmente, yo licencio todo mi software bajo GPL. Creada y protegida por la Free Software Foundation y el proyecto GNU, GPL es la licencia del kernel de Linux, GNOME, Emacs, y la gran mayoría de software GNU/linux. Es la elección obvia pero también creo que es buena. Cualquier fanático de BSD insistirá en recordarte sobre el aspecto viral de GPL que impide mezclar código GPL con código no GPL. Para mucha gente (yo incluido), es un beneficio, pero para otros, es un gran inconveniente.

1 Del original: Flame war.

Mucha gente escriben tres o cuatro frases en una documento y asumen que han escrito una licencia de software libre—mi larga experiencia con el mailing de manifiestos legales de debian, por lo general no es el caso. No te protegerá, no protegerá tu software, y hará las cosas muy difíciles a aquellos que quieran utilizar tu software pero presten mucha atención a los puntos sutiles legales de las licencias.

Si te apasiona una licencia "hecha en casa", antes de utilizarla compruébala con la gente de OSI o con la mailing-list de debian-legal para protegerte de efectos laterales inesperados de tu licencia.

Las tres mayores licencias pueden encontrarse en los siguientes enlaces:

- GNU General Public License.
- BSD license.
- Artistic License.

En cualquier caso, por favor lee cualquier licencia antes de liberar tu software bajo ella. Como programador principal, no puedes permitirte ninguna sorpresa respecto a la licencia.

Los mecanismos de licenciar.

El texto de GPL proporciona una buena descripción de los mecanismos de aplicación de una licencia a un componente de software. Mi lista rápida de comprobación al aplicar una licencia incluye:

- Declárate a ti mismo o a la FSF como titular del copyright del trabajo. En algunos pocos casos, podrías querer definir como titular a una organización patrocinadora (si es suficientemente grande y potente). Haciendo esto es tan simple como poner el nombre en el espacio a tal efecto cuando modificas el aviso de copyright que se presenta a continuación. Al contrario de lo que se cree, no es necesario que sea una organización. El aviso es suficiente por sí solo para registrar el copyright de tu trabajo.
- De ser posible, adjunta y distribuye una copia completa de la licencia con el código fuente y binario/s en un fichero separado.
- Incluye un aviso de copyright al principio de cada fichero de código fuente de tu programa, además de incluir también la información de dónde se puede encontrar la licencia completa. GPL recomienda que cada fichero empiece con:

```
one line to give the program's name and an idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

GPL también recomienda incluir información de cómo contactar contigo (el autor) via email o correo postal.

- GPL sugiere que si tu programa se ejecuta en modo interactivo, deberías escribir el programa de modo que presente un aviso cada vez que se inicia el modo interactivo que incluya el mensaje como el siguiente que indique la información completa sobre la licencia del programa:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

- Por último, podría ser de ayuda incluir una "limitación de responsabilidad del copyright" para un empresario o escuela si trabajas como programador o si piensas que tu empresario o escuela podría iniciar más adelante polémica sobre la propiedad de tu código. No es que sea muy necesario pero hay muchos desarrolladores de software libre que han tenido problemas sobre este punto y desearon tener una.

Una última advertencia sobre licenciar.

Por favor, por favor, por favor, registra tu software bajo alguna licencia. Puede no parecer importante, y para tí podría no serlo, pero las licencias son importantes. Para que un componente de software se incluya en la distribución Debian GNU/Linux, debe estar licenciado y que su licencia se ajuste a las pautas de la Debian Free Software. Si tu software no está licenciado, no puede distribuirse como paquete en Debian hasta que lo re-liberes bajo una licencia libre. Por favor, evítate problemas a ti mismo y a los demás liberando la primera versión de tu software bajo una licencia y de forma clara.

Eligiendo un método de numeración de versiones.

Lo más importante sobre un sistema de numeración de versiones es que tengas uno. Puede parecer pedante enfatizar este punto pero te sorprenderías cuántos scripts y pequeños programas aparecen sin ningún número de versión.

Lo segundo más importante sobre un sistema de numeración es que los números siempre van en sentido creciente. Los sistemas automáticos de versiones y el sentido del orden del universo se desintegrarán si los números de versiones no crecen. No importa si 2.1 es un gran salto y 2.0.005 es un pequeño salto pero sí importa que 2.1 sea más reciente que 2.0.005.

Sigue estas dos simples reglas y no fallarás (demasiado). A partir de aquí, la técnica más común parece ser la del esquema de numeración "nivel superior", "nivel inferior", "nivel de parche". Ya te sea o no familiar el nombre, estás relacionado con él todo el tiempo. El primer número es el número superior y significa cambios o recodificaciones importantes. El segundo número es el inferior y representa funcionalidad añadida al principio de una estructura lo bastante coherente. El tercer número es el número de parche y normalmente se relaciona con arreglo de errores.

El uso extendido de este esquema me permite saber la naturaleza y el grado relativo de las diferencias entre el release 2.4.12 del kernel de Linux y el 2.4.11, 2.2.12 o el 1.2.12 sin saber nada de estos releases.

Puedes romper con estas reglas, como hacen algunos. Pero ten cuidado, si lo haces, alguien se enfadará, asumirá que no sabes, e intentará enseñarte, seguramente de manera no agradable. Yo siempre sigo este método y te imploro que lo hagas tú también.

Existen varios sistemas de numeración conocidos, útiles, y que valen la pena mirarlos antes de liberar tu primera versión.

Numeración de las versiones del Kernel de Linux.

> El kernel de Linux utiliza un sistema de versiones que cualquier número de versión "inferior" impar se refiere a un release de desarrollo o prueba y cualquier número de versión "inferior" par se refiere a una versión estable. Piénsalo un momento. Bajo este sistema, los kernel 2.1 y 2.3 han sido y serán siempre kernels de desarrollo o prueba, y los kernel 2.0, 2.2 y 2.4 son código de producción con un grado superior de estabilidad y prueba.
>
> Tanto si piensas en tener un modelo de desarrollo dividido[2] (como se describe en la sección Ramas estables y en desarrollo) o liberar solo una versión cada vez, mi experiencia con varios proyectos de software libre y con el proyecto Debian me ha enseñado que el uso del sistema de numeración de versiones de Linux es digno de tenerse en cuenta. En Debian, todas las versiones "inferiores" son estables (2.0, 2.1, etc). No obstante, muchas personas asumen que 2.1 es una versión inestable o en desarrollo y continua usando una versión más antigua hasta que se ven tan frustrados por la carencia del progreso del desarrollo que se quejan y entonces entienden el sistema. Si nunca liberas una versión "inferior" impar sino solo versiones pares, nadie se perjudica, y hay menos personas confundidas. Vale la pena tener en cuenta esta idea.

Numeración de versiones de Wine:

> Debido a la naturaleza inusual del desarrollo de Wine en donde el no-emulador está siendo constantemente mejorado pero no siendo trabajado hacia un objetivo específico, wine se libera cada tres semanas. Wine lo hace etiquetando sus releases en un formato "Año Mes Día" de forma que cada release se etiquettaría "wine-XXXXXXXX" donde la versión del 04 de Enero de 2000 sería "wine-20000104". Para algunos proyectos, el formato "Año Mes Día" tiene mucho sentido.

Hitos del desarrollo de Mozilla:

> Cuando alguien tiene en cuenta Netscape 6 y las versiones del proveedor, la estructura del desarrollo del proyecto mozilla es uno de los más complicados modelos de proyectos de software libre. La numeración de versiones del proyecto refleja la especial circunstancia en la que se ha desarrollado.
>
> La estructura de la numeración de versiones de mozilla se ha elaborado históricamente en base a hitos. Desde el principio del proyecto mozilla, los objetivos del proyecto en el orden y grado de su consecución se trazaron en una serie de "road map". Las mayores logros y puntos a conseguir se marcan como hitos en estos "road maps". Por lo tanto, aunque Mozilla se compiló y distribuyó diariamente[3] como "compilaciones de un día para otro"[4], cuando un día se han

2 Del texto original "split development model"
3 Del original "nigthly".
4 Del original "nightly builds".

conseguido los objetivos de un hito en el “road-map”, esta compilación se marca como “hito release”[5].

Aunque no he visto este método empleado en otros proyectos hasta ahora, me gusta la idea y creo que podría ser valiosa en la división de pruebas o de desarrollo de una gran aplicación con un fuerte desarrollo.

Documentación.

Un gran número de aplicaciones software libres estupendas se han marchitado y muerto porque su autor fue la única persona que la conocía totalmente. Incluso si tu programa está escrito principalmente para un grupo de usuarios tecno-inteligentes, la documentación es de ayuda e incluso necesaria para la supervivencia de tu proyecto. Más adelante, en este documento, aprenderás en la sección llamada “liberando tu programa” que debes siempre liberar algo que se pueda utilizar. Un componente de software sin documentación no es utilizable.

Hay muchas personas diferentes que debes documentar y hay muchas maneras de documentar tu proyecto. La importancia de la documentación en el código fuente para facilitar el desarrollo por una gran comunidad es vital pero está fuera del alcance de este HOWTO. En nuestro caso, esta sección trata de tácticas útiles para la documentación dirigida a los usuarios.

Una combinación de tradición y necesidad ha tenido como consecuencia un sistema de documentación medio-habitual en la mayoría de proyectos de software libre que vale la pena seguir. Tanto usuarios como desarrolladores esperan poder acceder a la documentación de varias maneras y es esencial que proporciones la información que buscan de una manera que puedan leerla incluso si tu proyecto está a punto de “despegar”. Lo que se espera tener es:

Páginas Man.

Tus usuarios querrán poder teclear “man nombredetuproyecto” y que aparezca una página man bien formateada resaltando la utilización básica de tu aplicación. Comprueba que antes de liberar tu programa, lo hayas previsto.

Las páginas man no son difíciles de escribir. Hay una excelente documentación sobre el proceso de escritura de una página man en “The Linux Man-Page-HOWTO” que está accesible en el Linux Documentation Project (LDP) y está escrito por Jens Schweikhardt. Lo encontrarás en el Schweikhardt's site o en la LDP.

También es posible escribir página man utilizando DocBook SGML. Dado que las páginas man son tan sencillas y el método DocBook relativamente nuevo, no he podido experimentar esta vía y agradeceré cualquier ayuda de alguien que pueda darme más información sobre cómo hacerlo.

Documentación accesible desde la línea de comandos.

La mayoría de usuarios esperarán alguna información básica fácilmente disponible desde la línea de comandos. Para algunos pocos programas este tipo de documentación se extenderá más de una pantalla (24 o 25 líneas) pero debería cubrir la utilización básica, una breve (una o dos frases) descripción del programa, una lista de comandos con explicaciones, así como las opciones más

5 Del original “milestone release”.

importantes (también con explicaciones), además de la manera de acceder a la documentación más detallada para aquellos que la necesiten. La documentación de la línea de comandos para el apt-get de Debian sirve como un excelente ejemplo y como modelo útil.

```
apt 0.3.19 for i386 compiled on May 12 2000  21:17:27
Usage: apt-get [options] command
       apt-get [options] install pkg1 [pkg2 ...]

apt-get is a simple command line interface for downloading and
installing packages. The most frequently used commands are update
and install.

Commands:
   update - Retrieve new lists of packages
   upgrade - Perform an upgrade
   install - Install new packages (pkg is libc6 not libc6.deb)
   remove - Remove packages
   source - Download source archives
   dist-upgrade - Distribution upgrade, see apt-get(8)
   dselect-upgrade - Follow dselect selections
   clean - Erase downloaded archive files
   autoclean - Erase old downloaded archive files
   check - Verify that there are no broken dependencies

Options:
  -h  This help text.
  -q  Loggable output - no progress indicator
  -qq No output except for errors
  -d  Download only - do NOT install or unpack archives
  -s  No-act. Perform ordering simulation
  -y  Assume Yes to all queries and do not prompt
  -f  Attempt to continue if the integrity check fails
  -m  Attempt to continue if archives are unlocatable
  -u  Show a list of upgraded packages as well
  -b  Build the source package after fetching it
  -c=? Read this configuration file
  -o=? Set an arbitary configuration option, eg -o dir::cache=/tmp
See the apt-get(8), sources.list(5) and apt.conf(5) manual
pages for more information and options.
```

Se ha convertido en una convención GNU el hacer este tipo de información accesible con las opciones “-h” y “--help”. La mayoría de usuarios GNU/Linux esperarán poder recuperar la documentación básica por estas vías por lo que si elijes utilizar otros métodos diferentes, estáte preparado para el fuego y la lluvia radiactiva.

Ficheros que esperarán los usuarios.

Además de las páginas man y la ayuda de la línea de comandos, existen con seguridad ficheros donde se buscará documentación, especialmente en los paquetes que contienen código fuente. En una distribución, la mayoría de estos ficheros pueden estar almacenados en el directorio root de la distribución o en un subdirectorio de root llamado “doc” o “Documentation”. Los ficheros que comúnmente se incluyen son:

README o Readme

> Un documento que contiene la instalación básica, compilación, e incluso instrucciones básicas de utilización que maquilla la mínima información pura necesaria para conseguir poner en

marcha y ejecutar el programa. Lo más probable es que un README no sea muy detallado pero debe ser conciso y efectivo. Un README ideal tiene al menos 30 lineas y no más de 250.

INSTALL o Install

El fichero INSTALL debería ser mucho más pequeño que el README y debería describir de forma concisa y rápida cómo compilar e instalar el programa. Normalmente un fichero INSTALL simplemente ordena al usuario ejecutar "./configure; make; make install" y menciona cualquier opción inusual o acciones que puedan ser necesarias. Para la mayoría de procedimientos de instalación estándar y programas, los ficheros INSTALL son lo más pequeño posible y raramente sobrepasan las 100 líneas.

CHANGELOG, Changelog, ChangeLog o changelog

Un CHANGELOG es un fichero sencillo que todo software libre bien gestionado debería incluir. Un CHANGELOG es sencillamente el fichero que, como su nombre indica, registra o documenta los cambios que se hacen a tu programa. La forma más sencilla de mantener un CHANGELOG es conservar un fichero con el código fuente de tu programa y añadir una sección al principio de CHANGELOG con cada release describiendo qué se ha cambiado, arreglado, o añadido al programa. Es una buena idea enviar también el CHANGELOG a la website porque puede ayudar a otros a la hora de decidir si necesitan actualizar a una nueva versión o esperar una mejoría más importante.

NEWS

Un fichero NEWS y un ChangeLog son parecidos. A diferencia del CHANGELOG, un fichero NEWS no se actualiza con nuevas versiones normalmente. Siempre que se añaden nuevas características, el desarrollador responsable creará una anotación en el fichero NEWS. Los ficheros NEWS no deberían cambiarse antes de una release (deberían estar al día todo el tiempo) pero es una buena idea comprobarlo ya que a menudo los desarrolladores olvidan mantenerlos al día como deberían.

FAQ

Para aquellos que no lo sepan, FAQ son las siglas de Frequently Asked Questions y un FAQ es exactamente un conjunto de preguntas frecuentes. Los FAQ no son difíciles de crear. Solo crea una política tal que si te hacen una misma pregunta o la ves en una lista de correo dos o más veces, añade la pregunta (y su respuesta) a tu FAQ. Los FAQ son más opcionales que los ficheros anteriores pero te pueden ahorrar tiempo, aumentar la usabilidad, y en cualquier caso disminuir los dolores de cabeza.

Website

Se trata indirectamente de una cuestión de documentación pero un buen website se está convirtiendo rápidamente en una parte esencial de cualquier proyecto de software libre. Tu website debería proveer acceso a tu documentación (si es posible en HTML). Debería también incluir una sección de noticias y acontecimientos relacionados con tu programa y una sección que detalla el proceso de involucrarse en el desarrollo o pruebas y ofrecer una invitación a todos los que lo deseen. Debería también suministrar enlaces a listas de correo, websites similares, y proveer un

enlace directo a todas las formas posibles de descargar tu software.

Otros consejos sobre documentación.

- Toda tu documentación debería ser en texto sin formato, o, en los casos en que está principalmente en tu website, en HTML. Todo el mundo puede crear un fichero de texto, todo el mundo tiene un beeper[6], (casi) todo el mundo puede crear HTML. Puedes distribuir información en PDF, PostScript, RTF, o cualquier número de otros formatos muy utilizados pero esta información debe estar también disponible en texto sin formato o HTML o la gente se va a enfadar mucho contigo. Según mi opinión, el formato info entra en esta categoría. Existe mucha y muy buena documentación GNU que la gente simplemente no lee porque solo está en formato info. Y esto hace que la gente se enfade. No se trata de formatos importantes; se trata de accesibilidad y el status quo juega un papel importante en esta determinación.
- No hace daño distribuir cualquier documentación para tu programa de tu website (FAQs etc) incluida con tu programa. No vaciles en incluirlo en el tarball[7] del programa. Si la gente no lo necesita, lo borrará. Puedo repetirlo una y otra vez: **Demasiada documentación no es pecado.**
- A menos que tu software se haya desarrollado para un idioma que no sea el inglés (por ejemplo un editor de textos en japonés), por favor distribúyelo con la documentación en inglés. Si no hablas inglés o no confías en tus aptitudes, pide ayuda a un amigo. Te guste o no, sea justo o no, el inglés es el idioma del software libre. No obstante, esto no significa que debas limitar tu documentación a solo en inglés. Si hablas otro idioma, distribuye traducciones de la documentación con tu software si tienes tiempo y energías para hacerlo. Siempre será de utilidad para alguien.
- Para terminar, **por favor comprueba la ortografía de tu documentación**. Los errores ortográficos en la documentación son errores[8]. Yo soy propenso a cometer este error y es muy fácil caer en él. Si el inglés no es tu idioma nativo, consigue que alguien que sí lo sea lo revise o modifique tu documentación o páginas web. Una ortografía o gramática deficiente contribuirá a que tu código parezca poco profesional. En los comentarios del código fuente, este punto es menos importante pero en las páginas man y las páginas web estos errores no son aceptables.

Otras cuestiones sobre la entrega

Muchas de las restantes cuestiones alrededor de la creación de un nuevo programa software caen dentro de lo que mucha gente denomina como cuestiones de sentido común. Se ha dicho a menudo que la ingeniería del software es un 90% de sentido común combinado con un 10% de conocimiento especializado. De todos modos, son dignas de mencionar brevemente con la esperanza de que pudiesen hacer recordar algo a un desarrollador que pudiese haber olvidado.

Nombres del paquete.

Estoy de acuerdo con ESR[9] cuando dice que: “Es de ayuda para todo el mundo el que todos tus ficheros comprimidos[10] tengan nombre tipo GNU – prefijo con todos caracteres alfanuméricos en

6 Del original: Pager.
7 Paquete preparado para ser compilado e instalado, generalmente construidos con el comando tar.
8 Del original “bugs”.
9 Eric S. Raymond
10 Del original “archive files”.

minúsculas, seguido de un guión, seguido de un número de versión, extensión, y otros sufijos". Hay más información (incluyendo muchos ejemplos de qué es lo que no hay que hacer en su **Software Release Practices HOWTO** que está incluida en la bibliografía de este HOWTO y que puede encontrarse en el LDP[11].

Formatos del paquete.

Los formatos del paquete puede diferir dependiendo del sistema en el que desarrollas. Para el software basado en windows, los ficheros Zip (.zip) son normalmente el formato elegido. Si desarrollas para GNU/Linux, *BSD, o cualquier UN*X, asegúrate que tu código fuente esté disponible en formato tar y gzip (.tar.gz). La compresión UNIX (.Z) ha pasado tanto de moda como de su utilización y los ordenadores más rápidos han resaltadoel formato bzip2 (.bz2) como un medio de compresión más efectivo. Yo creo todos mis releases disponibles en tarballs tanto en formato gzip como bzip2.

Los paquetes binarios deben ser siempre específicos de una distribución. Si puedes crear paquetes binarios para una versión actual de una distribución importante, harás feliz a tus usuarios. Intenta promover relaciones con usuarios o desarrolladores de grandes distribuciones para desarrollar un sistema coherente de creación de paquetes binarios. A menudo es buena idea proveer RPMs de RedHat (.rpm), deb's para Debian (.deb) y fuentes RPM SRPM's si es posible. Recuerda: Aunque los paquetes binarios son buenos, tu prioridad debería ser siempre la de conseguir los fuentes empaquetados y liberados. Tus usuarios o colegas de desarrollo pueden y harán los paquetes binarios para tí.

Sistemas de control de versiones

Un sistema de control de versiones puede hacer que muchos de los problemas de empaquetado (y muchos de otros problemas mencionados en este HOWTO) sean menos problemáticos. Si estás utilizando *NIX, CVS es tu mejor apuesta. Recomiendo de todo corazón el libro de Karl Fogel referente a este tema (y su versión publicada en HTML).

Sea CVS o no, seguramente deberías invertir algo de tiempo aprendiendo sobre un sistema de control de versiones porque provee una forma automatizada de resolver muchos de los problemas descritos en este HOWTO. No conozco ninguna versión libre de sistemas de control de versiones para Windows o Mac OS pero sé que existen clientes CVS para ambas plataformas. Websites como SourceForge hacen un gran trabajo con una interface buena y fácil de utilizar para CVS.

Me hubiese gustado dedicar más espacio para CVS en este HOWTO porque me encanta (¡incluso utilizo CVS para mantener las versiones de este HOWTO!) pero creo que está fuera del alcance de este documento y ya existen sus propios HOWTOs. El más notable es el **CVS Best Practices HOWTO** [CVSBESTPRACTICES] que he incluido en la bibliografía adjunta.

Golosinas[12] prácticas y consejos para la entrega.

Otros consejos útiles son:

- **Comprueba que tu programa puede encontrarse siempre en un único lugar**. A menudo

11 Linux Documentation Project.
12 Del original "tidbit"

esto significa que tienes un único directorio accesible via FTP o la web donde la versión más nueva puede ser reconocida rápidamente. Una técnica efectiva es la de proveer una enlace simbólico llamado "tunombredeproyecto-latest" que esté siempre apuntando al release más reciente o a la versión de desarrollo de tu aplicación software. Ten en cuenta que este lugar recibirá muchas peticiones de descarga de releases por lo que asegúrate que el servidor que elijas tenga el ancho de banda adecuado.

- Comprueba que tienes una dirección de email coherente para los informes de errores. Normalmente es buena idea al hacer esto que no sea tu dirección primaria de email como tunombredeproyecto@host o tunombredeproyecto-bugs@host. De esta manera, si alguna vez decides transferir el mantenimiento o tu dirección de email cambia, solo tienes que cambiar a dónde redirecciona la dirección email. También permitirá a más de una persona tratar con la llegada de correo si tu proyecto llega a ser tan enorme que esperas que sea.

Manteniendo un proyecto: relacionándose con desarrolladores.

Una vez has puesto en marcha tu proyecto, has superado las más importantes trabas[13] en el proceso de desarrollo de tu programa. Tener unos buenos cimientos es esencial, pero el proceso de desarrollo es en sí mismo igualmente importante y provee las mismas oportunidades de fracaso. En las dos secciones siguientes, describiré el funcionamiento de un proyecto tratando el cómo mantener un esfuerzo de desarrollo a través de interacciones con desarrolladores y usuarios.

Liberando tu programa, éste se convierte en software libre. Esta transición es más que solamente una gran base de usuarios. Liberando tu programa como software libre, tu software se convierte en software de la comunidad de software libre. La dirección del desarrollo de tu software se reorganizará, desviará, y estará totalmente decidida por tus usuarios y, en gran parte, por otros desarrolladores de la comunidad.

La gran diferencia entre el desarrollo de software libre y el desarrollo de software propietario es la base de desarrolladores. Como líder de un proyecto de software libre, necesitas atraer y mantener desarrolladores de una manera que simplemente los líderes de proyectos de software propietario no tiene porqué preocuparse. Como la persona que lidera el desarrollo de un proyecto de software libre, tienes que aprovechar el trabajo de los colegas desarrolladores tomando decisiones responsables y por responsabilidad elegir no tomar decisiones. Debes dirigir a los desarrolladores sin ser prepotente o mandón. Necesitas esforzarte por ganar respeto y nunca olvidar darlo.

Delegando trabajo.

Ahora mismo, hipotéticamente me has seguido a través del comienzo de la programación de un componente de software, la creación de un sitio web y un sistema de documentación, hemos seguido adelante y (como se tratará en la Sección llamada Liberando Tu Programa) la hemos liberado al resto del mundo. El tiempo pasa, y si todo va bien, habrá personas interesadas y querrán ayudar. Los parches comienzan a llegar.

Como el padre de cualquier niño que crece, es el momento de estremecerse, sonreír y hacer lo más difícil en la vida de un padre: Es el momento de soltarlo.

Delegar es la manera política de describir este proceso de "soltar". Es el proceso de pasar alguna responsabilidad y poder sobre tu proyecto a otro desarrollador responsable e implicado. Es difícil

13 Del original "hurdles" (vallas).

para cualquiera que haya invertido mucho tiempo y energía en un proyecto pero es esencial para el crecimiento de todo proyecto de software libre. Una persona sola no puede hacer tanto. Un proyecto de software libre no es nada sin la implicación de un grupo de desarrolladores. Un grupo de desarrolladores sólo puede mantenerse con respeto y con liderazgo y delegación responsables.

Cuando tu proyecto progresa, notarás a personas dedicando cantidades significantes de tiempo y esfuerzo para tu proyecto. Serán las personas que enviarán más parches, que crearán más entradas en las listas de correo, y entablando conversación en largos emails de debate. Es tu responsabilidad contactar con estas personas e intentar traspasar hacia ellos algún poder o responsabilidad de tu puesto como mantenedor del proyecto (si quieren). Hay varias maneras fáciles de hacerlo:

Como descargo de responsabilidad, delegar no tiene porqué significar que sea regulado por comité. En muchos casos lo es y se ha probado que funciona. En otros casos ha creado problemas. Managing Projects The Open Source Way defiende que "Los proyectos de Software de Código Abierto funcionan mejor cuando una persona es el claro líder de un equipo y toma las decisiones importantes (cambios de diseño, fechas de liberación, etcétera)". Yo creo que esto es por lo general cierto pero quería instar a los desarrolladores a considerar las ideas de que el líder del proyecto no tiene porqué ser el fundador del proyecto y estos importantes poderes no necesitan recaer todos en una persona sino que el encargado de liberaciones puede ser diferente del jefe de desarrollo. Estas situaciones son políticamente delicadas por lo que sé prudente y asegúrate de que es necesario antes de correr a dar poderes a las personas.

Cómo delegar

Puedes encontrar que otros desarrolladores parecen incluso más experimentados o con más conocimientos que tú. Tu trabajo como mantenedor no significa que debas ser el mejor de los más brillantes. Significa que eres responsable de mostrar buen juicio y de reconocer qué soluciones son mantenibles y cuáles no.

Como todo, es más fácil observar a otros delegar que hacerlo tú mismo. En una frase: está pendiente de otros desarrolladores capacitados que demuestran interés e implicación prolongada en tu proyecto e intenta traspasar responsabilidad hacia ellos. Las siguientes ideas podrían ser buenas para comenzar o ser buenas fuentes de inspiración:

Permite a un gran grupo de personas el acceso de escritura a tu repositorio CVS y haz verdaderos esfuerzos hacia el gobierno por comité.

Apache es un ejemplo de un proyecto que es llevado por un grupo de desarrolladores que votan las cuestiones técnicas importantes y la admisión de nuevos miembros y todos ellos tienen acceso de escritura en el repositorio principal de fuentes. Su proceso está detallado online.

El proyecto Debian es un ejemplo extremo de gobierno por comité. En números actuales, más de 700 desarrolladores tienen responsabilidad total para aspectos del proyecto. Todos estos desarrolladores pueden subir ficheros al servidor FTP principal, y votar las cuestiones más importantes. La dirección del proyecto está determinada por el contrato social del proyecto y unos estatutos. Para facilitar este sistema, hay equipos especiales (por ejemplo el equipo de instalación, el equipo de Lengua Japonesa) así como un comité técnico y un líder de proyecto. La responsabilidad principal del líder es, "nombrar delegados o decidir delegaciones al comité técnico".

Aunque ambos proyectos operan en una escala que no será la de tu proyecto (al menos al principio),

su ejemplo es de ayuda. La idea de Debian sobre un líder de proyecto que no puede hacer más que delegar sirve como una caricatura de cómo un proyecto puede implicar y motivar un número enorme de desarrolladores y crecer a un tamaño enorme.

Nombra públicamente a alguien como el encargado de liberaciones para una liberación específica.

Un encargado de liberaciones normalmente responsable de coordinar pruebas, hacer respetar un congelamiento de código, ser responsable de la estabilidad y el control de calidad, crear el empaquetado del software, y colocarlo en el lugar apropiado para su descarga.

Esta utilización del encargado de liberaciones es una buena manera de darte un respiro y traspasar a otra persona la responsabilidad de aceptación o rechazo de parches. Es una buena manera de definir claramente una parte del trabajo en el proyecto haciéndolo pertenecer a cierta persona y es una gran manera de darte un espacio para respirar.

Delega el control de toda una división[14].

Si tu proyecto decide tener divisiones (como se detalla en la Sección de nombre Divisiones estables y de desarrollo), podría ser buena idea nombre a alguien para que sea el jefe de una división. Si prefieres centrar tu energía en liberaciones de desarrollo y en la implementación de nuevas funcionalidades, pasa el control total sobre liberaciones estables a un desarrollador idóneo.

El autor de Linux, Linus Torvalds, declaró y coronó a Alan Cox como " el hombre de los núcleos[15] estables". Todos los parches para nucleos estables van a Alan y, si Linus tuviese que dejar de trabajar en Linux por alguna razón, Alan Cox sería más que apropiado para adoptar ese rol como digno heredero del mantenimiento de Linux.

Aceptando y rechazando parches.

Este HOWTO ya ha tratado el hecho de que como mantenedor de un proyecto de software libre, una de tus principales y más importantes responsabilidades será aceptar o rechazar parches entregados a ti por otros desarrolladores.

Animando a practicar un buen sistema de parches.

Siendo la persona que dirige o mantiene el proyecto, no eres la persona que vaya a crear muchos parches. No obstante, vale la pena conocer la sección de ESR[16] Good Patching Practice de su Software Release Practices HOWTO [ESRHOWTO][17]. No estoy de acuerdo con la afirmación de ESR que dice que vale la pena tirar la mayoría de parches malos o indocumentados nada más verlos —no ha sido así mi experiencia, sobre todo tratando de arreglos de errores que normalmente no llegan ni como parches. Por supuesto, no significa que me guste tener parches poco elaborados. Si obtienes malos parches -e, si obtienes parches totalmente indocumentados, y sobre todo si son algo más que arreglos triviales de errores, podría valer la pena juzgar el parche por algunos criterios del HOWTO de ESR y entonces enviar el enlace del documento a las personas para que puedan hacerlo "adecuadamente".

14 Del original "branch".
15 Del original "kernel".
16 Eric S. Raymond.
17 Se mantiene el texto original.

Technical judgment.

En Open Source Development with CVS, karl Fogel hace un argumento convincente en el que las cosas más importantes a tener e cuenta a la hora de rechazar o aceptar parches son:

- Un conocimiento sólido del ámbito de tu programa (esta es la "idea" de la que hablo en la sección Escogiendo un proyecto);+
- La capacidad de reconocer, facilitar, dirigir la "evolución" de tu programa de manera que el programa pueda crecer y cambiar e incorporar la funcionalidad no prevista en un principio.
- La necesidad de evitar las digresiones que podrían ampliar demasiado el ámbito del programa y ocasionar que el proyecto muera prematuramente debido a su propio peso y falta de mantenibilidad.

Estos son los criterios que como mantenedor del proyecto debes tener en cuenta cada vez que recibes un parche.

Fogel profundiza y establece "las preguntas para hacerse cuando considera tanto implementar (o aprobar) un cambio son:"

- ¿Beneficiará a un porcentaje significativo de la comunidad de usuarios del programa?.
- Encaja en el dominio del programa o en una extensión intuitiva o natural de ese dominio?.

Las respuestas a esta preguntas nunca son sencillas y es muy posible (e incluso probable) que la persona que envió el parche difiera contigo sobre la respuesta a estas preguntas. No obstante, si consideras que la respuesta a alguna de estas preguntas es "no", es tu responsabilidad rechazar el cambio. Si fallas en esto, el proyecto será difícil de mantener y a la larga fracasará.

Rechazando parches.

Rechazar parches es probablemente el trabajo más difícil y sensible que el mantenedor de todo proyecto de software libre tiene que enfrentarse. Pero a veces tiene que hacerse. Mencioné antes (en la sección Manteniendo un proyecto: interactuando con desarrolladores y en la sección Delegando trabajo) que necesitas tratar de equilibrar tu responsabilidad y poder para hacer lo que tú crees como las mejores decisiones técnicas con la seguridad de que perderás soporte de otros desarrolladores si pareces tener aires de grandeza[18] o excesivamente autoritario o posesivo en la comunidad del proyecto. Recomiendo que tengas en cuenta estos tres importantes conceptos cuando rechaces parches (u otros cambios):

Llévalo a la comunidad.

Una de las mejores maneras de justificar una decisión de rechazo de un parche y conseguir no parecer que actúas con mano de hierro en tu proyecto es no tomar las decisiones solo. Podría tener sentido traspasar los cambios más grandes o las decisiones más difíciles a una lista de correo de desarrollo donde puedan ser discutidas y debatidas. Habrán algunos parches (arreglo de errores, etc.) que se aceptarán definitivamente y otros que pensarás que son tan equivocados[19] que no merecen ninguna discusión. Son aquellos que están en el área gris entre estos dos grupos los que podrían merecer un rápido envío a una lista de correo.

18 Del original: seem like on a power trip.
19 Del original: off base.

Recomiendo encarecidamente este proceso. Como mantenedor del proyecto estás preocupado por tomar la mejor decisión para el proyecto, para los usuarios del proyecto y para los desarrolladores, y para ti mismo como líder responsable del proyecto. Entregar cosas a una lista de correo demostrará tu responsabilidad y liderazgo receptivo al servir a los intereses de la comunidad de software.

Las cuestiones técnicas no son siempre una buena justificación

Especialmente hacia el principio de la vida de tu proyecto, te encontrarás que muchos cambios son difíciles de implementar, crean nuevos errores, o tienen otros problemas técnicos. Intenta ignorarlos. Especialmente con las nuevas funcionalidades, las buenas ideas no provienen siempre de buenos programadores. El valor técnico es una razón válida para posponer la aplicación de un parche pero no siempre es una buena razón para rechazar un cambio de forma rotunda. Incluso con pequeños cambios vale la pena el esfuerzo de trabajar con el desarrollador enviando el parche para eliminar errores e incorporar el cambio si crees que parece una buen añadido a tu proyecto. El esfuerzo por tu parte contribuirá a que tu proyecto sea una comunidad y arrastrará a tu proyecto a un desarrollador nuevo o con menos experiencia e incluso enseñarles algo que podría ayudarles durante la creación de su siguiente parche.

Simple y pura educación.

Quizá no sea necesario decirlo pero, antes de nada y siempre, sé amable. Si alguien tiene una idea y le tiene tanto cariño que escribe código y envía un parche, les importa, están motivados, y ya están involucrados. Tu objetivo como mantenedor es conseguir que vuelvan a enviar. Quizá te hayan enviado una porquería esta vez pero la próxima podría ser la idea o funcionalidad que revolucione tu proyecto.

Es tu responsabilidad justificar tu decisión de no incorporar su cambio concisa y claramente. Agradéceselo. Házles saber que aprecias su ayuda y que te sientes mal por no poder incorporar su cambio. Házles saber que te gustaría que siguiesen involucrados en el proyecto porque aprecias su trabajo y quieres verlo reflejado en tu aplicación. Si alguna vez has tenido un parche rechazado después de haberle dedicado una gran cantidad de tiempo, pensamiento, y energía en él, recordarás qué se siente ya que te sientes mal. Acuérdate de esto cuando tengas que decepcionar a alguien. Nunca es fácil pero tienes que hacer todo lo que puedas para que sea lo menos desagradable posible.

Divisiones estable y en desarrollo.

La idea de divisiones estable y en desarrollo se ha descrito ya de forma breve en la sección denominada Eligiendo un método de numeración de versiones y en la sección denominada Delega el control de toda una división. Estas alusiones atestiguan algunas de las maneras que múltiples divisiones pueden afectar a tu software. Las divisiones pueden ahorrarte (hasta cierto punto) algunos de los problemas sobre rechazar parches (como se describe en la sección denominada Aceptando y rechazando parches) permitiéndote comprometer de forma temporal la estabilidad de tu proyecto sin afectar aquellos usuarios que necesitan esta estabilidad.

La forma más común de dividir tu proyecto es tener una división que es estable y una que está en desarrollo. Este es el modelo seguido por el kernel de Linux descrito en la sección denominada Eligiendo un método de numeración de versiones. En este modelo, siempre hay una división que es estable y siempre hay una en desarrollo. Antes de cualquier nueva liberación, la división en

desarrollo entra en “funcionalidad congelada”[20] como se describe en la sección denominada Congelando[21] donde loa cambios importantes y funcionalidades añadidas se rechazan o se dejan aparcados hasta que el desarrollo del kernel se libera como la nueva división estable y los desarrollos importantes se reanudan en la división en desarrollo. La corrección de errores y los pequeños cambios que sea improbable que tengan gran repercusión negativa se incorporan en la división estable así como en la división en desarrollo.

El modelo de Linux provee un ejemplo extremo. En muchos proyectos, no hay necesidad de tener dos versiones constantemente disponibles. Puede tener sentido tener dos versiones sólo cerca de una liberación. Históricamente el proyecto Debian ha creado distribuciones estable e inestable disponibles pero las ha ampliado para incluir: estable, inestable, pruebas, experimental, y (cerca del momento de liberación) una distribución congelada que sólo incorpora arreglo de errores durante la transición de inestable a estable. Hay algunos proyectos que debido a su tamaño podrían necesitar un sistema como el de Debian pero este uso de divisiones ayuda a demostrar como pueden ser utilizados para equilibrar un desarrollo consistente y efectivo con la necesidad de crear liberaciones periódicas y utilizables.

Intentando tú mismo definir un árbol de desarrollo, hay varios puntos que podrían ser útiles tener en mente:

Minimiza el número de divisiones

> Debian puede estar capacitado para hacer un buen uso de cuatro o cinco divisiones pero contiene gigabytes de software en más de 5000 paquetes compilados para 5-6 arquitecturas diferentes. Para ti, dos es un buen límite. Demasiadas divisiones confundirá a tus usuarios (¡no puedo saber cuántas veces he tenido que explicar el sistema de Debian cuando sólo tiene 2 o a veces 3 divisiones!), desarrolladores potenciales e incluso tú mismo. Las divisiones pueden ayudarte pero tienen un precio así que utilízalas con mucha moderación.

Asegúrate que todas las divisiones están explicadas

> Tal y como he mencionado en el párrafo anterior, tener varias divisiones confundirá a tus usuarios. Haz todo lo que puedas por evitarlo explicando claramente las diferentes divisiones en una página destacada de tu sitio web y en un archivo LEEME en el FTP o directorio web.
>
> También recomendaría no hacer lo que yo creo un error cometido por Debian. Los términos “inestable”, “test”, y “experimental” son vagos y difíciles de clasificar por orden de estabilidad (o inestabilidad). Intenta explicar a alguien que “estable” significa de hecho “ultra estable” y que “inestable” de hecho no significa que incluya software inestable sino que se trata de software estable que todavía no ha sido testeado como distribución.
>
> Si vas a utilizar divisiones, sobre todo al principio, ten en cuenta que las personas son propensas a entender términos como “estable” y “desarrollo” y seguramente no te equivocarás al elegir esta simple y común estrategia de divisiones.

Asegúrate que todas tus divisiones están siempre disponibles

> Como mucho de lo escrito en este documento, probablemente no debería ser necesario decirlo

20 Del original feature freeze
21 Del original Freezing

pero la experiencia me ha enseñado que no siempre es tan obvio para todos. Es buena idea separar físicamente la diferentes divisiones en diferentes directorios o árboles de directorios en tu FTP o sitio web. Linux lleva a cabo esto teniendo los núcleos en directorios v2.2 y v2.3 donde es directamente obvio (una vez conoces el esquema de numeración de versiones) qué directorio estable es el más actualizado y la actual liberación en desarrollo. Debian lo logra poniendo nombre a todas sus distribuciones (por ejemplo woody, potato, etc) y cambiando los enlances simbólicos llamados "estable", "inestable" y "congelado" apuntándolos a la distribución correspondiente (por el nombre) y al nivel correspondiente. Ambos métodos funcionan y también hay otros. En cualquier caso, es importante que las diferentes divisiones estén disponibles, accesibles en localizaciones constantes, y que las diferentes divisiones se distinguen claramente unas de otras para que los usuarios sepan exactamente lo que quieren y dónde conseguirlo.

Otras cuestiones de gestión de proyecto.

Hay más cuestiones alrededor de la interacción con desarrolladores en un proyecto de software libre que no puedo abordar en gran detalle en un HOWTO de este tamaño y alcance. No dudes en contactar conmigo si ves que he pasado por alto algo importantes

Otras pequeñas cuestiones que valen la pena mencionar son:

Congelando

Para esos proyectos que eligen adoptar un modelo separado de desarrollo (la sección llamada Divisiones estables y en desarrollo), congelar es un concepto que seguro será familiar.

La congelación puede ser de dos tipos importantes. Una "congelación de funcionalidad"[22] es un periodo en el que no se ha añadido funcionalidad importante al programa. Es un periodo en el que la funcionalidad establecida (incluso esquemas de funcionalidad apenas funcional) puede ser mejorada y perfeccionada. Es un periodo en el que se arreglan errores. Este tipo de congelación se aplica normalmente durante un tiempo (un mes o dos) antes de una liberación. Es fácil retrasar una liberación porque esperas "una funcionalidad más" y una congelación ayuda a evitar esta situación marcando muy bien el límite[23]. Provee a los programadores de un espacio que necesitan para conseguir un programa listo para su liberación.

El segundo tipo de congelación es una "congelación de código"[24] que se parece más a un componente liberado de software. Una vez que un componente de software pasa estar en "congelación de código", se pone freno a todos los cambios en el código y sólo se permiten cambios que arreglen errores. Este tipo de congelación sigue normalmente a una "congelación de funcionalidad" y justo precede a la liberación. La mayoría de software liberado se podría interpretar como del tipo de alto nivel "código congelado".

Incluso si nunca nombras a alguien como encargado de una liberación (la sección llamada nombra públicamente a alguien como encargado de liberaciones para una liberación específica), tendrás una manera fácil de justificar el rechazo o aplazamiento de parches (la sección llamada Aceptando o rechazando parches) antes de una liberación con la publicación de poner efectivo un estado de congelación.

22 Del original: feature freeze.
23 Del original: by drawing the much needed line in the sand.
24 Del original: code freeze.

Bifurcaciones[25].

No estaba seguro de cómo trataría el tema de las bifurcaciones en este documento (o si lo iba a tratar). Una bifurcación es cuando un grupo de desarrolladores coge código de un proyecto de software libre y empieza un nuevo proyecto de software con él. El ejemplo más famoso de una bifurcación fue entre Emacs y Xemancs. Ambos emacs se basan en un código base idéntico pero debido a razones técnicas, políticas y filosóficas, el desarrollo se dividió en dos proyectos que ahora compiten entre sí.

La versión corta de la sección de bifurcaciones es, no las hagas. Las bifurcaciones fuerzan a los desarrolladores a elegir un proyecto en el que trabajar, causan divisiones políticas desagradables, y redundancia en el trabajo. Afortunadamente, normalmente la amenaza de una bifurcación basta para asustar al mantenedor o mantenedores de un proyecto y hacerles cambiar la forma en la que llevan su proyecto.

En su capítulo de "El proceso de código abierto", Karl Fogel explica cómo hacer una bifurcación si no tienes más remedio. Si has decidido que es absolutamente necesario y que las diferencias entre tú y las personas que amenazan con la bifurcación son absolutamente irresolubles, recomiendo el libro de Fogel como un buen punto de partida.

Manteniendo un proyecto: interactuando con los usuarios.

Si has llegado hasta aquí, felicidades, estás llegando casi al final de este documento. Esta sección final explica algunas de las situaciones en las que tú, al estar capacitado como mantenedor del proyecto, tendrás al interactuar con los usuarios. Esta sección te recomienda cómo deberían llevarse a cabo de manera efectiva.

Interactuar con los usuarios es difícil. En nuestro debate sobre la interacción con los desarrolladores, asumimos que en un proyecto de software libre, un mantenedor del proyecto debe estar constantemente esforzándose por atraer y conservar a desarrolladores que pueden dejar el proyecto fácilmente en cualquier momento.

Los usuarios de la comunidad de software libre son diferentes respecto a los desarrolladores y son también diferentes respecto a los usuarios del mundo del software propietario y por tanto deben ser tratados de forma diferente respecto esos grupos. Algunas puntos importantes en los que estos grupos difieren son:

- La linea que separa los usuarios de los desarrolladores es difusa en el sentido de que es totalmente extraña para todo modelo de desarrollo propietario. Tus usuarios son a menudo tus desarrolladores y viceversa.
- En el mundo del software libre, a menudo tu única opción de usuarios eres tú mismo. Debido a que se enfatiza el no repetir el trabajo de otros en la comunidad de software libre y debido a que no existe el componente de competencia presente en el modelo de software propietario (o al menos en una forma sumamente diferente) en el modelo de desarrollo de software libre, seguramente serás el único proyecto que hace lo que tú haces (o por lo menos el único que hace lo que el tuyo de la manera que tú lo haces). Esto significa que la sensibilidad hacia tus usuarios es incluso más importante que en el mundo del software

25 Del original "Forks".

propietario.

- Paradójicamente, en general los proyectos de software libre tienen menos consecuencias inmediatas o graves al ignorar a sus usuarios. De hecho es algo bastante común. Dado que normalmente no necesitas competir con otro producto, es muy probable que no tengas que batallar para mejorar las funcionalidades del nuevo programa de tu competidor. Esto significa que tu proceso de desarrollo deberá dirigirse de forma interna, con responsabilidad a tus usuarios, o las dos cosas juntas.

Intentar tratar esta caso único puede hacerse sólo de forma indirecta. Los desarrolladores y mantenedores necesitan escuchar a los usuarios e intentar ser lo más responsable que puedas. Un conocimiento sólido de la situación contada antes es la mejor herramienta para cualquier desarrollador de software libre para poder cambiar su manera de desarrollar o de liderar y ajustarla al singular proceso de gestión de un proyecto de software libre. Estos capítulos intentarán presentar algunos de los puntos más difíciles o importantes sobre la interacción de proyectos con los usuarios y proveer algunas pistas de cómo tratarlos.

Pruebas y probadores.

Además de que tus usuarios sean tus desarrolladores, también son (quizás más que otra cosa) tus probadores. Antes de que me quemen, voy a expresar de otra manera mi frase: algunos de tus usuarios (los explícitamente voluntarios) son tus probadores.

Es importante que esta distinción se haga lo antes posible ya que no todos tus usuarios quieren ser probadores. Muchos usuarios quieren utilizar software estable y no les importa si no tienen el más nuevo, maravilloso software con la última, maravillosa funcionalidad. Estos usuarios asumirán un componente de software estable y probado, aún con errores obvios o importantes, pero se enfadarán si ellos mismos los encuentran probando. Este es otro ejemplo en el que un modelo de desarrollo dividido (como se mencionó en la sección de nombre Divisiones Estable y En desarrollo) puede ser útil.

“Gestionando proyectos de la forma de código abierto” describe cómo debería ser una buena prueba:

Condiciones límite.

> Tamaño máximo de buffers, conversión de datos, límites superior e inferior, etcétera.

Comportamiento inadecuado.

> Es buena idea averiguar lo que el programa hará si un usuario le proporciona un valor inesperado, toca el botón erróneo, etc. Pregúntate un buen número[26] de preguntas tipo “Qué pasa si” y piensa en cualquier cosa que podría fallar o podría ir mal y averigua que es lo que hará tu programa en estos casos.

Fallos elegantes.

> Normalmente, la respuesta a un número de “Qué pasa si” es un “fallo”, el cual es a menudo la única respuesta. Ahora asegúrate de que ocurre de manera correcta. Asegúrate que cuando peta[27], hay alguna indicación de porqué lo ha hecho para que el usuario o desarrollador sepan

26 Del original: “bunch”.

27 Traducción coloquial. Del original “crashes”.

qué está pasando.

Cumplimiento de los estándares.

Si es posible, asegúrate de que tus programas cumplen los estándares. Si es interactivo, no seas demasiado creativo con la interfaz. Si no es interactivo, asegúrate que comunica con otros programas y con el resto del sistema utilizando los canales adecuados y establecidos.

Pruebas automatizadas.

Para muchos programas, muchas equivocaciones típicas pueden ser detectadas de forma automática. Las pruebas automatizadas pueden llegar a ser bastante buenas detectando errores con los que te has topado varias veces antes o las cosas que simplemente olvidas. No son muy buenas encontrando errores, incluso los más importantes, que son del todo imprevistos.

CVS tiene un Bourne shell script llamado sanity.sh que vale la pena considerar. Debian utiliza un programa llamado lintian que chequea los paquetes Debian en busca de los errores más comunes. Aunque el uso de estos scripts pueda no ser de ayuda, hay un montón de software de chequeo de errores[28] en la red que puede ser aplicable (no dudes en enviarme un mail con cualquier recomendación). Ninguno de ellos creará una liberación libre de errores pero al menos evitará algunos descuidos importantes. Para acabar, si tu programa llega a ser un gran periodo de esfuerzo, te darás cuenta que cometes algunos errores una y otra vez. Empieza una colección de scripts que chequeen estos errores para ayudar a mantenerlos fuera de futuras liberaciones.

Probando mediante probadores.

Para cualquier programa que dependa de la interactividad con el usuario, muchos errores sólo serán descubiertos a través de pruebas de usuario presionando teclas y botones del ratón. Para ello necesitas tantos probadores como puedas.

La parte más difícil de las pruebas es buscar probadores. Normalmente es una buena táctica poner un mensaje en una lista de correo de importancia o en un grupo de noticias anunciando una propuesta específica de fecha de liberación y esquematizando la funcionalidad de tu programa. Si dedicas algo de tiempo a tu anuncio, seguro que tendrás algunas respuestas.

La segunda parte más difícil de las pruebas es mantener tus probadores y mantenerlos involucrados de manera activa en el proceso de pruebas. Afortunadamente, hay algunas tácticas probadas que pueden aplicarse hacia este objetivo:

Haz las cosas fáciles para tus probadores

Tus probadores te están haciendo un favor de manera que haz que sea lo más fácil para ellos. Esto significa que deberías tener cuidado en empaquetar tu software de manera que sea fácil de encontrar, desempaquetar, instalar, e desinstalar. Esto también significa que deberías explicar lo que esperas de cada probador y crear los medios para hacer los informes de errores simples y bien definidos. La clave es proveer tanta estructura como sea posible para hacer fácil el trabajo de tus probadores y para mantener tanta flexibilidad como sea posible para aquellos que quieras hacer cosas de una manera un poco diferente.

28 Del original: Sanity checking software

Demuestra interés hacia tus probadores.

> Cuando tus probadores envían errores, respóndeles y rápido. Incluso si eres el único que les responde que el error ha sido arreglado, respuestas rápidas y constantes les hace sentir que su trabajo es escuchado, importante, y agradecido.

Sé agradecido con tus usuarios.

> Agradéceles personalmente cada vez que te envían un parche. Agradéceles públicamente en la documentación y en la sección "about" de tu programa. Estás agradecido a tus probadores ya que tu programa no sería posible sin su ayuda. Asegúrate de que lo saben. Dáles públicamente un palmada en la espalda para asegurarte también que el resto del mundo lo sabe. Lo agradecerán más de lo que piensas.

Creando la infraestructura de soporte.

Aunque las pruebas son importantes, la mayor parte de tus interacciones y responsabilidad hacia tus usuarios está en la categoría de soporte. La mejor manera de estar seguro de que tus usuarios tienen un soporte adecuado al utilizar tu programa es crear una buena infraestructura para este propósito de forma que tus desarrolladores y usuarios se ayuden mútuamente y que tú tengas menor carga de trabajo. De esta forma, las personas tendrán las respuestas a sus preguntas antes y mejor. Esta infraestructura se presenta de varias maneras importantes:

Documentación

No debería sorprender que el elemento clave de toda infraestructura de soporte es una buena documentación. Este tema se abordó ampliamente en la Sección llamada Documentación y no se será repetido en esta.

Listas de correo.

Además de la documentación, las listas de correo efectivas serán tu mejor herramienta para proveer soporte a los usuarios. Llevando bien una lista de correo es más complicado que instalar un software de lista de correo en una máquina.

Listas separadas.

Una buena idea también es separar tu listas de correo de usuarios y desarrolladores (quizás en proyecto-usuario@host y proyecto-desarrollo@host) y hacer cumplir la división. Si las personas envían un mensaje con una pregunta de desarrollo en -usuario, solicítales cortésmente que lo reenvíen a -desarrollo y viceversa. Subscíbete en ambos grupos y anima a todos los desarrolladores principales a que hagan lo mismo.

Este sistema consigue que ninguna persona esté atascada haciendo todo el trabajo de soporte y funciona de forma que si los usuarios aprenden más sobre el programa, pueden ayudar a los nuevos usuarios con sus preguntas.

Elige bien tu software de lista de correo

Te recomiendo que no elijas tu software de lista de correo de forma impulsiva. Ten en cuenta la fácil accesibilidad a usuarios con poca experiencia técnica ya que te interesa que sea lo más fácil posible. Es también importante el acceso vía Web a un archivo de la lista.

Los dos programas libres más grandes de lista de correo son majordomo y GNU Mailman. Después de mucho tiempo siendo defensor de majordomo, quisiera ahora recomendar la elección de GNU Mailman a todo proyecto. Cumple los criterios descritos antes y lo hace de forma fácil. Provee un buen programa de lista de correo para el mantenedor de un proyecto de software libre a diferencia de una buena aplicación de lista de correo para un administrador de listas de correo.

Hay otras cosas a tener en cuenta al crear tu lista. Si es posible abrir[29] tu lista de correo a Usenet y proveerla en formato de boletín además de acerla accesible en la web, harás un favor a algunos usuarios y conseguirás que la infraestructura de soporte sea un poco más accesible.

Otras ideas de soporte.

Una lista de correo y la documentación accesible están lejos de ser lo único que puedes hacer para crear una buena infraestructura de soporte al usuario. Sé creativo. Si das con algo que funciona, envíame un email y lo incluiré aquí.

Sé accesible.

No puedes presentar pocas maneras de ponerse en contacto contigo. Si te conectas a un canal IRC, no dudes en ponerlo en la documentación de tu proyecto. Lista tus direcciones de email y correo postal[30], y la manera de ponerse en contacto contigo mediante ICQ, AIM, o Jabber si corresponde.

Software de gestión de errores.

Para muchos proyectos de gran tamaño, es esencial el uso de software de gestión de errores para llevar el control de qué errores se han solucionado, qué errores no, y qué errores se han solucionado por qué personas. Debian utiliza Debian Bug Tracking System (BTS) aunque puede no ser la mejor opción para todos los proyectos (actualmente parece caer por su propio peso). Además de ser un condenado buen navegador web, el proyecto Mozilla ha generado un subproyecto cuyo resultado es un sistema de control de errores llamado bugzilla que es muy capaz y me encanta.

Estos sistemas (y otros como ellos) pueden ser difíciles de manejar así que los desarrolladores deberían tener cuidado de no dedicar más tiempo al sistema control de errores que a los mismos errores o el proyecto en sí. Si un proyecto sigue creciendo, el uso de un sistema de control de errores puede proveer una posibilidad fácil y estándar para que los usuarios y probadores informen sobre errores y para que los desarrolladores y mantenedores los solucionen de forma ordenada.

Liberando tu programa.

Como se ha mencionado antes en el HOWTO, la primera regla a la hora de liberar es, libera algo útil. Software que no funciona o no es útil no atraerá a nadie a tu proyecto. Las personas se “desconectarán” de tu proyecto y lo más probable es que no le den importancia y esperen el anuncio

29 Del original “to gate”
30 Del original “snailmail”

de una nueva versión. Un software medio funcionando, si es útil, intrigará a la gente, abrirá su apetito para las versiones siguientes, y los animará a unirse al proceso de desarrollo.

Cuándo liberar.

Tomar la decisión de liberar tu software por primera vez es una decisión extremadamente importante y estresante. Pero es necesario hacerla. Mi consejo es intentar hacer algo que sea lo suficientemente completo como para utilizarse lo suficientemente incompleto como para permitir flexibilidad y dar un espacio de imaginación a tus futuros desarrolladores. No es una decisión fácil. Pide ayuda a una lista de correo de un grupo local de usuarios de Linux o a un grupo de desarrolladores amigos.

Una táctica es hacer primero una liberación “alpha” o “beta” como se describe más adelante en la sección de nombre Alpha, beta, y liberaciones de desarrollo. No obstante, la mayoría de las pautas descritas antes siguen siendo aplicables.

Cuando tu instinto te diga que es el momento y sientes que has considerado la situación varias veces, cruza los dedos y lánzate.

Después de haber liberado por primera vez, saber cuándo liberar no es tan estresante, sólo difícil de calcular. Prefiero el criterio propuesto por Robert Krawitz en su artículo, “Gestión de proyectos de software libre para mantener un buen ciclo de liberaciones. Recomienda que te preguntes si esta liberación...

- Contiene suficiente nueva funcionalidad o soluciona suficientes errores como para valer la pena el esfuerzo.
- Está suficientemente espaciada en el tiempo respecto a la anterior como para que el usuario tenga suficiente tiempo para trabajar con la última liberación.
- Es suficientemente funcional como para que el usuario pueda hacer su trabajo (calidad).

Si la respuesta a todas estas cuestiones es si, es probable que sea el momento de liberar. Si dudas, recuerda que el pedir consejo no hace daño.

Cómo liberar.

Si ha seguido las pautas descritas hasta ahora en este HOWTO, la mecánica para hacer una liberación serán la parte más fácil de las liberaciones. Si has creado lugares constantes de distribución y otras infraestructuras descritas en secciones precedentes, liberar debería ser tan simple como crear el paquete, chequearlo al menos una vez, y subirlo al lugar adecuado para entonces que tu website refleje el cambio.

Liberaciones alpha, beta y desarrollo.

Al pensar en liberaciones, vale la pena tener en cuenta el hecho de que no todas las liberaciones necesitan ser una liberación con un número de versión definitiva. Los usuarios de software están acostumbrados a las pre-liberaciones pero debes tener cuidado con etiquetar estas liberaciones correctamente o causarán más problemas de los necesarios[31].

31 Del original: “they are worth”.

Un comentario que se hace bastante es que muchos desarrolladores de software libre parecen confusos en lo que respecta al ciclo de liberaciones. “Gestionando proyectos al estilo de código abierto” sugiere que memorices la siguiente frase, “Alpha no es Bet. Beta no es una liberación” y reconozco que seguramente esta es una buena idea.

Liberaciones alpha.

> Un software alpha está completa respecto a los requerimientos pero a veces está parcialmente funcional.
>
> De una liberación alpha se espera que sea inestable, quizás un poco insegura, pero utilizable. Pueden tener errores conocidos y problemas por resolver. Antes de liberar una alpha, acuérdate de que una alpha sigue siendo una liberación y que la gente no va a esperar una versión diaria del CVS. Una alpha debería funcionar y tener hechos un mínimo test y el arreglo de errores.

Liberaciones beta.

> El software beta es funcional y completo respecto a sus características, pero se encuentra en el período de pruebas y sigue teniendo errores por resolver.
>
> De las liberaciones beta se espera por lo general que sean utilizables y ligeramente inestables, aunque definitivamente no inseguras. Las beta impiden normalmente la salida de una liberación completa durante menos de un mes. Pueden contener pequeños y conocidos errores pero no importantes. La funcionalidad importante tiene que estar implementada totalmente aunque su mecánica necesite resolverse. Las beta son una gran utilidad para abrir el apetito[32] de usuarios potenciales dándoles una visión muy realista de lo que será tu proyecto dentro de poco y puede ayudar a mantener el interés dándoles algo.

Liberaciones en desarrollo.

> “Una liberación en desarrollo” es un término más vago que “alpha” o “beta”. Normalmente elijo reservar el término para el debate de una división de desarrollo aunque hay otras maneras de utilizar el término. De hecho tantas, que me parece que el término ha perdido valor. El popular gestor de ventanas Enlightenment sólo ha liberado liberaciones en desarrollo. Por lo general, el término se utiliza para describir liberaciones que no son ni alpha ni beta y cuando se libera una versión pre-alpha de un componente de software con la intención de mantener un interés vivo en mi proyecto, es probablemente cómo yo mismo lo etiquetaría.

Dando a conocer tu proyecto.

Bueno, ya lo has hecho. Has (al menos para el propósito de este HOWTO) diseñado, creado, y liberado tu proyecto de software libre. Lo que te queda por hacer es comunicar al mundo para que lo conozca, lo prueben y con suerte subirse al tren del desarrollo. Si todo está como lo descrito antes, será un proceso sencillo y rápido. Un rápido comunicado es todo lo necesario para ponerte en el radar de la comunidad de software libre.

Listas de correo y Usenet.

---

32 Del original: “whet the appetites”

Da a conocer tu software en el Usenet comp.s.linu.announce. Si solo lo das a conocer en dos lugares, que sean c.o.l.a y freshmeat.

No obstante, el email sigue siendo la forma que la mayoría de gente en Internet consigue la información. Es una buena idea enviar un mensaje dando a conocer tu programa a toda lista de correo de importancia que conozcas y también todo grupo de debate de Usenet de importancia.

Karl Fogel recomienda utilizar un asunto sencillo describiendo que el motivo del mensaje es un anuncio, el nombre del programa, la versión, y media linea con una descripción de su funcionalidad. De esta forma, tu anuncio atraerá de forma inmediata a todo usuario o desarrollador interesado. El ejemplo de Fogel tiene el siguiente aspecto:

Subject: ANN: aub 1.0, a program to assemble Usenet binaries

El resto del email debes describir la funcionalidad de los programas concisa y rápidamente en no más de dos parágrafos y debe proveer enlaces a las páginas web del proyecto y enlaces directos a las descargas para aquellos que quieran probarlo enseguida. Esta estructura servirá tanto para Usenet como para las listas de correo.

Debes repetir este anuncio de forma consecuente en los mismos lugares para las siguientes liberaciones.

f**reshmeat.net**

Mencionado antes en la sección Buscando proyectos similares, hoy día en la comunidad de software libre, los anuncios de tu proyecto en freshmeat son caso más importantes que los anuncios de las listad de correo.

Visita la web de freshmeat o su página de proyecto para incluir el anuncio de tu proyecto en su site y su base de datos. Además de un gran website, freshmeat provee un boletín diario que destaca las liberaciones de todos los días y llega a una gran audiencia (personalmente lo leo por encima cada noche en busca de nuevas liberaciones interesantes).

Lista de correo del proyecto.

Si has seguido adelante y has creado listas de correo para tu proyecto, siempre debes anunciar las nuevas versiones en estas listas. He visto que muchos proyectos, los usuarios piden una única lista de correo pequeña para ser notificados cuando nuevas versiones son anunciadas. freshmeat.net permite a los usuarios subscribirse a un proyecto en particular para que puedan recibir emails a su sistema cada vez que una nueva versión se anuncia. Según mi opinión, no hace daño.