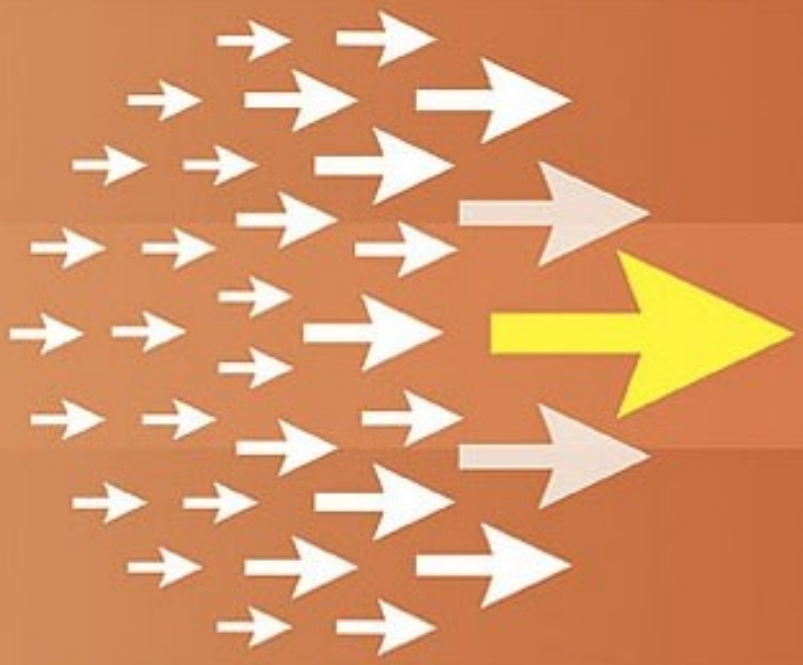
Karl Fogel

producir software de código abierto



COMO LLEVAR A BUEN PUERTO UN PROYECTO DE CÓDIGO LIBRE

O'REILLY®

Producir Software de Código Abierto

Como Llevar a Buen Puerto un Proyecto de Código Libre

Karl Fogel
Rafael Martilotti
Alejandro Ayuso
José Manuel Puerta Peña
Pedro Andrés Bonilla Polo
Aldo Vadillo Batista
Francisco Urbano García
Christian López Espínola
Emilio Casbas Jimenez

Producir Software de Código Abierto: Como Llevar a Buen Puerto un Proyecto de Código Libre

por Karl Fogel, Rafael Martilotti, Alejandro Ayuso, José Manuel Puerta Peña, Pedro Andrés Bonilla Polo, Aldo Vadillo Batista, Francisco Urbano García, Christian López Espínola, y Emilio Casbas Jimenez

Dedicatoria

Este libro está dedicado a dos queridos amigos sin los cuales esta obra no hubiera sido posible: Karen Underhill y Jim Blandy.

Tabla de contenidos

Prefacio

¿Por qué escribir éste libro?

Ahora cuando estoy en una fiesta, la gente no se queda con la mirada en blanco cuando les digo que escribo software libre. "Oh sí, código abierto ¿cómo Linux?" me dicen mientras asiento con aprovación. "¡Sí, exactamente! Eso es lo que hago." Es agradable no estar más completamente aislado. Antes, la siguiente pregunta era predecible: "¿Cómo ganas dinero haciendo eso?" Para responder, resumiría así la economía del código abierto: que existen organizaciones interesadas en que cierta clase de software exista, pero que no necesitan vender copias, sólo quieren asegurarse de que esté disponible y mantenido, como una herramienta en lugar de como un servicio.

En cambio, últimamente, la siguiente pregunta no siempre ha tenido que ver con el dinero. El *Business Case* del software Open Source [1] ya no es tan misterioso, y muchos no-programadores ya entienden—o al menos no se sorprenden—que haya gente empleada en ello a tiempo completo. En su lugar, la pregunta que voy escuchando cada vez más es "*¿Cómo funciona todo esto?*"

No tenía una respuesta satisfactoria lista y cuan más duro intentaba pensar en una, más me daba cuenta de cuan complejo realmente es el tema. Llevar un proyecto de software libre no es exactamente como dirigir un negocio (imaginemos tener que negociar constantemente la naturaleza de nuestro producto con un grupo de voluntarios, muchos de los cuales ¡ni siquiera conocemos!). Tampoco es, por varias razones, como llevar una organización sin ánimo de lucro tradicional o un gobierno. Es similar a todas ellas, pero poco a poco he llegado a la conclusión de que el software libre es *sui generis*. Existen muchas cosas con las que puede ser comparado por su utilidad, pero con ninguna puede ser igualado. En realidad, asumir que el software libre puede ser dirigido es iluso. Un proyecto de software libre puede ser *iniciado* y puede ser influenciado, fuertemente, por algunas partes interesadas. Pero sus activos no pueden ser hechos propiedad de un sólo dueño, así que mientras haya gente en alguna parte—cualquier parte— interesada en continuar con el proyecto, no puede ser cancelado unilateralmente. Todos tienen poder infinito; nadie tiene poder. Una dinámica muy interesante.

Es por esto que quería escribir éste libro. Los proyectos de software libre han permitido a una nueva cultúra evolucionar, un *ethos* en el cual la libertad de hacer que el software haga cualquier cosa que deseamos sea el eje central, sin embargo, el resultado de ésta libertad no es la dispersión de los individuos, cada uno trabajando por su cuenta en el código, sino la colaboración entusiasta. De hecho, ser un cooperador competente es en si, una de las cualidades mas valoradas en el software libre. Dirigir uno de estos proyectos es abordar un tipo de cooperación hipertrofiada, donde la habilidad de, no sólo trabajar con otros, pero de ingeniar nuevas maneras de trabajar en conjunto, pueden producir beneficios tangibles para el desarrollo. Este libro intenta describir las técnicas con las que esto se puede lograr. No es de ninguna manera completo, pero al menos es un inicio.

El buen software libre es ya en si mismo un objetivo y espero que aquellos lectores que vengan buscando como lograrlo esten satisfechos con lo que van a encontrar aquí. Pero más allá de esto, espero transmitir algo del doloroso placer de trabajar con un equipo motivado de desarrolladores de código abierto y de la interacción con los usuarios en la maravillosa manera directa que el Open Source anima. Participar en un proyecto de software libre exitoso es *divertido* y en última instancia es esto lo que mantiene a todo el sistema funcionando.

¿Quíen debería leer éste libro?

[1] Los términos "código abierto" y "libre" son sinónimos esenciales en éste contexto; son discutidos en mayor profundidad en el “#Libre# vs #Abierto#”, en Capítulo 1, *Introducción*.

Este libro está enfocado a desarrolladores y directores quienes esten considerando iniciar un proyecto de software libre o que ya hayan iniciado uno y esten planeado qué hacer ahora. Tambien debería ser útil para aquellas personas que quieren participar en un proyecto Open Source y que nunca lo han hecho.

El lector no necesita ser un programador, pero debe conocer conceptos básicos de ingéniera informática como código fuente, compiladores y parches.

Experiencia anterior con software Open Source como usuario o desarrollador no es necesaria. Quienes hayan trabajado en proyectos de software libre con anterioridad probablemente encuentren algunas partes del libro algo obvias y quizas deseen saltar esas secciones. Dado que potencialmente existe una amplia audiencia experimientada, he hecho un esfuerzo para etiquetar claramente cada sección y decir cuando algo puede ser omitido por quienes ya estan familiarizados en la materia.

Fuentes

Mucha de la materia prima para éste libro viene de trabajar durante cinco años con el proyecto Subversion (http://subversion.tigris.org/). Subversion es un sistema de código abierto para el control de versiones, escrito desde cero con la intención de reemplazar a CVS como el sistema de control de versiones *de facto* utilizado por la comunidad Open Source. El proyecto fue iniciado por la empresa en la que trabajo, CollabNet (http://www.collab.net/), a principios del año 2000 y gracias a Dios, CollabNet entendio desde el inicio a llevarlo como un esfuerzo colaborativo y distribuido. Desde el principio tuvimos muchos desarrolladores voluntarios; hoy somos unos 50 en el proyecto, de los cuales sólo unos pocos son empleados de CollabNet.

Subversion es de muchas maneras un clásico ejemplo de un proyecto Open Source y termine aproximandome más de lo que originalmente esperaba. Parte de esto fue una cuestión de conveniencia: cada vez que necesitaba un ejemplo de un fenómeno en partícular, usualmente podía recordar alguno sobre Subversion. Pero tambien fue una cuestión de verificación. Aunque estoy inmerso en otros proyecto de software libre a diversos niveles, y que converso con amigos y conocidos envueltos en muchos otros, rapidamente me he dado cuenta que al escribir para la imprenta, todas las afirmaciones deben ser verificadas con hechos. No deseaba hacer declaraciones acerca de situaciones presentes en otros proyectos basandome sólo en lo que podía leer en las listas de correo. Si alguien intentase algo así con Subversion sé que sólo estaria en lo correcto la mitad de las veces y equivocado la otra mitad. Así que al buscar inspiración o ejemplos en proyectos con los que no tenía experiencia directa, intentaba primero hablar con algún informador, alguien en quien confiará para explicarme qué estaba sucediendo realmente

Subversion ha sido mi trabajo durante los ultimos cinco años pero he estado involucrado en el software libre durante otros doce. Otros proyectos que han influenciado éste libro son:

- El proyecto de la Free Software Foundation GNU Emacs, un editor de texto del cual mantengo algunos páquetes pequeños
- Sistema de versiones concurrentes, del íngles Concurrent Version System (CVS) en el que trabajé intensamente en 1994#1995 con Jim Blandy y en el que sigo trabajando intermitentemente desde entonces.
- La colección de proyectos Open Source conocidos como la Fundación de Software Apache, especialmente en el Apache Portable Runtime (APR) y en el servidor Apache HTTP.
- OpenOffice.org, las bases de datos Berkeley de Sleepycat y MySQL; No he estado envuelto personalmente en estos proyectos, pero los he observado y, en algunos casos, hablado con personas en ellos.
- GNU Debugger (GDB) (igual que con los anteriores).
- Proyecto Debian (igual que con los anteriores).

Ésta no es la lista completa, por supuesto. Como muchos de los programadores de Open Source, mantengo varios frentes abiertos en diferentes proyectos, sólo para tener una visión del estado general. No los voy a nombrar a todos aquí, pero seran mencionados a lo largo del libro cuando sea apropiado.

Reconocimientos

Éste libro me tomó cuatro veces el tiempo que esperaba para escribirlo, y durante mucho tiempo sentia como si un piano estuviese suspendido sobre mi cabeza a cada lugar al que iba. Sin la ayuda de mucha gente, no habría podido completarlo y seguir cuerdo.

Andy Oram, mi editor en O'Reilly fue el sueño de todo escritor. Aparte de conocer el tema intimamente (él sugirió muchos de los temas), tiene el raro don de saber lo que se intenta decir y ayudar a encontrar la manera correcta de decirlo. Ha sido un honor trabajar con él. Gracias tambien a Chuck Toporek por pasarle ésta propuesta a Andy Right desde el principio.

Brian Fritzpatrick revisó casí todo el material mientras lo escribía, lo que no sólo hizo el libro mejor, pero me mantuvo escribiendo cuando quería estar en cualquier lugar menos frente a un ordenador. Ben Collins-Sussman y Hike Pilato tambien revisaban de vez en cuando el progreso y siempre se contentaban con discutir—algunas veces en profundidad—cualquier tema que intentaba cubrir esa semana. Tambien se daban cuenta cuando reducia la marcha y gentilmente me regañaban cuando era necesario. Gracias tios.

Biela Coleman estaba escribiendo su tésis al mismo tiempo que yo escribía éste libro. Ella sabe lo que significa sentarse cada día a escribir, dandome un ejemplo de inspiración como un oido amigo. Tambien tiene una fascinante vista antropologica del movimiento free software, dandome ídeas y referencias que podría utilizar en el libro. Alex Golub—otro antropólogo con un pie en el mundo del software libre—fue un apoyo excepcional que me ayudo inmensamente.

Micah Anderson de alguna manera nunca parecio oprimido por su propio trabajo de escritor, el cual me inspiraba en alguna forma enfermiza y envidiable, pero siempre estuvo listo con su amistad, conversación y (al menos en una ocasión) soporte técnico. ¡Gracias Micah!

Jon Trowbridge and Sander Striker gave both encouragement and concrete help—their broad experience in free software provided material I couldn't have gotten any other way.

Thanks to Greg Stein not only for friendship and well-timed encouragement, but for showing the Subversion project how important regular code review is in building a programming community. Thanks also to Brian Behlendorf, who tactfully drummed into our heads the importance of having discussions publicly; I hope that principle is reflected throughout this book.

Thanks to Benjamin "Mako" Hill and Seth Schoen, for various conversations about free software and its politics; to Zack Urlocker and Louis Suarez-Potts for taking time out of their busy schedules to be interviewed; to Shane on the Slashcode list for allowing his post to be quoted; and to Haggen So for his enormously helpful comparison of canned hosting sites.

Thanks to Alla Dekhtyar, Polina, and Sonya for their unflagging and patient encouragement. I'm very glad that I will no longer have to end (or rather, try unsuccessfully to end) our evenings early to go home and work on "The Book."

Thanks to Jack Repenning for friendship, conversation, and a stubborn refusal to ever accept an easy wrong analysis when a harder right one is available. I hope that some of his long experience with both software development and the software industry rubbed off on this book.

CollabNet was exceptionally generous in allowing me a flexible schedule to write, and didn't complain when it went on far longer than originally planned. I don't know all the intricacies of how management arrives at such decisions, but I suspect Sandhya Klute, and later Mahesh Murthy, had something to do with it—my thanks to them both.

The entire Subversion development team has been an inspiration for the past five years, and much of what is in this book I learned from working with them. I won't thank them all by name here, because there are too many, but I implore any reader who runs into a Subversion committer to immediately buy that committer the drink of his choice—I certainly plan to.

Many times I ranted to Rachel Scollon about the state of the book; she was always willing to listen, and somehow managed to make the problems seem smaller than before we talked. That helped a lot—thanks.

Thanks (again) to Noel Taylor, who must surely have wondered why I wanted to write another book given how much I complained the last time, but whose friendship and leadership of Golosá helped keep music and good fellowship in my life even in the busiest times. Thanks also to Matthew Dean and Dorothea Samtleben, friends and long-suffering musical partners, who were very understanding as my excuses for not practicing piled up. Megan Jennings was constantly supportive, and genuinely interested in the topic even though it was unfamiliar to her—a great tonic for an insecure writer. Thanks, pal!

I had four knowledgeable and diligent reviewers for this book: Yoav Shapira, Andrew Stellman, Davanum Srinivas, and Ben Hyde. If I had been able to incorporate all of their excellent suggestions, this would be a better book. As it was, time constraints forced me to pick and choose, but the improvements were still significant. Any errors that remain are entirely my own.

My parents, Frances and Henry, were wonderfully supportive as always, and as this book is less technical than the previous one, I hope they'll find it somewhat more readable.

Finally, I would like to thank the dedicatees, Karen Underhill and Jim Blandy. Karen's friendship and understanding have meant everything to me, not only during the writing of this book but for the last seven years. I simply would not have finished without her help. Likewise for Jim, a true friend and a hacker's hacker, who first taught me about free software, much as a bird might teach an airplane about flying.

Disclaimer

Los pensamientos y opiniones expresadas en este libro son propias. No representan los puntos de vista de CollabNet o del proyecto Subversion.

Capítulo 1. Introducción

La mayoría de los proyectos de software libre fracasan.

Tratamos de no prestar mucha atención a los fracasos. Solamente los proyectos exitosos llaman la atención, y hay tantos proyectos de software[1] que aún cuando solo un pequeño porcentaje tiene éxito, el resultado es de una apreciable cantidad de proyectos. Pero tampoco prestamos atención a los fracasos porque no los contamos como un evento. No existe un momento puntual en el que un proyecto deja de ser viable; simplemente se los deja de lado y se deja de trabajar en ellos. Puede haber un momento en que se hace un cambio final al proyecto, pero quienquiera que lo haga, normalmente no sabe en ese momento que ese cambio fue el último. Tampoco hay una definición clara del momento en que un proyecto se acaba. ¿Podrá ser cuando se haya dejado de trabajar en él por seis meses? ¿O cuando su base de usuarios deja de crecer, sin antes haber excedido la base de programadores? ¿Y qué pasaría si los programadores de un proyecto lo abandonan porque se dan cuenta que estaban duplicando el trabajo de algún otro— y si se unen todos en el otro proyecto, y lo amplían para incluir ahí su esfuerzo realizado? ¿Acaso el primer proyecto finalizó, o simplemente cambió de lugar de residencia?

Dada ésta complejidad, es imposible obtener un número preciso para un promedio de fracasos. Pero la evidencia de lo que ha ocurrido en más de un decenio con proyectos con fuente abierta y curioseando un poco en SourceForge.net y otro poco en Google, se llega siempre a la misma conclusión: el porcentaje es muy alto, probablemente algo así como el 90–95%. Este numero crece aún más si se incluyen los proyectos que sobreviven pero son disfuncionales: aquellos que *producen* un código que funciona, pero no son placenteros ni amigables, o no progresan tan rápidamente ni son tan confiables como tendrían que ser.

En este libro se habla de cómo evitar los fracasos. Se examina no solamente cómo se hacen bien las cosas, sino también cómo se hacen mal, para que se puedan reconocer desde el comienzo, y se corrijan los problemas. Tengo la esperanza que después de que se lea este libro, se adquiera un repertorio de técnicas no sólo para evitar los errores comunes en el desarrollo de programas de fuente abierta, sino también para manejar el crecimiento y el mantenimiento de un proyecto exitoso. El éxito no es un juego para que haya un solo ganador, y este libro no busca producir un solo ganador que salga airoso de una competición. Así pues, una parte importante de impulsar un proyecto de fuente abierta es trabajar en armonía con otros proyectos relacionados entre si. Y a la larga, cada proyecto exitoso contribuye al bienestar de todo el mundo del software libre.

Sería muy tentador afirmar que los proyectos de software libre fracasan por las mismas razones que los proyectos de software propietario. Ciertamente el software libre no tiene el monopolio de los requisitos descabellados, las especificaciones vagas, del manejo pobre de los recursos, fases de diseño insuficientes, y tantas otras complicaciones ya conocidas en la industria del software. Se va a hablar mucho de estos asuntos en este libro, y ahora hay que tratar de no multiplicar las referencias a dichos asuntos. Más bien se intentará describir los problemas particulares al software libre. Cuando un proyecto de software libre se estanca, a menudo es porque los programadores (o la dirección) no caen en cuenta de los problemas típicos del desarrollo de software de fuente abierta, aunque pareciera que estan muy bien preparados para las dificultades más conocidas del desarrollo de software de fuente cerrada.

Uno de los errores más comunes es tener expectativas desproporcionadas sobre los beneficios propios de la fuente abierta. Una licencia abierta no es una garantía de tener una legión de programadores activos que de repente se ofrecen para el proyecto, ni tampoco un proyecto con problemas se cura por el sólo hecho de pasarlo a fuente abierta. De hecho es todo lo contrario: abrir un proyecto puede agregar una serie de complicaciones, y resultar a corto plazo *más* costoso que manejarlo dentro de casa. Abrirlo va a significar acomodar el código para que sea comprensible a gente extraña, estableciendo un sitio en la red y

[1] El sitio de hosting SourceForge.net, tiene 79 225 proyectos registrados a mediados de abril de 2004. Por supuesto que este número no se acerca para nada al número total de proyectos en Internet, sólo el número que usa SourceForge.

una lista de correos, y a menudo redactando la documentación del proyecto por primera vez. Todo esto significa mucho trabajo. Y además, si *aparece* algún programador interesado, habrá que soportar el peso agreqado de contestar sus preguntas por un tiempo, antes de ver el beneficio que se recibe por su presencia. Como dijo el programador Jaime Zawinski comentando los días ajetreados cuando se lanzaba el proyecto Mozilla:

> *La fuente abierta anda, pero no es definitivamente la panacea. Hay que advertir con cautela que no se puede encarar un proyecto moribundo, rociarlo con el polvo mágico de la #fuente abierta# y tener de repente todo en funcionamiento. El software es difícil. Las cosas no son tan simples.*
>
> (de **http://www.jwz.org/gruntle/nomo.html**)

Una equivocación relacionada es escatimar en la presentación y el empaquetado, creyendo que esto se puede hacer después, cuando el proyecto esté encaminado. La presentación y el empaquetado comprenden una amplia serie de tareas, todas en torno a reducir la barrera de ingreso al proyecto. Hacer un proyecto atractivo para un no iniciado significa documentarlo para el usuario y el programador, establecer un sitio web para los recien llegados, automatizar cuanto sea posible la compilación e instalación del software, etc. Desgraciadamente muchos programadores dan a este trabajo una importancia secundaria comparado con el código. Hay un par de razones para esto. De entrada se puede percibir como trabajo no productivo, porque aparentemente beneficia más a los que no están familiarizados con el proyecto. De cualquier modo, los que desarrollan el código no necesitan realmente del empaquetado. Ya conocen como instalar, administrar y usar el software, porque ellos lo escribieron. En segundo lugar, los conocimientos para hacer bien la presentación y el empaquetado son a menudo completamente diferentes a los que se requieren para escribir el código. La gente tiende a concentrarse en lo que más sabe, aún cuando podría ser más útil al proyecto que se dediquen un poco a lo que no les resulta tan familiar. En el Capítulo 2, *Primeros Pasos* se trata la presentación y el empaquetado en detalle, y explica por qué es importante que sean una prioridad desde el comienzo del proyecto.

Después se introduce la falacia de que no se requiere una dirección del proyecto cuando es de fuente abierta, o a la inversa, que las mismas prácticas de gestión usadas para un proyecto hecho en casa van a funcionar bien en un proyecto de fuente abierta. El manejo de un proyecto de fuente abierta no siempre resulta visible, pero cuando éste es exitoso tiene lugar detrás de las bambalinas de una u otra forma. Un pequeño experimento mental será suficiente para mostrar por qué. Un proyecto de fuente abierta consiste en una colección de programadores al azar —los que ya de por sí son gente con categorías independientes— que muy probablemente nunca se van a encontrar juntos, y que quizás tienen objetivos personales muy diferentes para trabajar en el proyecto. El experimento consiste en imaginar sencillamente qué va a pasarle a dicho grupo *sin* dirección. Si no creemos en milagros, el proyecto va a colapsar y diluirse muy rápidamente. Las cosas no funcionarán simplemente por si solas, por más que los deseos sean grandes. Pero la administración, aún cuando sea muy activa, es a menudo informal, sutil, y de bajo perfil. Lo único que mantiene unido al grupo de desarrollo es el convencimiento compartido de que juntos pueden hacer más que individualmente. Entonces el objetivo de la dirección es mayormente asegurar que continúen con ese convencimiento, estableciendo estándares de comunicación, cuidando que los programadores útiles no queden marginados debido a idiosincrasias personales, y en general procurando que el proyecto sea un lugar acogedor para los programadores. Las técnicas específicas para realizar esto se discuten a lo largo de este libro.

Finalmente, hay una categoría general de los problemas que podría llamarse #fallas de orientación cultural.# Hace diez años, o quizás sean sólo cinco, hubiera sido prematuro hablar de una cultura global de software libre, pero ahora ya no es así. Lentamente ha emergido una cultura visible, y aún cuando esta no es monolítica —por lo menos es tan propensa al disentimiento interno y al corporativismo como cualquier cultura limitada geográficamente— tiene ciertamente un núcleo básico consistente. Los proyectos de fuente abierta más exitosos muestran algo o el total de las características de ese núcleo. Se premian ciertos tipos de conductas y se castigan otros. Se crea una atmósfera que incita a la participación espontánea, a veces a expensas de una coordinación central. Se tienen conceptos de lo que es ser amable o ser rudo que difieren substancialmente de lo que prevalece fuera. Lo más importante es que los participantes que son asiduos tienen ya interiorizados esos conceptos y comparten un cierto consenso sobre la conducta que es aceptable. Los proyectos no exitosos a menudo se desvían apreciablemente de ese núcleo, a ve-

ces intencionalmente, y no tienen un consenso sobre lo que razonablemente constituye una conducta predeterminada. Esto quiere decir que cuando surgen los problemas la situación se viene abajo rápidamente, porque los participantes carecen de un conjunto de reflejos culturales determinados que les permita resolver sus diferencias.

Este libro es una guía práctica, no un estudio antropológico o un libro de historia. Sin embargo, un conocimiento efectivo de los orígenes del software libre actual es una base esencial para cualquier consejo práctico. Una persona que entienda esta cultura puede viajar sin límites en este mundo de la fuente abierta, encontrándose con muchas variaciones en costumbres y dialectos, y a la vez estar en la condición de participar cómoda y efectivamente en cualquier lado. Por el contrario, una persona que no entiende esta cultura encontrará que el proceso de organizar y participar en un proyecto es algo difícil y lleno de sorpresas. Puesto que el número de gente que desarrolla software libre sigue creciendo a grandes saltos, habrá muchos en ésta última categoría— ésta es mayormente una cultura de inmigrantes recientes, y continuará así por mucho tiempo. Si crees que eres uno de estos, en el próximo título se presentarán algunos antecedentes útiles para las discusiones que vendrán después, tanto en este libro como en Internet. (Por otro lado, si ya has trabajado en proyectos de fuente abierta por algún tiempo, puede ser que conozcas mucho sobre esta historia, y y sea mejor saltar a la siguiente sección.)

La Historia

Compartir el software tiene tanta historia como el software mismo. En los primeros tiempos de los ordenadores, los fabricantes se dieron cuenta que vendrían avances competitivos en la innovación del hardware y no prestaron mucha atención al software como una ventaja para el desarrollo de sus negocios. Muchos de los usuarios de las primeras máquinas eran científicos o técnicos que podían modificar y ampliar el software que incluía la máquina. A veces los usuarios distribuían sus aportes no solamente al fabricante, sino también a otros usuarios que tenían máquinas similares. A menudo los fabricantes toleraban esto, e incluso lo estimulaban: para ellos cualquier mejora en el software, fuera cual fuese su procedencia, contribuía a que las máquinas resultasen más atractivas para otros usuarios potenciales.

Aunque esta primera época se parece de muchas maneras a la cultura actual del software libre, difiere fundamentalmente en dos aspectos: primero que había poca estandarización del hardware— era un momento de mucha innovación en el diseño de los ordenadores, pero la diversidad en las arquitecturas hacía que cualquier cosa resultara incompatible con la otra. Así que el software que se escribía para una máquina generalmente no servía para otra. Los programadores se inclinaban hacia una arquitectura en particular o familia de arquitecturas y en ellas se hacían expertos (mientras que hoy se adquiere experiencia en un lenguaje de programación o una familia de lenguajes y se espera que esa experiencia se pueda luego transferir a cualquier hardware en que se vaya a trabajar). Puesto que un experto se inclinaba a sólo un tipo de ordenador, la acumulación de sus conocimientos tenía el efecto de hacer más atractivo ese ordenador tanto para él como para sus colegas. Por lo que los fabricantes tenían gran interés en difundir tanto como pudieran la codificación y el conocimiento de alguna máquina específica.

En segundo lugar Internet no existía. Aunque tenían menos restricciones legales que hoy para compartir, había más restricciones técnicas: Hablando comparativamente, los medios para transmitir datos de un lado a otro eran dificiles y engorrosos. Había algunas pequeñas redes locales, aptas para compartir información entre empleados del mismo laboratorio de investigación o compañía. Pero quedaban por superar una serie de trabas si se quería compartir con alguien, estuviere donde estuviere. Estas trabas se *superaban* en muchos casos. A veces eran grupos varios que se contactaban independientemente, enviándose discos o cintas por correo, y a veces eran los fabricantes mismos que servían como centrales de intercambio de los aportes individuales. También ayudaba que muchos de los que desarrollaban los primeros ordenadores trabajasen en las universidades, en donde era costumbre publicar los avances. Pero la realidad de la transmisión de datos implicaba que siempre que se los quería compartir se topaba con un impedimento que era proporcional a la distancia (física u organizacional) que el software tenía que viajar. Era imposible compartir algo con todo el mundo sin resistencias, tal como se puede hacer hoy.

El Florecimiento del Software Propietario y del Software

Libre

A medida que maduraba la industria ocurrían simultáneamente algunos cambios interrelacionados. La gran diversidad de los diseños del hardware finalmente cedieron el paso a unos pocos ganadores —ganadores por tener una tecnología superior, o una comercialización superior, o una combinación de ambas cosas. Al mismo tiempo, no coincidente en su totalidad, el desarrollo de los así llamados lenguajes de programación de #alto nivel# significaba que se podía escribir un programa de una sóla vez en un lenguaje, y luego traducirlo automáticamente (#compilarlo#) para que funcione en diferentes tipos de ordenadores. Las consecuencias de esto no se quedaron perdidas en los fabricantes de hardware: un usuario podía ahora emprender un mayor esfuerzo de ingeniería de software sin encerrarse en una arquitectura particular. Cuando esto se combinaba con la disminución gradual de las diferencias en la calidad de funcionamiento entre los ordenadores, y mientras los diseños menos eficientes eran eliminados, un fabricante que se centraba en el hardware como único beneficio podía divisar una disminución de sus ganancias en el futuro. La potencia de computación pura se convertía en un bien fungible, mientras que el software se convertía en el diferenciador. Aparecía como una buena estrategia vender software, o al menos, tratarlo como parte integral de las ventas del hardware.

Esto significó que los fabricantes tuvieran que ser más estrictos defendiendo los derechos de copia de los códigos. Si los usuarios hubieran continuado simplemente con su costumbre de compartir y modificar los códigos de manera libre y gratis, hubieran instalado en forma independiente las mejoras que ahora empezaban a ser vendidas como #valor agregado# por los proveedores. Peor aún, el código compartido podría haber caído en las manos de los competidores. La ironía de esto es que ocurría al mismo tiempo que Internet estaba ganando terreno. Justamente, cuando se hacía técnicamente posible compartir el software y se caían los obstáculos, los cambios en el mundo de los negocios hacían del compartir algo económicamente indeseable, por lo menos desde el punto de vista propio de una compañía. Los proveedores imponían sus controles, ya sea negando el acceso al código a los usuarios que corrían el programa en sus máquinas, o mediante acuerdos de no difundir el código, lo que hacía que el compartir fuera imposible.

Una resistencia conciente

Mientras se extinguía el mundo del intercambio de códigos se cristalizaba una contra reacción al menos en la mente de un programador. Richard Stallman trabajaba en el laboratorio de inteligencia artificial en el Instituto Tecnológico de Massachussets en la década de 1970 e inicios de 1980, la época y el lugar de oro para la costumbre de compartir los códigos. El laboratorio de IA tenía una fuerte "ética de hackers"[2] y no sólo se estimulaba al personal de los proyectos sino que era de esperar que todos los avances hechos en el sistema fueran compartidos. Como luego escribiría Stallman:

> *No le llamábamos #software libre# a nuestro software porque ese término no existía; pero era precisamente eso. Toda vez que alguien de otra universidad quería llevar y usar un programa, nosotros se lo ofrecíamos con gusto. Si se veía que alguien usaba un programa distinto e interesante, se le podía pedir el código fuente, para poder leerlo, cambiarlo o fusionar partes de él en un programa nuevo.*
>
> (de **http://www.gnu.org/gnu/thegnuproject.html**)

Esta comunidad edénica colapsó con Stallman poco después de 1980, cuando los cambios que venían ocurriendo en el resto de la industria finalmente alcanzaron al laboratorio de IA. Una compañía que se iniciaba incorporaba a muchos de los programadores del laboratorio para trabajar en un sistema operativo similar al que habían desarrollado allí, pero ahora bajo una licencia exclusiva. Al mismo tiempo, el laboratorio de IA adquiría nuevos equipos que llegaban con un sistema operativo de marca registrada.

Stallman vio la gran trama de lo que estaba sucediendo:

> *Los ordenadores modernos de la época, como el VAX o el 68020, venían con sus siste-*

[2] Stallman usa la palabra "hacker" con el significado de "alguien que ama la programación y disfruta si hace algo inteligente" y no con el sentido más reciente de "alguien que se conecta como un intruso en los ordenadores"

> *mas operativos propios, pero ninguno era un software libre: se debía firmar un acuerdo de no revelar los contenidos para poder recibir una copia ejecutable.*
>
> *Lo cual significaba que el primer paso para usar un ordenador era prometer que no había que ayudar al vecino. La comunidad de cooperación estaba prohibida. La regla que establecían los dueños del software propietario era: "si compartes con tu vecino, eres un pirata. Si quieres cambios, nosotros los haremos, si nos lo pides.#*

Y por su personalidad peculiar decidió ofrecer resistencia a esta nueva ola. En lugar de continuar trabajando en el diezmado laboratorio de IA, o aceptar el trabajo de escribir código en alguna de las compañías nuevas, en las que su trabajo iba a quedar encerrado en una caja, renunció al laboratorio y comenzó el proyecto GNU y la Fundación de Software Libre (FSF por sus siglas en Inglés). El objetivo del GNU[3] era desarrollar un sistema operativo y un conjunto de aplicaciones completamente libres y abiertas, donde nunca se impediria a la gente hackear o compartir sus cambios. En esencia, estaba empeñado en recrear lo que se había destruido del laboratorio de IA, pero a una escala global, y sin las vulnerabilidades que ponían a la cultura del laboratorio de IA en un estado de posible desintegración.

Además de trabajar en el nuevo sistema operativo, Stallman inventó una licencia de copyright cuyos términos garantizaban que los códigos permanecerían gratis en perpetuidad. La Licencia Pública General GNU es una ingeniosa pieza de judo legal: dice que los códigos pueden ser copiados y modificados sin ninguna restricción y que ambas copias y trabajos derivados (a saber, las versiones modificadas) deben ser distribuidas bajo la misma licencia que el original, sin poner restricciones adicionales. En efecto, se usan las leyes del copyright para conseguir un efecto contrario al que apunta el copyright tradicional: en lugar de limitar la distribución del software, prohíbe que *nadie*, ni siquiera el autor, lo limite. Para Stallman, esto era mejor que si hubiera puesto su código en el dominio público. Si hubiera estado en el dominio público, cualquier copia podría haber sido incorporada a los programas propietarios (como ya se sabía que había sucedido con códigos que tenían licencias permisivas). Aunque una incorporación como éstas no hubiera disminuido la disponibilidad de los códigos originales, hubiera significado que los esfuerzos de Stallman iban a beneficiar al enemigo— al software propietario. La Licencia Pública General puede entenderse como una forma de proteccionismo del software libre, porque impide que el software no-libre se aproveche de los códigos que están bajo esta licencia. La Licencia Pública General y su relación con otras licencias del software libre se discuten en detalle en el Capítulo 9, *Licencias, Copyrights y Patentes*.

Con la ayuda de nuevos programadores, alguno de los cuales compartían la ideología de Stallman y otros que simplemente querían ver abundante código disponible en forma gratuita, el Proyecto GNU comenzó entregando versiones libres para reemplazar muchos de los componentes críticos de sistemas operativos. Gracias a la estandarización expandida del hardware y software para ordenadores, se hizo posible usar los reemplazos GNU en sistemas no-libres, y mucha gente lo hizo. El editor de texto de GNU (Emacs) y el compilador C (GCC) tuvieron especial éxito, ganando muchos seguidores leales, no por términos ideológicos, sino simplemente por los méritos técnicos. Alrededor del año 1990, GNU había producido la mayor parte de un sistema operativo libre, con excepción del núcleo —la parte por la que realmente la máquina arranca y se hace responsable de manejar la memoria, el disco y otros recursos del sistema.

Desafortunadamente el proyecto GNU había elegido un diseño de núcleo que resultó más difícil de implementar de lo esperado. La consiguiente demora impedía que la Fundación de Software Libre ofreciera la primera versión de un sistema operativo enteramente libre. La pieza final fue instalada en su lugar por Linus Torvalds, un estudiante de computación finlandés quien con la ayuda de voluntarios de todo el mundo había completado un núcleo libre usando un diseño más conservador. Le llamó Linux, y cuando fue combinado con los programas GNU existentes tuvo como resultado un sistema operativo completamente libre. Por primera vez se podía arrancar un ordenador y hacerlo trabajar sin usar ningún software propietario.[4]

[3] Siglas que significan "GNU No es Unix" donde GNU significa... lo mismo.

[4] Un sistema operativo libre para ordenadores compatibles con IBM, llamado 386BSD, había aparecido poco antes que Linux. Sin embargo, era mucho mas difícil conseguir un 386BSD y hacerlo funcionar. Linux tuvo tanta resonancia no solo porque era libre, sino porque realmente tenía una probabilidad alta de hacer arrancar al ordenador una vez que se instalaba.

Muchas partes del software de este nuevo sistema operativo no fueron producidas por el proyecto GNU. De hecho, el GNU no fue el único grupo que trabajaba para producir un sistema operativo libre (por ejemplo, el código que luego fue NetBSD y FreeBSD estaba ya en desarrollo en ese momento). La importancia de la Fundación de Software libre no solamente residía en los códigos que se escribían, sino en el tratamiento político del tema. Al hablar del software libre como una causa en lugar de una conveniencia, era casi imposible que los programadores *no* tomasen una postura política de ello. Aún los que no estaban de acuerdo con la Fundación de Software Libre tuvieron que enfrentar la causa, aunque más no sea para proponer una posición diferente. La efectividad que tuvo la Fundación de Software Libre en el proceso de difusión residió en la vinculación del código al mensaje, por medio de la Licencia Pública General y de otros textos. Al mismo tiempo que se difundían los códigos, se distribuía también el mensaje.

Resistencia accidental

Habían muchas otras cosas sucediendo en la escena naciente del software libre, sin embargo, pocas eran tan explícitas ideológicamente como el Proyecto GNU de Stallman. Una de los sucesos mas importantes fue la *Berkeley Software Distribution* (*BSD*), una reimplementación gradual del sistema operativo Unix, que hasta finales de la década de los 70' había sido un proyecto de investigación sin restricciones de AT&T— hecho por programadores de la Universidad de Berkeley en California. Este grupo BSD no hizo una declaración política sobre la necesidad de que los programadores se unan y compartan unos con otros, pero *practicaron* la idea con talento y entusiasmo, coordinando un esfuerzo de desarrollo distribuido masivamente en el cual fueron reescritos los recursos de línea de comando y las bibliotecas del Unix y eventualmente también el núcleo del sistema operativo, en su mayoría por voluntarios que los tomaban de borradores. El proyecto BSD resultó un primer ejemplo de desarrollo de un software libre no-ideológico, y también sirvió como campo de entrenamiento para muchos desarrolladores que continuarían activos en el mundo del software libre.

Otro proyecto de desarrollo cooperativo fue el *X Window System*, un entorno gráfico de computación libre y transparente en la red, desarrollado en el MIT a mediados de la década de 1980 en coparticipación con empresas que tenían el interés común de estar en condiciones de ofrecer a sus clientes un sistema operativo con ventanas. Lejos de oponerse al software propietario, la licencia X permitía deliberadamente que se hicieran extensiones propietarias encima del nucleo libre — cada miembro del consorcio quería tener la oportunidad de mejorar la distribucion X predeterminada y consiguientemente ganar una ventaja competitiva con respecto a los otros miembros. El X Windows[5] era un software libre, pero fundamentalmente como una manera de nivelar el campo de juego entre intereses de las empresas competidoras, y no por el deseo de poner fin a la dominación del software propietario. Todavía hay otro ejemplo, el TeX de Donald Knuth, un sistema de tipografía, que se alimentaba del proyecto GNU. Ofreció una versión bajo una licencia que permitía que cualquiera modifique y distribuya el código, pero que no se llamara "TeX" a no ser que superara una serie de tests de compatibilidad muy estrictos (este es un ejemplo de una clase de licencias libres "protectoras de marcas registradas" de las que se hablará más en el Capítulo 9, *Licencias, Copyrights y Patentes*) Knuth no estaba tomando partido para un lado ni para el otro en la cuestión del software libre contra el propietario, solo necesitaba un sistema mejor de impresión para cumplir con su objetivo *real* —un libro sobre programación de ordenadores— y no encontró escollos para presentar al mundo su sistema una vez que estuvo hecho.

Aún sin tener un listado completo de proyectos y licencias, se puede afirmar con seguridad que para el fin de la década de los 80' había una buena cantidad de software libre y una amplia variedad de licencias. La diversidad de licencias reflejaba una diversidad de motivaciones correspondientes. Incluso algunos de los programadores que eligieron la Licencia Pública General de GNU estaban mucho menos motivados ideológicamente que el proyecto GNU mismo. Aunque disfrutaban trabajando en el software libre, muchos desarrolladores no consideraron que el software propietario era una lacra social. Había quienes sentían un impulso moral de liberar al mundo del #acaparamiento de software# (un término que usaba Stallman para el software no libre), pero otros estaban más motivados por un entusiasmo técnico, o por el placer de trabajar con colaboradores de pensamiento afín, o simplemente por el deseo humano de la gloria. Pero las motivaciones disparatadas no intervinieron en forma destructiva en todo este con-

[5]Preferían que se llamara "X Windows System", pero en la práctica se le llama comunmente "X Windows", porque tres palabras es demasiado complicado.

fín. Esto se explica en parte porque, en oposición a los que acontece en otras formas creativas como la prosa o las artes visuales, el software debe superar pruebas semi-objetivas para ser considerado un éxito: debe funcionar y estar razonablemente libre de errores. Esto otorga a todos los participantes del proyecto una especie de pie de igualdad común, una razón y un encuadre para trabajar juntos sin preocuparse mucho de otros títulos que no sean los conocimientos técnicos.

Además, los desarrolladores tenían otra razón para permanecer juntos: acontecía que el mundo del software libre estaba produciendo códigos de muy alta calidad. En algunos casos se podía demostrar que eran técnicamente superiores a las alternativas del software no libre que se les acercaban; en otros casos eran al menos comparables y por supuesto, costaban menos. Mientras que solo unos pocos pudieron estar motivados para usar software libre por razones estrictamente filosóficas, la gran mayoría se sentía feliz de usarlos porque cumplían mejor con las tareas. Y entre los usuarios, algún porcentaje estaba siempre listo para donar su tiempo y habilidad para ayudar a mantener y mejorar el software.

Esta tendencia de producir buenos códigos no era ciertamente universal, pero se repetía por todas partes con frecuencia en aumento en los proyectos de software libre. Las empresas que dependían fuertemente del software lo empezaron a notar gradualmente. Muchos de ellos descubrieron que ya estaban usando software libre en las operaciones de todos los días, sólo que no lo sabían (los gerentes de alto rango no siempre saben todo lo que ocurre en las dependencias de la tecnología informática). Las corporaciones comenzaron a tomar cartas activas en los proyectos del software libre, contribuyendo con tiempo y equipos, y a veces subvencionando directamente al desarrollo de programas libres. Estas inversiones podían, en el mejor de los casos, devolverles muchas horas de tiempo extra. Las subvenciones solo pagaban a una cantidad pequeña de programadores expertos para que dedicaran su trabajo de tiempo completo, pero cosechaban los beneficios de las contribuciones de *todos*, incluso de voluntarios no pagos, y programadores pagados por otras corporaciones.

#Libre# vs #Abierto#

Cuando las corporaciones prestaron mayor atención a los programadores de software libre se enfrentaron con nuevas formas de presentación. Una de ellas fue la palabra #libre#. Al escuchar por primera vez el término #software libre# muchos pensaron erróneamente que solamente significaba #software de costo cero#. Es verdad que todo software libre tiene un costo cero[6], pero no todo el software gratis es libre. Por ejemplo, durante la guerra de los navegadores de la década de los '90 Netscape y Microsoft repartían gratis sus navegadores en la disputa por ganar la mayor participación en el mercado. Ninguno de estos navegadores era libre en el sentido que tiene el "software libre". No se dispone del código fuente, y si se lo tuviera, no se tiene el derecho de modificarlo o redistribuirlo.[7] Lo único permitido era bajar los programas ejecutables y hacerlos funcionar. Los navegadores no eran más libres que los softwares empaquetados y comprimidos que se compran en un negocio; sólo que el precio era mas bajo.

Esta confusión en la palabra #libre# se debe a una desafortunada ambigüedad de lenguaje, en este caso del inglés. En otras lenguas romances aparece la diferencia entre precio bajo y libertad porque existen las palabras *gratis* y *libre* que se distinguen con facilidad. Pero siendo el inglés el lenguaje puente dentro de Internet, pasó esto a significar que un problema con el inglés era también un problema para los demás. Este malentendido suscitado por la palabra #libre# era tan penetrante para los angloparlantes que los programadores de software desarrollaron una formula estándar que repetían: "Es *libre* (free) como la *libertad*, no como la cerveza *gratis* (free)" Aún ahora, tener que explicar esto una y otra vez resulta fatigante. Muchos programadores sentían, no sin razón, que la palabra ambigua (en inglés) #libre# (free) estaba obstaculizando la comprensión del público en relación a este software.

Pero este problema se profundizó más aún. La palabra #libre# llevaba consigo una inevitable connotación moral: si la libertad era un bien en si mismo, no era importante si el software era mejor o más conveniente para ciertos asuntos o ciertas circunstancias. Estos últimos efectos aparecían como secundarios, por otras motivaciones que no eran en el fondo ni técnicas ni comerciales, sino morales. Más todavía, la

[6] Se podría cobrar algo por repartir las copias del softwre libre, pero puesto que no se puede parar a los que lo reciben si éstos quieren ofrecerlo gratis después, el precio vuelve a cero inmediatamente.

[7] El código fuente del Navegador de Netscape *apareció* eventualmente bajo una licencia de fuente abierta, en 1998, vino a ser la base del navegador Mozilla. Ver http://www.mozilla.org/.

postura de #libre como la libertad# llevaba a una flagrante incoherencia de las corporaciones que subvencionaban algunos programas libres para algunas áreas de sus negocios, pero continuaban comercializando software propietario en otras.

Estos dilemas llovían sobre de una comunidad que ya estaba aplastada por una crisis de identidad. Los programadores que *escriben* actualmente el software libre no se sienten necesariamente identificados con el objetivo central #si lo hay- del movimiento del software libre. Sería engañoso decir que las opiniones van de un extremo al otro, porque esto implicaría la falsedad de imaginar que nos movemos en una línea de pensamiento, cuando en realidad es una distribución multidimensional. Sin embargo, si estamos dispuestos a obviar las sutilezas, por el momento pueden diferenciarse dos amplias categorías. Un grupo se alinea bajo el punto de vista de Stallman, para quien la libertad de participar y modificar es lo mas importante, y por lo tanto si no se habla de libertad se está esquivando el núcleo principal de la cuestión. Otros piensan que el software es el argumento más importante a su favor, y se sienten incómodos con la proclamación del software propietario como algo inherentemente malo. Algunos de los programadores de software, auque no todos, creen que el autor (o el empleador, en el caso de trabajo pagado) *debería* tener el derecho de controlar las cláusulas de la distribución y que no se necesita agregar un juicio moral en la selección de algunas cláusulas particulares.

Por mucho tiempo no se necesitó examinar o articular estas diferencias, pero el éxito floreciente del software libre hizo que esta cuestión fuera inevitable. En 1998 un grupo de programadores creó el término *fuente abierta* como una alternativa para "libre" y fueron ellos quienes crearon la Iniciativa por el Código Abierto(OSI por sus siglas en Inglés). [8] La Iniciativa por el Código Abierto creía que el término "software libre" llevaba a una confusión potencial, y que la palabra "libre" era justamente un síntoma del problema general: que el movimiento necesitaba un programa de mercado para lanzarlo en el mundo de las grandes empresas, y que hablar de moral y de los beneficios sociales del compartir no iba a tener vuelo en las salas de las empresas. Tomando sus propias palabras:

> *La Iniciativa por el Código Abierto es un programa de mercado para el software libre. Significa fundar el #software libre# sobre bases sólidas y prácticas más que en una discusión acalorada. La sustancia ganadora no ha cambiado, sí en cambio la actitud de perdedores y su simbolismo. ...*
>
> *La aclaración que debe hacerse a muchos técnicos no es acerca del concepto de fuente abierta, sino sobre el nombre. ¿Por qué no llamarle, como se ha hecho tradicionalmente, software libre?*
>
> *Una razón definitiva es que el término #software libre# se confunde fácilmente de manera que lleva a terrenos conflictivos. ...*
>
> *Pero la verdadera razón del cambio de cartel es una razón de comercialización. Estamos ahora tratando de lanzar nuestro concepto al mundo corporativo. Tenemos un producto ganador, pero nuestra posición, en el pasado, ha sido terrible. El término "software libre" se ha malentendido entre las personas de negocios, quienes confunden el deseo de compartir con una conducta anticomercial, o peor todavía, con un robo.*
>
> *Los CEOs y CTOs de grandes corporaciones ya establecidas nunca compraran "software libre." Pero si manteniendo la misma tradición, la misma gente y las mismas licencias de software libre y les cambimos el nombre poniéndole #código abierto#, entonces sí lo compraran.*
>
> *Algunos hackers encuentran esto difícil de creer, porque son técnicos que piensan en concreto, con términos substanciales, y no entienden la importancia de la imagen de algo cuando uno lo está vendiendo.*
>
> *Para el mercado la apariencia es la realidad. La apariencia de que estamos dispues-*

[8] El sitio web de la OSI es http://www.opensource.org/.

> *tos a bajarnos de nuestras barricadas y a trabajar con el mundo corporativo importa tanto como la realidad de nuestras conductas o convicciones, y de nuestro software.*
>
> (de **http://www.opensource.org/advocacy/faq.php** y **http://www.opensource.org/advocacy/case_for_hackers.php#marketing**)

En este libro aparecen las puntas de muchos icebergs de la controversia. Se refiere a #nuestras convicciones#, pero discretamente evita decir con exactitud de que convicciones se trata. Para algunos, puede ser la convicción de que el código desarrollado en concordancia con un proceso abierto será un código mejor; para otros pudiera ser la convicción de que toda información debiera ser compartida. Aparece el uso del término #robo# para referirse (posiblemente) al copiado ilegal —una costumbre que muchos objetan alegando que pese a todo no es un robo si el propietario original todavía tiene el artículo. Hay una sospecha inquietante que el movimiento de software libre podría ser acusado por equivocación de anticomercialismo, aunque queda por examinar detenidamente la cuestión de si esta acusación tendría alguna base en los hechos.

Esto no quiere decir que el sitio web de la OSI sea incoherente o engañoso. No lo es. En realidad es un ejemplo de lo que la OSI reclama como perdido por el movimiento de software libre. Una buena comercialización, donde #buena# significa viable en el mundo de los negocios. La Iniciativa de Fuente Abierta brindó a mucha gente exactamente lo que buscaban —un vocabulario para referirse al software libre como una metodología de desarrollo y una estrategia para los negocios, en lugar de una cruzada moral.

La aparición de la Iniciativa por el Código Libre cambió el panorama del software libre. Formalizó una dicotomía que por mucho tiempo no tuvo un nombre, y al hacerlo forzaba al movimiento a reconocer que tenía una política interna al mismo tiempo que una externa. Hoy, el efecto es que ambos lados han tenido que encontrar un terreno común, puesto que la mayoría de los proyectos incluye a programadores de ambos campos, como también otros participantes que no encajan claramente en una categoría. Esto no impide que se hable de motivaciones morales —por ejemplo, a veces aparecen convocatorias con recaídas en la tradicional #ética de hackers#. Pero es raro que un desarrollador de software libre / fuente abierta entre a cuestionar abiertamente las motivaciones básicas de otros. La contribución encubre al contribuyente. Si alguien escribe un buen código, no se le pregunta si lo hace por razones morales o porque su empleador le paga, o porque está engrosando su currículum, o lo que sea. Se evalúa la contribución en términos técnicos, y se responde con fundamentos técnicos. Inclusive organizaciones políticas como el proyecto Debian, cuyo objetivo es ofrecer un entorno computacional 100% libre (#libre como la libertad#), no tienen peros para integrarse con el código no libre y cooperar con programadores que no comparten exactamente los mismos objetivos.

La situación de Hoy

Cuando se maneja un proyecto libre, no se necesita hablar todos los días sobre esos enfoques filosóficos. Los programadores no pretenden que todos los integrantes del proyecto estén de acuerdo con sus puntos de vista en todos los aspectos (aquellos que insisten en hacerlo se encuentran rápidamente incapacitados para trabajar en cualquier proyecto). Pero se necesita estar advertido que la cuestión de #libre# contra #fuente abierta# existe, en parte para evitar decir cosas que pueden enemistarlo a uno con algún otro participante, y en parte porque un entendimiento con los demás y sus motivaciones es la mejor manera, y —en cierto sentido— la *única* manera de llevar adelante el proyecto.

El software libre es una cultura por elección. Para trabajar con éxito en esta cultura hay que entender por qué hay personas que la eligen en primer lugar. Las técnicas coercitivas no tienen efecto. Si hay alguien que no se siente cómodo en un proyecto, recurre a otro. El software libre se distingue incluso entre las comunidades de voluntarios por sus inversiones limitadas. Muchos de los participantes nunca se encuentran cara a cara, y simplemente hacen donación de algún tiempo cada vez que se sienten motivados. Los conductos normales que conectan a los seres humanos entre si y se concretan en grupos duraderos se reducen a un pequeño canal: la palabra escrita, trasmitida por cables eléctricos. Por esto puede llevar mucho tiempo para formar un grupo dedicado y unido. Y a la inversa, es muy fácil que un proyecto pierda un voluntario potencial en los cinco primeros minutos de haberse encontrado. Si el proyecto no impacta

con una buena impresión, raras veces los recién llegados le darán una segunda oportunidad.

La transitoriedad real o *potencial* de las relaciones es quizás la tarea más desalentadora que se debe enfrentar en un nuevo proyecto. ¿Qué va a persuadir a toda esa gente a permanecer juntos el tiempo suficiente necesario para producir algo útil? La respuesta es tan compleja como para ocupar el resto de este libro, pero si se tiene que expresar en una sola frase, sería la siguiente:

> *Las personas deben sentir que su conexión con un proyecto, y su influencia sobre él, es directamente proporcional a sus contribuciones.*

Ningún desarrollador, real o potencial, debe sentir que no es tenido en cuenta o es discriminado por razones que no sean técnicas. Con claridad, los proyectos con apoyo de empresas y/o desarrolladores pagos tienen que ser especialmente cuidadosos en este aspecto, como se expresa en detalle en el Capítulo 5, *Dinero*. Por supuesto, esto no quiere decir que si no hay apoyo de empresas no hay nada de que preocuparse. El dinero es sólo uno de los tantos factores que pueden afectar el éxito de un proyecto. Otras cuestiones son el lenguaje que se va a elegir, la licencia, cuál será el proceso de desarrollo, qué tipo de infraestructura hay que instalar, cómo promocionar efectivamente el arranque del proyecto, y muchas otras cosas más. El contenido del próximo capítulo será cómo dar el primer paso con el pié correcto al comenzar un proyecto.

Capítulo 2. Primeros Pasos

El clásico modelo de cómo los proyectos de software libre deben iniciar fue propuesto por Eric Raymond, en un artículo ahora famoso sobre procesos de código abierto titulado *La catedral y el bazar*. Él escribió:

> *Todos los trabajos buenos en software comienzan tratando de paliar un problema personal de quien los programa*
>
> (de **http://www.catb.org/~esr/writings/cathedral-bazaar/**)

Es de notar que Raymond no estaba diciendo que los proyectos de código abierto no sólo suceden cuando cierto individuo tiene una necesidad. En cambio, nos está diciendo que los *buenos* programas son resultado de que un programador tenga un interés personal en ver el problema resulto. La relevancia de esto para el software libre ha sido que ésta necesidad personal sea frecuentemente a motivación para iniciar un proyecto de software libre.

Esto sigue siendo la manera cómo muchos de los proyectos libres se inician, pero menos ahora que en 1997, cuando Raymond escribió esas palabras. Hoy, tenemos el fenómeno de organizaciones —incluidas corporaciones con fines de lucro—iniciando desde cero, proyectos Open Source centralizados y a gran escala. El desarrollador solitario, tecleando algo de código para resolver un problema local y luego dándose cuenta de que los resultados tienen un mayor aplicación, sigue siendo la fuente de muchos software libre, pero esa no es la única historia.

De todas formas, el objetivo de Raymond sigue siendo profundo. La condición esencial es que los productores de software libre tengan un interés directo en su éxito, porque ellos mismos lo utilizan. Si el software no hace lo que se supone debería hacer, la persona u organización que lo han producido sentirán insatisfacción en su labor diaria. Por ejemplo, el proyecto OpenAdapter (http://www.openadapter.org/), el cual fue iniciado por el banco de inversiones Dresdner Klienwort Wasserstein es un marco de trabajo para la integración de sistemas de información financieros dispares, poco de esto puede ser considerado como un problema personal de un programador. En particular éste problema surge directamente de la experiencia de la institución y sus socios, por lo cual si el proyecto falla en aliviarlos, ellos lo sabrán. Este arreglo produce buenos programas porque el bucle de críticas fluye en la dirección correcta. El programa no está siendo escrito para ser vendido a alguien más, es solo para que sean ellos quienes resuelvan *sus* problemas. Está siendo desarrollado para resolver su *propio* problema, luego compartiéndolo con todo el mundo como si el problema fuera una enfermedad y el software la medicina, la cual debe ser distribuida para erradicar la epidemia.

Este capítulo trata de cómo introducir un nuevo proyecto de software libre al mundo, pero muchas de sus recomendaciones sonarán familiares a una organización sanitaria distribuyendo medicinas. Los objetivos son muy similares: quieres dejar claro lo que hace la medicina, hacerla llegar a las manos correctas y asegurarte de que aquellos quienes la reciben saben como usarla. Pero con el software, también deseas incitar a algunos de los receptores a unirse al esfuerzo de investigación para mejorar la medicina.

La distribución del software libre es una tarea a dos bandas. El programa necesita usuarios y desarrolladores. Estas dos necesidades no tienen por que estar en conflicto, pero si que añaden cierta complejidad a la presentación inicial de un proyecto. Alguna información es útil para las dos audiencias, alguna sólo lo es para alguna u otra. Ambos tipos de información deben suscribirse al principio de las presentaciones en escala, esto es, el grado de detalle con el que se presenta cada etapa debe corresponder directamente a la cantidad de tiempo y esfuerzo puesto por el lector. Un mayor esfuerzo debe tener siempre una mayor recompensa. Cuando los dos no se relacionan conjuntamente, las personas pueden perder rápidamente su fe y perder el impulso.

El corolario a esto es:*las apariencias importan*. Los programadores en particular, no desean creer esto. Su amor por la sustancia sobre la forma es casi un punto de orgullo profesional. No es un accidente que tantos desarrolladores exhiban una antipatía hacia los trabajos en marketing y en relaciones públicas o

que diseñadores gráficos profesionales usualmente se sientan horrorizados de lo que los desarrolladores ingenian.

Esto es penoso, ya que hay situaciones en las que la forma *es* la sustancia y la presentación de proyectos es una de estas. Por ejemplo, lo primero que un visitante descubre sobre un proyecto es como se ve su sitio web. Esta información es absorbida antes de que el contenido en si sea comprendido—antes de que cualquier línea haya sido leída o enlaces pulsados. Aunque parezca injusto, las personas no pueden evitar el formarse una opinión inmediatamente después de la primera impresión. La apariencia del sitio señala si se ha tomado cuidado en la organización de la presentación del proyecto. Los humanos tenemos una antena extremadamente sensible para detectar el empeño en el cuidado. Muchos de nosotros podemos decir con sólo un vistazo si un sitio web ha sido ensamblado rápidamente o ha sido diseñado con cuidado. Ésta es la primera pieza de información que el proyecto muestra y la impresión que cree será asociada al resto del proyecto por asociación.

Aunque mucho de éste capítulo habla acerca del contenido con el que se debería iniciar el proyecto, recuerde que la presentación también importa. Ya que el sitio web debe funcionar para dos tipos diferentes de visitantes—usuarios y desarrolladores— hay que ser directo y conciso. A pesar de que este no es el lugar para un tratado general acerca de diseño web, un principio es suficientemente importante para merecer nuestra atención, particularmente cuando sirve a múltiples audiencias: la gente debe tener una idea de a donde lleva un enlace antes de pulsar en el. Por ejemplo, debe ser obvio que *con sólo ver el enlace* a la documentación para los usuarios, que les lleve a la documentación para los usuarios, sin mencionar la documentación para los desarrolladores. Dirigir un proyecto se basa parcialmente en suministrar información, pero también en suministrar comodidad. La mera presencia de ofrecer ciertos estándares, en lugares obvios, tranquiliza a usuarios y desarrolladores quienes están decidiendo si desean involucrarse. Dice que este proyecto funciona, ha anticipado las preguntas que la gente puede hacer y ha hecho un esfuerzo en responderlas sin la necesidad del más mínimo esfuerzo por parte del visitante. Al dar ésta aura de preparación, el proyecto envía un mensaje: "Su tiempo no será malgastado si se involucra", lo que es exactamente lo que la gente desea escuchar.

Primero investiga

Antes de iniciar un proyecto Open Source hay un importante advertencia:

Siempre investiga si existe un proyecto que hace lo que deseas. Las posibilidades son muy buenas de que cualquier problema que desees resolver ahora alguien más lo haya deseado resolver con anterioridad. Si han sido capaces de resolverlo y han liberado bajo una licencia libre entonces hoy, no será necesario inventar la rueda. Existen excepciones claro: si deseas iniciar un proyecto como experiencia educativa, el código pre-existente no es de ayuda o quizás el proyecto que deseas iniciar es muy especializado y sabes que no existe la posibilidad de que alguien más lo haya hecho ya. Pero generalmente, no hay necesidad en no investigar ya que las ganancias pueden ser grandiosas. Si los buscadores más utilizados no muestran nada, intenta tus búsquedas en: http://freshmeat.net/(un sitio sobre noticias de proyectos open source y del cual hablaremos un poco más adelante), en http://www.sourceforge.net/ y en el directorio de proyectos de la Free Software Foundation http://directory.fsf.org/.

Incluso si no se encuentra exactamente lo que estamos buscando, podría encontrar algo parecido, a lo que tiene más sentido unirse a ese proyecto y añadir funcionalidad en lugar de empezar desde cero por si mismo.

Empezando con lo que se tiene

Has investigado, sin encontrar nada que realmente se adapte a tus necesidades, y decides iniciar un nuevo proyecto.

¿Ahora qué?

Lo más difícil acerca de lanzar un proyecto de software libre es transformar una visión privada a una pública. Tú y tu organización quizás sepan exactamente lo que deseas pero expresar ese objetivo de una

manera comprensiva al resto del mundo tiene su trabajo. De hecho, es esencial, que te tomes tu tiempo para hacerlo. Tú y los otros fundadores deben decidir sobre qué va realmente el proyecto—eso es, decidir sus limitaciones, lo que *no podrá* hacer como lo que sí—y escribir una declaración de objetivos. Ésta parte no suele ser usualmente difícil, aunque puede revelar afirmaciones y desacuerdos sobre la naturaleza del proyecto, lo cual esta bien: mejor resolver esto ahora que luego. El próximo paso es empaquetar el proyecto para el consumo público, y esto es, básicamente, trabajo puro y duro.

Lo que lo hace laborioso es porque consiste principalmente en organizar y documentar lo que ya todo el mundo sabe—todos aquellos involucrados en el proyecto hasta ahora. Así que, para las personas trabajando ya, no existen beneficios inmediatos. Estos no necesitan de un fichero `README` que resuma el proyecto ni de un documento de diseño o manual de usuario. No necesitan de un árbol de código cuidadosamente ordenado conforme a los estándares informales, ampliamente utilizados para las distribuciones de fuentes. De cualquier forma como esté ordenado el código fuente estará bien, porque ya estarán acostumbrados de todas formas, y si el código funciona, saben cómo usarlo. Ni siquiera importa si las afirmaciones fundamentales sobre la arquitectura del proyecto siguen sin documentar, ya están familiarizados con lo que deben hacer.

En cambio, los recién llegados, necesitan de todas estas cosas. Afortunadamente, no las necesitan todas a la vez. No es necesario proporcionar todos los recursos posibles antes de tomar un proyecto público. Quizás en un mundo perfecto, todo nuevo proyecto open source empezaría su vida con un riguroso documento de diseño, un manual de usuario completo (marcando especialmente las características planeadas pero que aun no han sido implementadas), código empaquetado hermosamente y portable, capaz de ejecutar en cualquier plataforma y así sucesivamente. En realidad, cuidar de todos estos detalles consumiría demasiado tiempo, y de todas maneras, es trabajo con el que podrían ayudar voluntarios una vez que el proyecto esté en marcha.

Por otro lado, lo que *sí* es necesario, es que se realice una inversión apropiada en la presentación, de forma que los recién llegados puedan superar el obstáculo inicial de no estar familiarizados con el proyecto. Pensemos en ello como en el primer paso en un proceso de inicio (bootstrapping), llevar al proyecto a un tipo de activación de energía mínima. He escuchado llamar a este umbral como *hacktivation energy*: la cantidad de energía que debe aportar un recién llegado antes de recibir algo a cambio. Mientras menor sea ésta energía, mejor. La primera tarea es hacer descender ésta *hacktivation energy* a niveles que animen a la gente a involucrarse.

Cada una de las siguientes secciones, describen un aspecto importante de iniciar un nuevo proyecto. Están presentadas casi en el mismo orden en el que un nuevo visitante las encontraría, aunque claro, el orden en el cual sean implementadas puede ser diferente. Incluso pueden ser tratadas como una lista de tareas. Cuando se inicie un proyecto, asegúrese de revisar la lista y de que cada uno de los elementos sean cubiertos, o al menos asegurar cierta comodidad con las posibles consecuencias de dejar alguna aparte.

Escoger un buen nombre

Coloque se en la posición de alguien que acaba de escuchar acerca de su proyecto, quizás por alguien quien fortuitamente tropezó con éste mientras buscaba por alguna aplicación para resolver un problema. Lo primero que encontraran será el nombre del proyecto.

Un nombre genial no hará que automáticamente el proyecto tenga éxito, y un nombre malo no significa que éste acabado—bueno, en realidad un mal nombre probablemente podría hacer eso, pero empecemos asumiendo que nadie está activamente intentando hacer que su proyecto falle. De todos modos, un mal nombre puede desacelerar la adopción del programa porque la gente no se lo tome seriamente o porque simplemente les cueste recordarlos.

Un buen nombre:

- Da cierta idea de lo que el proyecto hace, o al menos está relacionado de una manera obvia, como si alguien conoce el nombre y sabe lo que hace, después lo recordaran rápidamente.

- Es fácil de recordar. Veamos, no hay nada de falso en el hecho de que el ingles se a convertido en el lenguaje por defecto de Internet: "fácil de recordar" significa "fácil para alguien que sepa leer en ingles de recordar." Nombres que son calambures dependientes en la pronunciación de ingleses nativos, por ejemplo, serán opacos para muchos lectores no nativos en ingles. Si el calambur es particularmente llamativo y memorable, quizás sí valga la pena. Sólo recuerde que muchas personas al ver el nombre no lo escucharán en sus mentes de la misma manera que un ingles nativo lo haría.

- No tiene el mismo nombre que otro proyecto y no infringe ninguna marca comercial. Esto es por buenos modales, y tener un buen sentido legal. No desea crear confusiones de identidad. Ya es bastante difícil mantenerse al día con todo lo que hay disponible en la red, sin tener diferentes cosas con el mismo nombre.

  Los enlaces mencionados anteriormente en “Primero investiga” son muy útiles en descubrir si algún otro proyecto ya tiene el mismo nombre en el que estábamos pensando. Podemos encontrar buscadores gratuitos de marcas registradas en http://www.nameprotect.org/ y http://www.uspto.gov/.

- Está disponible como un nombre de dominio `.com`, `.net`, y `.org`. Hay que escoger alguno, probablemente `.org`, para promocionarse como el sitio oficial para el proyecto. Los otros dos deben reenviar allí simplemente para evitar que terceras partes creen una confusión de identidad sobre el nombre del proyecto. Incluso si piensa en hospedar el proyecto en otro sitio (vea “Hosting enlatado”) puede registrar los dominios específicos del proyecto y direccionarlos al sitio del hospedaje. Ayuda mucho a los usuarios tener que recordar sólo un URL.

Tener los objetivos claros

Una vez que se ha encontrado el sitio del proyecto, lo siguiente que la gente hace es buscar por una descripción rápida, una declaración de objetivos, para poder decidir (en menos de 30 segundos) si están o no interesados en aprender más. Esto debe estar en un lugar prioritario en la página principal, preferiblemente justo debajo del nombre del proyecto.

La declaración de los objetivos debe ser concreta, limitada y sobre todo, corta. Aquí tenemos un buen ejemplo, de http://www.openoffice.org/:

> *Crear, como una comunidad, una suite ofimática líder a nivel internacional, que funcione en las mayores plataformas y proporcionar acceso a toda la funcionalidad y datos a través de API's basadas en componentes abiertos y un formato de ficheros basado en XML.*

En pocas palabras, han logrado la máxima puntuación, sobretodo al basarse en los conocimientos previos de los lectores. Al decir "*como una comunidad*", señalan que ninguna corporación dominará el desarrollo. "*Internacional*" significa que la aplicación permitirá a personas con múltiples lenguas y localidades trabajar. "*En las mayores plataformas* significa que será portable a Unix, Macintosh y Windows. El resto señala que las interfaces abiertas y formatos de ficheros fáciles de comprender son una parte importante de sus objetivos. De buenas a primeras, no intentan declarar ser una alternativa libre a Microsoft Office, aunque seguramente la mayoría puede leer entre lineas. Aunque ésta declaración de objetivos pueda parecer demasiado amplia a primera vista, el hecho es que está bien circunscrita: las palabras "*suite ofimática* " significan algo muy concreto para aquellos familiarizados con este tipo de programas. Otra vez, el asumir sobre los conocimientos previos del lector (en este caso probablemente de MS Office) permite mantener la declaración concisa.

El ámbito de una declaración de objetivos depende en gran parte de quien la escriba, no sólo del programa que intenta describir. Por ejemplo, tiene sentido para OpenOffice.org utilizar las palabras "*como una comunidad*", porque el proyecto fue iniciado, y sigue estando patrocinado, por Sun Microsystems. Al incluir esas palabras, Sun esta indicado sensibilidad a preocupaciones de que intente dominar el proceso de desarrollo. Con este tipo de cosas, simplemente demostrar un conocimiento ambiguo del *potencial* de un problema ayuda enormemente a evitar el problema completamente. Por otra parte, aquellos proyectos

que no son patrocinados por una sola corporación probablemente no tengan que utilizar este lenguaje, después de todo, el desarrollo comunitario es la norma, así que normalmente no debería haber ninguna razón para señalar esto como una parte de los objetivos.

Declara que el proyecto es libre

Aquellos que sigan interesados después de leer la declaración de objetivos querrán más detalles, quizás un poco de documentación para usuarios o desarrolladores, y eventualmente querrán descargar algo. Pero antes que nada, necesitaran estar seguros de que es open source.

La página principal debe poner claramente y sin ambigüedades que el proyecto es open source. Esto puede parecer obvio, pero es sorprendente cuantos proyectos se olvidan de esto. He visto sitios de proyectos de software libre donde la página principal no sólo no decía bajo cual licencia libre se distribuía la aplicación sino que ni siquiera declaraban que el software fuese libre. A veces, estas piezas cruciales de información eran relegadas a la página de descargas o a la página de los desarrolladores o a algún otro lugar el cual requería más de un enlace para llegar. En casos extremos, la licencia no se mostraba en ninguna parte del sitio—la única forma de encontrarla era descargando la aplicación e investigando.

No cometáis estos errores. Una omisión como ésta puede haceros perder muchos desarrolladores y usuarios potenciales. Declarad desde el principio, justo debajo de la declaración de objetivos, que el proyecto es "software libre" u "open source", y mostrad la licencia exacta. Una guía rápida para escoger una licencia se encuentra en “Escogiendo una licencia y aplicándola” más adelante en éste capítulo, y algunos detalles sobre las licencias serán discutidos en el Capítulo 9, *Licencias, Copyrights y Patentes*.

Llegados a este punto, nuestro visitante hipotético ha determinado— probablemente en un minuto o menos—que está interesado en utilizar, digamos, al menos cinco minutos más investigando el proyecto. La próxima parte describe qué debería encontrar durante esos cinco minutos.

Lista de características y requerimientos

Debería haber una breve lista de las características que el software soporta (si algo aun no ha sido completado, se puede listar de todas formas, pero señalando "*planeado*" o "*en progreso*") y el tipo de entorno necesario para ejecutar la aplicación. Hay que pensar en ésta lista como algo que daríamos a alguien que requiere un resumen de nuestro programa. Por ejemplo, la declaración de objetivos podría decir:

> *Crear un controlador y sistema de búsqueda con una API, para ser utilizada por programadores suministrando servicios de búsqueda para grandes colecciones de ficheros de texto.*

La lista de características y requerimientos daría detalles que permitirían esclarecer el alcance de la declaración de objetivos:

> *Características*
>
> - *Búsquedas en texto plano, HTML y XML*
> - *Búsqueda de palabras o frases*
> - *(planeado) Emparejando borroso (Fuzzy Matching)*
> - *(planeado) Actualización incremental de índices*
> - *(planeado) Indexado de sitios web remotos*
>
> *Requerimientos:*

- *Python 2.2 o mayor*
- *Espacio en disco suficiente para contener los índices (aproximadamente 2x el tamaño original de los datos)*

Con ésta información, los lectores podrán rápidamente tener una idea de si éste programa tiene alguna esperanza de trabajar para ellos, y también pueden considerar involucrarse como desarrolladores.

Estado del desarrollo

La gente siempre quiere saber cómo va un proyecto. Para proyectos nuevos, desean saber la separación entre las promesas del proyecto y la realidad del momento. Para proyectos maduros, desean saber cuan activamente es mantenido, cuan seguido sacan nuevas versiones, la facilidad para reportar fallos, etc.

Para responder a estas dudas, se debe suministrar una página que muestre el estado del desarrollo, listando los objetivos a corto plazo del proyecto y las necesidades (por ejemplo, quizás se estén buscando desarrolladores con un expertos en un tema en particular). Ésta página también puede dar una historia de versiones anteriores, con listas de las características, de manera que los visitantes obtengan una idea de cómo el proyecto define su "progreso" y de cuan rápidamente se hacen progresos de acuerdo a esas definiciones.

No hay que asustarse por parecer no estar preparado y no caer en la tentación de inflar el estado del desarrollo. Todos saben que el software evoluciona por etapas; no hay que avergonzarse en decir "Esto es software alfa con fallos conocidos. Ejecuta, y funciona algunas veces, así que uselo bajo su responsabilidad." Este lenguaje no asustará el tipo de desarrolladores que son necesarios en esta etapa. En cuanto a los usuarios, una de las peores cosas que un proyecto puede hacer es atraer usuarios antes de que el software éste listo para estos. Una reputación por inestabilidad y fallos es muy difícil de hacer desaparecer una vez adquirida. La paciencia da sus frutos a largo plazo; siempre es mejor que el software sea *más* estable de lo que espera el usuario ya que las sorpresas gratas producen el mejor boca a boca.

Alfa y Beta

El término *alfa* usualmente significa, la primera versión, con lo que los usuarios pueden realizar todos el trabajo teniendo todas la funcionalidad esperada, pero que se sabe tiene fallos. El principal propósito de el software alfa es generar una respuesta, de forma que los desarrolladores sepan en qué trabajar. La próxima etapa, *beta*, significa que han sido resueltos todos los fallos más importantes, pero que aun no ha sido intensivamente probado como para ser la versión oficial. El propósito de las betas es la de convertirse en la versión oficial, asumiendo que nuevos fallos no sean encontrados, o de suministrar un feedback para los desarrolladores para que logren la versión oficial más rápido. La diferencia entre alfa y beta es más una cuestión de juicio.

Descargas

EL software debe poder ser descargable como código fuente en formatos estándares, paquetes binarios (ejecutables) no son necesarios, a menos que el programa tenga requerimientos muy complicados para su compilado o dependencias que hagan hacerlo funcionar sea muy laborioso para la mayoría de las personas. (¡Aunque si es éste es el caso, el proyecto va a tenerlo muy difícil atrayendo programadores de todas maneras!)

El mecanismo de distribución debe de ser de lo más conveniente, estándar y sencillo posible. Si se estuviese intentando erradicar una enfermedad, no distribuiría la medicina tal que requiriese de una jeringuilla especial para administrarse. De igual manera, un programa debe ser conforme a métodos de compilación e instalación estándar; entre más se desvíe de estos estándares, mayor será la cantidad de usuarios y

desarrolladores potenciales que se den por vencidos y abandonen el proyecto confundidos.

Esto parece obvio, pero muchos proyectos no se molestan en estandarizar sus procedimientos de instalación hasta mucho después, diciéndose a si mismos que esto lo pueden hacer en cualquier momento: *"Ya resolveremos todas esas cosas cuando el código éste casi listo."* De lo que no se dan cuenta es de que al dejar de lado el trabajo aburrido de terminar los procedimientos de compilado e instalación, en realidad están ralentizando todo—porque desalientan a los programadores que de otra manera habrían contribuido al código. Más dañino aun, no *saben* que están perdiendo a todos esos desarrolladores, porque el proceso es una acumulación de eventos que no suceden: alguien visita un sitios web, descarga el programa, intenta compilarlo, falla, deja de intentarlo y abandona. ¿Quién sabrá que ocurrió exceptuando a ésta persona? Nadie en el proyecto se dará cuenta que el interés y la buena voluntad de alguien a sido silenciosamente malgastada.

Las tareas aburridas con un alto beneficio siempre deben ser hechos al principio y disminuyendo de manera significativa las barreras de entrada a un proyecto utilizando buenos paquetes brindan altos beneficios.

Cuando se lanza un paquete descargable, es vital que se le dé un número de versión único a éste lanzamiento, de manera que la gente pueda comparar dos versiones cualquiera diferentes y saber cual reemplaza a cual. Una discusión detallada sobre la numeración de versiones puede ser encontrada en “Release Numbering”, y detalles sobre la estandarización de los procedimientos de compilado e instalación serán cubiertos en “Packaging”, ambos en el Capítulo 7, *Packaging, Releasing, and Daily Development*.

Control de versiones y acceso al Bug Tracker

Descargar paquetes con el código fuente está bien para aquellos que sólo desean instalar y utilizar un programa, pero no es suficiente para aquellos que desean buscar fallos o añadir nuevas mejoras. Instantáneas nocturnas del código fuente pueden ayudar, pero esto no es suficiente para una prospera comunidad de desarrollo. Estas personas necesitan de acceso en tiempo real a los últimos cambios, y la manera de proporcionarles esto es utilizando un sistema de control de versiones (version control system). La presencia de fuentes controladas, accesibles anónimamente es una señal de—para ambos, usuarios y programadores—que éste proyecto ésta haciendo un esfuerzo en proporcionar todo lo necesario para que otros participen. Si no se puede ofrecer control de versiones desde el principio, comunique la intención de montarlo pronto. La infraestructura de control de versiones es discutida en detalle en “Control de Versiones” en el Capítulo 3, *Infraestructura Técnica* .

Lo mismo se aplica para el seguimiento de errores del proyecto. La mayor importancia que se le dé a ésta base de datos, lo mejor que parecerá el proyecto. Esto puede parecer contra intuitivo, pero hay que recordar que el número de fallos registrados, en realidad depende en tres cosas: el número absoluto de errores presentes en el programa, el número de usuarios utilizándolo y la conveniencia con la cual esos usuarios registran nuevos fallos. De estos tres factores, los dos últimos son más significativos que el primero. Cualquier aplicación con suficiente tamaño y complejidad tiene una cantidad arbitraria de fallos esperando a ser descubiertos. La verdadera cuestión es, cuan bien serán registrados y priorizados estos errores. Un proyecto con una base de datos de fallos amplia y bien mantenida (errores importantes son atacados rápidamente, fallos duplicados son unificados, etc.) generan una mejor impresión que un proyecto sin una o vacía.

Claro está, que si un proyecto está empezando, que la base de datos de fallos contenga algunos pocos, y no hay mucho que se pueda hacer al respecto. Pero si la página donde se indica el estado del proyecto, enfatiza en la juventud del proyecto y si las personas mirando los fallos pueden observar que muchos de estos han sido incluidos recientemente, pueden asumir que el proyecto tiene una *proporción* saludable de entradas y no serán alarmados por el mínimo absoluto de fallos registrados.

Hay que señalar que los bug trackers no sólo son usados para fallos en los programas pero también para peticiones de mejoras, cambios en la documentación, tareas pendientes y mucho más. Los detalles de ejecutar un sistema de seguimiento de fallos será cubierto en “Seguimiento de errores” en el Capítulo 3,

Infraestructura Técnica, así que no vamos a entrar en detalles. Lo importante desde la perspectiva de la presentación está en *tener* un bug tracker y asegurarse de que es visible desde la página principal del proyecto.

Canales de comunicación

Usualmente los visitantes desean saber cómo pueden contactar con los seres humanos detrás del proyecto. Hay que suministrar direcciones de listas de correo, salas de chat, canales en IRC y cualquier otro foro donde aquellos involucrados puedan ser contactados. Hay que dejar claro que los autores del proyecto están suscritos a estas listas, de manera que la gente vea una forma de dar feedback a los desarrolladores. La presencia de estos en las listas no implica obligación alguna de responder a todas las preguntas que se formulan o de implementar todas las peticiones. A la larga, muchos de los usuarios probablemente ni siquiera se unan a los foros de todas maneras, pero estarán conformes con saber que *podrían* si fuese necesario.

En la primeras etapas de cualquier proyecto, no existe la necesidad de que haya una diferenciación entre los foros de los usuarios y los de los desarrolladores. Es mejor tener a todos los involucrados en el proyecto hablando en conjunto en una sala. Dentro de los primeros en adoptar el proyecto, la distinción entre usuario y desarrollador será muchas veces borrosa, hasta tal punto que la distinción no se puede hacer y la proporción entre programadores y usuarios usualmente es mayor al principio que al final. Mientras que no se puede asumir que todos quienes utilicen el programa sean programadores que quieren modificarlo, sí se puede asumir que al menos estan interesados en seguir las discusiones sobre el desarrollo y en obtener una visión de la dirección del proyecto.

Ya que éste capítulo es sólo sobre iniciar un proyecto, es suficiente decir que al menos estos foros de comunicación deben existir. Luego en “Manejando el crecimiento” en el Capítulo 6, *Communications*, examinaremos dónde y cómo montar estos foros, cómo deben ser moderados o cualquier otro tipo de dirección y cómo separar los foros de usuarios de los foros de los desarrolladores, cuando llegue el momento, sin crear un espacio infranqueable.

Pautas de Desarrollo

Si alguien considera contribuir al proyecto, buscará por pautas de desarrollo. Estas pautas son más sociales que técnicas: explican como los desarrolladores interactúan entre ellos y con los usuarios y últimamente como hacer las cosas.

Este tema es tratado en detalle en “Tomando Nota de Todo” en Capítulo 4, *Infraestructura Social y Política*, pero los elementos básicos de unas pautas de desarrollo son:

- enlaces a los foros para la interacción de los desarrolladores
- instrucciones en cómo reportar fallos y enviar parches
- alguna indicación de *cómo* el desarrollo es usualmente llevado a cabo—es el proyecto una dictadura benevolente, una democracia o algo más

Ningún sentido peyorativo es intencional por lo de "dictadura" por cierto. Es perfectamente aceptable ser un tirano donde un desarrollador en particular tiene el poder de veto sobre todos los cambios. Muchos proyectos exitosos funcionan de ésta manera. Lo importante es que el proyecto sea consciente de esto y lo comunique. Una tiranía pretendiendo ser una democracia desalentara a las personas; una tiranía que dice serlo funcionará bien siempre que el tirano sea competente y de confianza.

Un ejemplo de unas pautas de desarrollos particularmente exhaustivas están en http://svn.collab.net/repos/svn/trunk/www/hacking.html o en http://www.openoffice.org/dev_docs/guidelines.html tenemos unas pautas más amplias que se concentran más en la forma de gobierno y el espíritu de participación y menos en temas técnicos.

Proveer una introducción a la aplicación para los programadores es otro tema y será discutido en “Documentación para Desarrolladores” más adelante en éste capítulo .

Documentación

La documentación es esencial. Debe haber *algo* para que la gente lea, aunque sea algo rudimentario e incompleto. Esto entra de lleno en la categoría antes referida y usualmente es la primera área donde un proyecto falla. Conseguir una declaración de objetivos y una lista de requerimientos, escoger una licencia, resumir el estado de desarrollo—son todas tareas relativamente pequeñas que pueden ser completadas y a las que usualmente no es necesario volver una vez terminadas. La documentación, por otra parte, nunca está terminada realmente, lo cual puede que sea una de las razones por las cuales se retrase su inicio.

La cuestión más insidiosa sobre la utilidad de la documentación es que es inversamente proporcional para quienes la escriben y para quienes la leen. Lo más importante de la documentación para un usuario inicial es lo más básico: cómo configurar la aplicación, una introducción de cómo funciona y quizás algunas guías para realizar las tareas más comunes. Pero a la vez son estas cosas las más sabidas por aquellos quienes *escriben* la documentación— tan bien sabidas que puede ser difícil para estos ver las cosas desde el punto de vista de los lectores, dificultando listar los pasos que (para los escritores) parecen tan obvios que no merecen especial atención.

No existe una solución mágica para éste problema. Alguien debe sentarse y escribir todo esto para luego presentárselo a un usuario nuevo tipo y probar la calidad. Hay que utilizar un formato simple y fácil de modificar como HTML, texto plano, Tex o alguna variante de XML—algo que sea conveniente para mejoras rápidas, ligeras e imprevisibles. Esto no es sólo para eliminar cualquier trabajo innecesario a los escritores originales realizar cambios incrementales, sino que también para quienes se unan al proyecto después y desean trabajar en la documentación.

Una manera de asegurarse de que la documentación básica inicial se hace, es limitando su alcance. Al menos de ésta manera no parecerá que se está escribiendo una tarea sin fin. Una buena regla es seguir unos criterios mínimos:

- Avisar al lector claramente el nivel técnico que se espera que tenga.
- Describir clara y extensivamente cómo configurar el programa y en alguna parte al inicio de la documentación comunicarle al usuario cómo ejecutar algún tipo de prueba de diagnóstico o un simple comando para confirmar que todo funciona correctamente. La documentación inicial es a veces más importante que la documentación de uso. Mientras mayor sea el esfuerzo invertido en instalar y tener funcionando la aplicación, mayor será la persistencia en descubrir funcionalidades avanzadas o no documentadas. Cuando alguien abandona, abandonan al principio; por ello, las primeras etapas como la instalación, necesiten la mayor ayuda.
- Dar un ejemplo estilo tutorial de como realizar alguna tarea común. Obviamente, muchos ejemplos para muchas tareas sería mejor, pero si el tiempo es limitado, es mejor escoger una tarea en específico y llevar al usuario de la mano paso por paso. Una vez que se ve que la aplicación *puede* ser utilizada , empezarán a explorar qué más es lo que puede hacer—y si se tiene suerte empezar a documentarlo ellos mismos. Lo que nos lleva al siguiente punto...
- Indicar las áreas donde se sabe que la documentación es incompleta. Al mostrar a los lectores que se es consciente de las deficiencias, nos alineamos con su punto de vista. La empatía les da confianza en que no van a tener que luchar para convencer al proyecto de su importancia. Estas indicaciones no necesitan representar promesa alguna de completar los espacios en blanco en una fecha en particular—es igualmente legitimo tratarlas como requisitos abiertos para ayudantes voluntarios.

Ese último criterio es de una especial importancia, y puede ser aplicado al proyecto entero, no sólo a la documentación. Una gestión exacta de las deficiencias conocidas es la norma en el mundo Open Source.

No se debe exagerar en las faltas del proyecto, solo identificarlas escrupulosa y desapasionadamente cuando sea necesario (sea en la documentación, en la base de datos de fallos o en discusiones en la lista de correos). Nadie verá esto como derrotismo por parte del proyecto, ni como una responsabilidad explícita. Ya que cualquiera que utilice la aplicación descubrirá sus deficiencias por si mismos, es mejor que estén psicológicamente preparados—entonces parece que el proyecto tiene un sólido conocimiento acerca de como va progresando.

Manteniendo un FAQ (Preguntas Más Frecuentes)

Un *FAQ* (del ingles "Frequently Asked Questions") puede ser uno de las mejores inversiones que un proyecto puede hacer en términos de beneficios educativos. Los FAQs están enfocados a las preguntas que desarrolladores y usuarios podrían formular—opuesto a aquellas que *se espera* que hagan—por lo cual un FAQ bien cuidado tiende a dar a aquellos quienes lo consultan exactamente lo que están buscando. Por lo general es el primer lugar en el que se busca cuando se encuentran con un problema, incluso con preferencia sobre el manual oficial y es probablemente el documento más propenso a ser enlazado desde otros sitios.

Desafortunadamente, no se puede hacer un FAQ al principio del proyecto. Los buenos FAQs no son escritos, crecen. Son por definición documentos reactivos, evolucionando con el tiempo como respuesta al uso diario del programa. Ya que es imposible anticipar correctamente las preguntas que se podrían formular, es imposible sentarse a escribir un FAQ útil desde cero.

Así que no hay que malgastar el tiempo en intentarlo. En cambio, podría ser útil crear una plantilla casi en blanco del FAQ, de forma que haya un lugar obvio donde las personas contribuyan con preguntas y respuestas después de que el proyecto esté en progreso. En ésta etapa lo más importante no es tenerlo todo completo, sino la conveniencia: si es sencillo agregar contenido al FAQ, la gente lo hará. (Un mantenimiento correcto de un FAQ es un problema no trivial e intrigante y es discutido más a fondo en “FAQ Manager” en el Capítulo 8, *Coordinando a los Voluntarios*.)

Disponibilidad de la documentación

La documentación debe ser accesible desde dos sitios: en línea (directamente desde el sitio web), *y* en la distribución descargable de la aplicación (consultar “Packaging” en el Capítulo 7, *Packaging, Releasing, and Daily Development*). Debe estar en línea y navegable porque a menudo se lee la documentación *antes* de descargar el programa por primera vez, como una ayuda en la decisión de descargarlo o no. Pero también debe acompañar al programa, bajo la premisa de que la descarga debe suministrar todo lo necesario para utilizar el paquete.

Para la documentación en línea, hay que asegurarse de que hay un enlace que muestra *toda* la documentación en una página HTML (indicando algo como "monolito" o "todo-en-uno" o "sólo un gran fichero" al lado del enlace, de tal manera que se sepa que puede tardar un poco en cargar). Esto es muy útil porque a veces sólo desean buscar una sola palabra o frase en la documentación. Generalmente, las personas ya saben qué es lo que están buscando, sólo que no recuerdan en cual sección está. Para estas personas, nada es más frustrante que encontrar una página para la tabla de contenidos, luego otra diferente para la introducción, luego otra diferente para las instrucciones de instalación, etc. Cuando las páginas están divididas de esta manera, la función de búsqueda de sus navegadores es inútil. Este estilo de páginas separadas es útil para quienes ya saben cual es la sección que necesitan, o que desean leer toda la documentación de principio a fin en secuencia. Pero esta *no es* la forma más común en que la documentación es leída. Ocurre más a menudo que alguien que conoce algo básico de la aplicación vuelve para buscar una palabra o frase. Fallar al suministrarles un sólo documento en el que se puedan realizar búsquedas, es hacerles la vida más dura

Documentación para Desarrolladores

La documentación para los desarrolladores es escrita para ayudar a los programadores a entender el có-

digo y puedan arreglarlo o extenderlo. Esto es algo diferente a las *pautas de desarrollo* discutidas anteriormente, que son más sociales que técnicas. Estas pautas para los desarrolladores le dicen a los programadores como deben desenvolverse entre ellos. La documentación les dice como deben desenvolverse con el código en si mismo. Por conveniencia las dos vienen juntas en un sólo documento (como sucede con el ejemplo anterior http://svn.collab.net/repos/svn/trunk/www/hacking.html) pero no es obligatorio.

A pesar de que la documentación para los desarrolladores puede ser de mucha ayuda, no existe ninguna razón para retrasar un lanzamiento por hacerla. Es suficiente para empezar que los autores originales estén disponibles (y dispuestos) a responder a preguntas sobre el código. De hecho, tener que responder la misma pregunta varias veces es una motivación muy común para escribir dicha documentación. Pero antes de que sea escrita, determinados contribuyentes serán capaces de desenvolverse con el código ya que la fuerza que hace que las persones utilicen su tiempo en leer el código base es que éste código les resulta útil. Si las personas tienen fé en ello, ninguna cantidad de documentación hará que vengan o los mantendrá.

Así que si hay tiempo para escribir documentación sólo para una audiencia, que sea para los usuarios. Toda la documentación para los usuarios es, en efecto, documentación para desarrolladores también. Cualquier programador que vaya a trabajar en un proyecto necesita estar familiarizado con su uso. Luego, cuando se vea a los programadores preguntando las mismas preguntas una y otra vez, habrá que tomarse el tiempo de escribir algunos documentos aparte sólo para estos.

Algunos proyectos utilizan wikis para su documentación inicial o incluso para su documentación principal. En mi experiencia, esto es efectivo si y sólo si, el wiki es editado activamente por algunas personas que se ponen de acuerdo en como la documentación debe ser organizada y la voz que debe tener. Más en “Wikis” en el Capítulo 3, *Infraestructura Técnica*.

Ejemplos de salidas y capturas

Si el proyecto implica una interfaz gráfica para el usuario o si produce una salida gráfica o distintiva, habrá que poner algunos ejemplos en el sitio web del proyecto. En el caso de las interfaces, esto significa capturas. Para salidas, pueden ser capturas o sólo ficheros. Ambos dotan al usuario de gratificación instantánea: una sola captura puede ser más convincente que párrafos de texto descriptivo y cháchara de listas de correo, porque una captura es la prueba indiscutible de que el programa *funciona*. Puede que tenga fallos, quizás sea difícil de instalar o que la documentación esté incompleta, pero esa captura sigue siendo la prueba de que con el esfuerzo necesario, se puede hacer funcionar.

Capturas

Ya que hacer las capturas puede ser algo desalentador, aquí tenéis unas instrucciones básicas sobre como hacerlas. Utilizando The Gimp (http://www.gimp.org/), pinchad en Archivo->Adquirir->Captura de pantalla, escoged Capturar una sola ventana o Toda la pantalla, luego pinchad en Capturar. La próxima vez que pinche la ventana o la pantalla será capturada como una imagen en The Gimp Recortad y cambiad el tamaño de la imagen según sea necesario siguiendo las instrucciones en http://www.gimp.org/tutorials/Lite_Quickies/#crop.

Existen muchas otras cosas que se pueden poner en el sitio web del proyecto, si se tiene el tiempo, o si por alguna razón u otra son especialmente apropiadas: página de noticias, historia, enlaces relacionados, función de búsqueda, enlace para donaciones, etc. Ninguno de estos es necesarios al principio, pero hay que tenerlos en mente para el futuro.

Hosting enlatado

Existen algunos sitios que proveen hosting gratuito e infraestructura para proyectos open source: un área web, control de versiones, gestor de errores, zona de descargas, salas de chat, backups regulares, etc. Los detalles varían entre sitio y sitio, pero los servicios básicos son ofrecidos por todos. Al utilizar uno

de estos sitios, se obtiene mucho por nada, dando a cambio, obviamente, el control sobre la experiencia del usuario. Quien provee el hosting decide cuales programas el sitio acepta y puede controlar o al menos influenciar el aspecto de las páginas del proyecto.

Vaya a “Soluciones de hospedaje” en el Capítulo 3, *Infraestructura Técnica* para una discusión más detalladas acerca de las ventajas y desventajas del hosting enlatado y una lista de sitios que lo ofrecen.

Escogiendo una licencia y aplicándola

Esta sección está concebida para ser una guía rápida y amplia sobre como escoger una licencia. Leed el Capítulo 9, *Licencias, Copyrights y Patentes* para entender en detalle las implicaciones legales de diferentes licencias y como la licencia escogida puede afectar la capacidad de otras personas de mezclar el programa con otros.

Existen muchas licencias libres de donde escoger. Muchas de ellas no necesitamos tenerlas en consideración aquí, ya que han sido escritas para satisfacer las necesidades legales específicas de alguna corporación o persona, así que no serian apropiadas para nuestro proyecto. Por ello nos vamos a restringir a las más usadas. En la mayoría de los casos, querrás escoger una de ellas.

Las licencias "Haz lo que quieras"

Si se está conforme con que el código del proyecto sea potencialmente usado en programas propietarios, entonces se puede utilizar una licencia estilo *MIT/X*. Es la más sencilla de muchas licencias mínimas que no hacen más que declarar un copyright nominal (sin restringir la copia) y especificar que el código viene sin ninguna garantía. Id a “La MIT / X Window System License” para más detalles.

Licencia GPL

Si no desea que el código sea utilizado en aplicaciones propietarias utilice la Licencia Pública General o GPL (del ingles General Public License) (http://www.gnu.org/licenses/gpl.html). La GPL es probablemente la licencia para software libre más utilizada a nivel mundial hoy en día. Esto es en si mismo una gran ventaja, ya que muchos usuarios potenciales y voluntarios ya estarán familiarizados con ella, por lo cual, no tendrán que invertir tiempo extra en leer y entender la licencia utilizada. Más detalles en “La GNU General Public License” en el Capítulo 9, *Licencias, Copyrights y Patentes*.

Cómo aplicar una licencia a nuestro software

Una vez que ha sido escogida una licencia, se debe exponer en la página principal del proyecto. No se tiene que incluir el texto de la licencia aquí, sólo hay que dar el nombre de la licencia y un enlace al texto completo de ésta en otra página.

Esto informa al público bajo cual licencia se *pretende* publicar la aplicación. Para ello, el programa en si debe incluir la licencia. La manera estándar de hacer esto es poniendo el texto completo en un fichero llamado `COPYING` (o `LICENSE`) y luego colocar un aviso al principio de cada fichero con el código fuente, listando la fecha del copyright, titular y licencia y explicando donde encontrar el texto completo de la misma.

Hay muchas variaciones de éste patrón, así que miraremos a un sólo ejemplo. La GPL de GNU indica que se debe colocar un aviso como éste al principio de cada fichero con código fuente:

```
Copyright (C) <year>  <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
```

No específica que una copia de la licencia adjuntada al programa está en el fichero `COPYING`, pero comúnmente es en éste donde se pone (Se puede cambiar lo anterior para indicar esto directamente). Esta plantilla también nos da una dirección física a donde solicitar una copia de la licencia. Otro método muy común es suministrar un enlace a una página web que contiene la licencia. Sólo hay que utilizar el sentido común y señalar algún sitio donde se crea habrá una copia permanente de la licencia. Por lo general, el aviso que se coloca al principio de cada fichero con código fuente no debe ser exacto al anteriormente expuesto, siempre y cuando se empiece con el mismo aviso de copyright, titular y fecha, se especifique el nombre de la licencia y se deje claro donde encontrar la licencia completa.

Ajustar el tono

Hasta ahora hemos cubierto tareas que se hacen sólo una vez durante el proyecto: escoger la licencia, acomodar el sitio web inicial, etc. Pero los aspectos más importantes al empezar un nuevo proyecto son dinámicos. Escoger la dirección para la lista de correos es fácil; asegurarse de que las conversaciones en ésta se mantengan en contexto y sean productivas es otro tema. Si el proyecto es abierto después de años de desarrollo cerrado propio, sus procesos de desarrollo cambiaran y habrá que preparar a los desarrolladores existentes para éste cambio.

Los primeros pasos son los más duros, porque los precedentes y las expectaciones sobre la conducta futura aun no se han definido. La estabilidad de un proyecto no viene de políticas formales, sino de un conocimiento colectivo compartido muy difícil de definir y que se desarrolla con el tiempo. A veces existen unas reglas escritas, pero tienden a ser un destilado de los acuerdos intangibles y siempre cambiantes que realmente guían el proyecto. Las políticas escritas no definen la cultura del proyecto mas que describirla, he incluso así, sólo se aproximan.

Hay algunas razones por las cuales las cosas funcionan de ésta manera. El crecimiento y los grandes cambios no son tan dañinos para la acumulación de las normas sociales como se puede pensar. Mientras que el cambio no ocurra *demasiado* rápido, hay tiempo para que los novatos aprendan como funcionan las cosas y después de que aprendan, ellos mismos ayudaran a reforzar este funcionamiento. Consideremos cómo las canciones infantiles sobreviven a lo largo de los siglos. Hay niños hoy en día cantando casi las mismas rimas que los niños de hace cien años, aunque no haya ninguno vivo hoy en día que haya vivido entonces. Los más pequeños escuchan estas canciones de otros niños mayores y cuando son mayores, las cantarán frente a otros niños menores que ellos. Conscientemente los niños no están iniciando un programa de transmisión, por supuesto, pero la razón por la cual las canciones sobreviven es nada más y nada menos porque son transmitidas regular y repetidamente. La escala de tiempo de un proyecto de software libre quizás no sea medido en siglos (aún no lo sabemos) pero las formas de transmisión son las mismas. Aunque el índice de cambios es más rápido y debe ser compensado con un esfuerzo deliberado de comunicación más activo.

A este esfuerzo le ayuda el hecho de que las personas por lo general se presentan esperando y buscando normas sociales. Así es como los humanos estamos construidos. En cualquier grupo unido por un mismo objetivo, las personas que se unen, instintivamente buscan conductas las cuales los marcarán como parte del grupo. El objetivo temprano de sentar precedentes es hacer de esas conductas de grupo útiles para el proyecto; una vez establecidas serán perpetuas por si mismas.

A continuación hay algunos ejemplos específicos de lo que se puede hacer para establecer buenos prece-

dentes. No se supone que sea una lista exhaustiva, mas es una ilustración de la idea de que establecer un ambiente de colaboración desde el principio ayuda enormemente al proyecto. Físicamente, cada desarrollador puede que trabaje en solitario, pero se puede hacer mucho para hacerlo *sentir* como si todos estuviesen trabajando juntos en la misma habitación. Mientras mayor sea ésta sensación mayor será el tiempo que quieran invertir en el proyecto. He escogido estos ejemplos en particular porque han surgido en el proyecto de Subversion (http://subversion.tigris.org/), en el cual participé y observé desde sus inicios. Pero estas no son únicas a Subversion, situaciones como estas surgen en casi todos los proyectos open source, y deben ser tomadas como oportunidades para empezar de la manera correcta.

Evitar discusiones privadas

Incluso después de haber hecho público el proyecto, usted y los otros fundadores del proyecto se encontrarán a menudo intentado resolver preguntas difíciles vía comunicaciones privadas dentro de un circulo interno. Esto es especialmente cierto en los primeros días del proyecto, cuando hay tantas decisiones importantes que tomar y usualmente pocos voluntarios cualificados para resolverlas. Todas las obvias desventajas de una lista pública de discusión se perfilan palpablemente frente a ti: el retraso inherente en las conversaciones por correo, la necesidad de dejar que se forme un consenso, las dificultades de tratar con voluntarios crédulos que piensan que entienden todos los problemas pero que no es así (todo proyecto tiene de estos; a veces son el voluntario estrella del próximo año, a veces permanecen ingenuas durante el resto del proyecto), la persona que no puede entender por qué quieres resolver el problema X cuando es obviamente una parte del más grande problema Z y muchos otros. La tentación de tomar decisiones a puerta cerrada y presentarlas como *faits accomplis*, o al menos como firmes recomendaciones de un bloque unido e influyente serian geniales la verdad.

No lo hagas.

Por muy lentas y engorrosas que puedan ser las discusiones publicas, casi siempre son preferibles a largo plazo. Tomar decisiones importantes en privado es como esparcir repelente anti-voluntarios sobre el proyecto. Ningún voluntario serio se quedaría mucho tiempo en un ambiente donde un consejo secreto toma todas las grandes decisiones. Además, las discusiones publicas tienen efectos secundarios beneficiosos que durarán más que cualquier pregunta técnica que fuese el problema:

- La discusión ayudará a entrenar y educar a nuevos desarrolladores. Nunca se sabe cuantos ojos están viendo una conversación así; Incluso si muchas de las personas no participan, muchas podrían estar monitorizando silenciosamente, deduciendo información acerca de la aplicación.
- La discusión te entrenará en el arte de explicar temas técnicos a personas que no están tan familiarizadas con el programa. Esta es una capacidad que requiere de práctica y no se puede entrenar hablando con personas que ya saben lo mismo que tu.
- La discusión y sus conclusiones estarán disponibles en un archivo público para siempre, evitando que futuras discusiones caigan en los mismos problemas. Más en “Sobresaliente uso de los archivos” en el Capítulo 6, *Communications*.

Finalmente, existe la posibilidad de que alguien en la lista haga una contribución real a la conversación, ingeniando una idea nunca antes anticipada. Es difícil decir cuan probable es que esto suceda; depende en la complejidad del código y el nivel de especialización requerida. Pero si se me permite utilizar evidencia anecdótica, apostaría a que esto es más probable de lo que podemos esperar. En el proyecto Subversion, nosotros (los fundadores) creíamos encontrarnos ante una serie compleja y profunda de problemas, en los cuales habíamos estado pensando durante meses, y francamente, dudábamos de que alguien en la recientemente creada lista de correos fueses a dar alguna contribución útil a la discusión. Así que tomamos el camino más fácil y empezamos a lanzar ideas técnicas a diestra y siniestra en correos privados, hasta que alguien observando el proyecto [1] descubrió lo que estaba pasando y pidió que se moviera la discusión a la lista pública. Torciendo un poco los ojos, lo hicimos—y fuimos asombrados por la cantidad de comentarios inspiradores y sugerencias que rápidamente resultaron. En muchos casos ofrecien-

do ideas que no se nos habían ocurrido anteriormente. Al final resultó que había gente *muy* inteligente en esa lista, sólo estaban esperando el anzuelo apropiado. Es cierto que las discusiones tomaron más tiempo de haberlas hechas en privado, pero eran mucho más productivas, lo cual hacía que valiera la pena el tiempo extra.

Sin entrar en generalizaciones como "el grupo es siempre más listo que el individuo" (ya hemos conocido muchos grupos para saberlo) debe ser apuntado que hay ciertas actividades en las que un grupo sobresale. Las revisiones distribuidas masivas son una de estas. Generar un gran número de ideas rápidamente es otra. La calidad de las ideas depende en la calidad del pensamiento que se ha aplicado a estas, por supuesto, pero no vas a saber qué clase de pensadores hay hasta que los estimules con problemas desafiantes.

Naturalmente, hay discusiones que deben ser llevadas a cabo en privado; a lo largo de éste libro veremos algunos ejemplos. Pero el principio que debe guiar siempre es: *Si no existe razón alguna para que sea privada, debe ser pública.*

Hacer que esto suceda requiere acciones. No es suficiente con simplemente asegurarse de que todos los comentarios van a la lista pública. También hay que atenerse a las conversaciones privadas innecesarias en la lista. Si alguien intenta iniciar una conversación privada, y no existe razón alguna para que así sea, entonces es de tu incumbencia el abrir la discusión apropiada inmediatamente. Ni siquiera intentes comentar el tema original hasta que se haya direccionado exitosamente la conversación a un sitio público, o asegurado que el tema era necesariamente privado. Si se hace esto consistentemente, las personas se darán cuenta rápidamente y empezara# a utilizar los foros públicos por defecto.

Echad a volar la mala educación

Desde el primero momento de la existencia pública de un proyecto se deberá mantener una política de tolerancia cero ante la mala educación o las actitudes insultantes en los foros. Tolerancia cero no implica esfuerzos técnicos per se. No se deben eliminar personas de la lista de correos cuando ataquen a otros usuarios, o quitarles sus accesos para realizar commits porque hayan hecho comentarios peyorativos. (En teoría, habría que llegar a tomar estas acciones, pero sólo después de que todas las otras vías hayan fallado—lo cual, por definición, no significa que sea al principio del proyecto.) Tolerancia cero simplemente significa nunca permitir que este tipo de conductas pasen desapercibidas. Por ejemplo, cuando alguien envía un comentario técnico mezclado con un ataque *ad hominem* contra otros desarrolladores del proyecto, es imperativo que tu respuesta sea primero dirigida a ese ataque *ad hominem* como un tema aparte y sólo después entrar en el tema técnico.

Desafortunadamente es muy fácil y típico, que conversaciones constructivas terminen en una guerra. Las personas dirán cosas en un correo electrónico que nunca dirían cara a cara. Los temas de discusión sólo ayudan a ampliar éste efecto: en cuestiones técnicas, la gente cree a menudo que sólo existe una sola respuesta correcta para la mayoría de las preguntas y que el desacuerdo ante la respuesta sólo puede ser explicado por la ignorancia o la estupidez. Hay una corta distancia entre llamar la propuesta técnica de alguien estúpida y llamar a esa persona estúpida. De hecho, es difícil definir cuando un debate técnico lo deja de ser y se convierte en ataques personales, por lo cual una respuesta drástica y el castigo no son buenas ideas. En su lugar, cuando creas que lo estas viviendo, envía un mensaje que remarque la importancia de mantener la discusión amistosa, sin acusar a nadie de ser deliberadamente venenoso. Este tipo de "política amable" de mensajes tienen la desafortunada tendencia a parecer consejos de un profesor de kindergarten sobre la buena conducta en el aula:

> *Primero, vamos a dejar a un lado los comentarios (potenciales) ad hominem por favor; por ejemplo, decir que el diseño para la capa de seguridad de J es "simple e ignorante de los principios de la seguridad informática." Quizás sea cierto o no, pero en cualquier caso no es la manera de mantener una discusión. J hizo su propuesta de buena fe y estoy seguro de que M no deseaba insultar a J, pero las maneras han sido inadecuadas y lo único que deseamos es mantener las cosas constructivas.*

[1]No hemos llegado a la sección de los agradecimientos aún, pero sólo para practicar lo que luego voy a enseñar: el nombre del observador era Brian Behlendorf, y fue él quien no indicó la importancia de mantener todas las discusiones públicas a menos de que existiera alguna necesidad de privacidad

Ahora, vamos con la propuesta de J. Creo que J tenía razón en decir que...

Por muy artificial que parezcan respuestas como estas, tienen un efecto notable. Si se llama la atención constantemente acerca de estas malas actitudes, pero no se pide una disculpa o conocimiento de la parte ofensora, entonces se deja a la gente calmarse y mostrar una mejor cara comportándose con más decoro la próxima vez—y lo harán. Uno de los secretos para hacer esto con éxito es nunca hacer de la discusión el tema principal. Siempre debe ser tratado a parte, una breve introducción a la mayor parte de tu respuesta. Hay que señalar que "aquí no hacemos las cosas de ésta manera" y luego continuar con el tema real, de manera que no dejemos nada a lo que los demás puedan responder. Si alguien protesta diciendo que no merecían ese reproche, simplemente hay que negarse a entrar en una disputa sobre esto. O no respondas (si crees que sólo están liberando tensión y que no requiere de una respuesta) o responde disculpándote por haber sobreactuado y que es difícil detectar matices en el correo electrónico, y ahora de vuelta al tema principal. Nunca insistas en un reconocimiento, público o privado, de alguien que se haya comportado inadecuadamente. Si deciden por voluntad propia enviar una disculpa, genial, pero solicitar que lo hagan en contra de su voluntad, sólo causará resentimiento.

El objetivo principal es de hacer que la buena educación se vea como una de las actitudes del grupo. Esto ayuda al proyecto, porque otros desarrolladores pueden ser espantados (incluso de proyectos que les gustan y en los que quieren ayudar) por una flame war. Quizás ni siquiera se llegue a saber que han sido espantados; pueden estar merodeando las listas de correo, descubrir que se necesita de un grueso pelaje para participar en el proyecto y decidir en contra de involucrarse de cualquier manera. Mantener los foros amistosos es una estrategia de supervivencia a largo plazo y es más fácil mientras el proyecto siga siendo pequeño. Una vez sea parte de la cultura general, no será necesario ser la única persona promocionando esto. Será mantenido por todos.

Practicad revisiones visibles del código

Una de las mejores formas de fomentar una comunidad productiva de desarrollo es hacer que cada uno pueda ver el código de los demás. Una infraestructura técnica es necesaria para hacer esto efectivamente—en particular, se deben activar los correos con los avisos de cambios; más detalles en “Correos de cambios”. El efecto de los correos electrónicos con los cambios es que cada vez que se envíe un cambio al código fuente, un correo es enviado mostrando un registro y las diferencias de los cambios (mirad *diff* en “Vocabulario”). *La revisión del código* es la práctica de revisar los correos con cambios mientras van llegando, buscando fallos y posibles mejoras. [2]

Revisar el código sirve varios propósitos simultáneamente. Es el ejemplo más obvio de revisión en nodos en el mundo del open source y directamente ayuda a mantener la calidad del programa. Cada fallo que se envía junto a un programa llego allí después de ser comprometido y no haber sido detectado; es por esto que mientras más ojos estén revisando los cambios, menos fallos serán empaquetados. Pero indirectamente, las revisiones tienen también otro propósito: confirmar a las personas que lo que hacen importa, porque obviamente nadie se tomaría el tiempo de revisar un cambio a menos que le importara su efecto. La gente realiza una mejor labor cuando saben que otros van a tomarse el tiempo de evaluarla.

En el proyecto Subversion, no hicimos de la revisión del código una práctica regular. No existía ninguna garantía de que después de cada commit éste sería revisado, aunque a veces alguien se interesa en un cambio que se realiza sobre una parte del código en el que se tiene particular interés. Fallos que deberían y podrían haber sido detectados, se colarón. Un desarrollador llamado Greg Stein, quien sabia la importancia de las revisiones del código de trabajos anteriores, decidió que iba a ser él quien diera el ejemplo revisando cada línea de *uno y cada uno de los commits* que hayan llegado al repositorio. Cada vez que alguien envía un cambio era seguido de un correo electrónico de Greg a las lista de los desarrolladores, diseccionándolos, analizando posibles problemas y ocasionalmente elogiando ingeniosas piezas de código. De ésta manera, estaba atrapando fallos y prácticas poco óptimas de programación que de otra manera habrían pasado desapercibidas. Deliberadamente, nunca se quejó de ser la única persona revisando ca-

[2]Comunmente es así como las revisiones del código se hacen en los proyectos open source, por lo menos. En proyectos más centralizados, la revisión del código puede significar que muchas personas se sienten juntas y lean impresiones del código fuente, buscando por problemas y patrones específicos.

da commit, a pesar de que esto le tomaba una gran cantidad de tiempo, pero siempre alababa las revisiones de código cada vez que tenía oportunidad. Muy pronto, otros, yo incluso, empezamos a revisar los cambios regularmente también. ¿Cuál era nuestra motivación? No había sido porque Greg conscientemente nos avergonzó hacia esto. Nos había probado que revisar el código era una manera muy valiosa de utilizar nuestro tiempo y que se podía contribuir tanto al proyecto revisando los cambios de otros como escribiendo código nuevo. Una vez demostrado esto, se volvió una conducta anticipada, hasta el punto en el que cada commit que no generaba alguna reacción hacía que quien la realizaba se preocupara e incluso que preguntase a la lista si alguien había tenido la oportunidad de revisarlo aun. Luego, Greg consiguió un trabajo que no le dejaba mucho tiempo libre para Subversion y tuvo que dejar de hacer revisiones regulares. Pero llegados a éste punto, el habito se había integrado en el resto de nosotros tanto, que parecía como algo que se hacía desde tiempos inmemoriables.

Hay que empezar a realizar las revisiones desde el primer commit. El tipo de problemas que son más fáciles de descubrir con sólo revisar las diferencias son las vulnerabilidades de seguridad, desbordamientos de memoria, comentarios insuficientes o documentación del API, errores *off-by-one*, emparejamientos mal hechos y otros problemas que requieren de un mínimo de contexto para encontrar. Aunque incluso problemas a larga escala como el fallar en abstraer patrones repetitivos a un sólo sitio sólo se pueden descubrir después de llevar mucho tiempo realizando revisiones regularmente, porque el recuerdo de diferencias anteriores ayuda a revisar las diferencias presentes.

No hay que preocuparse al no poder encontrar nada sobre lo que comentar o de saber lo suficiente acerca de todas las áreas del código. Usualmente habrá algo que decir sobre casi todos los cambios; incluso donde no hay nada que criticar, se puede encontrar algo que elogiar. Lo importante es dejar claro a cada programador, que lo que hacen se ve y es entendido. Por supuesto, el revisar código no absuelve a los desarrolladores de la responsabilidad de revisar y probar su código antes de enviar los cambios; nadie debe depender en las revisiones para encontrar cosas que debería haber encontrado.

Al abrir un proyecto cerrado, hay que ser sensible acerca de la magnitud de los cambios

Si se reabre un proyecto existente, uno que ya tiene desarrolladores activos acostumbrados a trabajar en un ambiente de código cerrado, habrá que asegurarse de que todos entienden que grandes cambios se avecinan—y asegurarte de que entiendes como se siente desde su punto de vista.

Intenta imaginar como la situación se presenta ante ellos: antes, todas las decisiones sobre el código y diseño eran hechas con un grupo de programadores quienes conocían el software más o menos al mismo nivel, quienes compartían la misma presión de los mismos directores y quienes conocían entre todos sus fuerzas y debilidades. Ahora se les pide que expongan su código al escrutinio de extraños al azar, quienes formarán un juicio basado sólo en el código, sin la conciencia de las presiones bajo las cuales se tomaron ciertas decisiones. Estos forasteros harán muchas preguntas, preguntas que harán que los desarrolladores existentes se den cuenta que la documentación en la que se han esclavizado tan duramente *sigue siendo* inadecuada (esto es inevitable). Para cerrar con broche de oro, todos estos forasteros son entidades desconocidas y sin cara. Si alguno de los desarrolladores ya se siente de por si inseguro sobre sus habilidades, imaginemos como éste sentimiento es exacerbado cuando recién llegados empiezan a señalar fallos en el código que han escrito, y aun peor, frente a sus colegas. A menos que se tenga un equipo con programadores perfectos, esto es inevitable—de hecho, puede que le suceda a todos ellos al principio. Esto no es porque sean malos programadores; es solo que todo programa de cierto tamaño tiene fallos y una revisión distribuida descubrirá algunos de estos fallos (Id a “Practicad revisiones visibles del código” anteriormente en éste capítulo). En algún momento, los recién llegados no serán sujetos a muchas revisiones al principio, ya que no pueden contribuir con código hasta que estén más familiarizados con el proyecto. Para tus desarrolladores, podrá parecer que todas las críticas van hacia ellos y no por su parte. Por esto, existe el peligro de que los viejos programadores se sientan asediados.

La mejor manera de prevenir esto, es advertir a todos acerca de lo que se avecina, explicarlo, decirles que el desconcierto inicial es perfectamente normal y asegurar que todo va a mejorar. Algunas de estas advertencias deberán hacerse en privado, antes de que el proyecto se haga público. Pero también puede llegar a ser útil recordarle a la gente de las listas publicas que ésta es una nueva dirección en el desarro-

llo del proyecto y que tomará algo de tiempo adaptarse. Lo mejor que se puede hacer es enseñar con el ejemplo. Si no ves a tus desarrolladores respondiendo suficiente preguntas a los nuevos, decirles que deben responder más preguntas no será de gran ayuda. Quizás no tengan aún una noción acerca de que requiere una respuesta y de que no, o puede que no sepan como dar diferentes prioridades a escribir código y las nuevas tareas de comunicación exterior. La manera de hacerlos participantes es hacerlo uno mismo. Hay que estar en las listas publicas y responder algunas preguntas. Cuando no se tenga la experiencia necesaria en una materia para responder a una pregunta entonces transfierela visiblemente a un desarrollador quien pueda responderla—y vigila para asegurarte de que continua con una respuesta. Naturalmente será tentador para los desarrolladores más antiguos enfrascarse en discusiones privadas ya que a esto es a lo que están acostumbrados. Asegurate de suscribirte a las listas internas en las cuales estas discusiones puedan dar lugar, de manera que puedas pedir que la discusión se continúe en las listas publicas inmediatamente.

Existen otros asuntos a largo plazo con abrir un proyecto cerrado. Capítulo 4, *Infraestructura Social y Política* explora técnicas para mezclar exitosamente desarrolladores asalariados y voluntarios y en Capítulo 9, *Licencias, Copyrights y Patentes* se discute la necesidad de ser diligente al abrir una base de código que puede contener programas que han sido escritos o que pertenecen a otras personas.

Anunciar

Una vez que el proyecto está presentable—no perfecto, sólo presentable—se está listo para anunciarlo al mundo. Esto es un proceso bastante sencillo: id a http://freshmeat.net/, pulsamos en Submit en la barra de navegación superior y rellenad el formulario anunciando el nuevo proyecto. Freshmeat es el sitio que todos miran a la espera de anuncios sobre nuevos proyectos. Sólo hace falta atrapar unas cuantas miradas allí para que noticias sobre el proyecto sean esparcidas de boca en boca.

Si se conocen listas de correos o grupos de noticias donde el anuncio del proyecto sea un tema de interés, entonces publicalo allí, pero hay que tener cuidado en publicar sólo *una* vez en cada foro y dirigir a las personas a los foros del proyecto para más discusiones (configurando la cabecera `Reply-to`). Los comentarios en los foros deben ser cortos y directos al grano:

```
To: discuss@lists.example.org
Subject: [ANN] Scanley full-text indexer project
Reply-to: dev@scanley.org

Este es un sólo mensaje para anunciar la creación del proyecto
Scanley, un indexador y buscador de texto open source con
un extenso API para el uso de programadores quienes desean
crear servicios de búsqueda en grandes colecciones de
ficheros de texto. Scanley ejecuta, está siendo desarrollado
activamente y buscamos nuevos desarrolladores y testers.

Sitio Web: http://www.scanley.org/

Características:
   - Busca texto plano, HTML y XML
   - Búsqueda de palabras o frases
   - (planeado) Búsquedas borrosas
   - (planeado) Actualización incremental de los índices
   - (planeado) Indexación de sitios web remotos

Requerimientos:
   - Python 2.2 o mayor
   - Suficiente espacio en disco para contener los índices
     (aproximadamente dos veces el tamaño ocupado en disco)

Para más información, visitad scanley.org

Gracias
```

```
-J. Random
```

(Más información “Publicity” enCapítulo 6, *Communications* para consejos sobre como anunciar lanzamientos posteriores u otros eventos.)

Existe un debate en el mundo del software libre sobre si es necesario empezar con código funcional o si el proyecto puede empezar a beneficiarse aun cuando está en la fase de diseño y discusión. Solía pensar que empezar con código funcional era el factor más importante, que esto es lo que separaba proyectos exitosos de los juguetes y que solo desarrolladores serios se verían atraídos que hacia algo concreto ya.

Esto resulto no ser del todo cierto. En el proyecto Subversion, empezamos con un documento de diseño, un núcleo de desarrolladores interesados e interconectados, mucha fanfarria y *nada* de código funcional. Para mi completa sorpresa, el proyecto recibió participantes activos desde el principio y para el momento en que teníamos algo funcional ya habían unos cuantos desarrolladores voluntarios involucrados profundamente. Subversion no es el único ejemplo; el proyecto Mozilla también fue iniciado sin código funcional y ahora es un navegador exitoso y popular.

En vista de ésta evidencia debo retirar mi afirmación sobre que es necesario tener código funcional para lanzar un proyecto. EL código funcional sigue siendo la mejor base para el éxito y una buena regla del pulgar sería esperar a tener el código antes de anunciar el proyecto. Por otra parte, pueden haber circunstancias cuando un anuncio temprano puede tener sentido. Si creo que al menos un documento de diseño bien desarrollado o algún otro tipo de marco de trabajo, es necesario— claro que puede ser modificado en base a las respuestas publicas, pero debe haber algo tangible, en el que las personas puedan hincar sus dientes.

Cuando sea que se anuncie un proyecto, no hay que esperar una horda de voluntarios listos para unirse inmediatamente. Usualmente, el resultado de anunciar es que se obtiene algunas preguntas casuales, algunas otras personas se unen a la lista de correos y aparte de esto, todo continua como antes. Pero con el tiempo, podréis notar un incremento gradual en la participación tanto de usuarios como de nuevo código de voluntarios. Anunciar es solo plantar una semilla, puede tomar un largo tiempo para que la noticia se extienda. Si el proyecto recompensa constantemente a quienes se involucran, las noticias se extenderán, pues la gente desea compartir algo cuando han encontrado algo bueno. Si todo va bien, la dinámica de las redes exponenciales de comunicación lentamente transformaran el proyecto en una compleja comunidad donde no se conoce el nombre de todos y no se puede seguir cada una de las conversaciones. Los próximos capítulos son acerca de como trabajar en éste ambiente.

Capítulo 3. Infraestructura Técnica

Los proyectos de software libre dependen en la tecnología que aportan la captura selectiva e integral de información. Mientras mejor se sea usando estas tecnologías y persuadiendo a otros para utilizarlas, mayor será el éxito del proyecto. Esto se vuelve más cierto mientras el proyecto crece. Un buen manejo de la información es lo que previene a un proyecto open source de colapsar bajo el peso de la Ley de Brook,[1] la cual afirma que asignar fuerza de trabajo adicional a un proyecto retrasado lo demorará aún más. Fred Brooks observó que la complejidad de un proyecto se incrementa al*cuadrado*del número de participantes. Cuando solo unas pocas personas están involucradas, todos pueden hablar entre todos fácilmente, pero cuando cientos de personas están involucradas, ya no es posible que cada uno de los individuos se mantengan constantemente al tanto de lo que todos los demás están haciendo. Si dirigir bien un proyecto de software libre se trata de hacer que todos se sientan como si estuviesen trabajando juntos en la misma habitación, es obvio preguntar: ¿Qué sucedería si todas las personas en una habitación atestada de gente hablase a la vez?

Este problema no es nuevo. En una habitación no metafórica atestada, la solución es *un procedimiento parlamentario* : guías formales acerca de como tener discusiones en tiempo real en grupos grandes, como asegurarse de que las disensiones más importantes no se pierdan entre comentarios irrelevantes, como formar subcomites, como reconocer cuando se toman decisiones, etc. Las partes más importantes en un procedimiento parlamentario especifican como deben interactuar los grupos con su sistema de manejo de información. Algunos comentarios se hacen para el registro, otros no. El registro mismo es sujeto a manipulación directa y se entiende que no es una transcripción literal de lo que ha ocurrido, sino que es una representación a lo que el grupo está dispuesto a *acordar* sobre lo sucedido. El registro no es monolítico, sino que toma diferentes formas para diferentes propósitos. Comprende los minutos de encuentros individuales, una colección completa de todos los minutos de todos los encuentros, sumarios, agendas y sus anotaciones, reportes de comités, reportes de corresponsales no presentes, listas de acción, etc.

Dado que Internet no es realmente una habitación, no debemos preocuparnos acerca de replicar aquellas partes de los procesos parlamentarios que mantiene a algunas personas calladas mientras las demás hablan. Pero cuando nos referimos a técnicas de manejo de la información, proyectos open source bien dirigidos son como procesos parlamentarios en esteroides. Ya que todas las comunicaciones en los proyectos open source suceden por escrito, sistemas muy elaborados han evolucionado para enrutar y etiquetar apropiadamente los datos, para minimizar las repeticiones de forma que se eviten divergencias espurias, para almacenar y buscar los datos, para corregir información incorrecta u obsoleta y para asociar bits dispares de información con cada uno mientras que nuevas conexiones son observadas. Los participantes activos en los proyectos open source integran muchas de estas técnicas y a menudo realizaran complejas labores manuales para asegurar que la información sea dirigida correctamente. Pero todo el esfuerzo depende al final de un sofisticado soporte informático. Tanto que sea posible, los mismos medios de comunicación deben realizar éste enrutamiento, etiquetado y registro y debería mantener la información al alcance de los humanos de la manera más conveniente posible. En la práctica, por supuesto, los humanos siguen necesitando intervenir en muchos puntos durante el proceso y también es importante que estos programas hagan ésta intervención lo más conveniente. Pero por lo general, si los humanos se encargan de etiquetar y enrutar información acertadamente desde su primera entrada en el sistema, entonces el software debería estar configurado para dar el máximo uso posible a esa metadata.

El consejo de éste capítulo es intensamente práctico, basado en las experiencias con aplicaciones y patrones específicos. Pero el objetivo no es sólo enseñar una colección particular de técnicas. Es también demostrar, utilizando pequeños ejemplos, la actitud general que mejor fomentará el correcto uso de los sistemas de manejo de información en el proyecto. Esta actitud incluye una combinación de habilidades técnicas y don de gentes. Las habilidades técnicas son esenciales porque las aplicaciones de manejo de información siempre requieren cierta configuración y además una cierta cantidad de mantenimiento y puesta apunto mientras nuevas necesidades vayan surgiendo (por ejemplo, mirad la discusión de como manejar el crecimiento del proyecto en “Pre-filtrado del gestor de fallos” más adelante en éste capítulo). El don de gentes es necesario porque la comunidad humana también requiere de cierto mantenimiento:

[1] De su libro*El mes mítico del hombre*, 1975. Más en http://en.wikipedia.org/wiki/The_Mythical_Man-Month y en http://en.wikipedia.org/wiki/Brooks_Law.

no siempre es inmediatamente obvio como utilizar estas herramientas para obtener una ventaja completa y en algunos casos los proyectos tienen convenciones conflictivas (por ejemplo, la discusión de como crear cabeceras `Reply-to` en los mensajes salientes de las lista de correos, en “Listas de correo”). Todos los involucrados en el proyecto van a necesitar ser animados, en el momento correcto de la forma correcta, para que sigan manteniendo la información del proyecto bien organizada. Mientras más involucrado esté el contribuyente, más complejas y especializadas serán las técnicas que se esperará que aprendan.

El manejo de información no tiene soluciones rápidas ya que existen demasiadas variables. Pueden que finalmente se tenga todo configurado justo como se desea y tener a la mayoría de la comunidad participando pero luego el crecimiento del proyecto hace de estas practicas no escalables. El puede que el crecimiento del proyecto se estabilice y que la comunidad de usuarios y desarrolladores acuerden una relación confortable con la infraestructura técnica pero llega alguien e inventa un nuevo servicio de manejo de información completo y pronto muchos de los recién llegados empezarán a preguntar que por qué no es utilizado en el proyecto— por ejemplo, esto está sucediendo mucho últimamente en proyectos de software libre que son anteriores a la invención del Wiki (más en http://en.wikipedia.org/wiki/Wiki). Muchas cuestiones son materia de juicio, incluyendo las desventajas entre la conveniencia de aquellos generando información y la conveniencia de aquellos quienes la consumen o entre el tiempo requerido para configurar el software de manejo de la información y los beneficios que le brinda al proyecto.

Cuidado con la tentación de automatizar demasiado, esto es, automatizar cosas que realmente requieren de atención por parte de los humanos. La infraestructura técnica es importante, pero lo que hace que los proyectos de software libre funcionar es el cuidado—y la expresión inteligente de éste cuidado—de los humanos involucrados. Principalmente, la infraestructura técnica está para ofrecer medios convenientes para hacer esto.

Lo que necesita un proyecto

La mayoría de los proyectos open source ofrecen al menos un mínimo y estándar conjunto de herramientas para manejar información:

Sitio Web
: Principalmente, conducto de información centralizado en un sentido del proyecto para el público. El sitio web puede también servir como una interfaz para otras herramientas del proyecto.

Listas de Correo
: Usualmente es el foro de comunicación más activo del proyecto y el "medio de registro."

Control de Versiones
: Permite a los desarrolladores realizar cambios al código convenientemente, incluso retroceder y exportar cambios. Le permite a todos mirar lo que está sucediendo con el código.

Gestión de fallos
: Permite a los desarrolladores mantener un registro de en qué están trabajando, coordinandose entre ellos y planear lanzamientos. Permite que todo el mundo pueda realizar búsquedas acerca del estado de los fallos y registrar información (p.e. recetas reproducibles) acerca de fallos en particular. Puede ser utilizado para seguir no solo fallos, sino también lanzamientos, tareas, nuevas características,etc.

Chat en tiempo real
: Un sitio para discusiones rápidas, sencillas e intercambios de preguntas/respuestas. No siempre se archiva completamente.

Cada una de estas herramientas está dirigida a distintas necesidades, pero sus funciones están también interrelacionadas y se debe hacer que estas herramientas trabajen en conjunto. Más abajo examinaremos como podemos lograr esto y más importante aun como hacer que las personas se acostumbren a usarlas. El sitio web no se discute hasta el final, ya que actúa más como un pegamento para otros componentes

que como una herramienta en sí.

Se pueden evitar muchos dolores de cabeza por escoger y configurar estas herramientas si en su lugar utilizamos un *hosting enlatado* : un servicio que ofrece todas las herramientas necesarias para un proyecto open source ya listas para su uso gracias a plantillas y empaquetado. Más en “Soluciones de hospedaje” a continuación en éste mismo capítulo para una discusión más profunda acerca de ventajas y desventajas de estas soluciones.

Listas de correo

Las listas de correo son el pan y la mantequilla de las comunicaciones del proyecto. Si algún usuario es expuesto a algún foro aparte de las paginas web, probablemente sea la lista de correos del proyecto. Pero antes de trabajar con las listas en si mismas, deben tomar experiencia con la interfaz—esto es, el mecanismo por el cual se pueden unir ("suscribirse a") a la lista. Esto nos brinda la regla número uno de las listas de correo:

> *No intentes dirigir las listas de correo a mano—consigue un software de manejo de listas.*

Será tentador dejar esto de lado. Configurar un software para listas de correo puede parecer demasiado difícil al principio. Manejar listas pequeñas de bajo tráfico a mano puede parecer seductor al principio: sólo hay que montar una lista de suscripción que te reenvía todo y cuando alguien envía un mensaje, se agrega (o elimina) su dirección de correo en algún tipo de fichero de texto que almacena todas las direcciones de la lista. ¿Qué podría ser más sencillo?

El truco está en hacer un buen manejo de las listas de correo—lo cual no es lo que la gente espera— no es nada sencillo. No es solo sobre suscribir y de suscribir usuarios cuando lo solicitan. También es sobre moderar para prevenir SPAM, ofrecer la lista resumida en lugar de mensaje por mensaje, proporcionar una lista estándar e información del proyecto a través de auto respuestas y muchas otras cosas. Un ser humano monitorizando las direcciones de suscripción es solo una pequeña parte del mínimo de funcionalidad e incluso así, no es la forma más segura y puntual que un software podría ofrecer.

Un software para el manejo de listas de correo usualmente ofrece las siguientes características:

Suscripción a través de correos o basada en web
: Cuando un usuario se suscribe a la lista, debería recibir una respuesta de bienvenida *sin demora*, explicándole como seguir interactuando con el software y (más importante) con eliminar la suscripción. Esta respuesta automática puede ser modificada para contener información específica del proyecto, por supuesto, como el sitio web, localización del FAQ, etc.

Suscripción al modo de resúmenes o al modo de mensaje por mensaje
: En modo resumen, el suscriptor recibe un correo conteniendo toda la actividad de la lista en ese día. Para aquellos quienes desean seguir la lista indirectamente, sin participar, el modo resumen es a menudo el preferible, porque les permite revisar todos los temas a la vez y evitar las distracciones de los correos que llegan en momentos al azar.

Características para la moderación
: Moderar es revisar los mensajes para asegurar que: a) no es SPAM y b) en tema, antes de que lleguen a la lista. La moderación incluye necesariamente a seres humanos, pero el software puede hacer mucho para hacerlo más sencillo. Se discute más acerca de la moderación luego.

Interfaz Administrativa
: Entre otras cosas, le permite a un administrador eliminar direcciones obsoletas fácilmente. Esto puede hacerse urgentemente cuando la dirección del receptor empieza a enviar respuestas automáticas del tipo "Ya no tengo ésta dirección de correo" a la lista en respuesta a cada mensaje. (Algunas apli-

caciones para listas de correo pueden incluso detectar esto por si mismas y eliminar la suscripción de ésta persona automáticamente.)

Manipulación de las cabeceras
: Muchas personas tienen sofisticados filtros y reglas de respuestas configuradas en sus clientes de correo. Las aplicaciones de listas de correo pueden añadir y manipular ciertas cabeceras estándar de las que estas personas se puedan beneficiar (más detalles a continuación).

Archivo
: Todos los mensajes enviados a las listas son almacenados y hechos públicos en la web. Alternativamente, algunas aplicaciones de software para listas de correo ofrecen interfaces especiales para conectar alguna herramienta externa de archivo como MHonArc (http://www.mhonarc.org/). Al igual "Sobresaliente uso de los archivos" en Capítulo 6, *Communications* se discute que el archivo es crucial.

El objetivo de todo esto es sencillamente enfátizar que la administración de las listas de correo es un problema complejo sobre el cual se ha pensado mucho y que está casi resuelto. Ciertamente no es necesario convertirse en un experto, pero hay que reseñar que siempre hay lugar para el aprendizaje y que la administración de las listas ocupara algo de atención de vez en cuando durante la duración del proyecto. A continuación examinaremos algunos de los problemas más comunes que podemos encontrar al configurar las listas de correo.

Prevenir el Spam

Entre el momento cuando ésta frase es escrita y cuando es publicada, el problema a lo largo y ancho de Internet el problema del Spam probablemente sea el doble de severo—o al menos parecerá que es así. Hubo una época, no mucho tiempo atrás, cuando se podía administrar una lista de correos sin la necesidad de tomar medidas para prevenir el Spam. Algún mensaje extraviado ocasional aparecía pero con tan poca frecuencia que solo era una molestia de bajo nivel. Esa época ya es historia. Hoy, las listas de correo que no toman medidas preventivas en contra del Spam se verá sumergida rápidamente en correo basura hasta el punto de ser inútil. Prevenir el Spam es una prioridad.

La prevención de Spam se divide en dos categorías: prevenir que mensajes basura aparezcan en la lista y prevenir que la lista sea utilizada como fuente de nuevas direcciones de correo para los spammers. La primera es la más importante, así que la examinaremos primero.

Filtrado de los mensajes

Existen tres técnicas básicas para prevenir mensajes basura y muchas aplicaciones para listas ofrecen las tres. Lo mejor es utilizarlas en tandem:

1. **Sólo permitir automáticamente mensajes de los suscriptores a la lista.**

   Esto es efectivo hasta cierto punto y necesita de poca administración ya que usualmente es sólo cuestión de cambiar algunos puntos en la configuración de la aplicación de listas. Hay que apuntar que aquellos mensajes que no son aprobados automáticamente no deben ser desechados. En su lugar, deben ser moderados por dos razones. Primero, se deben permitir mensajes de quienes no están suscritos. Alguna persona con una pregunta o sugerencia no debería tener que suscribirse a la lista para enviar un solo mensaje. Segundo, incluso quienes están suscritos envían mensajes desde cuentas diferentes de la que han utilizado para suscribirse. Las direcciones de correo electrónico no son un método eficaz para identificar a las personas y no debe ser utilizado para esto.

2. **Filtrar los mensajes utilizando un programa de filtro de spam.**

   Si la aplicación de listas de correo lo permite (la mayoría lo hace) se pueden filtrar los mensajes utilizando un filtro anti-spam. El filtrado automático de Spam no es perfecto, y nunca lo será, ya que exis-

te un pulso sin fin entre los spammers y los escritores de filtros. A pesar de esto, se puede reducir enormemente la cantidad de Spam que llega a la cola de moderación y dado que mientras más larga sea ésta cola, se necesitaran más tiempo examinándola, así que cualquier filtrado automático es beneficioso.

No hay lugar suficiente para unas instrucciones detalladas sobre como configurar filtros de Spam. Habrá que consultar la documentación de la aplicación de listas de correo para esto (en “Software” más adelante en éste capítulo). Las aplicaciones para listas vienen con características para la prevención de Spam, pero quizás sería una buena idea añadir una solución de un tercero. He tenido buenas experiencias con estas dos: SpamAssassin (http://spamassassin.apache.org/) y SpamProbe (http://spamprobe.sourceforge.net/). Esto no es una crítica contra otros filtros anti spam open source que al parecer son muy buenos también. Sucede que sólo he tenido la oportunidad de utilizar estos dos y estar satisfecho con ellos.

3. **Moderación.**

   Para aquellos correos que no son automáticamente aceptados por su virtud de ser enviados por un suscriptor a la lista y que pasan a través del filtro anti-spam, si es que lo hay, la ultima fase es la *moderación*: el correo es enrutado a una dirección especial, donde alguien lo examina y lo confirma o rechaza.

   Confirmar un mensaje se puede hacer de dos formas: se puede aceptar el mensaje sólo una vez o se le puede indicar a la aplicación que acepte éste y todos los mensajes futuros de éste remitente. Casi siempre deseamos hacer lo último de manera que podamos reducir la carga futura en la moderación. Los detalles sobre como confirmar esto, varían entre sistemas pero usualmente es una cuestión de responder a una dirección en especial con el comando "aceptar" (lo que significa que sólo se aceptará éste mensaje) o "permitir" (permitir éste y todos los mensajes futuros).

   Rechazar un mensaje se hace simplemente ignorando el correo de moderación. Si la aplicación nunca recibe confirmación de que algo es un mensaje valido entonces no pasará a la lista, así que con solo ignorar el correo de moderación creará el efecto deseado. En algunos casos, existe la opción de responder con los comandos "rechazar" o "denegar" para que automáticamente se desaprueben los mensajes del remitente sin siquiera pasarlos por la moderación. Raramente existe una razón para hacer esto ya que la moderación es por lo general para prevenir el spam y los spammers no suelen utilizar una misma dirección dos veces.

Hay que asegurarse de que la moderación *sólo* se utiliza para filtrar el spam y mensajes fuera de contexto, como cuando alguien envía un correo a la lista equivocada. El sistema de moderación por lo general ofrece una manera de responder directamente al remitente pero es mejor no utilizarlo para responder a preguntas que realmente pertenecen a la lista, incluso si se sabe la respuesta inmediatamente. De hacer esto, se privaria a la comunidad del proyecto de una visión exacta de que tipo de preguntas la gente hace y privarlos de la oportunidad de responder ellos mismos a preguntas y/o ver las respuestas de otros. La moderación de las listas debe ser estrictamente para mantenerlas libres de basura y de correos fuera de contexto, nada más.

Ocultar las direcciones en los archivos

Para prevenir que los spammers utilicen las listas de correo como una fuente de direcciones, una técnica muy común es la de ocultar las direcciones de correo de la gente en el registro, reemplazándolas como por ejemplo:

```
jrandom@somedomain.com
```

por

```
jrandom_AT_somedomain.com
```

o

```
jrandomNOSPAM@somedomain.com
```

o algo similar igual de obvio (para un humano). Ya que los recolectores de direcciones por lo general funcionan reptando por paginas web—incluyendo el archivo de nuestra lista de correo— y buscando secuencias conteniendo "@", modificar las direcciones es una forma para que sean invisibles o inútiles para los spammers. Esto no hace nada para prevenir que se envié spam desde la lista, por supuesto, pero si evita que se incremente la cantidad de spam enviado directamente a las cuentas personales de los usuarios de la lista.

Ocultar las direcciones puede ser algo controversial. A algunas personas les puede gustar mucho y se sorprenderán si el registro no lo hace automáticamente. Otras pueden pensar que es demasiado inconveniente (porque los humanos también tenemos que traducir las direcciones antes de utilizarlas). Algunas veces las personas afirman que es inefectivo, porque los recolectores en teoría pueden compensar cualquier patrón de modificación consistente. No obstante, hay que señalar que existe evidencia empírica de que ocultar las direcciones *es* efectivo, como se puede ver en http://www.cdt.org/speech/spam/030319spamreport.shtml.

Lo ideal sería que la aplicación administrativa de la lista diese la posibilidad de escoger a cada individuo, utilizando una cabecera si/no especial o configurándolo en las preferencias de la cuenta del suscriptor. Sin embargo, no conozco ninguna aplicación que permita hacer esto para cada suscriptor o para cada mensaje, así que por ahora el administrador de la lista debe tomar la decisión en nombre de todos (asumiendo que el archivador ofrece ésta característica, lo cual no es siempre así). Yo me inclino ligeramente hacia ocultar las direcciones. Algunas personas son muy cuidadosas para evitar enviar sus direcciones de correo electrónico en paginas web o en cualquier lugar donde un recolector de spam pueda verla, y podrían ser decepcionante que todo ese cuidado sea perdido gracias al registro de la lista de correo. Mientras tanto, la inconveniencia al ocultar las direcciones que impone en los usuarios del registro es muy pequeña, dada la trivialidad de transformar las direcciones al formato correcto si se necesita contactar con esa persona. Pero hay que seguir pensando en que, al final, sigue siendo una lucha sin fin: para cuando haya leído esto, los recolectores podrían haber evolucionado hasta el punto de reconocer la mayoría de formas comúnmente utilizadas para ocultar y tendremos que pensar en algo más.

Identificación y Administración de cabeceras

Por lo general, los suscriptores de las listas mueven estos correos a una carpeta específica para el proyecto, separados de su otro correo personal. Sus clientes de correo hacen esto automáticamente al examinar las *cabeceras* de los mensajes. La cabecera son los campos que se encuentran en la parte superior de los correos, los cuales indican el remitente, destinatario, asunto, fecha e información variada sobre el mensaje. Cabeceras certeras son bien conocidas y obligatorias:

```
From: ...
To: ...
Subject: ...
Date: ...
```

Otras son opcionales, aunque de cierta manera estándar. Por ejemplo, no es estrictamente requerido que un correo electrónico tenga la cabecera

```
Reply-to: sender@email.address.here
```

pero muchas lo tienen, porque da al destinatario una manera a prueba de errores de responder al remitente (es especialmente útil cuando el remitente ha tenido que enviar un correo desde una dirección diferente a la cual las respuestas deben ser dirigídas).

Algunos clientes de correo ofrecen una interfaz fácil de usar para rellenar correos basados en patrones en la cabecera Asunto. Esto lleva a que la gente pida que la lista de correo añada automáticamente un prefijo a todos los Asuntos, de forma que puedan configurar sus clientes para que busquen esos prefijos y archivar los correos en el directorio correcto. La idea es que el autor original escribiría:

```
Asunto: Trabajando en la versión 2.5
```

pero el correo aparecería en la lista así:

```
Asunto: [discuss@lists.example.org] Trabajando en la versión 2.5
```

Aunque la mayoría de las aplicaciones de administración de listas ofrecen la opción de hacer esto, yo recomiendo no utilizarla. El problema que resuelve puede ser resuelto de otras formas menos intrusas y los costes de utilizar espacio en el campo del Asunto son demasiado grandes. Los usuarios experimentados de las listas de correos revisan el asunto de los correos entrantes del día para decidir acerca de qué van a leer y qué van a responder. Fijar el nombre de la lista al Asunto puede mover hacia la derecha el verdadero Asunto y fuera de la pantalla, haciéndolo invisible. Esto oculta información necesaria para aquellos quienes dependen en la decisión de cuales correos van a abrir, reduciendo la funcionalidad conjunta de la lista para todos.

En lugar de sobrecargar el Asunto, hay que enseñar a los usuarios para que saquen ventajas de otras cabeceras estándar, empezando con el campo "Para", el cual debería contener el nombre de la lista de correos:

```
To: <discuss@lists.example.org>
```

Cualquier cliente de correo capaz de filtrar los mensajes basándose en el Asunto debe ser capaz de filtrar utilizando el campo Para facilmente.

Existen otras cabeceras opcionales pero estándar para las listas de correo. Filtrar utilizándolos es incluso más fiable que utilizar las cabeceras "Para" o "Cc" dado que estas cabeceras son añadidas a todos los mensajes por el programa de administración de la lista, así que algunos usuarios están contando con su presencia:

```
list-help: <mailto:discuss-help@lists.example.org>
list-unsubscribe: <mailto:discuss-unsubscribe@lists.example.org>
list-post: <mailto:discuss@lists.example.org>
Delivered-To: mailing list discuss@lists.example.org
Mailing-List: contact discuss-help@lists.example.org; run by ezmlm
```

La mayoría se explican en si mismos. En http://www.nisto.com/listspec/list-manager-intro.html se explican mejor o en http://www.faqs.org/rfcs/rfc2369.html para una especificación formal más detallada.

Hay que señalar como estas cabeceras implican que si se tiene una lista de correos llamada "list" entonces se tienen también unas direcciones administrativas "list-help" y "list-unsubscribe". Además de estas, es normal tener "list-subscribe" para unirse y "list-owner" para contactar con el administrador de la lista. Dependiendo en la aplicación administrativa que se use, estas y/o otras direcciones administrativas varias pueden ser configuradas; la documentación debería detallar esto. A menudo una explicación com-

pleta de todas estas direcciones especiales es enviada a cada nuevo suscriptor como parte de un mensaje de bienvenida automático. Probablemente usted mismo reciba una copia de esto correo de bienvenida. Si no lo ha recibido, pida una copia a alguien, de manera que pueda saber qué están recibiendo los nuevos suscriptores. Mantenga la copia a mano de manera que pueda responder preguntas acerca del funcionamiento de la lista, o mejor aun, ponerlo en una página web en alguna parte. Así, cuando alguien pierda su copia de las instrucciones y pregunte cómo pueden eliminarse de la lista, se les facilita la URL.

Algunas aplicaciones para listas de correos ofrecen la opción de agregar al final de cada mensaje las instrucciones para eliminar la suscripción. Si ésta opción está disponible, usela. Solo causa algunas lineas extra por mensaje en un sitio inofensivo y puede ahorrar mucho tiempo al reducir el número de gente que escriba —o peor aún, que escriban a la lista—preguntando cómo eliminar la suscripción.

El gran debate del Reply-To

Antes en “Evitar discusiones privadas” hice incapie en la importancia de asegurar que las discusiones se mantengan en foros públicos y hable acerca de porque a veces tomar medidas activas es necesario para prevenir que algunas conversaciones deriven a hilos privados. Este capítulo es acerca de todo lo relacionado con preparar el software de comunicación del proyecto para que realice la mayor cantidad de trabajo posible. Así que, si la aplicación para la administración de las listas de correo ofrece una manera automática de encausar las discusiones a la lista, habría que pensar que habilitarla es la opción correcta.

Bueno, quizás no. Existe tal característica, pero tiene algunas desventajas muy importantes. Usarla o no es uno de los debates más calientes en la administración de las listas de correo—admitamoslo, no es una controversia que vaya a aparecer en las noticias, pero se puede iniciar de vez en cuando en los proyectos de software libre. A continuación, voy a describir esta característica, proponer los argumentos de cada posición y hacer la mejor recomendación que puedo.

Esta característica en si misma es muy sencilla: la aplicación para las listas puede, si lo deseamos, automáticamente establecer la cabecera Reply-To en todos los mensajes para dirigir todas las respuestas a la lista. Así que, sin importar lo que escriba el remitente en este campo (o si ni siquiera lo establecen) para cuando los suscriptores a la lista vean el mensaje, éste contendrá la dirección de la lista en la cabecera:

```
Reply-to: discuss@lists.example.org
```

Esto puede parecer algo bueno, porque virtualmente todos los clientes de correo prestan atención a esta cabecera y ahora cada vez que alguien responda a algún mensaje, su respuesta será automáticamente dirigida a la lista, no sólo a quien ha enviado el mensaje que se responde. Por supuesto que el remitente puede manualmente modificar esto, pero lo importante es que *por defecto* las respuestas son enviadas a la lista. Este es un ejemplo perfecto de como utilizar la tecnología para animar la colaboración.

Desafortunadamente, existen algunas desventajas. La primera es conocida como el problema *No puedo llegar a casa*: a veces, el remitente original establece su dirección de correo real en el campo Reply-To porque por alguna razón u otra envían correos utilizando una dirección diferente a la que utilizan para recibirlos. Las personas que envían y reciben correos desde el mismo sitio no tienen éste problema y quizás se sorprendan de que siquiera existe. Pero para quienes utilizan configuraciones de correo inusuales o quienes no pueden controlar como se forma el campo From de su dirección de correo electrónico (quizás porque envían correos desde sus trabajos y no tienen ninguna influencia sobre el departamento de sistemas) el utilizar el campo Reply-To quizás sea la única manera que tienen para asegurarse de que las respuestas a sus mensajes les llegan (encuentran el camino a casa). Así que si la aplicación de las listas sobre escribe esto, esta persona puede que nunca vea las respuestas a sus mensajes.

La segunda desventaja se refiere a las expectativas y en mi opinión, el argumento más fuerte en contra del cambio del Reply-To. La mayoría de los usuarios experimentados de correo electrónico están acostumbrados a dos métodos básicos de responder: *Responder a todos* y *Responder al remitente*. Todos los clientes de correo tienen botones separados para estas dos acciones. Sus usuarios saben que para responder a todos los incluidos en la lista, deben escoger, responder a todos y que para responder sólo al remi-

tente en privado, deben seleccionar Responder al remitente. Aunque se desee animar a que la gente responda a la lista siempre que sea posible, existen ciertas circunstancias cuando un mensaje privado al remitente es prerrogativo—por ejemplo, desean compartir información confidencial, algo que sería inapropiado para una lista pública.

Ahora consideremos lo que sucede cuando la lista sobre escribe la cabecera Reply-To original del remitente. Quien responde pulsa la opción de Responder al remitente, con la esperanza de enviar un mensaje privado al autor original. Porque esta es la conducta esperada y quizás esta persona no se moleste en examinar cuidadosamente la dirección del destinatario en el nuevo mensaje. Redacta su correo privado, un mensaje confidencial, uno que puede diga algo embarazoso acerca de alguien de la lista y pulsa el botón de enviar. Inesperadamente el mensaje *llega a la lista* unos minutos después. Cierto, en teoría debería haber revisado cuidadosamente el destinatario y no debería haber asumido nada acerca del campo Reply-To. Pero por lo general este campo se compone con su dirección de correo personal (o en su lugar, los clientes de correo lo hacen) y muchos usuarios asiduos del correo electrónico dan esto por seguro. De hecho, cuando alguien determina deliberadamente el campo Reply-To a alguna otra dirección, como la de la lista, usualmente señalan esto en el contenido del mensaje, de forma que quienes respondan no se sorprendan de lo que sucede cuando lo hacen.

Dada la posibilidad de consecuencias muy severas de esta conducta inesperada, mi preferencia es la de configurar la aplicación de la lista para que nunca toque la cabecera Reply-To. Este caso de cuando se utiliza la tecnología para animar la colaboración tiene, a mi parecer, efectos colaterales potencialmente peligrosos. Por otro lado, existen argumentos concretos del otro lado de este debate. Sea lo que sea que se escoja, puede que en ocasiones algunas personas pregunten por qué no se ha escogido el otro camino. Dado que esto no es algo que se quiere sea el principal tema de discusión en la lista, puede ser conveniente tener una respuesta preparada del tipo que sea más propensa a poner fin a la discusión en lugar de animarla. Hay que asegurarse de *no* insistir en que esta decisión, sea cual sea, es obviamente la única correcta (incluso cuando se crea que esto es así). En cambio, hay que señalar que este es un viejo debate, que existen buenos argumentos de cada lado, que ninguna decisión iba a satisfacer a todos los usuarios y que por esto se ha tomado la mejor decisión que se podía. Amablemente se pide que el tema no vuelva a surgir a menos que alguien tenga algo realmente nuevo que decir, entonces hay que mantenerse alejado y esperar a que muera por causas naturales.

Alguien podría sugerir una votación. Se puede permitir esto si se quiere, pero personalmente no creo que contar manos sea una solución satisfactoria en este caso. El castigo para alguien que se vea sorprendido por este comportamiento es demasiado (accidentalmente enviar un correo privado a la lista publica) y las molestias para todos los demás es pequeña (ocasionalmente recordarle a alguien que deben responder a la lista) por esto no está claro de que la mayoría, aunque sean la mayoría, deban poner a una minoría bajo ese riesgo.

No he llegado a tocar todos los aspectos acerca de este tema, sólo los que me han parecido de especial importancia. Para una discusión completa, se pueden leer los siguientes documentos, los cuales son siempre citados cuando se entra en el debate:

- **Leave Reply-to alone**, *por Chip Rosenthal*

  http://www.unicom.com/pw/reply-to-harmful.html

- **Set Reply-to to list**, *por Simon Hill*

  http://www.metasystema.net/essays/reply-to.mhtml

A pesar de las benignas preferencias indicadas anteriormente, no creo que exista una única respuestas correcta y he participado felizmente de muchas listas que *cambiaban*el Reply-To. Lo mejor que se puede hacer, es centrarse en alguna de las dos vías desde el principio e intentar no verse enredado en debates sobre esto despues.

Dos fantasías

Algún día, alguien tendrá la brillante idea de implementar una opción *Responder a la lista* en su cliente de correo. Podría utilizar alguna de las cabeceras para listas mencionadas antes para descubrir la dirección de la lista de correos y luego direccionar las respuestas directamente a la lista, ignorando cualquier otro destinatario, ya que probablemente muchos estén suscritos a la lista de todas formas. Eventualmente, otros clientes implementarán esta característica y todo el debate desaparecerá. (De hecho, el cliente de correos Mutt [http://www.mutt.org/] ofrece esta opción.[2])

Una mejor solución sería que el tratamiento del campo Reply-To fuese una opción por suscriptor. Quienes deseen que la lista modifique sus cabeceras Reply-To (ya sea en sus mensajes o en los de otros) podría solicitarlo, y quienes no lo desean, se les deja tranquilos. Aunque no conozco ninguna aplicación para listas de correo que permita esto para cada suscriptor. Así que por ahora, parece que estamos atados a una configuración global.[3]

Archivo

Los detalles técnicos para configurar un archivo para la lista de correos son específicos de la aplicación utilizada y están fuera del alcance de este libro. Al escoger o configurar un archivador, es conveniente considerar lo siguiente:

Actualización rápida
: A menudo la gente querrá ser referida a un mensaje enviado durante la ultima hora o dos. Si es posible, el archivador deberá archivar cada mensaje instantáneamente, de tal manera de que cuando el mensaje aparezca en la lista de correos, ya esté en el archivo. Si esa opción no esta disponible entonces al menos hay que intentar configurar el archivado para que se realice cada hora o así. (Por defecto, algunos archivadores ejecutan el proceso de actualización cada noche, pero en la práctica esta demora es demasiado larga para una lista de correos.

Estabilidad referencial
: Una vez que un mensaje es archivado bajo una URL en particular, debe ser accesible desde esa URL para siempre. Incluso si el archivo es reconstruido o restaurado de un respaldo, cualquier URL que haya sido hecha publica debe permanecer igual. Las referencias estables hacen posible que los buscadores de Internet sean capaces de indexar el archivo, lo cual es una gran ventaja para los usuarios que buscan respuestas. Las referencias estables son también importantes porque los mensajes de la lista y los hilos son enlazados desde el gestor de fallos (“Seguimiento de errores”) más adelante en este capítulo o en la documentación de otros proyectos.

  Lo ideal sería que la aplicación de la lista de correos incluya la URL del mensaje archivado o al menos la porción de la URL específica del mensaje en una cabecera cuando este es distribuido. De esta manera la gente que haya recibido el mensaje podrá conocer su lugar en el archivo sin la necesidad de visitar el archivo, lo cual es de gran ayuda, ya que cualquier actividad que implique el uso del navegador web es automáticamente un consumo de tiempo. Que alguna aplicación de listas de correos ofrece esta posibilidad no lo sé. Desafortunadamente, los que he utilizado no la tienen. Pero esto es algo que hay que buscar (o si desarrolla una aplicación de listas, esta es una característica que debe considerar implementar, por favor).

Respaldos (Backups)
: Debe ser obvio como respaldar el archivo y la receta para restaurarlo no deben ser muy complicada. En otras palabras, no hay que tratar el archivo como una caja negra. Debe conocer donde se almacenan los mensajes y como restaurar las páginas del archivo del almacén si alguna vez es necesario. Estos archivos contienen datos muy preciados—un proyecto que los pierde, pierde buena parte de su memoria colectiva.

[2] Poco después de que este libro apareciera, Michael Bernstein [http://www.michaelbernstein.com/] me escribió para comentarme lo siguiente: "Existen otros clientes de correos que implementan una función de responder a la lista a parte de Mutt. Por ejemplo, Evolution tiene una combinación de teclas, pero no un botón (Ctrl+L)."

[3] Desde que escribí esto, he aprendido que existe al menos un sistema de gestión de listas que ofrece esta característica: Siesta [http://siesta.unixbeard.net/]. Hay un artículo sobre este en http://www.perl.com/pub/a/2004/02/05/siesta.html

Soporte de los hilos
: Desde cualquier mensaje debe ser posible ir al *hilo* (grupo de mensajes relacionados) al que pertenece el mensaje. Cada hilo debe tener su propia URL también, separado del URL de los mensajes del hilo.

Búsquedas
: Un archivo que no permita búsquedas—tanto en el cuerpo de los mensajes como por autor o según el asunto—es casi inútil. Hay que señalar que algunos archivadores permiten búsquedas al remitir la labor a un buscador externo como Google [http://www.google.com/]. Esto es aceptable, pero por lo general, las búsquedas directas son más finas, porque permiten a quien busca, especificar que los resultados sean mostrados, por ejemplo, según el asunto y no según el cuerpo del mensaje.

Lo anterior es sólo una lista técnica para ayudar a evaluar y configurar un archivador. Hacer que la gente de hecho *utilice* el archivo como ventaja para el proyecto es discutido en capítulos posteriores en particular en “Sobresaliente uso de los archivos”.

Software

Aquí hay algunas herramientas open source para la gestión de las listas de correo y su archivo. Si el hosting del proyecto ya tiene una configuración por defecto, quizás no sea necesario siquiera decidir cual herramienta utilizar. Pero si se tiene que instalar una, existen algunas posibilidades. Las que he utilizado son Mailman, Ezmlm, MHonArc e Hypermail, lo cual no significa que no haya otras que sean igual de buenas (y por supuesto, probablemente existan otras que no he logrado encontrar, así que no considere esto como una lista completa).

Aplicaciones de gestión de listas de correo:

- **Mailman** — http://www.list.org/

  (Tiene un archivador incorporado y la posibilidad de conectarse a archivadores externos.)

- **SmartList** — http://www.procmail.org/

  (Para ser utilizado con el sistema de procesamiento de correos Procmail.)

- **Ecartis** — http://www.ecartis.org/

- **ListProc** — http://listproc.sourceforge.net/

- **Ezmlm** — http://cr.yp.to/ezmlm.html

  (Diseñado para funcionar conQmail [http://cr.yp.to/qmail.html] .)

- **Dada** — http://mojo.skazat.com/

  (A pesar del bizarro intento de su sitio web, es un software libre, liberado bajo la licencia GNU GPL. También tiene un archivador incluido.)

Software para el archivo de las listas de correo:

- **MHonArc** — http://www.mhonarc.org/

- **Hypermail** — http://www.hypermail.org/

- **Lurker** — http://sourceforge.net/projects/lurker/

- **Procmail** — http://www.procmail.org/

  (Software que acompaña a SmartList, un sistema de procesado general de correos que puede, aparentemente, ser configurado como un archivo.)

Control de Versiones

Un *sistema de control de versiones* (o *sistema de control de revisiones*) es una combinación de tecnologías y practicas para seguir y controlar los cambios realizados en los ficheros del proyecto, en particular en el código fuente, en la documentación y en las páginas web. Si nunca antes se ha utilizado un control de versiones, lo primero que hay que hacer es conseguir a alguien que sí lo haya hecho y hacer que se una al proyecto. Hoy en día todo el mundo espera que al menos el código fuente del proyecto este bajo un control de versiones y probablemente no se tomen el proyecto seriamente si no se utiliza este sistema con un mínimo de competencia.

La razón por la cual el control de versiones es universal es porque ayuda virtualmente en todos los aspectos al dirigir un proyecto: comunicación entre los desarrolladores, manejo de los lanzamientos, administración de fallos, estabilidad entre el código y los esfuerzos de desarrollo experimental y atribución y autorización en los cambios de los desarrolladores. El sistema de control de versiones permite a una fuerza coordinadora central abarcar todas estas áreas. El núcleo del sistema es la *gestión de cambios*: identificar cada cambio a los ficheros del proyecto, anotar cada cambio con meta-data como la fecha y el autor de la modificación y disponer esta información para quien sea y como sea. Es un mecanismo de comunicación donde el cambio es la unidad básica de información.

Aun no hemos discutido todos los aspectos de utilizar un sistema de control de versiones ya que es un tema tan extenso que será introducido según el tópico a lo largo de este libro. Ahora, vamos a concentrarnos en seleccionar y configurar un sistema de control de versiones de forma que fomentemos un desarrollo cooperativo.

Vocabulario

En este libro no podemos enseñar como utilizar el control de versiones si nunca antes lo ha utilizado, pero sería imposible continuar sin conocer algunos términos clave. Estos son útiles independientemente del sistema particular utilizado: son definiciones básicas y verbos sobre la colaboración en red y serán utilizados a lo largo del libro. Incluso si no existiera ningún sistema de control de versiones, el problema del control de los cambios aun existiría y estas palabras nos dan un lenguaje para hablar acerca de este problema consistentemente.

> **"Versión" Versus "Revisión"**
>
> El termino *versión* es a veces utilizado como un sinónimo para "revisión", pero aquí no voy a utilizarla de esta forma, ya que se puede confundir fácilmente con "versión" en el sentido de una versión de un programa—así que, el número de lanzamiento o edición como en "Versión 1.0". Y aunque la frase "control de versiones" es un estándar, continuare utilizándolo como sinónimo para "control de revisiones" y para "control de cambios".

commit
: Realizar un cambio en el proyecto. Formalmente, almacenar un cambio en la base de datos del control de versiones de forma que pueda ser incorporado en lanzamientos futuros del proyecto. "Commit" puede ser utilizado como un verbo o como un sustantivo. Como un sustantivo, es esencialmente un sinónimo de "cambio". Por ejemplo: "He commited una reparación para un fallo reportado en Mac OS X que hacia que el servidor se colgara. Jóse ¿podrías por favor revisarlo y verificar que no

estoy haciendo mal la asignación?"

Mensaje de registro
: Un pequeño comentario añadido a cada commit que describe el tipo y el propósito del commit. Los mensajes de registro forman parte de los documentos más importantes de cualquier proyecto ya que son un puente entre el lenguaje altamente técnico de los cambios individuales en el código y el lenguaje más orientado al usuario de características, resolución de fallos y progreso del proyecto. Más adelante vamos a ver la forma de distribuir estos mensajes a la audiencia apropiada y también “Codifying Tradition” en Capítulo 6, *Communications* discutimos como ENCOURAGE a los voluntarios para que escriban mensajes de registro útiles y concisos.

update
: Solicitar los cambios (commits) que han realizado otros en la copia local del proyecto, esto actualiza esta copia a la ultima versión. Es una operación muy común ya que la mayoría de los desarrolladores actualizan su código varias veces al día y así saben que están ejecutando casi lo mismo que los otros desarrolladores, así que si se descubre un fallo es muy posible que este aun no haya sido resuelto. Por ejemplo: "Hey, he notado que el código del índice está fallando en el último byte. ¿Es esto un nuevo fallo?" "Sí, pero fue resuelto la semana pasada—prueba actualizar para resolverlo."

repositorio
: Una base de datos en la que los cambios son almacenados. Algunas versiones de sistemas de control de versiones son centralizados, es decir, existe un único repositorio maestro, el cual almacena todos los cambios en el proyecto. Otros sistemas son descentralizados, cada desarrollador tiene su propio repositorio y los cambios pueden ser intercambiados entre repositorios arbitrariamente. El sistema de control de versiones mantiene un registro de las dependencias entre estos cambios y cuando llega el momento de realizar un lanzamiento, un conjunto particular de cambios es aprobado para ese lanzamiento. La cuestión de cual sistema es mejor es otra de las guerras santas del desarrollo de software. Intentad no caer en la trampa de discutir sobre esto en las listas de correo del proyecto.

checkout
: El proceso de obtener una copia del proyecto desde el repositorio. Por lo general, un checkout produce un árbol de directorios llamado "copia funcional" desde el cual los cambios serán enviados de vuelta al repositorio original. En algunas versiones descentralizadas de sistemas de control, cada copia funcional es en si mismo un repositorio y los cambios son empujados (o atraídos) a cualquier repositorio que este dispuesto a aceptarlos.

copia funcional
: El árbol de directorios privado de cada desarrollador que que contiene el código fuente del proyecto y posiblemente las páginas web u otros documentos. Una copia funcional también contiene un pequeña cantidad de meta-data administrada por el sistema de control de versiones, informando a la copia funcional cual es repositorio central de procedencia, la revisión de los ficheros presentes, etc. Generalmente, cada desarrollador tiene su propia copia funciona en la cual realiza y prueba los cambios y desde la cual envía sus commits (cambios).

revisión, cambio, conjunto de cambios
: Una revisión es usualmente una encarnación específica de un fichero o directorio en particular. Por ejemplo, si el proyecto se inicia en la revisión 6 del fichero F y alguien envía un cambio al fichero F, esto produce la revisión 7 de F. Algunos sistemas también utilizan revisión (revision), cambio (change) o conjunto de cambios (changeset) para referirse a un conjunto de cambios enviados juntos como una unidad conceptual.

  Estos conceptos pueden tener distintos significados técnicos en cada sistema de control de versiones, pero en general, la idea es siempre la misma: dar un sistema para comunicar de manera precisa la historia de cambios en un fichero o conjunto de ficheros (inmediatamente antes y después de que se ha corregido un fallo). Por ejemplo: "Eso se ha resuelto en la revisión 10" o "Se ha corregido eso en la revisión 10 del fichero foo.c."

  Cuando se habla sobre ficheros o una colección de ficheros sin especificar una revisión en particular, por lo general se asume que nos referimos a la revisión disponible más reciente.

diff
: Una representación contextual de un cambio. Un diff muestra las lineas que han sido modificadas, como y además, algunas lineas contextuales rodeándolas a cada lado. Un desarrollador familiarizado con el código puede, con leer un diff de ese código, entender lo que hace el cambio e incluso detectar fallos.

etiqueta (tag)
: Una etiqueta para una colección particular de ficheros en una revisión específica. Los tags son utilizados para preservar capturas interesantes del proyecto. Por ejemplo, un tag es hecho para cada lanzamiento público, de forma que cada persona pueda obtener, directamente desde el sistema de control de versiones, el conjunto exacto de ficheros/revisiones que componen el lanzamiento. Algunos tags comunes son como `Release_1_0`, `Delivery_00456`, etc.

rama (branch)
: Es una copia del proyecto, bajo el control de versiones, pero aislado, de forma que los cambios realizados en esta rama no afecten al resto del proyecto y vice versa, excepto cuando los cambios sean deliberadamente "unidos" de un lado al otro. Las ramas también son conocidas como "lineas de desarrollo". Incluso cuando un proyecto no tiene ramas específicas se considera que el desarrollo se esta produciendo en la rama principal, también conocida como "línea primaria" o "*trunk*".

  Las ramas o branches, permiten aislar diferentes lineas de desarrollo de si mismas. Por ejemplo, una rama puede ser utilizada para un desarrollo experimental que sería demasiado inestable para la rama principal. O al contrario, una rama puede ser utilizada como sitio para estabilizar una versión para lanzamiento. Durante el proceso de lanzamiento, el desarrollo regular se mantendría ininterrumpida en la rama principal. Mientras tanto, en la rama de lanzamiento, ningún cambio es aceptado excepto aquellos aprobados por el responsable del lanzamiento. De esta manera, realizar un lanzamiento no tiene porque interferir con el trabajo de desarrollo que se está realizando. Para más información “Las ramas para evitar cuellos de botella” más adelante en el capítulo para una discusión más detallada sobre las ramas.

merge
: Mover un cambio de una rama a otra, lo que incluye unir desde la rama principal a otra rama o vice versa. De hecho, estos son las uniones más comunes y es rara la ocasión en la que esto se hace entre dos ramas no principales. Para más información sobre los merge “Singularidad de la información”.

  "Merge" tiene otro significado: es lo que hace el sistema de control de versiones cuando se encuentra con que dos personas han realizado cambios en un mismo fichero sin relación alguna. Ya que estos cambios no interfieren entre ellos, cuando alguna de estas personas actualizan su copia del fichero (el cual ya contiene los cambios) los cambios de la otra persona serán unidos automáticamente. Esto sucede muy a menudo, especialmente en proyectos con múltiples personas realizando cambios en el mismo código. Cuando dos cambios diferentes están relacionados, el resultado es un "conflicto".

conflicto
: Sucede cuando dos o más personas intentan realizar diferentes cambios en la misma porción de código. Todos los sistemas de control de versiones detectan estos conflictos automáticamente y notifican a al menos uno de los humanos involucrados de que sus cambios entran en conflicto con los de alguien más. Es entonces tarea de esta personas *resolver* el conflicto y comunicar esa resolución al sistema de control de versiones.

bloqueo (lock)
: Declaración de un intento exclusivo para cambiar un fichero o directorio en particular. Por ejemplo, "No puedo enviar cambios a las paginas web ahora mismo, ya que parece que Alfredo las tiene bloqueadas mientras arregla sus imágenes de fondo." No todos los sistemas de control de versiones ofrecen la posibilidad del bloqueo y aquellos que sí lo permiten, no es necesario que se utilice. Esto es porque el desarrollo paralelo y simultaneo es la norma y bloquear a la gente para que no puedan modificar los ficheros es contrario a la idea del sistema.

El modelo del sistema de control de versiones que requiere el bloqueo de ficheros suele ser llamado *bloqueo-modificación-desbloqueo* y el modelo que no requiere del bloqueo es llamado *copia-modificación-unión*. Una excelente explicación en profundidad y comparaciones puede ser encontrada en http://svnbook.red-bean.com/svnbook-1.0/ch02s02.html. En general, el modelo de copia-modificación-unión es el mejor para el desarrollo open source y todos los sistemas de control de versiones discutidos en este libro soportan este modelo.

Escoger un sistema de control de versiones

Hasta ahora, los dos sistemas de control de versiones más populares en el mundo del software libre son *Concurrent Versions System* (*CVS*), http://www.cvshome.org/) y *Subversion* (*SVN*, http://subversion.tigris.org/).

CVS lleva largo tiempo y los desarrolladores más experimentados ya están familiarizados con el. Hace más o menos lo necesario y ya que ha sido popular durante mucho tiempo es probable que no se termine discutiendo su utilización. A pesar de todo, CVS tiene algunas desventajas. No ofrece facilidad para hacer referencia a cambios en múltiples ficheros, no permite renombrar o copiar ficheros dentro del sistema de control (así que puede ser muy doloroso reorganizar el código después de iniciar el proyecto), el soporte para las uniones es algo pobre, no trabaja muy bien con ficheros muy grandes o con ficheros binarios y algunas operaciones son lentas cuando muchos ficheros están involucrados.

Ninguno de estos fallos de CVS son fatales y sigue siendo muy popular. Sin embargo, durante los últimos años Subversion ha venido ganando terreno, especialmente con nuevos proyectos. [4]. Si esta iniciando un nuevo proyecto, recomiendo Subversion.

Por otra parte, dado que estoy involucrado en el proyecto Subversion, mi objetividad puede ser razonablemente cuestionable y durante los últimos años han ido surgiendo un número de nuevos sistemas de control de versiones open source. Podemos encontrar una lista de todos los sistemas que conozco en Apéndice A, *Sistemas de Control de Versiones Libres* según su popularidad. Como muestra esta lista, escoger un sistema de control de versiones puede convertirse en una interminable tarea de investigación. Posiblemente esta decisión ya haya sido tomada por el sitio donde hospedamos el proyecto, liberándonos de esta carga, pero si es necesario tomar una decisión, lo mejor es consultar con otros desarrolladores, indagar un poco para tener una idea de las distintas experiencias que haya tenido la gente y luego escoger uno y mantenerse con este. Cualquier sistema de control de versiones estable y listo para entornos de producción será suficiente, no hay que preocuparse demasiado sobre tomar una decisión equivocada. Si simplemente no se puede decidir, entonces la opción es Subversion. Es relativamente fácil de aprender y es probable que se mantenga como el estándar por unos años más.

Utilizando el sistema de control de versiones

Las recomendaciones realizadas en esta sección no están enfocadas hacia un sistema de control de versiones en particular y debería ser sencillo implementarlas en cualquiera. La documentación específica del sistema debe ofrecer los detalles necesarios.

Versiones de todo

No solo hay que mantener el código del proyecto bajo el control de versiones también las paginas web, documentación, FAQ, notas de diseño y cualquier cosa que pueda ser necesario editar. Todo esto hay que mantenerlo cerca del código, en el mismo árbol que el repositorio. Se deben mantener versiones de cualquier pieza de información que pueda cambiar y archivar la que no cambie. Por ejemplo, un correo electrónico, una vez enviado, no cambia, por lo tanto, mantener versiones de este no tiene sentido (a menos que se convierta en una parte importante de la documentación).

[4]Más información en http://cia.vc/stats/vcs y para evidencia sobre su crecimiento en http://subversion.tigris.org/svn-dav-securityspace-survey.html .

La razón de mantener versiones de todo en un mismo sitio, es para que la gente sólo tenga que aprender un sistema para realizar cambios. A menudo, un voluntario se iniciara modificando algunas paginas web o partes de la documentación, para luego pasar a realizar pequeñas contribuciones al código, por ejemplo. Cuando el proyecto utiliza el mismo sistema para todo tipo de cambios, las personas sólo tendrán que aprender THE ROPES una vez. Mantener las versiones juntas significa que nuevas características pueden ser añadidas junto a la actualización de la documentación, y que al crear ramas del código, se crearan ramas de la documentación, etc.

No hace falta mantener los *ficheros generados* bajo el sistema de control de versiones ya que no son datos editables generados por otros programas. Por ejemplo, algunos sistemas de compilado generan los ficheros `configure` basandose en una plantilla `configure.in`. Para realizar cambios al fichero `configure` bastaría con modificar `configure.in` y volver a generarlo. Entonces, sólo el fichero `configure.in` es un fichero editable. Sólo es necesario mantener versiones de las plantillas—si se hace con los ficheros generados, la gente se olvidará de volver a generarlos cuando realicen algún cambio en las plantillas y las resultantes inconsistencias crearan una mayor confusión. [5]

La regla de que todos los datos editables deben ser mantenidos bajo el control de versiones tiene una excepción desafortunada: el gestor de fallos. La base de datos de fallos almacena una gran cantidad de datos editables pero generalmente, por razones técnicas, no se puede mantener bajo el control de versiones principal. (Algunos gestores tienen características primitivas de control de versiones, pero independiente de repositorio principal del proyecto.)

Navegabilidad

El repositorio del proyecto debe ser accesible desde Internet. Esto no solo significa la habilidad de visualizar la ultima revisión de los ficheros del proyecto, pero permitir volver atrás en el tiempo y ver en revisiones anteriores, mirar las diferencias entre revisiones, leer los mensajes de registro para cambios específicos, etc.

La navegabilidad es importante porque es un ligero portal a los datos del proyecto. Si el repositorio no es accesible desde un navegador web entonces alguien que desea inspeccionar un fichero en particular (por ejemplo, para mirar si una mejora ha sido incluida en el código) tendrá que instalar el mismo programa utilizado por el sistema de control de versiones, lo cual convierte una simple consulta de dos minutos en una tarea de medio hora o más.

También implica URLs CANONICAL para visualizar revisiones específicas de un fichero y para la ultima revisión en cualquier momento. Esto puede ser muy útil durante discusiones técnicas o para indicar alguna documentación a la gente. Por ejemplo, en lugar de decir "Para ayudas sobre como encontrar fallos en el servidor, mirad el fichero www/hacking.html en vuestra copia funcional" se puede decir "Para ayudas sobre como encontrar fallos en el servidor, mirad *http://svn.collab.net/repos/svn/trunk/www/hacking.html*," dando una URL que siempre lleva a la ultima revisión del fichero `hacking.html`. La URL es mejor porque no es nada ambigua y evita la cuestión de si existe una copia funcional actualizada.

Algunos sistemas de control de versiones incluyen un mecanismo que permite la navegación del repositorio, mientras que otros dependen de herramientas de terceros. Tres de estas herramientas son *ViewCVS* (http://viewcvs.sourceforge.net/), *CVSWeb* (http://www.freebsd.org/projects/cvsweb.html), and *WebSVN* (http://websvn.tigris.org/). La primera trabaja muy bien con CVS y con Subversion, la segunda sólo con CVS y la tercera sólo con Subversion.

Correos de cambios

Cada commit al repositorio debería generar un correo electrónico mostrando quien ha hecho el cambio, cuando, cuales ficheros y directorios han cambiado y como. Este correo debe ser dirigido a una lista de correos separada de las listas a las que envían los humanos. Los desarrolladores y todos aquellos intere-

[5] Alexey Mathotkin tiene una opinión diferente sobre el tema de controlar las versiones de los ficheros `configure` en un artículo llamado "*configure.in and version control*" en http://versioncontrolblog.com/2007/01/08/configurein-and-version-control/.

sados deben ser animados para suscribirse a las lista de commits ya que es la manera más efectiva de mantenerse al día con lo que sucede en el proyecto al nivel del código. Aparte de los obvios beneficios técnicos de la revisión por la comunidad (“Practicad revisiones visibles del código”), los correos con los cambios ayudan a crear un sentido de comunidad porque establecen un ambiente compartido en el que la gente puede reaccionar ante diferentes eventos (commits) que saben son visibles a otros tambien.

La configuración específica para habilitar estos correos varia dependiendo de la versión del sistema de control de versiones pero a menudo existe un script o paquete que facilita esto. Si se tiene algún problema para encontrar estos, intente buscar en la documentación el tema relacionado con los *hooks*, específicamente el *post-commit hook*, también llamado *loginfo hook* en CVS. Los Post-commit hooks son tareas automatizadas que se ejecutan como respuesta a los cambios enviados (commits). El hook es ejecutado por un cambio individual, se rellena con la información acerca del cambio y luego es liberada esa información para ser utilizada como se desee—por ejemplo, para enviar un correo electrónico.

Con los sistemas de correos con cambios ya listos para usar, quizás sea necesario modificar alguna de las siguientes conductas:

1. Algunos de estos sistemas no incluyen el diff en el correo que envían sino que enlazan con una URL para poder ver el cambio en la web utilizando el sistema de navegación del repositorio. Aunque esta bien dar una URL para que se pueda revisar el cambio luego, también es *muy* importante que el correo del commit incluya los diff. Leer el correo electrónico ya es parte de la rutina de la gente, así que si el contenido es visible allí mismo en el correo, los desarrolladores podrán revisar el commit en el mismo sitio sin la necesidad de abandonar sus clientes de correo. Si tienen que seguir un enlace a una página de revisiones, muchos no lo pulsarán, ya que esto requiere de una nueva acción en lugar de una continuación de lo que ya estaban haciendo. Por si fuera poco, si el lector desea preguntar algo acerca del cambio, es mucho más fácil responder al mensaje incluyendo el texto original y simplemente realizar anotaciones en el diff, en lugar de tener que visitar una página web y tomarse la molestia de copiar y pegar partes del diff en el navegador web al cliente de correo.

   (Por supuesto que si el diff es gigantesco, como cuando una gran parte de código nuevo ha sido añadido al repositorio, entonces tiene sentido omitir la parte del diff y ofrecer sólo la URL. Muchos de los sistemas permiten hacen esto automáticamente. Si el utilizado en el proyecto no es capaz de hacer esto, entonces sigue siendo mejor incluir los diffs completos. La conveniencia de la revisión y los comentarios es una piedra angular del desarrollo cooperativo, algo demasiado importante para olvidar.)

2. Los correos con los cambios deben tener su cabecera Reply-To direccionada hacia la lista regular de desarrollo, no a la lista de los cambios, de esta manera cuando alguien revise un cambio y escriba una respuesta, esta debe ser dirigida automáticamente a la lista de desarrolladores, donde los temas técnicos son discutidos normalmente. Existen varias razones para esto, primero, se quiere mantener todas las discusiones técnicas en la lista, porque es allí donde la gente espera que sucedan y porque así ésta es la única lista que será necesario archivar. Segundo, puede que existan partes interesadas no suscritas a la lista de cambios. Tercero, la lista de cambios es publicitada como una lista para los commits y no como una lista para los commits y las discusiones técnicas ocasionadas. Quienes se han suscrito sólo a la lista de cambios, no se han suscrito a nada más que commits, al enviarles correos con material sin relación utilizando ésta vía, es una violación del contrato implícito. Cuarto, algunas personas escriben programas que procesan los correos con los cambios (para publicarlos en una página web, por ejemplo). Estos programas están preparados para manejar correos con un formato consistente y son incapaces de trabajar con correos escritos por humanos.

   Hay que señalar que ésta recomendación no contradice las recomendaciones anteriores en “El gran debate del Reply-To”. Siempre esta bien que el remitente del mensaje configure la cabecera Reply-to. En este caso, el remitente es el sistema de control de versiones y su Reply-to lo configura de tal manera que indique que el lugar apropiado para responder es la lista de desarrollo y no la lista de cambios

CIA: Otro mecanismo de publicación de cambios

Los correos con los cambios no son la única forma de propagar las noticias de los cambios. Recientemente, otro mecanismo llamado CIA (http://cia.navi.cx/) ha sido desarrollador. Este es un distribuidor y AGGREGATOR de estádisticas de cambios. El uso más popular de CIA es el de enviar notificaciones de los commits a un canal IRC de forma que las personas en el canal pueden ver los commits en tiempo real. Aunque es una utilidad menos técnica que los correos electrónicos, ya que los observadores puede que estén o no conectados cuando las alertas sobre nuevos cambios llegan al canal, esta técnica tiene una inmensa utilidad *social*. La gente tiene la sensación de pertenecer a algo vivo y activo, y siente que pueden ver el progreso que se está haciendo ante sus propios ojos.

El programa de notificaciones del CIA es invocado por el post-commit hook, da formato al commit en un mensaje XML y lo envia al servidor central (por lo general `cia.navi.cx`). Este servidor luego distribuye la información a los otros foros.

CIA también puede ser configurado para enviar feeds RSS [http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html]. Más detalles en la documentación en http://cia.navi.cx/.

Para ver un ejemplo de CIA en acción conectese a `irc.freenode.net`, y al canal `#commits`.

Las ramas para evitar cuellos de botella

Los usuarios inexpertos del control de versiones pueden sentirse temerosos de crear ramas y uniones. Esto sea probablemente un efecto colateral de la popularidad de CVS: su interfaz de ramas y uniones puede ser poco intuitivo, así que muchas personas han aprendido a evitar estas operaciones por completo.

Si se encuentra entre estas personas, decidase ahora mismo a conquistar cualquier miedo que pueda tener y tómese el tiempo de aprender cómo funcionan las ramas y las uniones. No son operaciones muy complicadas una vez que se acostumbra a ellas y se vuelven muy importantes mientras el proyecto adquiere más desarrolladores.

Las ramas son muy importantes porque convierten un recurso escaso—espacio de trabajo en el código del proyecto—en uno abundante. Normalmente, todos los desarrolladores trabajan juntos en la misma caja de arena, construyendo el mismo castillo. Cuando alguien desea añadir un nuevo puente levadizo, pero no puede convencer a los demás de que sería una mejora, entonces las ramas hacen posible que vaya a una esquina aislada donde probar su puente. Si el esfuerzo tiene éxito, puede invitar a otros desarrolladores para que evalúen el resultado. Si todos están de acuerdo en que el resultado es bueno, pueden hacer que el sistema de control de versiones mueva ("merge") el puente levadizo de la rama del castillo a la rama principal del castillo.

Es fácil ver como esta habilidad ayuda al desarrollo colaborativo, ya que la gente necesita de cierta libertad para probar cosas nuevas sin sentir que están interfiriendo con el trabajo de otros. Igual de importante es cuando el código debe ser aislado del CHURN usual de desarrollo de manera que un fallo sea reparado o un lanzamiento sea estabilizado (más en “Stabilizing a Release” y en “Maintaining Multiple Release Lines” en Capítulo 7, *Packaging, Releasing, and Daily Development*) sin la preocupación de seguir un blanco en movimiento.

Hay que utilizar las ramas libremente y fomentar su uso entre otros. Pero también hay que asegurarse de que una rama en particular se mantenga activa exactamente durante el tiempo que sea necesaria. Incluso quienes no trabajan en la rama principal mantienen una visión periférica de lo que está sucediendo en ésta. Esta visión es deseable, por supuesto, y los correos con cambios deben salir de estas ramas como de cualquier otra. Pero las ramas no deben convertirse en un mecanismo que divida a la comunidad de de-

sarrolladores. Con raras excepciones, el objetivo eventual de la mayoría de las ramas debe de ser su unión a la rama principal y desaparecer.

Singularidad de la información

Las uniones tienen un corolario importante: nunca se debe enviar el mismo cambio dos veces, es decir, un cambio dado sólo debe ser introducido al sistema de control de versiones solo una vez. La revisión (o conjunto de revisiones) en la que el cambio es introducido es su identificador único desde ese momento. Si debe ser aplicado a otras ramas aparte de la cual en la que ha sido hecho, entonces deberá ser unido desde su punto de entrada original a sus otros destinos —al contrario de enviar cambios textualmente idénticos, que tendrían el mismo efecto en el código, pero harían del mantenimiento eficaz y de la gestión de lanzamientos una tarea imposible.

Los efectos prácticos de este consejo difieren entre sistemas de control de versiones. En algunos sistemas, las uniones son eventos especiales, fundamentalmente distintos de los commits y acarrean sus meta-datos propios. En otros, el resultado de las uniones son enviadas de la misma manera que los cambios son enviados así que la mejor manera de distinguir una unión de un nuevo cambio es leyendo los mensajes de registro. El mensaje de registro de una unión no repite el mensaje de registro del cambio original, en cambio, sólo indica que es una unión y da la identificación de la revisión del cambio original, con como mucho una línea de sumario del sus efectos. Si alguien desea ver el mensaje de registro completo, deberá consultar la revisión original.

La razón por la cual es importante evitar la repetición de los mensajes de registro es que estos pueden ser editados después de que se hayan enviado. Si un mensaje de registro es repetido en el destino de cada unión, entonces incluso si alguien edita el mensaje original, deja todos los otros mensajes sin corregir—lo cual sólo puede causar confusión a largo plazo.

El mismo principio se aplica al retirar un cambio. Si esto llegara a suceder, entonces el mensaje de registro para la retirada solo debe indicar que una revisión en particular está siendo retirada, no debe describir el cambio en el código resultante, pues la semántica del cambio se puede intuir al leer el mensaje de registro original del cambio. Por supuesto, el mensaje de registro del retiro también debe indicar la razón por la cual ese cambio ha sido retirado, pero no debe duplicar nada del mensaje de registro del cambio original. Si es posible, hay que volver y editar el mensaje de registro original para señalar que ha sido retirado.

Todo lo anterior implica que se debe utilizar una sintaxis consistente al referirnos a las revisiones. Esto es de gran ayuda no sólo en los mensajes de registro, sino en los correos electrónicos, en el gestor de fallos y en todas partes. Si se esta utilizando CVS, recomiendo "`directorio/al/fichero/del/proyecto/rama:REV`", donde REV es un número de revisión en CVS como "1.76". Si se esta utilizando Subversion, la sintaxis estándar para la revisión 1729 es "r1729" (el directorio de los ficheros no es necesario porque Subversion utiliza números globales para las revisiones). En otros sistemas, existe por lo general una sintaxis estándar para expresar el nombre del conjunto de cambios. Cualquiera que sea la sintaxis apropiada para el sistema utilizado, hay que animar a la gente a que lo utilicen al referirse a algún cambio. El uso consistente en el nombre de los cambios permiten que el mantenimiento del proyecto sea mucho más sencillo (como ya veremos en Capítulo 6, *Communications* y en Capítulo 7, *Packaging, Releasing, and Daily Development*), y dado que mucho de este mantenimiento será realizado por voluntarios, debe ser lo más sencillo posible.

Más información en “Releases and Daily Development” Capítulo 7, *Packaging, Releasing, and Daily Development*.

Autorizaciones

Muchos de los sistemas de control de versiones ofrecen la posibilidad por la cual a ciertas personas se les permite o no, realizar cambios en áreas específicas del repositorio. Siguiendo el principio de que cuando a las personas se les entrega un martillo empiezan a buscar clavos para golpear, muchos proyectos utilizan esta característica con ABANDON, permitiendo cuidadosamente el acceso solo a las áreas

donde tienen permiso de enviar cambio y asegurándose de que no lo puedan hacer en ningún otro sitio. (Más información en “Committers” Capítulo 8, *Coordinando a los Voluntarios* sobre como los proyectos deciden quienes pueden hacer cambios y donde.)

Probablemente hayan pequeños daños implementar un control así de estricto, pero una política un poco más relajada también esta bien. Algunos proyectos utilizan un sistema basado en el honor: cuando a una persona se le permite la posibilidad de realizar cambios, aunque sea a una pequeña área del repositorio, lo que reciben es una contraseña que les permite realizar cambios en cualquier otro sitio del repositorio y sólo se les pide que mantengan sus cambios en su área. Hay que recordar que no existe ningún peligro aquí: de todas formas, en un proyecto activo, todos los cambios son revisados. Si alguien hace un cambio donde no debía, alguien más se dará cuenta y dirá algo. Es muy sencillo si un cambio debe ser rectificado—todo está bajo el control de versiones de todas formas, así que sólo hay que volver atras.

Existen varias ventajas en tal aproximación tan relajada. Primero, mientras los desarrolladores se vayan expandiendo en las diferentes áreas (lo cual harán a menudo si siguen en el proyecto), no es necesario un trabajo administrativo extra de tener que dar y quitar privilegios. Una vez que la decisión es tomada, la persona puede empezar a enviar sus cambios a la nueva área sin problemas.

Segundo, la expansión se puede filtrar mejor, ya que generalmente, quienes realizan cambios en el área X y desean expandirse al área Y sólo tienen que empezar a enviar sus cambios contra Y y solicitar su revisión. Si alguien con acceso a cambios al área Y recibe alguno de estos parches y lo aprueba, puede pedir que el cambio sea enviado directamente (mencionando el nombre de quien ha revisado/aprobado el cambio en el mensaje de registro). De esta manera, el commit vendrá de quien ha hecho el cambio, lo cual es preferible desde un punto de vista administrativo y de credibilidad.

Por último, y quizás la razón más importante, al utilizar un sistema basado en el honor, se crea una atmósfera de confianza y respeto mutuo. Al darle a alguien permiso para enviar cambio a un subdominio se hace una declaración acerca de su preparación técnica—la cual dice: "Hemos visto que tienes la capacidad para realizar cambios en cierto dominio, así que a por ello". Pero imponer controles estrictos en las autorizaciones dice: "No sólo estamos juzgando tus limitadas capacidades, sino que también sospechamos de tus *intenciones*." Este no es el tipo de declaraciones que se desean hacer si pueden ser evitadas. Incluir a alguien dentro del grupo de desarrolladores del proyecto es una oportunidad de iniciarlos en un circulo de confianza mutua. Una buena manera de hacer esto es dar más poder del que se supone deben tener e informarles que es su responsabilidad mantenerse dentro de los límites impuestos.

El proyecto Subversion ha operado bajo este sistema por más de cuatro años, con 33 desarrolladores con privilegios completos y 43 con privilegios parciales. La única distinción que el sistema fuerza esta entre quienes envían cambios y quienes no, otras divisiones son mantenidas sólo por humanos. Incluso así, nunca hemos tenido ningún problema con que alguien realice un cambio deliberado fuera de su dominio. Una que otra vez han habido inocentes mal entendidos sobre la extensión de los privilegios de alguna persona, pero siempre es resuelto rápida y amigablemente.

Obviamente, en situaciones donde esto es poco práctico se debe depender en controles estrictos en las autorizaciones, pero dadas situaciones son raras. Incluso cuando hay millones de lineas de código y ciento o miles de desarrolladores, un commit hecho a cualquier módulo del código sigue siendo revisado por quienes trabajan en dicho módulo y son quienes pueden reconocer si quien lo ha intentado hacer puede hacerlo. Si una revisión regular no está sucediendo entonces el proyecto tiene problemas más importantes con los cuales lidiar que el sistema de autorizaciones.

Para concluir, no hace falta pasar mucho tiempo con las autorizaciones del sistema de control de versiones a menos que se tenga una razón en específico. Usualmente esto no trae beneficios tangibles y confiar en el control humano tiene sus ventajas.

Por supuesto que nada de esto significa que las restricciones mismas son poco importantes. Sería malo para un proyecto el animar a las personas a realizar cambios en áreas para las cuales no están cualificadas. Incluso en algunos proyectos, el acceso ilimitado tiene un status especial: implica derecho de voto en cuestiones que atañen al proyecto por completo. Este aspecto político del acceso es discutido en mayor profundidad en “¿Quién Vota?” en Capítulo 4, *Infraestructura Social y Política*.

Seguimiento de errores

El seguimiento de errores es un tema muy amplio y varios aspectos de este son discutidos a lo largo de este libro. Aquí intentare concentrarme principalmente en las consideraciones técnicas y en la instalación, pero para llegar a esto, debemos empezar con una política de preguntas: exactamente ¿qué tipo de información va a ser mantenida en el sistema de seguimiento?.

El término *seguimiento de errores* puede generar confusión ya que estos sistemas se utilizan frecuentemente para seguir solicitudes para nuevas características, tareas que se efectúan sólo una vez, parches no solicitados—en realidad se utilizan para cualquier cosa que pueda tener estados distinguibles de comienzo y final, con estados opcionales de transición entre estos y que acumulan información a lo largo de su existencia. Por esta razón, los sistemas de seguimiento de fallos también son llamados *de seguimiento de temas*, *de defectos*, *de solicitudes*, *trouble ticket system*, etc. Más información en Apéndice B, *Gestor de fallos libres* donde hay una lista de programas.

En este libro continuare utilizando "gestor de fallos" para la aplicación que hace el seguimiento, porque es así como la mayoría de la gente lo llama y utilizare *issue* al referirme a un punto en particular en la base de datos del gestor de fallos. Esto nos permitirá distinguir entre los buenos y malos comportamientos que el usuario se puede encontrar (el fallo en si mismo) y el *registro* en el gestor del descubrimiento, diagnostico y eventual resolución del fallo. Hay que recordar que aunque la mayoría de las entradas sean fallos, también pueden ser otras tareas.

El clásico ciclo de vida se parece al siguiente:

1. Alguien crea una entrada. Ofrecen un resumen, una descripción inicial (incluyendo como reproducir el fallo si es posible. En “Treat Every User as a Potential Volunteer” en Capítulo 8, *Coordinando a los Voluntarios* hay ejemplos de como se puede animar la correcta creación de reportes de fallos) y cualquier otra información que el gestor solicite. Quien crea la entrada puede ser un desconocido al proyecto—los reportes de fallos y las solicitudes de características provienen tanto de los usuarios como de los desarrolladores.

   Una vez enviada, la entrada entra en un estado llamado *abierto* porque ninguna acción ha sido tomada aun. Algunos gestores etiquetan las nuevas entradas como *sin verificar* o como *sin iniciar*. No está asignada a nadie, o en algunos sistemas, es asignada a un usuario fantasma que representa la falta de una asignación real. Llegado a este punto, la entrada se encuentra en el área de espera: ha sido registrada, pero aun no ha sido integrada en la conciencia del proyecto.

2. Otros leen la entrada, añaden comentarios y quizás soliciten el esclarecimiento de algunos puntos a quien realizo la entrada.

3. El fallo es *reproducido*. Este puede que sea el momento más importante en su ciclo vital, ya que incluso que el fallo aun no ha sido resuelto, el hecho de que alguien haya podido reproducirlo además de quien creo la entrada prueba que es genuino y, no menos importante, confirma al creador de la entrada que ha contribuido al proyecto reportando un fallo real.

4. El fallo es *diagnosticado*: su causa es identificada, y si es posible, es estimado el esfuerzo requerido para repararlo. Hay que asegurarse de que todo esto es registrado en la entrada, ya que en el case en que quien haya hecho el diagnostico abandona el proyecto (lo cual sucede a menudo con desarrolladores voluntarios), alguien más debe ser capaz de continuar con su trabajo.

   Llegados a este punto, o a veces en uno de los anteriores, puede que algún programador ya se haya "adueñado" de la entrada y se lo *asigne* a si mismo (el proceso es examinado en mayor detalle en “Distingue claramente entre pedir y asignar” en Capítulo 8, *Coordinando a los Voluntarios*). La *prioridad* de la entrada puede que también sea fijada en esta etapa. Por ejemplo, si el fallo es tan severo que debería retrasar el próximo lanzamiento, debe ser identificado desde el principio y el gestor debe proporcionar un mecanismo para hacer esto.

5. La entrada es programada para su resolución. Esto no implica necesariamente fijar una fecha para cuando debe ser resuelta. A veces sólo significa decidir para cual próximo lanzamiento (no necesariamente la siguiente) el fallo debe estar corregido o decidir si debe o no bloquear un lanzamiento en particular. Incluso nos podemos olvidar de planificar la reparación del fallo si es algo que se puede hacer rapidamente.

6. El fallo es reparado (o la tarea es completada, o el parche es aplicado o lo que sea). El cambio o conjunto de cambios que arreglan el fallo deben ser registrados en un comentario en la entrada, después de lo cual ésta es *cerrada* o marcada como *resuelta*.

Existen variaciones en este ciclo. A veces el problema es cerrado seguidamente después de ser archivado, porque resulta que no es un fallo, sino que es un malentendido por parte del usuario. Mientras el proyecto vaya ganando usuarios, más y más de estas entradas invalidas aparecerán, y los desarrolladores las cerraran con respuestas cada vez menos respetuosas. Hay que intentar protegerse de ésta tendencia, pues no le hace ningún bien a nadie, porque el usuario en cada caso no es responsable de las entradas invalidas previas. Esta tendencia estadísticas sólo es divisada por los desarrolladores, no por los usuarios. (En “Pre-filtrado del gestor de fallos” más adelante en este capítulo, examinaremos algunas técnicas para reducir el número de entradas invalidas.) También puede suceder que varios usuarios estén experimentando el mismo malentendido una y otra vez, lo cual significa que algún aspecto de la aplicación necesita volver a ser diseñada. Este tipo de patrones son los más sencillos de ver cuando se utiliza un gestor de entradas que monitorice la base de datos de fallos. Más en “Issue Manager” en Capítulo 8, *Coordinando a los Voluntarios*.

Otra variación muy común de este ciclo de vida es cuando la entrada es cerrada al ser un *duplicado* poco después del paso 1. Un duplicado aparece cuando alguien crea una entrada para un problema ya conocido por el proyecto. Los duplicados no están limitados a entradas abiertas: es posible que un fallo haya reaparecido después de haberlo reparado (esto es conocido como *regresión*), por lo cual, la vía preferida es usualmente reabrir la entrada original y cerrar cualquier nuevo reporte como duplicado de este. El sistema de gestión de fallo debe mantener un seguimiento de esta relación bidimensional, de forma que la información en los duplicados este disponible en la entrada original y vice versa.

Una tercera variación es cuando los desarrolladores cierran la entrada pensando que ya ha sido resuelta y el usuario que la ha reportado rechaza esa reparación y es reabierta. Por lo general esto es porque el desarrollador no tiene la capacidad de reproducir el fallo o porque no han probado su reparación siguiendo la misma receta para la reproducción descrita por el usuario.

A parte de estas variaciones existen pequeños detalles de este ciclo de vida que pueden variar dependiendo de la aplicación de seguimiento. Pero la forma básica es la misma e incluso cuando el ciclo de vida no es sólo para el software open source, tiene implicaciones acerca de cómo los proyectos utilizan sus sistemas de control de fallos.

Implícito en el paso 1, el sistema es una cara tan publica del proyecto, como lo pueden ser las listas de correo o las paginas web. Cualquiera puede crear una entrada, cualquiera puede ver una entrada y cualquiera puede navegar la lista de entradas abiertas. De tal manera que nunca se sabe cuantas personas están interesadas en ver el progreso en una entrada en particular. Aunque el tamaño y la capacidad de la comunidad de desarrolladores constriñe la frecuencia con la que los problemas son atacados, el proyecto debe al menos intentar reconocer cada entrada mientras vayan llegando. Incluso si el problema persiste por un tiempo, una repuesta anima al usuario a mantenerse involucrado porque siente que un humano ha visto lo que ha hecho (recordad que rellenar una entrada requiere mucho más tiempo que un correo electrónico). Incluso mejor, una vez que una entrada es vista por un desarrollador, entra en la conciencia del proyecto, en el sentido en que este puede mantenerse al acecho de otras instancias del mismo problema, puede comentarlo con otros desarrolladores, etc.

La necesidad de reacciones oportunas implica dos cosas:

- El sistema de seguimiento debe conectarse a la lista de correos de manera que cada cambio a una en-

trada, incluyendo su redacción inicial, genere un correo describiendo lo sucedido. Esta lista de correos es, a veces, diferente de la lista de desarrollo ya que quizás, no todos los desarrolladores quieran recibir correos automáticos con fallos, pero (al igual que con los correos con cambios) la cabecera Reply-to debe ser fijada a la lista de desarrollo.

- El formulario donde se rellena la entrada debe almacenar la dirección de correo electrónico de quien la reporta, de forma que pueda ser contactada para solicitar más información. (No obstante, no debe *requerir* la dirección ya que algunas personas prefieren realizar el reporte anónimamente. Más información sobre el anonimato en “Anonimato y participación” a continuación en este capítulo.

Interacción con las Lista de Correo

Hay que asegurarse de que el gestor de fallos no se convierte en un foro de discusiones. Aunque es importante mantener una presencia humana en el gestor, no está preparado para discusiones en tiempo real. Hay que pensar en éste como un archivador, una forma de organizar hechos y referencias a otras discusiones, principalmente aquellas que suceden en las listas de correo.

Hay dos razones por las cuales es importante hacer esta distinción. Primero, el gestor de fallos es un sistema más engorroso que las lista de correo (o que salas de chat). Esto no es porque estén mal diseñados, es sólo que sus interfaces han sido diseñadas para capturar y presentar estados discretos, no discusiones. Segundo, no todo el mundo que este involucrado en una discusión sobre una entrada en particular, esta revisando el gestor de fallos frecuentemente. Parte de una buena gestión de fallos (más en “Share Management Tasks as Well as Technical Tasks” en Capítulo 8, *Coordinando a los Voluntarios*) está en asegurarse de que cada problema es llevado a la atención de las personas indicadas en lugar de requerir que todos los desarrolladores monitoricen todos los problemas. En “No Conversations in the Bug Tracker” en Capítulo 6, *Communications*, veremos como asegurarnos de que la gente no desvíe accidentalmente las discusiones de los foros apropiados hacia el sistema de gestión de fallos.

Algunos gestores pueden monitorizar listas de correos y automáticamente registrar todos los correos que son acerca de un problema conocido. Por lo general, hacen esto reconociendo el número de identificación de la entrada en el asunto de los correos, como parte de una línea especial, así los desarrolladores pueden aprender a incluir estas lineas en sus correos para atraer la atención del gestor. El sistema puede guardar el correo completo o (incluso mejor) registrar un enlace al correo en el archivo regular de la lista de correos. De cualquier forma, ésta es una habilidad muy útil, así que si el sistema utilizado la aporta hay que utilizarla y hay que recordarle a la gente que la utilice.

Pre-filtrado del gestor de fallos

Muchas de las bases de datos de fallos sufren eventualmente del mismo problema: una cantidad devastadora de fallos duplicados o inválidos hechos por usuarios bien intencionados pero sin experiencia o poco informados. El primer paso para combatir esta tendencia es, por lo general, colocar un vistoso aviso en la página principal del gestor de fallos, explicando como saber si un bug es realmente un bug, como buscar si el bug ya está incluido y finalmente, como reportar efectivamente si aun se cree que es un nuevo fallo.

Esto reducirá el nivel de ruido por un tiempo, pero mientras el número de usuarios vaya creciendo, el problema regresara eventualmente. Ningún individuo puede ser culpado de esto, ya que cada uno está intentando contribuir en beneficio del proyecto e incluso cuando su primer reporte no sea de verdadera utilidad, se desea animarlos para que continúen involucrándose y para que puedan hacer mejores reportes en el futuro. Mientras tanto, el proyecto necesita mantener en lo posible la base de datos libre de basura.

Las dos cosas que tendrán el máximo efecto a la hora de prevenir este problema son: asegurarnos de que hay gente vigilando el gestor de fallos quienes tienen el conocimiento suficiente para cerrar problemas como inválidos o duplicados mientras vayan llegando y requiriendo (o fomentando duramente) a los usuarios que confirme su reporte con otras personas antes de reportarlos en el gestor.

La primera técnica parece ser utilizada universalmente. Incluso proyectos con gigantescas bases de datos de fallos (digamos, el gestor de Debian en http://bugs.debian.org/, el cual contenía 315,929 reportes al momento de escribir este libro) siguen ordenando todo de tal manera que *todos* puedan ver los reportes mientras llegan. Puede que sea una persona diferente dependiendo de la categoría del problema. Por ejemplo, el proyecto Debian es una colección de paquetes de software, de manera que el proyecto automáticamente enruta cada reporte a la persona que mantiene el paquete específico. Por supuesto, a veces los usuarios no identifican bien la categoría a la que pertenece el problema, con el resultado de que el reporte es enviado a la persona equivocada, quien entonces deberá redireccionarlo. No obstante, lo importante es que la carga sigue siendo distribuida—cada vez que un usuario crea correcta o incorrectamente al reportar, la vigilancia de las entradas sigue siendo distribuida más o menos uniformemente entre los desarrolladores, de manera que cada reporte es respondido en un tiempo justo.

La segunda técnica esta menos extendida, probablemente sea porque es más difícil de automatizar. La idea esencial es que cada nuevo reporte es apadrinado hacia la base de datos. Cuando un usuario cree haber encontrado un bug, se le pide que lo describa en una de las listas de correo o en algún canal de IRC para que reciba confirmación de alguien de que en realidad es un fallo. Al introducir este segundo par de ojos puede prevenir muchos reportes falsos. A veces esta segunda persona puede identificar que este comportamiento no es un fallo o que ha sido resuelto recientemente. O puede que este familiarizado con los síntomas gracias a problemas anteriores, evitando un duplicado al señalar al usuario el viejo reporte. A veces es tan sencillo como preguntar al usuario "¿Has revisado el gestor de fallos para asegurarte de que no ha sido reportado ya?" Muchas personas no piensan en esto, pero se contentan con hacer la búsqueda sabiendo que hay alguien a la *expectativa* de que lo hagan.

El sistema de apadrinamiento puede mantener la limpieza de los reportes en la base de datos, pero también tiene algunas desventajas. Muchas personas harán los reportes sin consultar, al no buscar o despreocupándose de las instrucciones de buscar a un padrino para el nuevo reporte. Aun así, es necesario que los voluntarios sigan vigilando las bases de datos y dado que la mayoría de los nuevos usuarios que reportan fallos no entienden la dificultad de mantenerlas, no es justo reprenderlos duramente por ignorar las directrices. Aun así, los voluntarios deben ser vigilantes y ejercitar restricciones en como se rechazan reportes sin apadrinar de vuelta a quien lo haya hecho. El objetivo es entrenar a cada reportero para que utilice el sistema de apadrinamiento en el futuro, de tal manera que haya una siempre creciente fondo de gente quienes entienden el sistema de filtrado de fallos. Al encontrarnos con un reporte sin padrino, los pasos ideales a tomar son:

1. Inmediatamente responder el reporte, agradeciendo al usuario por hacerlo, pero dirigiéndolo a las directrices de apadrinamiento (las cuales deberían, por supuesto, estar publicadas en un lugar prominente del sitio web.)

2. Si el reportes es claramente valido y no un duplicado, hay que aprobarlo de todas formas y de esta manera que inicie su ciclo de vida normal. Después de todo, quien ha realizado el reporte ya ha sido informado sobre el apadrinamiento, así que no tiene sentido perder el trabajo ya hecho al cerrarlo como invalido.

3. Si el problema no es claramente valido, hay que cerrarlo, pero solicitando que sea reabierto si reciben la confirmación por parte de un padrino. Cuando lo hagan, deberán colocar una referencia al hilo de confirmación (por ejemplo, una URL en el archivo de la listas de correo).

Hay que recordar que a pesar de que este sistema mejorara la proporción señal/ruido en la base de datos de problemas a lo largo del tiempo, nunca pondrá fin a los reportes inválidos. La única manera de evitar esto por completo es cerrar el gestor de fallos a todos quienes no sean desarrolladores—una cura que casi siempre es peor que la enfermedad. Es mejor aceptar que la limpieza de reportes inválidos siempre será una parte de la rutina de mantenimiento del proyecto e intentar obtener la mayor cantidad de ayuda para hacerlo.

Más en “Issue Manager” en el Capítulo 8, *Coordinando a los Voluntarios*.

IRC / Sistemas de Chat en Tiempo Real

Muchos proyectos ofrecen salas de chat utilizando *Internet Relay Chat* (*IRC*), foros donde los usuarios y desarrolladores pueden hacerse preguntas y obtener respuestas instantáneas. Mientras que se *puede* llevar un servidor de IRC para nuestro sitio web, por lo general no vale la pena. En cambio podemos hacer lo que todo el mundo: crear canales de IRC en Freenode (http://freenode.net/). Freenode proporciona el control necesario para administrar los canales IRC del proyecto, [6]mientras que nos evita la molestia de tener que mantener un servidor de IRC.

Lo primero que hay que hacer es decidir un nombre para el canal. La opción más obvia es utilizar el nombre del proyecto—si es que se encuentra disponible en Freenode. Si no, se puede utilizar algo lo más parecido al nombre del proyecto y que sea en lo posible, fácil de recordar. Hay que publicitar la disponibilidad del canal en el sitio web del proyecto, de manera que un visitante con una duda pueda verlo rápidamente. Por ejemplo, esto aparece en un contenedor prominente en la parte de arriba de la página principal de Subversion:

> *Si está utilizando Subversion, le recomendamos que se una a la lista* `users@subversion.tigris.org` *y lea el Libro de Subversion [http://svnbook.red-bean.com/] y el FAQ [http://subversion.tigris.org/faq.html]. También puede comentar sus dudas en IRC en el canal* `#svn` *en* `irc.freenode.net`

Algunos proyectos tienen varios canales, uno para cada tema. Por ejemplo, un canal para problemas de instalación, otro para dudas sobre su uso, otro para charlas sobre el desarrollo, etc. (“Manejando el crecimiento” en el Capítulo 6, *Communications* se discute como dividirse en múltiples canales). Cuando el proyecto es joven, sólo debe haber un canal en el que todos hablan juntos. Luego, mientras el proyecto vaya creciendo, la separación de canales será necesaria.

¿Cómo podrá la gente encontrar todos los canales disponibles y además, en cuales entrar? ¿Y al entrar, cómo sabrán los criterios de la sala?

La respuesta a todo esto es publicándolo en el *tópico del canal*.[7] El tópico del canal es un breve mensaje que ven todos los usuarios cuando entran en el canal. Da una guía rápida para los recién llegados y apunta a información necesaria. Por ejemplo:

```
Ha entrado en #svn

El tema para #svn es Foro para usuarios de Subversion. Más información en
http://subversion.tigris.org/. || Las discusiones sobre el desarrollo
están en #svn-dev. || Por favor, no pegue transcripciones muy largas,
para ello utilice un sitio como http://pastebin.ca/ || Noticias:
Subversion 1.1.0 ha salido, más en http://svn110.notlong.com/
```

Es algo tosco, pero informa a quienes entran al canal lo que necesitan saber. Dice exactamente para lo que es el canal, muestra la página web del proyecto (en caso de que alguien entre al canal sin antes haber visitado el sitio web del proyecto), menciona canales relacionados y da algunas directivas sobre el pegado.

Sitios de pegado

[6]No es un requerimiento ni tampoco se espera ninguna donación a Freenode, pero si usted o el proyecto se lo pueden permitir, por favor considerelo. Son una caridad exenta de impuestos en EE.UU. y proveen de un servicio muy valioso.

[7]Para establecer el tópico del canal se utiliza el comando "`/topic`. Todos los comandos en IRC empiezan con el signo "`/`". Si no se está familiarizado con la utilización y administración de IRC id a http://www.irchelp.org. Hay un excelente tutorial en http://www.irchelp.org/irchelp/irctutorial.html.

Un canal de IRC es un espacio compartido: todos pueden ver lo que todos escriben. Normalmente esto es algo bueno, ya que permite que la gente entre en una conversación cuando creen que tienen algo para contribuir y permite a los espectadores aprender leyendo. Pero puede tornarse problemático cuando alguien suministra una gran cantidad de información a la vez, como la transcripción de una sesión de debugging, porque al pegar muchas lineas de texto en el canal se interrumpen las conversaciones de otros.

La solución a esto es el uso de los llamados sitios de pegado *pastebin* o *pastebot*. Al requerir una gran cantidad de datos de alguien, pida que no los pegue en el canal, sino que vayan a (por ejemplo) http://pastebin.ca, peguen la información necesaria allí y suministren el nuevo URL resultante al canal de IRC. Así cualquiera puede visitar la URL y revisar los datos que allí se encuentran.

Existen muchos sitios gratuitos de pegado disponibles, demasiados para una lista comprensiva, pero aquí hay algunos que he utilizado: http://www.nomorepasting.com/, http://pastebin.ca/, http://nopaste.php.cd/ http://rafb.net/paste/ http://sourcepost.sytes.net/, http://extraball.sunsite.dk/notepad.php, y http://www.pastebin.com/.

Bots

Many technically-oriented IRC channels have a non-human member, a so-called *bot*, that is capable of storing and regurgitating information in response to specific commands. Typically, the bot is addressed just like any other member of the channel, that is, the commands are delivered by "speaking to" the bot. For example:

Muchos canales técnicos de IRC tienen un miembro no humano, un tal llamado *bot*, el cual es capaz de almacenar y regurgitar información en respuesta a comandos específicos. El bot se parece a un miembro más del canal, esto es, los comandos se hacen llegar "hablándole" al bot. Por ejemplo:"

```
<kfogel> ayita: learn diff-cmd = http://subversion.tigris.org/faq.html#diff-cmd
<ayita>  Thanks!
```

Esto le ha dicho al bot (el cual está en el canal como ayita) que recuerde cierto URL como la respuesta a la pregunta "diff-cmd". Ahora podemos dirigirnos a ayita pidiendole al bot que le diga a otro usuario acerca de diff-cmd:

```
<kfogel> ayita: tell jrandom about diff-cmd
<ayita>  jrandom: http://subversion.tigris.org/faq.html#diff-cmd
```

Lo mismo puede ser logrado con un comando más corto:

```
<kfogel> !a jrandom diff-cmd
<ayita>  jrandom: http://subversion.tigris.org/faq.html#diff-cmd
```

El conjunto exacto de comandos y conductas difieren entre bots. El ejemplo anterior utiliza `ayita` (http://hix.nu/svn-public/alexis/trunk/), del cual existe una instancia en #`svn` en Freenode. Otros bots son Dancer (http://dancer.sourceforge.net/) y Supybot (http://supybot.com/). No son necesarios privilegios específicos en el servidor para ejecutar un bot. Un bot es un programa cliente; cualquiera puede fijar y dirigirlo para que escuche en un servidor/canal en particular.

Si el canal del proyecto tiende a recibir las mismas preguntas una y otra vez, recomiendo utilizar un bot. Sólo un pequeño porcentaje de usuarios del canal adquirirán la habilidad necesaria para manejar el bot,

pero serán los que sí lo hagan quienes responderán a una cantidad desproporcionada de preguntas, porque el bot permite que sean respondidas con mayor eficiencia.

Archivando IRC

Aunque es posible archivar todo lo que sucede en los canales de IRC, no es algo necesario. Las conversaciones en IRC pueden ser públicas, por lo que muchas personas piensan en ellas como conversaciones informales semi-privadas. Los usuarios puede que no cuiden la gramática y a veces expresen opiniones (por ejemplo, acerca del software o sobre otros desarrolladores) que no querrán que sean preservadas eternamente en un archivo en línea.

Por supuesto que existen *extractos* que deberían ser preservados. Muchos de los clientes de IRC pueden registrar conversaciones a un fichero bajo demanda por el usuario, o si esto falla, se puede copiar y pegar la conversación del IRC a otro foro permanente (a menudo, el bug tracker). Pero el registro indiscriminado puede incomodar a algunos usuarios. Si se archiva todo, hay que declararlo claramente en el tópico del canal y proporcionar una URL del archivo.

Wikis

Un *wiki* es un sitio web que permite a cualquier visitante editar o extender su contenido; el término "wiki" (una palabra Hawaiana que significa "rápido" o "super-rápido") también es usado para la aplicación que permite este tipo de edición. Los wikis fueron inventados en 1995, pero su popularidad alzo vuelo a partir del año 2000 o 2001, impulsado parcialmente por el éxito de la Wikipedia (http://www.wikipedia.org/), un enciclopedia de contenido libre basada en un wiki. Imaginemos un wiki como algo entre IRC y las páginas web: los wikis no trabajan en tiempo real, así que la gente tiene la posibilidad de deliberar y pulir sus contribuciones, pero a la vez son muy sencillos de utilizar, facilitando más la edición que una página web.

Los wikis no son aun equipamiento estándar para los proyectos open source, pero probablemente pronto lo serán. Dado que son una tecnología relativamente nueva y la gente aún experimenta con las diferentes maneras de utilizarlos, sólo ofreceré algunas precauciones —llegados a este punto, es más fácil analizar los usos equivocados de los wikis en lugar de analizar sus exitos.

Si decide ofrecer un wiki, hay que poner un gran esfuerzo en tener una organización clara de las paginas y un diseño visual atractivo, de manera que los visitantes (p.e. editores potenciales) instintivamente sepan como incluir sus contribuciones. Igual de importante, hay que publicar estos estándares en el mismo wiki, de manera que la gente tenga un lugar a donde ir en busca de orientación. Muy a menudo, los administradores de los wikis caen en la trampa de creer que porque hordas de visitantes añaden individualmente contenido de alta calidad al sitio, el resultado de todo esto debe ser también de la más alta calidad y esto no es como funcionan los sitios web. Cada página individual o párrafo puede que sea bueno al ser considerado individualmente, pero no será tan bueno si está encuadrado dentro de un todo desorganizado o confuso. Demasiadas veces, los wikis sufren de:

- **Falta de principios de navegación.** Un sitio web bien organizado hace que los visitantes se sientan como si supieran donde se encuentran en todo momento. Por ejemplo, si las paginas están bien diseñadas, la gente puede intuir las diferencias entre una región con la "tabla de contenidos" y otra con el contenido. Los contribuyentes del wiki respetarán tales diferencias también si y solo si las diferencias están presentes.

- **Información duplicada.** Frecuentemente los wikis acaban con diferentes páginas con información similar, porque los contribuyentes individuales no se han dado cuenta de la duplicidad. Esto puede ser una consecuencia de la falta de principios de navegación mencionados antes, en que la gente puede que no encuentre contenido duplicado si este no se encuentra donde esperaban encontrarlo.

- **Audiencia objetivo inconsistente.** Hasta cierto punto este problema es inevitable cuando existen tantos autores, pero puede ser ralentizado si existen guías escritas acerca de como crear nuevo contenido.

También ayuda editar nuevas contribuciones al principio dando un ejemplo a seguir de manera que los estándares se vayan asentando.

La solución más común para todos estos problemas es el mismo: tener estándares editoriales y mostrarlos no sólo publicándolos sino que editando paginas y adheriendose a estos. En general, los wikis amplificaran cualquier fallo en el material original, ya que los contribuyentes imitaran cualquier patrón que vean. No sólo hay que configurar el wiki y esperar que todo funcione a la perfección. Se debe preparar con contenido bien escrito, de manera que la gente tenga una plantilla que seguir.

El ejemplo más brillantes de un wiki bien llevado es la Wikipedia, aunque esto sea parcialmente a que el contenido (artículos enciclopédicos) sea idóneo para el formato del wiki. Pero si se examina la Wikipedia en profundidad verá que sus administradores han establecido unas fundaciones *muy* estrictas para las contribuciones. Existe una extensa documentación acerca de como añadir nuevo contenido o de como mantener un punto de vista apropiado, los tipos de ediciones que hacer (involucrando varios grados, incluyendo una eventual moderación) y así sucesivamente. También tienen controles de autorización, de manera que si una página es el objetivo de ediciones inapropiadas, pueden bloquearla hasta que el problema sea resuelto. En otras palabras, no pusieron unas cuantas plantillas en un sitio web y se sentaron a esperar. La Wikipedia funciona porque sus fundadores pensaron cuidadosamente acerca de como conseguir que cientos de contribuyentes pudieran adaptar sus escritos a una visión común. Aunque puede que no necesite de toda esta preparación al montar un wiki para un proyecto de software libre, está bien emular el espíritu.

Para más información acerca de los wikis visitad http://es.wikipedia.org/wiki/Wiki. El primer wiki sigue vivo y coleando y contiene mucha información sobre los wikis: ???, http://www.c2.com/cgi/wiki?WhyWikiWorks, y http://www.c2.com/cgi/wiki?WhyWikiWorksNot para varios puntos de vista.

Sitio Web

No hay mucho que decir acerca de los aspectos técnicos del sitio web del proyecto: montar un servidor web y crear las páginas web son tareas sencillas, y los aspectos más importantes acerca del diseño y contenido ya han sido tratados en capítulos anteriores. La principal función del sitio web es ofrecer una visión general clara y unir las otras herramientas (Sistema de control de versiones, gestión de fallos, etc.). Si no se tiene la experiencia suficiente para configurar un servidor web, no será difícil encontrar a alguien que pueda hacerlo y desee ayudar. Sin embargo, para ahorrar tiempo y esfuerzos, es preferible utilizar uno de los sitios web enlatados.

Soluciones de hospedaje

Existen dos ventajas importantes de utilizar sitios preparados. La primera es la capacidad y ancho de banda del servidor. Sin importar cuan exitoso pueda a llegar a ser el proyecto, el espacio en disco no se va a acabar y la conexión no se verá superada. La segunda ventaja es sencillez. Estos sitios ya han seleccionado un gestor de fallos, un sistema de control de versiones, un gestor de listas de correos, archivador y todo lo que sea necesario para llevar un sitio web. Ya han configurado las herramientas y se realizan los respaldos necesarios de los datos almacenados por estas. No es necesario tomar decisiones. Sólo es necesario rellenar un formulario, presionar un botón y se tiene un sitio web así de fácil.

Estos son beneficios muy significativos. La desventaja, por supuesto, es que se debe aceptar *sus* opciones y configuraciones, incluso si algo diferente sería mejor para el proyecto. Por lo general, estos sitios se pueden ajustar bajo ciertos parámetros pero nunca se obtendrá el control total que se tendría si se hubiera hecho en casa teniendo acceso de administrador al servidor.

Un ejemplo perfecto de esto es la gestión de los ficheros generados. Ciertas paginas web del proyecto puede que sean ficheros creados—por ejemplo, existen sistemas para mantener los datos del FAQ en un formato fácil de modificar, desde el cual se pueden generar ficheros HTML, PDF y otros formatos. Al

igual como se explica en “Versiones de todo” anteriormente en este capítulo, no se desean diferentes versiones de los formatos generados, sólo del fichero maestro. Pero cuando el sitio web está hospedado en el servidor de otra persona, puede que sea imposible crear un hook personalizado que permita regenerar la versión HTML pública cada vez que el fichero maestro del FAQ sea modificado. La única solución es tener diferentes versiones de los ficheros generados de manera que aparezcan en el sitio web.

Pueden haber consecuencias más importantes también. Puede que no se tenga el control sobre la presentación deseado. Algunos sitios de hospedaje permiten editar las páginas web, pero el diseño original del sitio termina apareciendo en diversas formas. Por ejemplo, algunos proyectos hospedados en Sourceforge tienen páginas web totalmente personalizadas pero apuntan los enlaces a la página web de Sourceforge para más información. La página en Sourceforge sería la página principal del proyecto si no se hubiera utilizado una personalizada. La página de Sourceforge tiene enlaces al gestor de fallos, repositorio CVS, descargas, etc. Desafortunadamente, una página en Sourceforge también contiene una gran cantidad de ruido de fondo. La parte superior es un anuncio en banner, por lo general, una animación. El lado izquierdo es un arreglo vertical de enlaces con poca relevancia para alguien interesado en el proyecto. El lado derecho es por le general más publicidad. Sólo el centro de la página es dedicado a material específico del proyecto e incluso esto esta organizado de forma confusa lo cual hace que los visitantes no estén seguros de donde pulsar a continuación.

Detrás de cada aspecto individual del diseño de Sourceforge existe sin lugar a dudas una buena razón—buena desde el punto de vista de Sourceforge, como la publicidad. Pero desde el punto de vista individual del proyecto el resultado puede que sea una página web alejada de la ideal. No es mi deseo criticar a Sourceforge; estas mismas preocupaciones se aplican a muchos de estos sitios de hospedaje. El punto es que hay que hacer un sacrificio. Se obtiene el alivio de los aspectos técnicos de llevar el sitio del proyecto, pero con la condición de aceptar la forma de llevarlo de otra persona.

Sólo usted puede decidir acerca de cual sitio de hospedaje es el mejor para el proyecto. Si se decide utilizar un sitio de hospedaje, deje abierta la posibilidad de cambiar a un servidor propio más adelante utilizando nombres de dominio personalizados para la página principal del proyecto. Se puede remitir el URL al sitio hospedado o tener una página totalmente personalizada detrás de la URL pública y llevar a los usuarios al sitio hospedado para funcionalidades más sofisticadas. Sólo asegúrese de organizar las cosas de manera que si decide cambiar de solución para el hospedaje, la dirección del proyecto no deba ser modificada.

Escoger un sitio de hospedaje

El sitio de hospedajes más grande y conocido es SourceForge [http://www.sourceforge.net/]. Otros dos sitios que proveen los mismos servicios sonsavannah.gnu.org [http://savannah.gnu.org/] y BerliOS.de [http://www.berlios.de/]. Algunas organizaciones, como Apache Software Foundation [http://www.apache.org/] y Tigris.org [http://www.tigris.org/][8], ofrecen hospedaje a proyectos open source que encajan su misión y su comunidad con proyectos ya existentes.

Haggen So ha hecho una evaluación exhaustiva de varios sitios de hospedaje como parte de su investigación para su tesis de doctorado titulada *Construcción of an Evaluation Model for Free/Open Source Project Hosting (FOSPHost) sites*. Los resultados se encuentran en http://www.ibiblio.org/fosphost/, y en http://www.ibiblio.org/fosphost/exhost.htm hay un gráfico comparativo.

Anonimato y participación

Un problema que no está estrictamente limitado a los sitios de hospedaje pero que usualmente se encuentra en estos, es el abuso de sus funcionalidades de login. La funcionalidad es suficientemente sencilla en si misma: el sitio permite a cada visitante registrarse con un nombre de usuario y contraseña. A partir de ahí mantiene un perfil para este usuario de manera que los administradores del proyecto puedan asignar ciertos permisos a este usuario, por ejemplo, el derecho de enviar cambios al repositorio.

[8]Disclaimer: Soy empleado de CollabNet [http://www.collab.net/], la cual patrocina Tigris.org, y lo utilizo regularmente.

Esto puede ser extremadamente útil y de hecho es una de las principales ventajas de los sitios de hospedaje. El problema es que a veces, el login de los usuarios termina siendo requerido para tareas que deberían ser permitidas para visitantes anónimos, especialmente la habilidad de añadir bugs en el gestor de fallos o comentar en bugs ya existentes. Al requerir que sólo sean usuarios registrados quienes puedan llevar a cabo estas acciones, el proyecto eleva la vara de participación para lo que debería ser algo rápido y conveniente. Por supuesto, se desea poder contactar con alguien que ha introducido algún dato en un bug en el gestor de fallos, pero sólo con tener un campo donde introducir la dirección de correo electrónico (opcional) debería ser suficiente. Si un nuevo usuario encuentra un fallo y desea reportarlo, se verá molestado por tener que rellenar un formulario para crear una nueva cuenta antes de poder introducir el fallo. Puede que simplemente decida no hacerlo después de todo.

Las ventajas de la gestión de usuarios generalmente superan las desventajas. Pero si se pueden escoger cuales acciones pueden ser hechas anónimamente, asegúrese no sólo de que *todas* las acciones de sólo lectura sean permitidas a visitantes sin registro, pero también algunas acciones de introducción de datos, especialmente en el gestor de fallos y, si se tiene, en el wiki.

Capítulo 4. Infraestructura Social y Política

Las primeras preguntas que la gente se hace sobre el software libre son "¿Cómo funciona? ¿Cómo se mantiene el proyecto? ¿Quién toma las decisiones? Siempre quedo insatisfecho con respuestas conciliadoras sobre la estima del mérito, el espíritu de cooperación, el código que se expresa por si mismo, etc. El caso es que sobre esto no hay una respuesta fácil. La meritocracia, la cooperación, y un código que funciona son partes de ella, pero aportan muy poco para explicar como funciona realmente un proyecto en el andar de todos los días, y nada dice sobre cómo se resuelven los conflictos.

Este capítulo trata de mostrar la estructura subyacente que los proyectos exitosos tienen en comun. Me refiero con el término "exitosos" no solamente a la calidad técnica, sino también a la salud operacional y la capacidad de sobrevivencia. La salud operacional es la capacidad efectiva del proyecto de incorporar las contribuciones de nuevos códigos y nuevos desarrolladores, y de asumir la responsabilidad de los informes de errores que ingresan. Capacidad de sobrevivencia es la posibilidad de que el proyecto exista independientemente de algún participante o auspiciante en particular— tómelo como la posibilidad que tiene el proyecto para continuar aún cuando alguno de sus miembros fundadores tuviera que pasar a ocuparse de otras cosas. El éxito técnico no es difícil de alcanzar, pero sin una base robusta de desarrollo y un fundamento social, un proyecto puede resultar incapaz de manejar el crecimiento que el éxito inicial aporta, o la ausencia de algún individuo carismático.

Hay varias maneras de alcanzar este tipo de éxito. Algunas suponen una estructura formal de supervisión, por la que se resuelven los debates, se aceptan (o rechazan) nuevos desarrolladores, se planifican nuevas características, etc. Otras requieren menos estructura formal, pero más aplicación en conciencia, para producir una atmósfera de armonía en la que la gente puede confiar como una forma*de facto* de supervisión. Ambos caminos llevan al mismo resultado: un sentido de permanencia institucional, ayudado por los hábitos y procedimientos que son bien comprendidos por todos los que participan. Estas características son todavía más importantes en los sistemas que se organizan a si mismos que en aquellos que están controlados centralmente, porque en los sistemas que se organizan a si mismos, cada uno es consciente que unas pocas manzanas pueden arruinar todo el cajón, al menos por un tiempo.

Forkability

El ingrediente indispensable que une a los desarrolladores en un proyecto de software libre, y que los lleva a comprometerse cuando es necesario es la *"forkabilidad"* del código: la capacidad de cada uno de tomar una copia del código fuente y usarlo para abrir un proyecto que compita con el original, evento que se conoce como *"fork"*. Lo que aparece como paradójico aquí es que *la posibilidad* de los "forks" es una fuerza mucho mayor en los proyectos de software libre que los "forks" reales, los que son muy raros. Puesto que un "fork" es malo para todos (por razones que se examinan en detalle en “Forks” en Capítulo 8, *Coordinando a los Voluntarios*), cuanto más seria sea la amenaza de un "fork", tanto mas son las personas que se comprometen a evitarlo.

Los "forks", o más bien la posibilidad de que se produzca un "fork", es la razón por la cual no hay verdaderos dictadores en los proyectos de software libre. Esto puede ser una expresión sorprendente, considerando que es muy común oir que alguien es llamado el "dictador" o el "tirano" en algún proyecto de fuente abierta. Pero esta tiranía es especial, muy diferente de lo que comúnmente se entiende por esa palabra. Imaginaos un rey cuyos súbditos pudieran copiar todo su reino en cualquier momento y trasladarse a la copia para gobernarla como creen que corresponde. ¿No sería el gobierno de ese rey muy diferente de otro cuyos súbditos están obligados a permanecer bajo su gobierno, sin importar lo que él haga?

Por esta razón aún aquellos proyectos que no están organizados formalmente como democracias, son en la práctica democracias en el momento en que se toman las decisiones importantes. La replicabilidad incluye a la "forkability"; "forkability" incluye al consenso. Podría bien darse el caso de que todos quieran apoyarse en un líder (el ejemplo más famoso es el de Linus Torvalds durante el desarrollo del kernel de

Linux), pero esto es porque ellos *así lo eligen*, de una manera ajena a todo cinicismo y en una forma no siniestra. El dictador no tiene un dominio mágico sobre el proyecto. Una propiedad de todas las licencias de fuente abierta es que no se le da a una parte más poder que a cualquier otra para decidir cómo se debe usar o cambiar el código. Si el dictador de repente comenzara a tomar malas decisiones, se produciría una agitación, seguida eventualmente por un levantamiento y por un "fork". Excepto que, por supuesto, muy rara vez las cosas llegan tan lejos, porque antes el dictador busca soluciones de compromiso.

Pero, sólo porque la forkability pone un límite al abuso de poder que uno puede ejercer en un proyecto, eso no quiere decir que no hayan diferencias importantes en el modo como se gobiernan los proyectos. Nadie desea que en todas las decisiones se llegue a la pregunta de última instancia de quien está considerando un fork. Eso pasaría rápidamente a ser muy agobiante, restando energía necesaria para el trabajo efectivo. Las dos secciones que siguen examinan los modos de organizar los proyectos para que la mayoría de las decisiones se tomen naturalmente. Estos dos ejemplos son los casos extremos idealizados; muchos proyectos quedan de alguna manera incluidos entre esos casos.

Dictadores Benevolentes

El modelo de un *dictador benevolente* es precisamente lo que se describe así: La autoridad final de la toma de decisiones reside en una persona, de quien se espera que, por la fuerza de su personalidad o experiencia, la use sabiamente.

Auque el término estándar de esta función es "dictador benévolo" (o *DB*), sería mejor que lo imaginemos como un "árbitro aprobado por la comunidad" o un "juez". En general, los dictadores benevolentes no toman realmente las decisiones, ni siquiera la mayoría de las decisiones. No es probable que una persona pueda tener todo el conocimiento para tomar decisiones buenas y coherentes en todas las áreas de un proyecto, y además, los desarrolladores de calidad no se acercarán al proyecto a no ser que tengan alguna influencia en su dirección. Por lo que los dictadores benevolentes no se comportan como mandones. Por el contrario, dejan que las cosas funcionen por sí solas por el intercambio de ideas y la experimentación, siempre que eso sea posible. Ellos mismos participan en esas discusiones, como un desarrollador cualquiera, a menudo delegando a un administrador de area que tenga mas conocimiento. Solamente cuando queda claro que no se puede alcanzar un consenso, y cuando la mayoría del grupo *desea* que alguien guíe la decisión para que el desarrollo pueda seguir adelante, pisan firme y dicen: "Esta es la forma que tiene que ser". Una característica compartida por casi todos los dictadores benevolentes exitosos es que tienen un rechazo a tomar decisiones con un "así tiene que ser"; esta es una de las razones por la permanecen en la función.

¿Quién puede ser un Buen Dictador Benevolente?

Ser un DB requiere una combinación de características. Se necesita, antes que nada, una cierta delicadeza para juzgar su propia influencia en el proyecto, lo que a su vez lleva a sujetar los primeros impulsos. En los primeros pasos de una discusión uno no debe expresar opiniones y conclusiones con tanta seguridad que los otros sientan que es inútil opinar en contra. La gente debe sentirse libre de ventilar sus ideas, aunque sean tontas. Es inevitable que el DB sugiera alguna idea tonta de vez en cuando, y por lo tanto esta función requiere la disponibilidad de reconocer cuando uno haya tomado una mala decisión— si bien es ésta una característica sencilla que *cualquier* buen desarrollador debe tener, especialmente si permanece en el proyecto por mucho tiempo. Pero la diferencia es que el DB puede darse el lujo de equivocarse de vez en cuando sin tener que lamentar daños permanentes en su credibilidad. Los desarrolladores más jóvenes pueden no tener tanta seguridad, y por eso los DB deben expresar sus críticas o decisiones en contra con mucha delicadeza para contrapesar la fuerza psicológica y técnica que tienen sus palabras.

El DB *no* necesita tener una habilidad técnica superior que supere a todos los que están en el proyecto. Tiene que saber lo suficiente como para trabajar en el código, y entender y comentar cualquier cambio en consideración, y eso es todo. La posición del DB no se adquiere ni mantiene en virtud a una habilidad de codificar intimidatoria. Lo que *si* es importante es la experiencia y un sentido general del diseño —no necesariamente la habilidad de producir un buen diseño a pedido, pero si la habilidad de reconocer el buen diseño, provenga de donde proveniere.

Es común que un dictador benevolente sea el fundador del proyecto, pero esto es más una correlación que una causa. El tipo de cualidades que permite poner en marcha con éxito un proyecto son exáctamente las cualidades que cualquier DB debe tener— competencia técnica, habilidad de persuadir para que otro se una, etc.—. Y por supuesto, los fundadores se inician con una cierta senioridad automática, que puede ser suficiente a menudo para que el dictador benevolente aparezca por el camino de menor resistencia para todos aquellos a quienes les incumbe.

Recordar que la amenaza de un fork vale para los dos sentidos. Un DB puede hacer un fork de un proyecto tan facilmente como cualquier otro, y ocasionalmente lo han hecho, cuando sienten que la dirección que está tomando el proyecto es diferente de donde la mayoría de los desarrolladores quieren ir. Por causa de la forkabilidad, poco importa si el dictador benevolente tiene privilegios de root (que corresponden al administrador del sistema) en el servidor principal del proyecto. A veces la gente se refiere al control del servidor como si fuera la mayor fuente de poder en un proyecto, pero de hecho es irrelevante. La posibilidad de agregar o quitar las palabras clave para hacer commit en un servidor afecta solo a la copia del proyecto que reside en el servidor. Un abuso constante de ese poder, sea por el DB o por cualquier otro, va a terminar simplemente con un cambio del desarrollo en un servidor diferente.

Si el proyecto tendrá un dictador benevolente o si va a funcionar mejor con un sistema menos centralizado, depende ampliamente de quién es el que va a cumplir con esa función. Por lo general es algo muy obvio desde el comienzo saber quién va a ser el DB, y entonces todo se encamina en ese sentido. Pero si no hay un candidoto obvio para el DB, puede ser que el proyecto se incline a usar un proceso descentralizado de tomas de decisión, como se va a describir en la prósima sección.

Democracia basada en el Consenso

A medida que el proyecto avanza, se tiende a pasar del modelo del dictador benevolente a los sistemas más abiertaente democráticos. Este paso no se produce necesariamente por la insatisfacción causada por un DB. Es que el gobierno basado en el grupo llega a ser estable en su evolución, para usar así una metáfora biológica. Siempre que un dictador benevolente se baja o intenta difundir la responsablidad de tomar decisiones entre todos por igual, se da la oportunidad para que el grupo se asiente en un nuevo sistema no-dictatorial—estableciendo una constitución, por así decirlo. Puede ser que el grupo no aprovecha la primera oportunidad, ni quizás tampoco la segunda, pero en algún momento lo hará; y una vez hecho, es muy difícil que esta decisión se vuelva atrás. Y el sentido comun lo explica: si un grupo de N individuos tuviera que investir una persona con poderes especiales, eso significaría que N - 1 personas tuvieron que aceptar que sus influencias individuales se disminuyan. Normalmente la gente no quiere hacer cosas como esa. Y si las hiciera, todavía la dictadura que de allí resulte sería condicional: el grupo que unge a un DB, es claramente el grupo que puede deponer al DB. Por lo tanto, una vez que el proyecto a pasado de un liderazgo carismático individual a un sistema más formal basado en el grupo, muy rara vez vuelve para atrás.

Los detalles de cómo funcionan esos sistemas varían ampliamente, pero hay en ellos dos elementos comunes: uno, el grupo funciona por consencio la mayoría del tiempo; dos, hay un mecanismo formal de votaciones para los casos en que el consenso no puede alcanzarse.

Consenso significa solamente un acuerdo que todos aceptan de una vez por todas. No es un estado ambiguo: un grupo alcanza el consenso en un asunto particular cuando alguien expresa que se ha alcanzado un consenso y nadie contradice esa afirmación. La persona que propone el consenso debe, por cierto, dejar en claro cual es el consenso alcanzado, y que acciones deben tomarse en consecuencia de él, si es que ésto no resulta obvio.

La mayoría de las conversaciones de un proyecto son sobre los asuntos técnicos, como el modo correcto de corregir algún error, la conveniencia o no de agregar un asunto, la forma estricta como un documento se enlaza, etc. Un gobierno basado en el consenso funciona bien porque se entrelaza con la discusión técnica y se confunde con ella silenciosamente. Al terminar una discusión, generalmente hay acuerdo sobre cual es el camino a seguir. Alguien hace una intervención conclusiva, que es al mismo tiempo un resumen de lo que se ha ido decidiendo y queda como una propuesta implícita de consenso. Esto ofrece una última oportunidad para que alguien diga "Un momento, no estoy de acuerdo. Debemos reconside-

rar esto un poco más"

En decisiones de poca importancia que no ofrecen discusión, la propuesta de consenso es implícita. Por ejemplo, cuando un desarrollador hace un commit de una reparación de error, el mismo commit es la propuesta de consenso: "Supongo que todos estamos de acuerdo en que este error debe ser corregido, y esta es la manera de hacerlo." Por supuesto, el desarrollador no lo dice; simplemente hace el commit de la reparación, y los demás no se preocupan de manifestar su acuerdo, porque el silencio es el consentimiento. Si alguien hace el commit de un cambio que resulta *no* tener consenso, se produce simplemente una discusión sobre el cambio como si todavía no estuviera incluido como cambio. La explicación de por qué esto funciona es el tema de la próxima sección.

Control de Versión Significa que Uno Puede Evitar el Estrés

Mantener el código fuente del proyecto bajo el control de versión significa que la mayoría de las decisiones pueden fácilmente deshacerse. La manera corriente para que esto pase es que alguien haga commit de un cambio pensando que todos van a aceptarlo con gusto, y después encontrarse con las objeciones ante el hecho. Una forma típica de esas objeciones es comenzar con las disculpas del caso por no haber intervenido en discusiones anteriores, aunque esto se puede omitir si el discrepante no encuentra registros de tales discusiones en los archivos de la lista de correos. En cualquier caso, no hay motivos para que el tono de la discusión sea diferente después del cambio introducido que antes. Cualquier cambio puede ser revertido, al menos antes de que se introduzcan cambios dependientes (es decir, nuevo código que se daña si el cambio original es quitado de repente). El sistema de control de versión permite que el proyecto deshaga los efectos de malas ideas o propuestas ligeras. Esto, a su vez, le da la libertad a la gente para que confíe en sus instintos y aprenda cuanta consulta es necesaria antes de hacer algo.

También significa que el proceso de consensuar no necesita ser muy formal. Muchos proyectos manejan esto por instinto. Los cambios menores pueden ir sin discusión, o con una discusión mínima seguida por algunos acuerdos. En cambios de mayor importancia, especialmente aquellos que pueden desestabilizar una parte del código, la gente espera uno o dos días antes de suponer que hay consenso. La razón es que nadie puede ser dejado de lado en una conversación importante simplemente por no haber inspeccionado su correo con la frecuencia debida.

Entonces, cuando alguien se siente seguro que sabe lo que tiene que hacer, no para en mientes y lo hace. Esto se aplica no sólo al software fijo, sino a las actualizaciones de la Web, a cambios en la documentación y a cualquier otra cosa que no sea controversial. Generalmente se darán pocos casos en los que la acción tenga que ser deshecha, y estos pueden ser tratados individualmente en cada caso. Por supuesto que no se debe incentivar a la gente para que sea obstinada. Hay todavía una diferencia psicológica entre una decisión bajo discusión y una que ya haya tenido efecto, por más que se diga que es técnicamente reversible. La gente siente que el momento es un aliado de la acción, y que se sentirán más reacios a revertir un cambio que a prevenirlo en el primer instante. Si un desarrollador se abusa de este principio y rápidamente hace commits de cambios que generan controversia, ciertamente la gente puede y debe quejarse, y mantener a ese desarrollador en un estándar estricto hasta que las cosas mejoren.

Cuando No Se Puede Tener Consenso, Vote

Inevitablemente, algunos debates no llegarán al consenso. Cuando no haya otro medio de salir del callejón, la solución es votar. Pero antes que se llegue a la votación, debe aclararse unas cuantas opciones del ballotage. De nuevo en este caso el proceso de discusión técnica se integra suavemente con los procedimientos de toma de decisión del proyecto. El tipo de asuntos que llega a votación implican a menudo temas complejos, llenos de facetas. En cualquiera de tales discusiones complicadas, hay a menudo una o dos personas que hacen las veces de *negociador honesto*: aportan periódicamente la síntesis de los argumentos y siguen las líneas de los puntos centrales del desacuerdo (y del acuerdo). Estas síntesis ayudan a que todos estimen el progreso que se va haciendo, y les recuerda a todos cuáles asuntos quedan pendientes. Estas síntesis podrán servir como modelos para una propuesta de votación, en caso de que ésta se vuelva necesaria. Si los negociadores honestos se han desempeñado bien en su oficio, estarán en condi-

ciones de llamar a votación cuando llegue el tiempo, y todos querrán usar las propuestas vertidas en esas síntesis para organizar la votación. Los negociadores también serán partícipes del debate; no es necesario que ellos queden fuera de la votación, en tanto puedan entender y representar los puntos de vista de los demás, y no dejen que sus sentimientos partidarios les impidan producir síntesis del estado del debate en una forma neutral.

Normalmente la organización de la votación no cae en la controversia. Cuando llega el tiempo de votar, el desacuerdo ha sido analizado y reducido a unas pocas cuestiones, bien etiquetadas y acompañadas de descripciones concisas. De vez en cuando un desarrollador hará una objeción sobre la forma de votar. A veces esta preocupación es legítima, por ejemplo, cuando una opción importante ha sido dejada de lado o no ha sido presentada con precisión. Pero otras veces un desarrollador puede tratar de impedir lo inevitable, quizás porque se da cuenta que el voto no va acompañar su idea. Ver“Gente difícil” en Capítulo 6, *Communications* para ver como tratar este tipo de obstruccionismo.

Recuerde de especificar el sistema de votación, puesto que hay varias formas, y la gente puede tener falsas expectativas sobre el procedimiento que va a ser usado. Una buena opción es la *votación por aprobación*, en la que cada votante puede votar por todas las opciones que quiera, dentro de las opciones presentadas. La votación por aprobación se resuelve simplemente explicando y contando, y a diferencia de otros métodos, solo requiere una ronda de votación. Ver http://en.wikipedia.org/wiki/Voting_system#List_of_systems para mas detalles acerca de la votación por aprobación y otros sistemas de votación, pero tratar de no caer en un debate largo sobre cuál deba ser el sistema que se use (ya que se verán atrapados en el círculo de tener que votar para decidir cómo votar!) Una razón para defender la votación por aprobación como una buena opción es que es difícil que alguien se oponga—es lo más transparente que puede ser una votación.

Finalmente, voto secreto, voto abierto. No hay necesidad de guardar secretos o aparecer como anónimos en una votación sobre asuntos que se han debatido públicamente. Cada participante pone su voto en la lista de correo del proyecto, de modo que cualquier observador pueda hacer el conteo y verificar el resultado, y que todo quede archivado.

Cuando Se Debe Votar

Lo más difícil en la votación es determinar cuando se debe votar. Generalmente la votación tiene que ser algo fuera de lo común—el último resorte cuando todas las otras opciones han fallado. No tome a la votación como el gran camino para resolver los debates. No lo es. Finaliza la discusión, y por tanto finaliza el pensamiento creativo sobre el problema. Mientras la discusión está en el tapete, existe la posibilidad de que alguien aporte una solución nueva, que sea del agrado de todos. Sorprendentemente, esto ocurre a menudo: un debate abierto puede producir un giro nuevo del pensamiento sobre el problema, y llevar a una propuesta que eventualmente satisfaga a todos. Aún cuando no surja una propuesta nueva, todavía es mejor negociar una solución de compromiso que poner un voto. Luego de una solución de compromiso, todos quedan algo insatisfechos, mientras que después de una votación unos quedan contentos y otros en desánimo. Desde un punto de vista político, la primera situación es preferible: al menos cada uno puede sentir que su desánimo es el precio de su accionar. Puede estar insatisfecho, pero todos lo están.

La ventaja principal de la votación es que se cierra la cuestión y se puede seguir adelante. Pero el arreglo se hace por un conteo de votos, en lugar de un diálogo racional que conduzca a todos a la misma conclusión. Cuanto más experiencia tiene la gente en proyectos de fuente abierta, les encuentro menos dispuestas a querer arreglar las cuestiones por medio de la votación. Tratarán primero de explorar las soluciones que previamente no hayan sido consideradas, o entrar en soluciones de compromiso más ajustadas de lo que planearon en un comienzo. Hay varias técnicas para prevenir una votación prematura. La más obvia es decir simplemente "no creo que ya estemos listos para una votación", y explicar por qué no. La otra es pedir que sin compromiso se levanten las manos. Si la respuesta tiende claramente hacia un lado, necesariamente va a inclinar al otro grupo a querer encontrar soluciones de compromiso, obviando así la necesidad de la votación formal. Pero la manera más efectiva es simplemente ofrecer una solución nueva, o un nuevo punto de vista para una sugerencia antigua, de modo que la gente se re-conecte con los temas en lugar de repetir meramente los mismos argumentos.

En algunos casos raros, todos pueden concordar que las soluciones de compromiso presentadas son perores que cualquiera de las soluciones en consideración. Cuando esto ocurre, la votación no es tan objetable, por un lado porque es muy probable que se va a llegar a una solución superior, y por otro porque la gente no se va a desanimar con el resultado, cualquiera sea la opción que gane. Aún en estos casos, no hay que apurarse en votar. La discusión que arriba en una votación es lo que educa al electorado, y detener pronto la discusión puede disminuir la calidad del resultado.

(Fijarse que este consejo de ser reacio a las votaciones no se aplican a la votación sobre cambio-inclusión que se describe en “Stabilizing a Release” en Capítulo 7, *Packaging, Releasing, and Daily Development*. Allí, la votación es más bien un mecanismo de comunicación, un medio de registrar el propio compromiso en el proceso de revisión de cambio de modo que todos puedan decir cuánta revisión ha recibido un cambio dado.)

¿Quién Vota?

Al tener un sistema de votación aparece la cuestión del electorado: ¿A quién le corresponde votar? Este asunto puede convertirse en delicado, porque fuerza a que el proyecto reconozca oficialmente que hay gente con mayor compromiso, o con mejores apreciaciones que los otros.

La mejor solución es simplemente tomar la distinción existente, el acceso a los commits, y asociar los privilegios del voto en eso. En proyectos en que existan accesos completos y parciales a los commits, la cuestión de permitir el voto a los que tienen commit parcial dependerá en gran manera de los procesos por los que el commit parcial fue otorgado. Si el proyecto lo maneja con liberalidad, por ejemplo como una manera de mantener muchas herramientas de contribución de terceras partes en el repositorio, entonces debe dejarse en claro que el acceso al commit parcial hace referencia a los commits, no a la votación. Naturalmente la implicación inversa se mantiene: puesto que los que tienen commit completo *tendrán* privilegios de votación, deben elegirse no solo como programadores, sino también como miembros del electorado. Si alguien muestra tendencias disruptivas u obstruccionistas en la lista de correo, el grupo debe ser muy cauto en incluirlo entre los que hacen commits, auque sea una persona capacitada técnicamente.

E sistema de votación debe ser usado para elegir a los nuevos miembros que hacen commit, sea completo o parcial. Y aquí aparece una de las circunstancias raras en donde el voto secreto es apropiado. No pueden ponerse los votos para los que hacen commits en una lista de correo pública, porque se pueden herir los sentimientos y la reputación de un candidato. En lugar de eso, la forma común es que los que tienen voto lo pongan en una lista de correo privada donde solamente estén los que pueden hacer commits, para proponer que alguien sea habilitado para hacer commits. De esta manera todos pueden expresarse libremente, sabiendo que la discusión es privada. A menudo no habrá desacuerdo, y no se necesitará votar. Luego de esperar unos días para asegurarse que todos tuvieron oportunidad de responder, el proponente envía un mail al candidato y le ofrece el acceso a los commits. Si hay desacuerdo, se inicia una discusión como para cualquier otro asunto, con la posibilidad de terminar en una votación. Para que este proceso sea abierto y transparente, tambien tiene que ser secreto el hecho que hay una discusión en curso. Si la persona en consideración sabe lo que está ocurriendo, y luego no se le ofrece un acceso de commit, puede concluir que él ha perdido el voto, y sentirse herido por ello. Por supuesto, si alguien explícitamente pide el acceso al commit, entonces no hay nada que hacer sino considerar la propuesta y explícitamente aceptarle o rechazarle. Si ocurre lo segundo, tiene que hacerse con sumo tacto, con una explicación clara: "Nos agradan tus aportes, pero todavía no hemos visto lo suficiente", o "Hemos tenido en cuenta todos tus aportes, pero se han tenido que hacer considerables ajustes antes de poder aplicarlos, por lo que todavía no nos sentimos confiados para darte el acceso al commit. Esperamos que esto cambie con el tiempo". Recordar que lo que se dice puede caer como un golpe, dependiendo del grado de confianza que se tenga con la persona. Tratar de verlo desde su punto de vista, en el momento que se escribe el mail.

Puesto que agregar un nuevo miembro que pueda hacer commits es una decisión más secuencial que otras decisiones, algunos proyectos tienen requerimientos especiales para el voto. Por ejemplo, puede requerirse que la propuesta reciba por lo menos *n* votos positivos y que no tenga ningún voto negativo, o que cierta supermayoría vote a favor. Los parámetros exactos no son importantes; la idea principal es

que el grupo debe ser cuidadoso al otorgar acceso a los commits. Similarmente, o todavía más estrictamente, se aplican requerimientos especiales a la votación para *quitar* el acceso a los commits, y ojalá que eso nunca sea necesario. Ver“Committers” en Capítulo 8, *Coordinando a los Voluntarios* para más aspectos sobre la no votación para agregar o quitar acceso a los commits.

Encuestas Versus Votaciones

Para ciertas clases de votaciones, puede ser útil expandir el electorado. Por ejemplo, si los desarrolladores no tienen una idea clara para decidir si una interfase dada se adapta al modo como la gente realmente usa el software, una solución es hacer una votación entre todos los suscriptos en la lista de correo del proyecto. Estas son realmente *encuestas* más que votaciones, pero los desarrolladores pueden acordar que los resultados sean vinculantes. Como en cualquier votación, hay que asegurarse de informar a los participantes que hay una opción escrita: si a alguien se le ocurre una opción mejor que no está en la lista, su respuesta puede llegar a ser el resultado más importante de la votación.

Vetos

Algunos proyectos permiten un tipo especial de voto que se conoce como *veto*. El veto es la manera que tiene un desarrollador para detener un cambio apresurado o mal considerado, por lo menos por un tiempo suficiente para que todos puedan discutirlo más. Entender el veto como algo que está entre una objeción fuerte y una discusión sin fin. El sentido exacto del veto varía de un proyecto a otro. Algunos proyectos hacen que sea difícil contrarrestar un veto; otros permiten que sea superado por el voto de una simple mayoría, quizás luego de una forzada demora producida por más discusión. Un veto debe ser acompañado por una explicación exhaustiva; el veto presentado sin esa explicación debe ser considerado inválido.

Junto con los vetos se introduce el problema del abuso del veto. A veces los desarrolladores están demasiado ansiosos en levantar la presión con el pedido de veto, cuando lo que realmente se requiere es más discusión. Se puede evitar el abuso del veto empezando por ser uno mismo contrario al uso del veto, y haciendo notar con tacto cuando alguien usa el veto muy a menudo. Si fuera necesario, se puede recordar para el grupo que los vetos tienen fuerza de obligación siempre y cuando el grupo esté de acuerdo—después de todo, si una gran mayoría de desarrolladores quieren X, de una u otra manera van a conseguir X. O bien el desarrollador que propuso el veto lo retira, o el grupo va a quitarle fuerza al significado del veto.

A veces se escribe un "-1" para contabilizar el veto. Esta costumbre viene de la Fundación del Software Apache, quienes tienen un proceso muy estructurado de votos y votaciones, que está en http://www.apache.org/foundation/voting.html. Las normas de Apache se han difundido a otros proyectos, y se pueden ver sus acuerdos usados de distinta forma en muchos lugares del mundo de la fuente abierta. Técnicamente "-1" no siempre indica que hay un veto formal de acuerdo a las normas de Apache, pero informalmente se considera que representa un veto, o por lo menos una objeción muy fuerte.

Igual que con las votaciones, los vetos se pueden aplicar con efectos retroactivos. No es correcto rechazar un veto porque el cambio en cuestión haya sido puesto en commit, o porque la acción está asumida (a no ser que se trate de algo irrevocable, como por ejemplo una edición de prensa). Por otro lado, un veto que llega semanas, o meses tarde no tiene la posibilidad de que se lo tome muy en serio, ni tendría que ser así.

Tomando Nota de Todo

En cierto momento, el número de acuerdos y arreglos que circulan por el proyecto pueden llegar a ser tan grandes que se necesita registrarlos en algún lugar. Y para dar legitimidad a esos documentos, hay que tener bien claro que están basados el las discusiones de las listas de correo y en acuerdos que ya estaban en vigencia. Cuando se los escribe, se hace referencia a las líneas de los archivos de la lista de correo y cada vez que no se está seguro sobre un punto, se pregunta de nuevo. El documento no debe contener sorpresas: Éste no es fuente de los acuerdos, sino solamente la descripción de ellos. Por supuesto,

si es aceptado, la gente comenzará a citarlo como si fuera una fuente de autoridad, pero eso sólo significa que refleja con exactitud la voluntad de todos los del grupo.

Este es el documento aludido “Pautas de Desarrollo” en Capítulo 2, *Primeros Pasos*. Naturalmente, cuando el proyecto recién comienza, se tendrá que esbozar una guía, sin que por esto se excluya la confección de una posterior historia del proyecto. Pero, a medida que la comunidad madura, se pueden hacer ajustes del lenguaje para reflejar la manera final a la que se ha llegado.

No se debe pretender que uno lo ha dicho todo. Ningún documento puede captar todo lo que la gente necesita saber para participar en un proyecto. Muchos de los acuerdos del proyecto permanecen tácitos, nunca explicitados, sin embargo aceptados por todos. Algunas cosas son muy obvias para incluirlas, y resultarían distractivas al lado del material que no es obvio y es importante. Por ejemplo, no tiene sentido escribir en la guía una instrucción como "Sea educado y respetuoso con los otros miembros de la lista de correos, y no incite a las discusiones acaloradas" o "Escriba código sin errores, claros y limpios." Por supuesto que son cosas deseables, pero no existe un universo concebible donde estas cosas *no* sean deseables, por lo que no vale la pena mencionarlas. Si alguien es grosero en la lista de correos, o escribe el código con errores, no van a dejar de hacerlo sólo porque la guía del proyecto lo dice. Estas situaciones requieren atención en el momento que aparecen, y no bastan las normas generales que dicen que hay que ser buenos. Además, si el proyecto tiene líneas específicas sobre *cómo* escribir un código bueno, entonces esas líneas de la guía deber escribirse con todo el detalle que sea posible.
.

Una buena manera de determinar qué debe incluirse en el documento es referirse a las preguntas que los recién llegados hacen, y a las quejas de los desarrolladores con experiencia. Esto no quiere decir que necesariamente tienen que convertirse en un informe FAQ—posiblemente el documento necesita una estructura narrativa más coherente que la que puede ofrecer el FAQ. Tiene entonces que seguir el mismo principio basado en la práctica, que hay que incluir asuntos que realmente se producen, y no tanto tratar de anticiparse a los asuntos que pueden producirse.
.

Si el proyecto tiene un dictador benévolo, o si tiene miembros con poderes especiales (presidente, secretario general, o lo que sea), entonces el documento es una buena oportunidad de escribir los procedimientos de la sucesión de poderes. A veces eso puede ser tan simple como nombrar cierta gente como reemplazantes en el caso en que el DB abandone el proyecto por alguna razón. Generalmente, si hay un DB, es sólo él quien puede decidir el nombre de un sucesor. Si se elige una comisión, entonces el procedimiento de la elección y el nombramiento de los integrantes de la comisión tiene que estar descrito en el documento. Si al comienzo no existe un procedimiento, entonces hay que conseguir un consenso en la lista de correos *antes* de escribir sobre el procedimiento. Hay gente que puede ser sensible con las estructuras jerárquicas, por lo que el asunto tiene que ser tratado con delicadeza.

Quizás lo mas importante es dejar en claro que las reglas pueden ser reconsideradas. Si los acuerdos descritos en el documento comienzan a frenar el proyecto, recordar a todos que fue pensado como una reflexión viviente de las intenciones del grupo, no para provocar frustración y bloqueo. Si alguien toma por hábito pedir que las reglas se reconsideren cada vez que una regla se considera, no siempre conviene debatir el tema con ella—a veces el silencio es la mejor táctica. Si hay mas de uno que se une a las quejas, la campana ha sonado, y será lógico suponer que algo necesita ser cambiado. Si nadie se une a la queja, entonces esa persona no representa a nadie, y las reglas quedarán como están.

Dos buenos ejemplos de una guía para un proyecto es Subversion `hacking.html` en http://svn.collab.net/repos/svn/trunk/www/hacking.html, y los documentos de gobierno de la Fundación de Software Apache, en http://www.apache.org/foundation/how-it-works.html y http://www.apache.org/foundation/voting.html. La Fundación de Software Apache es de hecho una colección de proyectos de software, organizada legalmente como una corporación sin fines de lucro, de modo que sus documentos tienden a describir los procedimientos de gobierno más que las convenciones de desarrollo. Vale la pena leerlas, porque representan una experiencia acumulada en muchos proyectos de fuente abierta.

Capítulo 5. Dinero

Este capitulo examina como conseguir fondos en un entorno de software libre. Esta dirigido no solo a los desarrolladores que se les paga por trabajar en proyectos de software libre, sino tambien a los directores, quienes necesitan comprender la dinámica social de el entorno de desarrollo. En las secciones que siguen, el destinatario ("tu") significa tanto un desarrollador que cobra como a aquel que coordina a tales desarrolladores. El consejo a menudo será el mismo para ambos; cuando no sea así, la audiencia pretendida quedará clara con el contexto.

Los fondos corporativos de un desarrollo de software libre no son un nuevo fenomeno. Muchos de los desarrollos han estado siempre informalmente subvencionados. Cuando un administrador de sistemas escribe una herramienta de análisis de sistemas para ayudarle en su trabajo, entonces la pone online y consigue corregir bugs y contribuciones con nuevas características de otros administradores de sistemas, lo que ha ocurrido es que se ha creado un consorcio no oficial. Los fondos del consorcio provienen de los sueldos de los sysadmins, y su espacio de oficina y ancho de banda son donados, aunque desconociéndolo la organización para la que ellos trabajan. Aquellas organizaciones se benefician de la inversión aunque ellas, institucionalmente no son conscientes de ello al principio.

Hoy la diferencia, es que muchos de estos esfuerzos estan siendo formalizados. Las corporaciones se están concienciando de los beneficios de el software open source, y por ello empiezan a involucrarse ellas mismas en su desarrollo. Los desarrolladores tambien llegan a esperar que los proyectos importantes atraigan al menos donaciones, y posiblemente incluso sponsors de gran duración. Mientras que la presencia del dinero no ha cambiado la dinámica básica del desarrollo del software libre, ha cambiado mucho la escala a la cual ocurren las cosas, ambas en términos de número de desarrolladores y tiempo por desarrollador. Tambien ha tenido efecto en como son organizados los proyectos, y en como las partes envueltas en ellos interactuan. La cuestión no es meramente sobre como se gasta el dinero, o en medir como se devuelven las inversiones. Sino tambien en las administraciones y procesos: como pueden las estructuras de mando jerárquico de las corporaciones y las comunidades de voluntarios semi-descentralizados de proyectos de software libre trabajar productivamente uno con otro? ¿Tendrán ellos que acordar incluso el significado de "productivo"?

El patrocinio financiero es, en general, bienvenido por las comunidades de desarrollo de open source. Puede reducir la vulnerabilidad de un proyecto a las fuerzas del Caos, el cual arrebata tantos proyectos antes de que ellos salgan a la tierra, y de ahí puede hacer a la gente más dispuesta a darle al software una oportunidad; ellos sienten que estan invirtiendo su tiempo en algo que todavía les llevará seis meses desde ahora. Después de todo, la credibilidad es contagiosa, hasta cierto punto. Cuando se dice, IBM apoya un proyecto Open Source, la gente más o menos asume que al proyecto no se le permitirá fallar, y su buena voluntad resultante dedicará los esfuerzos a ello para que pueda hacerse como una profecía que se cumple por su propia naturaleza.

Sin embargo, los fondos tambien traen una percepción de control. Si no se manejan cuidadosamente, el dinero puede dividir un proyecto en grupos incluyentes y grupos excluyentes de desarrolladores. Si los voluntarios no remunerados tienen el sentimiento que las decisiones de diseño o adición de características están simplemente disponibles para el mejor postor, se marcharan a un proyecto que se parezca más a una meritocracia y menos a un trabajo sin pagar para el beneficio de alguien. Puede que ellos nunca se quejen patentemente en las listas de correo. En vez de eso, simplemente habrá menos y menos ruido de fuentes externas, como los voluntarios gradualmente pararán de intentar ser tomados seriamente. El rumor de la actividad a pequeña escala continuará, en la forma de informes de fallos y ocasionalmente pequeños arreglos. Pero no habrá ninguna contribución con gran código o participación externa en discusiones de diseño. La gente siente que es lo que se espera de ellos, y viven (o se deprimen) en esas esperanzas.

Aunque el dinero necesita ser usado cuidadosamente, esto no significa que no se pueda comprar influencia. Desde luego puede. El truco es que no puede comprar la influencia directamente. En una transación comercial sencilla, cambias dinero por lo que quieras. Si necesitas añadir una característica, firmas un contrato, pagas por ello, y lo tienes hecho. En un proyecto Open Source no es tan simple. Tu puedes fir-

mar un contrato con algunos desarrolladores, pero ellos serían idiotas consigo mismos, y tú, si ellos garantizan que el trabajo por el que tu has pagado será aceptado por la comunidad de desarrollo simplemente porque tu pagaste por él. El trabajo únicamente puede ser aceptado por sus propios méritos, y es como encaja en la visión de la comunidad por el software. Puede que tengas algo que decir en esta visión, pero no serás la única voz.

Por lo tanto, el dinero no puede comprar influencia, pero puede comprar cosas que *llevan a* la influencia. El ejemplo más obvio son los programadores. Si los buenos programadores son contratados, y aguantan bastante como para conseguir experiencia con el software y credibilidad en la comunidad, entonces ellos pueden influenciar en el proyecto de la misma manera que cualquier otro miembro. Tendrán voto o si hay muchos de ellos, tendrán un bloque de votaciones. Si ellos son respetados en el proyecto, tendrán influencia más alla de sus votos. No hay necesidad de que los desarrolladores con sueldo disimulen sus motivos, tampoco. Después de todo, todo el mundo que quiere que se haga un cambio en el software lo quiere por alguna razón. Las razones de tu compañia no son menos legítimas que las de cualquiera. Es simplemente que el peso dado a los objetivos de tu compañia será determinado por el estatus de sus representantes en el proyecto, no por el tamaño de la compañia, presupuesto o plan de negocios.

Tipos de participación

Existen múltiples razones diferentes por las cuales los proyectos open source consiguen fondos. Los elementos de esta lista no se excluyen mutuamente; a menudo, la financiación de un proyecto será el resultado de muchos, o incluso todas de estas motivaciones:

Compartiendo la carga
: Distintas organizaciones con necesidades de software similares, a menudo se encuentran a si mismas duplicando esfuerzos, tanto escribiendo código interno similar, o comprando productos similares de vendedores propietarios. Cuando se dan cuenta de lo que ocurre, las organizaciones pueden reunir sus recursos y crear (o entrar) en un proyecto Open Source adaptado a sus necesidades. Las ventajas son obvias: el costo de desarrollo se divide pero los beneficios se acumulan entre todos. Aunque este escenario parezca más intuitivo para no lucrarse, puede crear un sentido estratégico incluso para los competidores que quieren sacar beneficio.

  Ejemplos: http://www.openadapter.org/, http://www.koha.org/

Aumentando servicios
: Cuando una compañía vende servicios de los cuales depende, o se hacen más atractivos por, programas open source particulares, naturalmente en los intereses de esta compañía está asegurar que esos programas sean activamente mantenidos.

  Ejemplo: CollabNet's [http://www.collab.net/] soporte de http://subversion.tigris.org/ (descargo: este es mi trabajo diario, pero es tambien un ejemplo perfecto de este modelo).

Apoyando las ventas de hardware
: El valor de los ordenadores y sus componentes está directamente relacionado con la cantidad de software disponible para ellos. Los vendedores de hardware no sólo venden máquinas completas, pero tambien los creadores de dispositivos periféricos y microchips han descubierto que teniendo software libre de gran calidad para ejecutarse en su hardware es tambien una parte importante para los clientes.

Socavando a la competencia
: A algunas empresas patrocinan ciertos proyectos OSS como una manera de socavar los productos de la competencia, que puede que sean o no OSS. Quitar cuotas de mercado de la competencia no es por lo general la única razón para involucrarse en un proyecto, pero si puede ser un factor importante.

  Ejemplo: http://www.openoffice.org/ (no, esta no es la única razón por la cual OpenOffice existe,

pero el software en si es parcialmente una respuesta a Microsoft Office).

Marketing
: Ser asociado con un proyecto OSS popular puede que genere muy buena publicidad.

Licencias Duales
: *Licenciamiento Dual* es una práctica bajo la cual se ofrece el software utilizando una licencia propietaria tradicional para clientes quienes deseen revenderlo como parte de otra aplicación propietaria, y simultaneamente bajo una licencia libre para aquellos quienes pretenden utilizarlo bajo los terminos del software libre (más en “Dual Licensing Schemes” en Capítulo 9, *Licencias, Copyrights y Patentes*). Si la comunidad de desarrolladores de software libre es activa, el programa recibe los beneficios del desarrollo y búsqueda de fallos de amplio espectro mientras la compañia sigue obteniendo beneficios por las regalías para mantener algunos desarrolladores a tiempo completro.

  Dos ejemplos muy conocidos son MySQL [http://www.mysql.com/], creadores de la base de datos con el mismo nombre y Sleepycat [http://www.sleepycat.com/], que distribuye y brinda soporte para la base de datos Berkeley. No es ninguna coincidencia que las dos sean empresas de bases de datos. Las bases de datos suelen ser integradas dentro de otras aplicaciones en lugar de ser vendidas directamente a los usuarios, por lo que son perfectas para el modelo de licencia dual.

Donaciones
: Un proyecto popular puede a veces obtener contribuciones significativas, tanto de individuos como de organizaciones, sólo con colocar un botón de donaciones en línea o a veces vendiendo productos promocionales del proyecto como tazas de cáfe, camisetas, alfombrillas, etc. Pero precaución, si el proyecto ha de aceptar donaciones hay que planear como será utilizado ese dinero *antes* de que llegue y publicar esto en la página web del proyecto. Las discusiones acerca de hacia donde debe ser dirigido el dinero tienden a ser más distendidas antes de que este se tenga; de todas formas, si existen importantes desacuerdos, es mejor averiguarlo mientras se sigue siendo algo académico.

El módelo de negocio del beneficiario no es el único factor en como este se relaciona con la comunidad open source. La relación historica entre los dos es tambien importante: ¿inicio la empresa el proyecto o se ha unido luego? En cualquiera de los dos casos, el beneficiario deberá ganar credibilidad, pero, no sorprendentemente, será necesario un mayor esfuerzo en el segundo caso. La organización debe tener claros objetivos con respecto al proyecto. ¿Intenta la empresa mantener una posición de liderazgo o simplemente intenta ser una voz dentro de la comunidad para guiar sin gobernar la dirección del proyecto? ¿O sólo desea tener un par de colaboradores que sean capaces de resolver los problemas de los usuarios e incluir sus cambios en la distribución pública sin mucho jaleo?

Mantened estas preguntas en mente mientras continua leyendo las siguientes directrices. Estan pensadas para ser aplicadas a cualquier tipo de participación empresarial dentro de un proyecto open source, pero teniendo en cuenta que todo proyecto es un entorno humano, por lo cual, ninguno es igual. Hasta cierto grado, habrá que seguir nuestro instinto, pero seguir estos principios aumentaran las posibilidades de que las cosas funcionen como queremos.

Contratos Indefinidos

Si está dirigiendo un equipo de desarrolladores en un proyecto de software libre, intente mantenerlos el tiempo suficiente para que adquieran experiencia técnica y política—un par de años como mínimo. Por supuesto que ningún proyecto, sea de código abierto o cerrado, se beneficia del intercambio incesante de programadores. La necesidad de que un recien llegado deba aprender todo de nuevo cada vez puede crear un ambiente disuasorio. Pero el castigo puede ser mayor para un proyecto de código abierto porque quienes abandonan el proyecto no sólo lo hacen con el conocimiento del código, tambien se llevan un status en la comunidad y las relaciones que hayan hecho.

The credibility a developer has accumulated cannot be transferred. To pick the most obvious example, an incoming developer can't inherit commit access from an outgoing one (see “Money Can't Buy You Love” later in this chapter), so if the new developer doesn't already have commit access, he will have to

submit patches until he does. But commit access is only the most measurable manifestation of lost influence. A long-time developer also knows all the old arguments that have been hashed and rehashed on the discussion lists. A new developer, having no memory of those conversations, may try to raise the topics again, leading to a loss of credibility for your organization; the others might wonder "Can't they remember anything?" A new developer will also have no political feel for the project's personalities, and will not be able to influence development directions as quickly or as smoothly as one who's been around a long time.

La credibilidad acumulada por un desarrollador no puede ser transferida. El ejemplo más obvio es que un desarrollador recien incorporado no puede heredar los mismo accesos al código de quien se va (más en “Money Can't Buy You Love”), así que si el nuevo desarrollador no tiene permisos para realizar cambios, deberá enviar parches hasta que tenga estos permisos. Pero este nivel de acceso es sólo una manifestación cuantitativa de la perdida de influencia. Un desarrollador veterano tambien conoce los viejos temas que han sido tratados una y otra vez en las listas de discusión. Uno nuevo, sin tener conocimiento de estas conversaciones quizas intente sacar a flote de nuevo estos temas, llevando a una perdida de credibilidad; otros pueden que piensen: "¿Acaso esta gente no puede recordar nada?". Una nueva persona tampoco tendrá ninguna sensación política hacia las personalidades del proyecto, y no podrá influenciar la dirección del desarrollo tan rápida o sin problemas como alguien quien lleva largo tiempo en el proyecto.

Train newcomers through a program of supervised engagement. The new developer should be in direct contact with the public development community from the very first day, starting off with bug fixes and cleanup tasks, so he can learn the code base and acquire a reputation in the community, yet not spark any long and involved design discussions. All the while, one or more experienced developers should be available for questioning, and should be reading every post the newcomer makes to the development lists, even if they're in threads that the experienced developers normally wouldn't pay attention to. This will help the group spot potential rocks before the newcomer runs aground. Private, behind-the-scenes encouragement and pointers can also help a lot, especially if the newcomer is not accustomed to massively parallel peer review of his code.

When CollabNet hires a new developer to work on Subversion, we sit down together and pick some open bugs for the new person to cut his teeth on. We'll discuss the technical outlines of the solutions, and then assign at least one experienced developer to (publicly) review the patch that the new developer will (also publicly) post. We typically don't even look at the patch before the main development list sees it, although we could if there were some reason to. The important thing is that the new developer go through the process of public review, learning the code base while simultaneously becoming accustomed to receiving critiques from complete strangers. But we try to coordinate the timing so that our own review comes immediately after the posting of the patch. That way the first review the list sees is ours, which can help set the tone for the others' reviews. It also contributes to the idea that this new person is to be taken seriously: if others see that we're putting in the time to give detailed reviews, with thorough explanations and references into the archives where appropriate, they'll appreciate that a form of training is going on, and that it probably signifies a long-term investment. This can make them more positively disposed toward that developer, at least to the degree of spending a little extra time answering questions and reviewing patches.

Appear as Many, Not as One

Your developers should strive to appear in the project's public forums as individual participants, rather than as a monolithic corporate presence. This is not because there is some negative connotation inherent in monolithic corporate presences (well, perhaps there is, but that's not what this book is about). Rather, it's because individuals are the only sort of entity open source projects are structurally equipped to deal with. An individual contributor can have discussions, submit patches, acquire credibility, vote, and so forth. A company cannot.

Furthermore, by behaving in a decentralized manner, you avoid stimulating centralization of opposition. Let your developers disagree with each other on the mailing lists. Encourage them to review each other's code as often, and as publicly, as they would anyone else's. Discourage them from always voting as a bloc, because if they do, others may start to feel that, just on general principles, there should be an orga-

nized effort to keep them in check.

There's a difference between actually being decentralized and simply striving to appear that way. Under certain circumstances, having your developers behave in concert can be quite useful, and they should be prepared to coordinate behind the scenes when necessary. For example, when making a proposal, having several people chime in with agreement early on can help it along, by giving the impression of a growing consensus. Others will feel that the proposal has momentum, and that if they were to object, they'd be stopping that momentum. Thus, people will object only if they have a good reason to do so. There's nothing wrong with orchestrating agreement like this, as long as objections are still taken seriously. The public manifestations of a private agreement are no less sincere for having been coordinated beforehand, and are not harmful as long as they are not used to prejudicially snuff out opposing arguments. Their purpose is merely to inhibit the sort of people who like to object just to stay in shape; see “Cuanto más blando sea el tema, más largo será el debate” in Capítulo 6, *Communications* for more about them.

Be Open About Your Motivations

Be as open about your organization's goals as you can without compromising business secrets. If you want the project to acquire a certain feature because, say, your customers have been clamoring for it, just say so outright on the mailing lists. If the customers wish to remain anonymous, as is sometimes the case, then at least ask them if they can be used as unnamed examples. The more the public development community knows about *why* you want what you want, the more comfortable they'll be with whatever you're proposing.

This runs counter to the instinct—so easy to acquire, so hard to shake off—that knowledge is power, and that the more others know about your goals, the more control they have over you. But that instinct would be wrong here. By publicly advocating the feature (or bugfix, or whatever it is), you have *already* laid your cards on the table. The only question now is whether you will succeed in guiding the community to share your goal. If you merely state that you want it, but can't provide concrete examples of why, your argument is weak, and people will start to suspect a hidden agenda. But if you give just a few real-world scenarios showing why the proposed feature is important, that can have a dramatic effect on the debate.

To see why this is so, consider the alternative. Too frequently, debates about new features or new directions are long and tiresome. The arguments people advance often reduce to "I personally want X," or the ever-popular "In my years of experience as a software designer, X is extremely important to users / a useless frill that will please no one." Predictably, the absence of real-world usage data neither shortens nor tempers such debates, but instead allows them to drift farther and farther from any mooring in actual user experience. Without some countervailing force, the end result is as likely as not to be determined by whoever was the most articulate, or the most persistent, or the most senior.

As an organization with plentiful customer data available, you have the opportunity to provide just such a countervailing force. You can be a conduit for information that might otherwise have no means of reaching the development community. The fact that that information supports your desires is nothing to be embarrassed about. Most developers don't individually have very broad experience with how the software they write is used. Each developer uses the software in her own idiosyncratic way; as far as other usage patterns go, she's relying on intuition and guesswork, and deep down, she knows this. By providing credible data about a significant number of users, you are giving the public development community something akin to oxygen. As long as you present it right, they will welcome it enthusiastically, and it will propel things in the direction you want to go.

The key, of course, is presenting it right. It will never do to insist that simply because you deal with a large number of users, and because they need (or think they need) a given feature, therefore your solution ought to be implemented. Instead, you should focus your initial posts on the problem, rather than on one particular solution. Describe in great detail the experiences your customers are encountering, offer as much analysis as you have available, and as many reasonable solutions as you can think of. When people start speculating about the effectiveness of various solutions, you can continue to draw on your data to support or refute what they say. You may have one particular solution in mind all along, but don't single it out for special consideration at first. This is not deception, it is simply standard "honest broker"

behavior. After all, your true goal is to solve the problem; a solution is merely a means to that end. If the solution you prefer really is superior, other developers will recognize that on their own eventually—and then they will get behind it of their own free will, which is much better than you browbeating them into implementing it. (There is also the possibility that they will think of a better solution.)

This is not to say that you can't ever come out in favor of a specific solution. But you must have the patience to see the analysis you've already done internally repeated on the public development lists. Don't post saying "Yes, we've been over all that here, but it doesn't work for reasons A, B, and C. When you get right down to it, the only way to solve this is..." The problem is not so much that it sounds arrogant as that it gives the impression that you have *already* devoted some unknown (but, people will presume, large) amount of analytical resources to the problem, behind closed doors. It makes it seem as though efforts have been going on, and perhaps decisions made, that the public is not privy to, and that is a recipe for resentment.

Naturally, *you* know how much effort you've devoted to the problem internally, and that knowledge is, in a way, a disadvantage. It puts your developers in a slightly different mental space than everyone else on the mailing lists, reducing their ability to see things from the point of view of those who haven't yet thought about the problem as much. The earlier you can get everyone else thinking about things in the same terms as you do, the smaller this distancing effect will be. This logic applies not only to individual technical situations, but to the broader mandate of making your goals as clear as you can. The unknown is always more destabilizing than the known. If people understand why you want what you want, they'll feel comfortable talking to you even when they disagree. If they can't figure out what makes you tick, they'll assume the worst, at least some of the time.

You won't be able to publicize everything, of course, and people won't expect you to. All organizations have secrets; perhaps for-profits have more of them, but nonprofits have them too. If you must advocate a certain course, but can't reveal anything about why, then simply offer the best arguments you can under that handicap, and accept the fact that you may not have as much influence as you want in the discussion. This is one of the compromises you make in order to have a development community not on your payroll.

Money Can't Buy You Love

If you're a paid developer on a project, then set guidelines early on about what the money can and cannot buy. This does not mean you need to post twice a day to the mailing lists reiterating your noble and incorruptible nature. It merely means that you should be on the lookout for opportunities to defuse the tensions that *could* be created by money. You don't need to start out assuming that the tensions are there; you do need to demonstrate an awareness that they have the potential to arise.

A perfect example of this came up in the Subversion project. Subversion was started in 2000 by CollabNet [http://www.collab.net/], which has been the project's primary funder since its inception, paying the salaries of several developers (disclaimer: I'm one of them). Soon after the project began, we hired another developer, Mike Pilato, to join the effort. By then, coding had already started. Although Subversion was still very much in the early stages, it already had a development community with a set of basic ground rules.

Mike's arrival raised an interesting question. Subversion already had a policy about how a new developer gets commit access. First, he submits some patches to the development mailing list. After enough patches have gone by for the other committers to see that the new contributor knows what he's doing, someone proposes that he just commit directly (that proposal is private, as described in “Committers”). Assuming the committers agree, one of them mails the new developer and offers him direct commit access to the project's repository.

CollabNet had hired Mike specifically to work on Subversion. Among those who already knew him, there was no doubt about his coding skills or his readiness to work on the project. Furthermore, the volunteer developers had a very good relationship with the CollabNet employees, and most likely would not have objected if we'd just given Mike commit access the day he was hired. But we knew we'd be setting

a precedent. If we granted Mike commit access by fiat, we'd be saying that CollabNet had the right to ignore project guidelines, simply because it was the primary funder. While the damage from this would not necessarily be immediately apparent, it would gradually result in the non-salaried developers feeling disenfranchised. Other people have to earn their commit access—CollabNet just buys it.

So Mike agreed to start out his employment at CollabNet like any other volunteer developer, without commit access. He sent patches to the public mailing list, where they could be, and were, reviewed by everyone. We also said on the list that we were doing things this way deliberately, so there could be no missing the point. After a couple of weeks of solid activity by Mike, someone (I can't remember if it was a CollabNet developer or not) proposed him for commit access, and he was accepted, as we knew he would be.

That kind of consistency gets you a credibility that money could never buy. And credibility is a valuable currency to have in technical discussions: it's immunization against having one's motives questioned later. In the heat of argument, people will sometimes look for non-technical ways to win the battle. The project's primary funder, because of its deep involvement and obvious concern over the directions the project takes, presents a wider target than most. By being scrupulous to observe all project guidelines right from the start, the funder makes itself the same size as everyone else.

(See also Danese Cooper's blog at http://blogs.sun.com/roller/page/DaneseCooper/20040916 for a similar story about commit access. Cooper was then Sun Microsystem's "Open Source Diva"—I believe that was her official title—and in the blog entry, she describes how the Tomcat development community got Sun to hold its own developers to the same commit-access standards as the non-Sun developers.)

The need for the funders to play by the same rules as everyone else means that the Benevolent Dictatorship governance model (see “Dictadores Benevolentes” in Capítulo 4, *Infraestructura Social y Política*) is slightly harder to pull off in the presence of funding, particularly if the dictator works for the primary funder. Since a dictatorship has few rules, it is hard for the funder to prove that it's abiding by community standards, even when it is. It's certainly not impossible; it just requires a project leader who is able to see things from the point of view of the outside developers, as well as that of the funder, and act accordingly. Even then, it's probably a good idea to have a proposal for non-dictatorial governance sitting in your back pocket, ready to be brought out the moment there are any indications of widespread dissatisfaction in the community.

Contracting

Contracted work needs to be handled carefully in free software projects. Ideally, you want a contractor's work to be accepted by the community and folded into the public distribution. In theory, it wouldn't matter who the contractor is, as long as his work is good and meets the project's guidelines. Theory and practice can sometimes match, too: a complete stranger who shows up with a good patch *will* generally be able to get it into the software. The trouble is, it's very hard to produce a good patch for a non-trivial enhancement or new feature while truly being a complete stranger; one must first discuss it with the rest of the project. The duration of that discussion cannot be precisely predicted. If the contractor is paid by the hour, you may end up paying more than you expected; if he is paid a flat sum, he may end up doing more work than he can afford.

There are two ways around this. The preferred way is to make an educated guess about the length of the discussion process, based on past experience, add in some padding for error, and base the contract on that. It also helps to divide the problem into as many small, independent chunks as possible, to increase the predictability of each chunk. The other way is to contract solely for delivery of a patch, and treat the patch's acceptance into the public project as a separate matter. Then it becomes much easier to write the contract, but you're stuck with the burden of maintaining a private patch for as long as you depend on the software, or at least for as long as it takes you to get that patch or equivalent functionality into the mainline. Of course, even with the preferred way, the contract itself cannot require that the patch be accepted into the code, because that would involve selling something that's not for sale. (What if the rest of the project unexpectedly decides not to support the feature?) However, the contract can require a *bona fide* effort to get the change accepted by the community, and that it be committed to the repository if the

community agrees with it. For example, if the project has written standards regarding code changes, the contract can reference those standards and specify that the work must meet them. In practice, this usually works out the way everyone hopes.

The best tactic for successful contracting is to hire one of the project's developers—preferably a committer—as the contractor. This may seem like a form of purchasing influence, and, well, it is. But it's not as corrupt as it might seem. A developer's influence in the project is due mainly to the quality of his code and to his interactions with other developers. The fact that he has a contract to get certain things done doesn't raise his status in any way, and doesn't lower it either, though it may make people scrutinize him more carefully. Most developers would not risk their long-term position in the project by backing an inappropriate or widely disliked new feature. In fact, part of what you get, or should get, when you hire such a contractor is advice about what sorts of changes are likely to be accepted by the community. You also get a slight shift in the project's priorities. Because prioritization is just a matter of who has time to work on what, when you pay for someone's time, you cause their work to move up in the priority queue a bit. This is a well-understood fact of life among experienced open source developers, and at least some of them will devote attention to the contractor's work simply because it looks like it's going to *get done*, so they want to help it get done right. Perhaps they won't write any of the code, but they'll still discuss the design and review the code, both of which can be very useful. For all these reasons, the contractor is best drawn from the ranks of those already involved with the project.

This immediately raises two questions: Should contracts ever be private? And when they're not, should you worry about creating tensions in the community by the fact that you've contracted with some developers and not others?

It's best to be open about contracts, when you can. Otherwise, the contractor's behavior may seem strange to others in the community—perhaps he's suddenly giving inexplicably high priority to features he's never shown interest in in the past. When people ask him why he wants them now, how can he answer convincingly if he can't talk about the fact that he's been contracted to write them?

At the same time, neither you nor the contractor should act as though others should treat your arrangement as a big deal. Too often I've seen contractors waltz onto a development list with the attitude that their posts should be taken more seriously simply because they're being paid. That kind of attitude signals to the rest of the project that the contractor regards the fact of the contract—as opposed to the code *resulting* from the contract—to be the important thing. But from the other developers' point of view, only the code matters. At all times, the focus of attention should be kept on technical issues, not on the details of who is paying whom. For example, one of the developers in the Subversion community handles contracting in a particularly graceful way. While discussing his code changes in IRC, he'll mention as an aside (often in a private remark, an IRC *privmsg*, to one of the other committers) that he's being paid for his work on this particular bug or feature. But he also consistently gives the impression that he'd want to be working on that change anyway, and that he's happy the money is making it possible for him to do that. He may or may not reveal his customer's identity, but in any case he doesn't dwell on the contract. His remarks about it are just an ornament to an otherwise technical discussion about how to get something done.

That example shows another reason why it's good to be open about contracts. There may be multiple organizations sponsoring contracts on a given open source project, and if each knows what the others are trying to do, they may be able to pool their resources. In the above case, the project's largest funder (CollabNet) is not involved in any way with these piecework contracts, but knowing that someone else is sponsoring certain bug fixes allows CollabNet to redirect its resources to other bugs, resulting in greater efficiency for the project as a whole.

Will other developers resent that some are paid for working on the project? In general, no, particularly when those who are paid are established, well-respected members of the community anyway. No one expects contract work to be distributed equally among all the committers. People understand the importance of long-term relationships: the uncertainties involved in contracting are such that once you find someone you can work reliably with, you would be reluctant to switch to a different person just for the sake of evenhandedness. Think of it this way: the first time you hire, there will be no complaints, because clearly you had to pick *someone*—it's not your fault you can't hire everyone. Later, when you hire the same person a second time, that's just common sense: you already know him, the last time was success-

ful, so why take unnecessary risks? Thus, it's perfectly natural to have one or two go-to people in the community, instead of spreading the work around evenly.

Review and Acceptance of Changes

The community is still important to the success of contract work. Their involvement in the design and review process for sizeable changes cannot be an afterthought. It must be considered part of the work, and fully embraced by the contractor. Don't think of community scrutiny as an obstacle to be overcome—think of it as a free design board and QA department. It is a benefit to be aggressively pursued, not merely endured.

Case study: the CVS password-authentication protocol

In 1995, I was one half of a partnership that provided support and enhancements for CVS (the Concurrent Versions System; see http://www.cvshome.org/). My partner Jim and I were, informally, the maintainers of CVS by that point. But we'd never thought carefully about how we ought to relate to the existing, mostly volunteer CVS development community. We just assumed that they'd send in patches, and we'd apply them, and that was pretty much how it worked.

Back then, networked CVS could be done only over a remote login program such as `rsh`. Using the same password for CVS access as for login access was an obvious security risk, and many organizations were put off by it. A major investment bank hired us to add a new authentication mechanism, so they could safely use networked CVS with their remote offices.

Jim and I took the contract and sat down to design the new authentication system. What we came up with was pretty simple (the United States had export controls on cryptographic code at the time, so the customer understood that we couldn't implement strong authentication), but as we were not experienced in designing such protocols, we still made a few gaffes that would have been obvious to an expert. These mistakes would easily have been caught had we taken the time to write up a proposal and run it by the other developers for review. But we never did so, because it didn't occur to us to think of the development list as a resource to be used. We knew that people were probably going to accept whatever we committed, and—because we didn't know what we didn't know—we didn't bother to do the work in a visible way, e.g., posting patches frequently, making small, easily digestible commits to a special branch, etc. The resulting authentication protocol was not very good, and of course, once it became established, it was difficult to improve, because of compatibility concerns.

The root of the problem was not lack of experience; we could easily have learned what we needed to know. The problem was our attitude toward the volunteer development community. We regarded acceptance of the changes as a hurdle to leap, rather than as a process by which the quality of the changes could be improved. Since we were confident that almost anything we did would be accepted (as it was), we made little effort to get others involved.

Obviously, when you're choosing a contractor, you want someone with the right technical skills and experience for the job. But it's also important to choose someone with a track record of constructive interaction with the other developers in the community. That way you're getting more than just a single person; you're getting an agent who will be able to draw on a network of expertise to make sure the work is done in a robust and maintainable way.

Funding Non-Programming Activities

Programming is only part of the work that goes on in an open source project. From the point of view of the project's volunteers, it's the most visible and glamorous part. This unfortunately means that other activities, such as documentation, formal testing, etc., can sometimes be neglected, at least compared to the amount of attention they often receive in proprietary software. Corporate organizations are sometimes able to make up for this, by devoting some of their internal software development infrastructure to open source projects.

The key to doing this successfully is to translate between the company's internal processes and those of the public development community. Such translation is not effortless: often the two are not a close match, and the differences can only be bridged via human intervention. For example, the company may use a different bug tracker than the public project. Even if they use the same tracking software, the data stored in it will be very different, because the bug-tracking needs of a company are very different from those of a free software community. A piece of information that starts in one tracker may need to be reflected in the other, with confidential portions removed or, in the other direction, added.

The sections that follow are about how to build and maintain such bridges. The end result should be that the open source project runs more smoothly, the community recognizes the company's investment of resources, and yet does not feel that the company is inappropriately steering things toward its own goals.

Quality Assurance (i.e., Professional Testing)

In proprietary software development, it is normal to have teams of people dedicated solely to quality assurance: bug hunting, performance and scalability testing, interface and documentation checking, etc. As a rule, these activities are not pursued as vigorously by the volunteer community on a free software project. This is partly because it's hard to get volunteer labor for unglamorous work like testing, partly because people tend to assume that having a large user community gives the project good testing coverage, and, in the case of performance and scalability testing, partly because volunteers often don't have access to the necessary hardware resources anyway.

The assumption that having many users is equivalent to having many testers is not entirely baseless. Certainly there's little point assigning testers for basic functionality in common environments: bugs there will quickly be found by users in the natural course of things. But because users are just trying to get work done, they do not consciously set out to explore uncharted edge cases in the program's functionality, and are likely to leave certain classes of bugs unfound. Furthermore, when they discover a bug with an easy workaround, they often silently implement the workaround without bothering to report the bug. Most insidiously, the usage patterns of your customers (the people who drive *your* interest in the software) may differ in statistically significant ways from the usage patterns of the Average User In The Street.

A professional testing team can uncover these sorts of bugs, and can do so as easily with free software as with proprietary software. The challenge is to convey the testing team's results back to the public in a useful form. In-house testing departments usually have their own way of reporting test results, involving company-specific jargon, or specialized knowledge about particular customers and their data sets. Such reports would be inappropriate for the public bug tracker, both because of their form and because of confidentiality concerns. Even if your company's internal bug tracking software were the same as that used by the public project, management might need to make company-specific comments and metadata changes to the issues (for example, to raise an issue's internal priority, or schedule its resolution for a particular customer). Usually such notes are confidential—sometimes they're not even shown to the customer. But even when they're not confidential, they're of no concern to the public project, and therefore the public should not be distracted with them.

Yet the core bug report itself *is* important to the public. In fact, a bug report from your testing department is in some ways more valuable than one received from users at large, since the testing department probes for things that other users won't. Given that you're unlikely to get that particular bug report from any other source, you definitely want to preserve it and make it available to the public project.

To do this, either the QA department can file issues directly in the public issue tracker, if they're comfortable with that, or an intermediary (usually one of the developers) can "translate" the testing department's internal reports into new issues in the public tracker. Translation simply means describing the bug in a way that makes no reference to customer-specific information (the reproduction recipe may use customer data, assuming the customer approves it, of course).

It is somewhat preferable to have the QA department filing issues in the public tracker directly. That gives the public a more direct appreciation of your company's involvement with the project: useful bug reports add to your organization's credibility just as any technical contribution would. It also gives develo-

pers a direct line of communication to the testing team. For example, if the internal QA team is monitoring the public issue tracker, a developer can commit a fix for a scalability bug (which the developer may not have the resources to test herself), and then add a note to the issue asking the QA team to see if the fix had the desired effect. Expect a bit of resistance from some of the developers; programmers have a tendency to regard QA as, at best, a necessary evil. The QA team can easily overcome this by finding significant bugs and filing comprehensible reports; on the other hand, if their reports are not at least as good as those coming from the regular user community, then there's no point having them interact directly with the development team.

Either way, once a public issue exists, the original internal issue should simply reference the public issue for technical content. Management and paid developers may continue to annotate the internal issue with company-specific comments as necessary, but use the public issue for information that should be available to everyone.

You should go into this process expecting extra overhead. Maintaining two issues for one bug is, naturally, more work than maintaining one issue. The benefit is that many more coders will see the report and be able to contribute to a solution.

Legal Advice and Protection

Corporations, for-profit or nonprofit, are almost the only entities that ever pay attention to complex legal issues in free software. Individual developers often understand the nuances of various open source licenses, but they generally do not have the time or resources to follow copyright, trademark, and patent law in detail. If your company has a legal department, it can help a project by vetting the copyright status of the code, and helping developers understand possible patent and trademark issues. The exact forms this help could take are discussed in Capítulo 9, *Licencias, Copyrights y Patentes*. The main thing is to make sure that communications between the legal department and the development community, if they happen at all, happen with a mutual appreciation of the very different universes the parties are coming from. On occasion, these two groups talk past each other, each side assuming domain-specific knowledge that the other does not have. A good strategy is to have a liaison (usually a developer, or else a lawyer with technical expertise) stand in the middle and translate for as long as needed.

Documentation and Usability

Documentation and usability are both famous weak spots in open source projects, although I think, at least in the case of documentation, that the difference between free and proprietary software is frequently exaggerated. Nevertheless, it is empirically true that much open source software lacks first-class documentation and usability research.

If your organization wants to help fill these gaps for a project, probably the best thing it can do is hire people who are *not* regular developers on the project, but who will be able to interact productively with the developers. Not hiring regular developers is good for two reasons: one, that way you don't take development time away from the project; two, those closest to the software are usually the wrong people to write documentation or investigate usability anyway, because they have trouble seeing the software from an outsider's point of view.

However, it will still be necessary for whoever works on these problems to communicate with the developers. Find people who are technical enough to talk to the coding team, but not so expert in the software that they can't empathize with regular users anymore.

A medium-level user is probably the right person to write good documentation. In fact, after the first edition of this book was published, I received the following email from an open source developer named Dirk Reiners:

```
One comment on Money::Documentation and Usability: when we had some
money to spend and decided that a beginner's tutorial was the most
```

```
critical piece that we needed we hired a medium-level user to write it.
He had gone through the induction to the system recently enough to
remember the problems, but he had gotten past them so he knew how to
describe them. That allowed him to write something that needed only
minor fixes by the core developers for the things that he hadn't gotten
right, but still covering the 'obvious' stuff devs would have missed.

His case was even better, as it had been his job to introduce a bunch of
other people (students) to the system, so he combined the experience of
many people, which is something that was just a lucky occurrence and is
probably hard to get in most cases.
```

Providing Hosting/Bandwidth

For a project that's not using one of the free canned hosting sites (see “Soluciones de hospedaje” in Capítulo 3, *Infraestructura Técnica*), providing a server and network connection—and most importantly, system administration help—can be of significant assistance. Even if this is all your organization does for the project, it can be a moderately effective way to obtain good public relations karma, though it will not bring any influence over the direction of the project.

You can probably expect a banner ad or an acknowledgment on the project's home page, thanking your company for providing hosting. If you set up the hosting so that the project's web address is under your company's domain name, then you will get some additional association just through the URL. This will cause most users to think of the software as having *something* to do with your company, even if you don't contribute to development at all. The problem is, the developers are aware of this associative tendency too, and may not be very comfortable with having the project in your domain unless you're contributing more resources than just bandwidth. After all, there are a lot of places to host these days. The community may eventually feel that the implied misallocation of credit is not worth the convenience brought by hosting, and take the project elsewhere. So if you want to provide hosting, do so—but either plan to get even more involved soon, or be circumspect about how much involvement you claim.

Marketing

Although most open source developers would probably hate to admit it, marketing works. A good marketing campaign *can* create buzz around an open source product, even to the point where hardheaded coders find themselves having vaguely positive thoughts about the software for reasons they can't quite put their finger on. It is not my place here to dissect the arms-race dynamics of marketing in general. Any corporation involved in free software will eventually find itself considering how to market themselves, the software, or their relationship to the software. The advice below is about how to avoid common pitfalls in such an effort; see also “Publicity” in Capítulo 6, *Communications*.

Remember That You Are Being Watched

For the sake of keeping the volunteer developer community on your side, it is *very* important not to say anything that isn't demonstrably true. Audit all claims carefully before making them, and give the public the means to check your claims on their own. Independent fact checking is a major part of open source, and it applies to more than just the code.

Naturally no one would advise companies to make unverifiable claims anyway. But with open source activities, there is an unusually high quantity of people with the expertise to verify claims—people who are also likely to have high-bandwidth Internet access and the right social contacts to publicize their findings in a damaging way, should they choose to. When Global Megacorp Chemical Industries pollutes a stream, that's verifiable, but only by trained scientists, who can then be refuted by Global Megacorp's scientists, leaving the public scratching their heads and wondering what to think. On the other hand, your behavior in the open source world is not only visible and recorded; it is also easy for many people to check it independently, come to their own conclusions, and spread those conclusions by word of

mouth. These communications networks are already in place; they are the essence of how open source operates, and they can be used to transmit any sort of information. Refutation is usually difficult, if not impossible, especially when what people are saying is true.

For example, it's okay to refer to your organization as having "founded project X" if you really did. But don't refer to yourself as the "makers of X" if most of the code was written by outsiders. Conversely, don't claim to have a deeply involved volunteer developer community if anyone can look at your repository and see that there are few or no code changes coming from outside your organization.

Not too long ago, I saw an announcement by a very well-known computer company, stating that they were releasing an important software package under an open source license. When the initial announcement came out, I took a look at their now-public version control repository and saw that it contained only three revisions. In other words, they had done an initial import of the source code, but hardly anything had happened since then. That in itself was not worrying—they'd just made the announcement, after all. There was no reason to expect a lot of development activity right away.

Some time later, they made another announcement. Here is what it said, with the name and release number replaced by pseudonyms:

> *We are pleased to announce that following rigorous testing by the Singer Community, Singer 5 for Linux and Windows are now ready for production use.*

Curious to know what the community had uncovered in "rigorous testing," I went back to the repository to look at its recent change history. The project was still on revision 3. Apparently, they hadn't found a *single* bug worth fixing before the release! Thinking that the results of the community testing must have been recorded elsewhere, I next examined the bug tracker. There were exactly six open issues, four of which had been open for several months already.

This beggars belief, of course. When testers pound on a large and complex piece of software for any length of time, they will find bugs. Even if the fixes for those bugs don't make it into the upcoming release, one would still expect some version control activity as a result of the testing process, or at least some new issues. Yet to all appearances, nothing had happened between the announcement of the open source license and the first open source release.

The point is not that the company was lying about the community testing. I have no idea if they were or not. But they were oblivious to how much it *looked* like they were lying. Since neither the version control repository nor the issue tracker gave any indication that the alleged rigorous testing had occurred, the company should either not have made the claim in the first place, or provided a clear link to some tangible result of that testing ("We found 278 bugs; click here for details"). The latter would have allowed anyone to get a handle on the level of community activity very quickly. As it was, it only took me a few minutes to determine that whatever this community testing was, it had not left traces in any of the usual places. That's not a lot of effort, and I'm sure I'm not the only one who took the trouble.

Transparency and verifiability are also an important part of accurate crediting, of course. See “Credit” in Capítulo 8, *Coordinando a los Voluntarios* for more on this.

Don't Bash Competing Open Source Products

Refrain from giving negative opinions about competing open source software. It's perfectly okay to give negative *facts*—that is, easily confirmable assertions of the sort often seen in good comparison charts. But negative characterizations of a less rigorous nature are best avoided, for two reasons. First, they are liable to start flame wars that detract from productive discussion. Second, and more importantly, some of the volunteer developers in *your* project may turn out to work on the competing project as well. This is more likely than it at first might seem: the projects are already in the same domain (that's why they're in competition), and developers with expertise in that domain may make contributions wherever their expertise is applicable. Even when there is no direct developer overlap, it is likely that developers on your project are at least acquainted with developers on related projects. Their ability to maintain cons-

tructive personal ties could be hampered by overly negative marketing messages.

Bashing competing closed-source products seems to be more widely accepted in the open source world, especially when those products are made by Microsoft. Personally, I deplore this tendency (though again, there's nothing wrong with straightforward factual comparisons), not merely because it's rude, but also because it's dangerous for a project to start believing its own hype and thereby ignore the ways in which the competition may actually be superior. In general, watch out for the effect that marketing statements can have on your own development community. People may be so excited at being backed by marketing dollars that they lose objectivity about their software's true strengths and weaknesses. It is normal, and even expected, for a company's developers to exhibit a certain detachment toward marketing statements, even in public forums. Clearly, they should not come out and contradict the marketing message directly (unless it's actually wrong, though one hopes that sort of thing would have been caught earlier). But they may poke fun at it from time to time, as a way of bringing the rest of the development community back down to earth.

Capítulo 6. Communications

La capacidad de escribir claramente es quizás la más importante habilidad que se puede tener en un ambiente de código abierto. A largo plazo es más importante que el talento para programar. Un gran programador con pocas habilidades comunicativas puede realizar sólo una cosa a la vez, y puede tener problemas convenciendo a otros para que le presten atención. Pero un mal programador con buenas habilidades de comunicación puede coordinar y persuadir mucha gente para realizar diferentes cosas, y de tal modo tener un efecto significativo sobre la dirección y el ímpetu de un proyecto.

No parece haber mucha correlación, en cualquier sentido, entre la capacidad de escribir buen código y la capacidad de comunicarse con sus compañeros. Hay cierta correlación entre programar bien y describir bien cuestiones técnicas, pero describir asuntos técnicos es sólo una pequeña parte de las comunicaciones en un proyecto. Más importante es la capacidad de enfatizar con su audiencia, ver sus propios correos y comentarios como lo ven los demás, y hacer que los demás vean sus propios correos con objetividad similar. Igualmente importante es notificar cuando un medio o método de comunicación determinado no está funcionando bien, quizás porque no escala al ritmo que incrementa el número de usuarios, y tomar el tiempo para hacer algo al respecto.

Aquello que es obvio en teoría y que se hace duro en la práctica es que los ambientes de desarrollo de software libre son desconcertadamente diversos tanto en audiencias como en mecanismos de comunicación. ¿Debería una opinión dada ser expresada en un mensaje a la lista de correo, como una anotación en el gestor de fallos, o como un comentario en el código? Al contestar una pregunta en un foro público, ¿cuánto conocimiento puedes asumir por parte del lector?, en primer lugar dado que "el lector" no es el único que hizo la pregunta, ¿pueden todos ver tú respuesta? ¿Como pueden los desarrolladores permanecer en contacto constructivo con los usuarios, sin ser ahogado por peticiones de características, informes falsos de fallos, y charla en general? ¿Cómo dices cuando un medio ha alcanzado los límites de su capacidad, y que harías al respecto?

Las soluciones a estos problemas son usualmente parciales, ya que cualquier solucion particular se vuelve finalmente obsoleta por el crecimiento del proyecto o los cambios en la estructura del mismo. Son a menudo *ad hoc*, ya que son respuestas improvisadas a situaciones dinámicas. Todos los participantes necesitan darse cuenta de como y cuando la comunicacion puede volverse farragosa, y deben estar implicados en buscar soluciones. Ayudar a la gente a hacer esto es una gran parte de la direccion en un proyecto open source. Las secciones siguientes tratan sobre como conducir tu propia comunicacion y como hacer el mantenimiento de los mecanismos de comunicacion una prioridad para todo el mundo en el proyecto.[1]

Tú eres lo que escribes

Considera esto: la única cosa que cualquier persona sabe de ti en Internet viene de lo que tú escribes, o de lo que otros escriben acerca de ti. Puedes ser brillante, perceptivo, y carismático en persona pero si tus correos electrónicos son incoherentes y no estructurados, la gente asumirá que esé es el verdadero tú. O quizás realmente eres incoherente y no estructurado en persona, pero nadie tiene por que saberlo, si tus mensajes son claros e informativos.

Dedicar cierto cuidado a tu escritura valdrá enormemente la pena. El veterano hacker de software libre Jim Blandy narra la siguiente historia:

> Por el año 1993 trabajaba para la Fundación de Software Libre, y estábamos llevando a cabo el beta-testing de la versión 19 de GNU Emacs. Haríamos una publicación beta

[1] Se ha hecho alguna investigacion academica interesante en esta materia; por ejemplo, vease *Group Awareness in Distributed Software Development* por Gutwin, Penner, y Schneider (solia estar disponible on-line, pero parece que ha desaparecido, al menos temporalmente; utiliza una herramienta de busqueda encontrarla).

> más o menos cada semana, y la gente la probaría y nos enviaría informes de error. Había un chico que ninguno de nosotros conocía en persona pero que hizo un gran trabajo: sus informes de error siempre fueron claros y nos enfocaba hacia el problema y, cuando nos proporcionaba una corrección, casi siempre tenía razón. Era un fuera de serie.
>
> Ahora, antes que la FSF pueda utilizar código escrito por alguien, hay que realizar un papeleo legal para que el interés de esa persona hacia el copyright del código pase a la FSF. Simplemente tomando el código de completos extraños dejándolo dentro es una receta para el desastre legal.
>
> Por lo que le envié un correo al chico con los formularios diciéndole "Te envío algo de papeleo que necesitamos, esto es lo que significa, firmas este, haces que quien te tiene contratado firme este otro y, entonces podemos comenzar a utilizar tus correcciones. Muchas gracias."
>
> Me envió un mensaje de vuelta diciendo: "No trabajo para nadie."
>
> Por lo que le dije: "Bien, eso está bien, simplemente haz que firme tu universidad y envíamelo de vuelta."
>
> Después de un poco, me escribió de nuevo y me dijo: "Verás, realmente... tengo trece años y vivo con mis padres."

Debido a que ese chico no escribía como si tuviera trece años nadie supuso que los tuviera. A continuación se exponen también algunas cosas que conseguirán además que tu escritura de una buena impresión.

Estructura y formato

No caigas en la trampa de escribir todo como si fuera un mensaje de teléfono móvil. Escribe frases completas, poniendo en mayúsculas la primera palabra de cada frase, y usando separaciones de párrafo donde sea necesario. Esto es lo más importante en correos electrónicos y otras composiciones. En el IRC u otros foros efímeros similares, generalmente es correcto dejar de poner mayúsculas, utilizar formas comprimidas o expresiones comunes, etc. Simplemente no lleves esos hábitos a foros más formales o persistentes. Correos electrónicos, documentación, informes de error y otras piezas de escritura que suelen tener una larga vida deberían escribirse usando una gramática y una spelling estándar, y tener una estructura narrativa coherente. Esto no se debe a que haya algo inherentemente bueno siguiendo reglas arbitrarias, sino a que estas reglas *no* son arbitrarias: evolucionan en las formas presentes ya que hacen que el texto sea más leíble y, por esa razón, deberías seguirlas. La legibilidad no sólo es deseable para que la mayoría de gente entienda lo que escribes, sino porque hace que que parezcas la clase de persona que se toma su tiempo en comunicarse de una forma clara: es decir, alguien a quien vale la pena prestar atención.

En particular, para correos electrónicos, desarrolladores experimientados de open source han decidido ciertas convenciones:

Envía correos solo de texto plano, no en HTML, texto enriquecido u otros formatos ya que podrían no ser leidos por lectores que leen sólo texto plano. Formatea las líneas para que estén sobre las 72 columnas de largo. No excedas las 80 columnas, que ha sido *de facto* el ancho estándar del terminal (es decir, hay gente que utiliza terminales más anchos, pero nadie utiliza terminales no más estrechos). Al hacer las líneas un poco *menores* de 80 columnas da cabida a unos cuantos niveles de caracteres de citado para ser añadidos en otras respuestas sin forzar un estrechamiento de tu texto.

Utiliza saltos de línea reales. Algunos clientes de correo muestran un falso formateo de línea, mientras estás escribiendo un correo, viéndose en la pantalla saltos de línea donde en realidad no los hay. Cuando se envía el correo, no tendrá los saltos de línea que se pensaba y se presentará con un formato horroroso

en la pantalla de la gente. Si tu cliente de correo muestra falsos saltos de línea, busca posibilidad de quitar la opción para ver los saltos de línea reales a medida que escribes el correo.

Cuando incluyas salida de pantalla, trozos de código u otro texto preformateado, desplázalo claramente, de forma que a simple vista se pueda fácilmente ver los límites entre tu texto y el material que estés incluyendo. (Nunca esperé escribir este consejo cuando comencé el libro, pero en un número de listas de correo de código abierto posterior, he visto gente mezclando textos de diferentes fuentes sin dejar claro qué es qué. El efecto es muy frustante. Hacen los correos bastante difíciles de entender y, francamente, hace que esas personas parezcan un poco desorganizadas).

Cuando cites el correo de alguien, inserta tus repuestas donde sea más apropiado, en diferentes lugares si es necesario, y elimina las partes de su correo que no utilices. Si estás escribiendo un comentario rápido con referencia a todo el correo, es correcto hacerlo *top-post* (Es decir, poner tu respuesta encima del texto citado; de lo contrario, deberías citar primero la parte relevante del texto original, seguido de tu respuesta.

Construye el asunto de los nuevos correos con cuidado. Es la línea más importante de un correo, ya que permite a cualquier otra persona del proyecto decidir si leer más o no. Los lectores de correo modernos organizan los grupos de mensajes relacionados en hilos, que pueden no solo definirse por un asunto común sino por otras cabeceras (que a menudo no se muestran). Entienden que si un hilo comienza a derivar hacia un nuevo tema, puedes y debes ajustar el asunto adecuadamente cuando respondas. La integridad del hilo persistirá, debido a aquellas otras cabeceras, pero el nuevo asunto ayudará a la gente que mira un resumen del hilo a saber que el tema ha derivado. Asimismo, si realmente quieres comenzar un nuevo tema, hazlo creando un nuevo mensaje y no respondiendo uno ya existente y cambiándole el asunto. De esta forma, tu correo podría estar agrupado en el mismo hilo del correo que estás respondiendo y así volver loca a la gente pensando sobre algo que no es. Recuerda: la penalización no será la pérdida de tiempo, sino la pequeña hendidura en tu credibilidad como alguien fluido en el uso de las herramientas de comunicación.

Contenido

Correos electrónicos bien formateados atraen a los lectores, pero el contenido los mantiene. Ningún conjunto fijo de reglas puede garantizar el buen contenido, por supuesto, hay algunos principios que lo hacen más prometedor.

Hacer las cosas fáciles para tus lectores. Hay una tonelada de información flotando alrededor en cualquier proyecto activo de software libre, y los lectores no pueden esperar estar al corriente de la mayor parte de ella, de hecho, no siempre pueden esperar familiarizarse. En lo posible, tus correos deben sumunistrar información en la forma más conveniente para los lectores. Si tienes que pasar unos dos minutos extra buscando el URL de un hilo particular en los archivos de la lista de correo, atendiendo al objetivo de librar a tus lectores de hacerlo, vale la pena. Si tienes que pasar unos 5 o 10 minutos extra resumiendo las conclusiones de un hilo complejo, con la intención de brindarle a las personas el contexto en el cual compredérán tu correo, entonces hazlo. Piénsalo de esta manera: el mayor éxito en un proyecto, es aumentar el cociente lector-a-escritor en cualquier foro dado. Si cada correo tuyo es visto por *n* personas, entonces como *n* aumenta, la utilidad de realizar un esfuerzo adicional para ayudar a aquellas personas aumenta con el tiempo. Y como las personas te verán imponer este estándar, trabajarán imitándolo en sus propias comunicaciones. El resultado es, idealmente, un incremento en la eficiencia global del proyecto: cuando hay una elección entre *n* personas realizando un esfuerzo y una persona haciendolo, el proyecto prefiere el segundo.

No acostumbrarse a la hipérbole. La exageración de correos online es una clásica competencia de armamento. Por ejemplo, a una persona que reporta un fallo puede preocuparle que los desarrolladores no le presten la suficiente atención, así que lo describirá como grave, gran problema que es prevenirle (y a todos sus amigos/compañeros de trabajo/primos) de la utilización del software productivamente, cuando es solamente una molestia leve. Pero la exageración no está limitada a los usuarios; los programadores frecuentemente hacen lo mismo durante debates técnicos, especialmente cuando el desacuerdo es una cuestión de gustos más que de corección:

> "Hacerlo de esa manera haría el código totalmente ilegible. Sería una pesadilla para el mantenimiento, comparado a la propuesta de J. Random..."

El mismo sentimiento se vuelve más *fuerte* cuando está expresado de una forma menos brusca:

> "Pienso que eso funciona, pero menos de lo ideal en términos de legibilidad y mantenimiento. La propuesta de J. Random evita esos problemas ya que..."

No podrás librarte completamente de la hipérbole, y en general no es necesario hacerlo. Comparada con otras formas retóricas, la hipérbole no es globalmente dañina y perjudica principalmente al autor. Los destinatarios pueden comprender, solamente que el remitente pierde un poco más de credibilidad cada vez. Por lo tanto, para bien de tu influencia en el proyecto, intenta proceder con moderación. De esa manera, cuando *necesitas* presentar un punto fuerte, las personas te tomarán con seriedad.

Corregir dos veces. Para cualquier mensaje más largo que el tamaño medio de un párrafo, se recomienda volver a leerlo de arriba a abajo antes de enviarlo pero después de que lo tengas listo. Éste es un conocido consejo para cualquiera que haya tomado una clase de composición, pero es especialmente importante para las discusiones en línea. Ya que el proceso de composición en línea tiende a ser altamente discontinuo (en el transcurso de escritura de un mensaje, podrías necesitar retroceder y revisar otros correos, visitar ciertas páginas web, ejecutar un comando para capturar su salida de depuración, etc.), es especialmente fácil perder el sentido de tu papel narrativo. Mensajes que fueron escritos discontinuamente y no fueron revisados antes de ser enviados son frecuentemente reconocibles como tal, mucho el disgusto (o uno esperaría) de sus autores. Tómate el tiempo para examinar lo que envías. Cuanto más estructurados sean tus mensajes, más leidos serán.

Tono

Después de escribir miles de mensajes, probablemente notarás que tu estilo tiende a ser extremadamente conciso. Esto parece ser la norma en la mayoría de los foros técnicos, y no hay nada malo con ello. Un nivel de brevedad que sería inaceptable en interacciones sociales normales es sencillamente el común para los hackers de software libre. Aquí está una respuesta a la que yo recurrí una vez en una lista de correo acerca de cierto software gratuito de administración de contenido, citado en su totalidad:

```
¿Puedes explicar exactamente con que problema
te enfrentas?

Además:

¿Qué versión de Slash estás usando? No pude encontrarlo en
tu mensaje original.

¿Exactamente como compilaste el código de apache/mod_perl?

¿Probaste el parche de Apache 2.0 que fue colocado en
slashcode.com?

  Shane
```

Eso es ser conciso! No tiene bienvenida, ni despedida con excepción de su nombre, y el mensaje en sí es solamente una serie de preguntas expresadas de la forma más compacta. Su oración declarativa fue una crítica implícita de mi mensaje original. Aunque, me alegra ver el correo de Shane, y no tomar su brevedad como un producto de cualquier otro motivo que no sea el de ser una persona ocupada. El mero hecho de que él haga preguntas, en vez de ignorar mi mensaje, significa que él es esta dispuesto a dedicarle cierto tiempo a mi problema.

¿Reaccionarán positivamente todos los lectores a este estilo? No necesariamente; depende de la persona y el contexto. Por ejemplo, si una persona envía un correo reconociendo que cometió un error (quizás codificó un fallo), y sabes por experiencias pasadas que esta persona tiende a ser un poco insegura, entonces mientras puedas escribir una respuesta compacta, deberías asegurarte de dejarlo con algo de mención hacia sus sentimientos. La mayor parte de tu respuesta puede ser un breve análisis de la situación desde el punto de vista del ingeniero, tan conciso como quieras. Pero al final, deberías despedirte con algo que indique que la brevedad no debe ser tomada como frialdad. Por ejemplo, si sólo escribiste montones de consejos indicando exactamente como la persona debería corregir el fallo, entonces debes despedirte con "Buena suerte, NOMBRE_DE_LA_PERSONA" para indicar que le deseas suerte y que no eres malgeniado. Una carita sonriente colocada estratégicamente u otro emoticón, también puede con frecuencia ser suficiente para tranquilizar a un interlocutor.

Puede resultar un tanto extraño centrarse en el sentimiento de los colaboradores, asi como tambien en lo superficial de lo que dicen por decirlo de alguna manera sin rodeos, los sentimientos afectan a la productividad. Los sentimientos tambien son importantes por otras razones, porque incluso confinandonos a nosotros mismos a razones puramente utilitarias, podemos notar que la gente infeliz escribe peor software, y/o menos. Dada la naturaleza restrictiva de la mayoria de los medios electronicos, aunque, a menudo no habra indicios patentes de como se siente una persona. Tendras que realizar una adecuada suposicion basandote en a) como se sentiria la mayoria de la gente en esa situacion, y b) que es lo que conoces de esa persona particular a partir de interacciones pasadas. Algunas personas prefieren una actitud mas pasiva, y simplemente estan de acuerdo con todo el mundo sin cuestionarlos, la idea tras esto es que si un participante no dice abiertamente que es lo que piensa, entonces uno no tiene nada que hacer tratandole como pensaba que lo hacia. No comparto este enfoque, por un par de razones. Una, la gente no se comporta de esa manera en la vida real, asi que porque deberian hacerlo online? Dos, dado que la mayoria de las interacciones tienen lugar en foros publicos, la gente tiende a ser incluso mas moderada expresando las emociones que lo podrian ser en privado. Para ser mas preciso, a menudo estan deseando expresar emociones directamente a otros, tales como de agradecimiento o indignacion, pero no emociones directamente intimas como inseguridad u orgullo. Todavía, la mayoría de los humanos trabajan mejor cuando saben que los demás son conscientes de su estado de ánimo. Prestando atención a a pequeñas pistas, normalmente podrás suponerlo acertadamente la mayoría del tiempo, y motivar a la gente a estar involucrada con un mayor grado que de otra manera no podrían.

Por supuesto no quiero decir que, tú rol sea el de un terapeuta de grupo, ayudando constantemente a todo el mundo a estar al corriente de sus sentimientos. Pero poniendo una especial atención a patrones a largo-plazo en el comportamiento de la gente, empezarás a tener una sensación de ellos como individuos incluso aunque nunca los hayas conocido cara a cara. Y siendo sensible en el tono de tus mensajes escritos, podrás tener una cantidad sorprendente de influencia sobre los sentimientos de los demás, que es el último beneficio del proyecto.

Reconociendo la grosería

Una de las características que definen la cultura del código abierto son son las nociones distintivas de qué constituye grosería y qué no. Mientras que los convenios que se describen debajo no son únicos para el desarrollo de software libre, ni tampoco para el software en general debería ser familiar para cualquiera que trabaje en disciplinas de las matemáticas, ciencias puras o la ingeniería el software libre, con sus porosos límites y un constante influjo de recién llegados, es un entorno donde es especialmente probable encontrar estas convenciones por gente no familiarizada con ellas.

Comencemos con las cosas que *no* son groseras (maleducadas):

La crítica técnica, incluso cuando es directa y sin tacto, no es una grosería. De hecho, puede ser una forma de adulación: la crítica es decir, por implicación, que vale la pena tomarse en serio el destinatario,y vale la pena invertir tiempo en él. Es decir, cuanto más viable fuera simplemente ignorar el mensaje de alguien, se entiende más por un cumplido molestarse en criticarlo (a no ser que la crítica se convierta, por su puesto, en un ataque *ad hominem* o alguna otra forma de grosería obvia).

Preguntas directas, sin adornos, como la que Shane me hizo en el correo anterior tampoco es grosería.

Preguntas que, en otros contextos, pueden parecer frias, retóricas e incluso a modo de burla, son formuladas a menudo de una forma seria, y no tienen más intención que obtener información lo más rápido posible. La famosa pregunta del soporte técnico "¿Está su ordenador conectado?" es un ejemplo clásico de esto. La persona de soporte realmente necesita saber si tu ordenador está contectado y, después de unos pocos días en el trabajo, se ha cansado de adornar su pregunta de florituras ("Le pido disculpas, quisiera que me contestara unas simples preguntas para descartar algunas posibilidades. Algunas pueden parecer muy básicas, pero tenga paciencia..."). En este punto, no le importa seguir adornando más simplemente pregunta directamente: ¿está o no está conectado? Preguntas similares se hacen en todo momento en las lista de distribución del software libre. La intención no es insultar al destinatario, sino descartar rápidamente las explicaciones más obvias (y quizás más comunes). Los destinatarios que lo entiendan y reaccionen de ese modo ganarán puntos en tener una visión tolerante sin provocarse. Pero los destinatarios que reaccionen mal tampoco deberían ser reprendidos. Es simplemente una colisión de culturas, no es culpa de nadie. Explica amablemente que tu pregunta (o crítica) no tiene significados ocultos; que solo significaba obtener (o transmitir) la información de la forma más eficientemente posible, nada más.

Entonces, ¿qué es grosería?

Bajo el mismo principo por el cual las críticas a detalles técnicos es una forma de halago, no proporcionar críticas de calidad puede ser un tipo de insulto. No quiero decir simplemente que ignorando el trabajo de alguien, sea una propuesta, cambio en el código, nuevas informaciones o cualquier cosa. A menos que explicitamente prometas una reacción detallada más adelante, normalmente es OK simplemente no reaccionando de ninguna manera. La gente asumirá así que no tuviste tiempo de decir nada. Pero si tu *reaccionas* , no escatimes: tómate el tiempo para analizar detalladamente las cosas, proporcionar ejemplos concretos allá donde sea apropiado, rebuscar a través de los archivos para encontrar información relacionada del pasado, etc. O si no tienes tiempo para realizar todo ese esfuerzo, pero todavía necesitas escribir algún tipo de respuesta corta, entonces exponlo de manera abierta y breve en tu mensaje ("Creo que hay un tema abierto para esto, pero desafortunadamente no tuve tiempo para buscarlo, lo siento"). Lo principal es reconocer la existencia de la norma cultural, ya sea algo satisfactorio o reconociendo abiertamente que ha fallado ligeramente esta vez. Sea lo que sea, la norma es reforzar. Pero el no cumplir esta norma mientras que al mismo tiempo no se explica el *porque* fallaste en conecerlo, es lo mismo que decir el tópico (y aquellos que participan en ello) no mereció tu tiempo. Es mejor mostrar que tu tiempo es muy valioso siendo seco que siendo vago.

Hay muchas otras formas de grosería, por supuesto, pero la mayoría no es específica del desarrollo de software libre, y el sentido común es una buena forma de evitarlas. Véase también “Echad a volar la mala educación” en Capítulo 2, *Primeros Pasos*, si lo has hecho todavía.

Caras

Hay una parte en el cerebro humano dedicada específicamente a reconocer caras. Es conocida informalmente como "área de fusión de caras", y sus capacidades son mayoritariamente innatas , no se han aprendido. Resulta que reconocer a las personas individualmente es una técnica tan crucial de supervivencia que hemos desarrollado un hardware especializado para ello..

La colaboración basada en Internet es por ello psicologicamente curiosa porque implica una estrecha colaboración entre seres humanos que nunca se identificarían entre ellos por los más naturales e intuitivos métodos: reconocimiento facial el primero de todos, pero tambien por el sonido de la voz, postura, etc. Para compensar esto, intenta usar un consistente *Nombre* en todas partes. Debería ser la primera parte de tu dirección de email (la parte antes de el signo @), tu nombre del IRC, tu nombre para hacer commit en los repositorios, tu marca de nombre en cualquier lado y así. Este nombre es tu "cara" online : un tipo de cadena de identificación que sirve el mismo propósito que tu cara real, aunque no lo es, desafortunadamente, estimula el mismo hardware consitutido en el cerebro.

El nombre que muestras debería ser una permutación intuitiva de tu nombre real (el mío por ejemplo, es "kfogel"). En algunas situaciones estará acompañado de tu nombre completo, por ejemplo en las cabeceras del correo:

```
From: "Karl Fogel" <kfogel@whateverdomain.com>
```

Actualmente, hay dos puntos a tener en cuenta en ese ejemplo. Como ya he mencionado anteriormente, el nombre que mostraremos coincidirá con el nombre real de una manera intuitiva. Pero tambien, el nombre real es *real*. Esto es, no se compone de una denominación como:

```
From: "Wonder Hacker" <wonderhacker@whateverdomain.com>
```

Hay una famosa tira cómica de Paul Steiner, del 5 de Julio de 1993 publicada en *The New Yorker*, que muestra a un perro que ha iniciado sesión en un terminal de ordenador, menospreciando y contando a los demás de manera conspiratoria: "En Internet, nadie sabe que tú eres un perro." Este tipo de pensamiento es una mentira detrás de tanto ensalzamiento propio, significado de estar a la moda con las identidades online que la gente se atribuye a ellos mismos; como llamándose uno mismo "Wonder Hacker" causará que la gente piense que uno *es* un maravilloso hacker. Pero los hechos permanecen: incluso si nadie sabe que tu eres un perro, todavía seras un perro. Una fantástica identidad online nunca impresiona a los lectores. En vez de esto, les hace creer que eres más una imagen que una persona con fundamento, o que simplemente eres inseguro. Utiliza tu nombre real para todas las interacciones, o si por alguna razón necesitas un anónimo, entonces crea un nombre que se parezca perfectamente a un nombre real, y úsalo consistentemente.

Además de mantener tu imagen online consistente, hay algunas cosas más que puedes hacer para que resulte más atractiva. Si posees un título oficial (ejem., "doctor", "profesor", "director"), no hagas obstentación de ello, no lo menciones a menos que sea directamente relevante a la conversación. El mundo hacker en general y la cultura del Software Libre en particular, tienden a ver la muestra de títulos como un signo de exclusión y de inseguridad. Esta bien si tu título aparece como parte de un bloque de firma standard al final de cada mail que envías, pero no lo utilices como una herramienta para reforzar tu posición en una discusión; al intentarlo está garantizado el fracaso. Tu quieres que la gente te respete como persona, no por el título.

Hablando de bloques de firma: mantelos pequeños y con buen gusto, o mejor todavía, inexistentes. Evita largas responsabilidades legales fijadas al final de cada mail, especialmente cuando estos expresen sentimientos incompatibles con la participación en un proyecto de software libre. Por ejemplo, el siguiente clásico del género aparece al final de cada post que un usuario particular hace en una lista de mail pública donde yo estoy:

```
IMPORTANT NOTICE

If you have received this e-mail in error or wish to read our e-mail
disclaimer statement and monitoring policy, please refer to the
statement below or contact the sender.

This communication is from Deloitte & Touche LLP.  Deloitte &
Touche LLP is a limited liability partnership registered in England
and Wales with registered number OC303675.  A list of members' names
is available for inspection at Stonecutter Court, 1 Stonecutter
Street, London EC4A 4TR, United Kingdom, the firm's principal place of
business and registered office.  Deloitte & Touche LLP is
authorised and regulated by the Financial Services Authority.

This communication and any attachments contain information which is
confidential and may also be privileged.  It is for the exclusive use
of the intended recipient(s).  If you are not the intended
recipient(s) please note that any form of disclosure, distribution,
copying or use of this communication or the information in it or in
any attachments is strictly prohibited and may be unlawful.  If you
have received this communication in error, please return it with the
```

```
title "received in error" to IT.SECURITY.UK@deloitte.co.uk then delete
the email and destroy any copies of it.

E-mail communications cannot be guaranteed to be secure or error free,
as information could be intercepted, corrupted, amended, lost,
destroyed, arrive late or incomplete, or contain viruses.  We do not
accept liability for any such matters or their consequences.  Anyone
who communicates with us by e-mail is taken to accept the risks in
doing so.

When addressed to our clients, any opinions or advice contained in
this e-mail and any attachments are subject to the terms and
conditions expressed in the governing Deloitte & Touche LLP client
engagement letter.

Opinions, conclusions and other information in this e-mail and any
attachments which do not relate to the official business of the firm
are neither given nor endorsed by it.
```

Para alguien que únicamente se quiere presentar para preguntar alguna cuestión ahora y entonces, esta gran "renuncia" parece un poco fuera de lugar pero probablemente no hace ningún daño. Sin embargo, si esta persona quería participar activamente en el proyecto, este formalismo-legal empezaría a tener un efecto más insidioso. Enviaría al menos dos señales potencialmente destructivas: primero, qué esta persona no tiene un control total sobre sus herramientas; está atrapado dentro de una cuenta de correo corporativa que acarrea un mensaje molesto al final de cada mail, y el no tiene ningúna manera de evitarlo; y segundo, que tiene poco o ningún apoyo de su organización para contribuir en las actividades del software libre. Cierto, que la organización claramente no le ha prohibido completamente de postear en listas públicas, pero hace que sus posts se distingan con un mensaje frío, ya que el riesgo de dejar información confidencial debe figurarse sobre las demás prioridades.

Si trabajas para una organización que insiste en añadir tales bloques de firma en todos los mail salientes, entonces considera tener una cuenta de correo gratuito de, por ejemplo, gmail.google.com, www.hotmail.com, or www.yahoo.com, y utilizar esta dirección para el proyecto.

Evitando los obstáculos corrientes

No envíes un correo sin un propósito

Un obstáculo común en la participación de un proyecto online es pensar que tú tienes que responder a todo. No tienes que hacerlo. Lo primero de todo, normalmente se generarán más hilos de correo de los que tú puedas manejar, al menos después de que el proyecto ha pasado sus primeros meses. Segundo, incluso en los hilos de correo en los que has decidido tomar parte, mucho de lo que comenta la gente no requerirá una respuesta. Los foros de desarrollo en particular tiendes a ser dominados por tres tipos de mensajes:

1. Mensajes proponiendo algo que -no es trivial-
2. Mensajes mostrando apoyo u oposición a algo o a lo que alguien ha dicho.
3. Mensajes de recapitulación

Ninguno de esos*de manera inherente* requerirá una respuesta, particularmente si puedes ser justamente seguro, basándote en revisar el hilo desde el principio, que alguien más probablemente dirá lo que tu ibas a decir de cualquier manera. (Si te preocupa que te tomen en un bucle de esperar-esperar porque todos los demás están usando esta táctica tambien, no lo hagas; casi siempre habrá *alguien* por ahí que se

tenderá a crisparse.) Una respuesta debería ser motivada por un propósito definitivo. Pregúntate a ti mismo primero: ¿Sabes que es lo que quieres conseguir? Y segundo: ¿no se conseguirá a menos que digas algo?

Dos buenas razones para añadir tu voz a un hilo de corre son a) cuando veas un movimiento de proposición y sospeches que tú eres el único que así lo percibe, y b) cuando veas que no hay entendimiento entre otros, y sepas que puedes solucionarlo con un correo clarificándolo todo. Tambien generalmente está bien escribir únicamente para dar las gracias a alguien por algo, o para decir "Yo tambien!", porque un lector puede decir en seguida que tal correo no requiere ninguna respuesta ni acción adicional, y por lo tanto el esfuerzo mental demnadado por el post termina limpliamente cuando el lector llega a la última línea de el correo. Pero incluso entonces, piensalo dos veces antes de decir algo; es siempre mejor dejar a la gente deseando que escribas más a menudo que deseando que escribas menos. (Consulta la segunda parte de Apéndice C, *Why Should I Care What Color the Bikeshed Is?* para ver más ideas sobre como portarse en una lista de correo muy concurrida.)

Hilos productivos vs Hilos Improductivos

En una lista de correo muy concurrida, tienes dos imperativos. Uno, obviamente es comprender en que es lo que necesitas poner tu atención y que es lo que puedes ignorar. El otro es evitar de alguna manera el *causar* ruido: no sólo quieres que tus propios posts tengan un ratio de gran ruido/señal, sino que tambien quieres que sean el tipo de mensajes que estimulan a *otra* gente a escribir mails con un ratio similar de señal/ruido, o no escribir nada.

Para ver como hacer eso, vamos a considerar el contexto en el cual se hace. ¿Cuales son algunos de los sellos de un hilo improductivo?

- Argumentos que ya se han hecho antes se empiezan a repetir, porque el que los hace piensa que nadie le ha escuchado la primera vez.
- Se incrementan los níveles de exageración y participación mientras el interés se hace cada vez más pequeño.
- Una mayoría de comentarios que provienen de gente que hablan poco o nada, mientras que la gente que tiene a hacer las cosas permanece en silencio.
- Muchas ideas se discuten sin un propósito claro de que hacer con ellas. Por supuesto, cualquier idea interesante empieza con una visión imprecisa; la cuestión importante es que dirección tomará a partir de ahí. Parece que el hilo empieza a convertir la visión en algo más concreto, o está derivando en subvisiones y disputas ontológicas?)

Sólo porque un hilo de correo no sea productivo al principio no significa que sea una perdida de tiempo. Puede tratar sobre un tema importante, en cuyo caso el hecho de que no se está produciendo ningún progreseo es todo lo más molesto.

Guiar un hilo de correo hacia la utilidad sin ser agresivo es todo un arte. No funcionará simplemente amonestando a la gente para que pare de gastar su tiempo, o preguntándoles que no escriban a menos que tengan algo constructivo que decir. Por supuesto puedes pensar en esas cosas en privado, pero si lo dices en la lista de correo sonará ofensivo. En lugar de eso, tienes que sugerir condiciones para promover progresos; guía a la gente, un camino a seguir que lleve a los resultados que quieres, y todo ello sin que tu conducta parezca dictatoria. La distinción es en gran parte el tono. Por ejemplo, esto esta mal:

> *Esta discusión no va a ningún lado. Por favor podemos dejar este tema hasta que alguien tenga un parche que implemente una de esas proposiciones? No hay razón para mantenernos en ello todo el rato diciendo las mismas cosas. El código hace más ruido que las palabras, chicos.*

Donde esto esta bien:

> *Varias propuestas han estado flotando en este hilo, pero ninguno ha tenido todos los detalles completos, al menos no demasiados como para hacer una votación arriba-o-abajo. Y todavía no estamos diciendo nada nuevo; estamos simplemente reiterando lo que ya se ha dicho anteriormente. Así que lo mejor a partir de este punto será probablemente para posteriores correos contener tanto una especificación completa para la característica propuesta, o un parche. Entonces al menos tendríamos una acción definitiva que tomar (ejem, tener un consenso en la especificación, o aplicar el parche).*

Compara la segunda propuesta con la primera. La segunda manera no traza una línea entre tú y los demás, ni les acusa de mantener la discusión en una espiral. Habla sobre "nosotros", que es lo importante hayas participado o no en el hilo de correo anteriormente, porque recuerada a todo el mundo que incluso aquellos que han estado en silencio hasta entonces en el hilo de correo todavía pueden participar en el resultado del hilo de correo. Describe porque el hilo no va a ninguna parte, pero lo hace sin peyorativas ni juicios; simplemente muestra el estado de algunos hechos sin sentimiento. Lo más importante, ofrece un curso de acción positivo, de manera que en vez de que la gente sienta que la discusión esta siendo cerrada (una restricción contra la cual ellos pueden sólo estar tentados a rebelar), se sentirán como si se les estuviera ofreciendo una manera de tomar parte en la conversación a un nivel más constructivo. Este es un estándar con el cual la gente querrá quedarse.

Siempre no querrás convertir un hilo de correo en el siguiente nivel de construcción; otras veces querrás dejarlo pasar. El propósito de tu correo, entonces, es hacer una cosa o la otra. Si puedes decir el camino que deberá tomar el hilo de correo de manera que nadie lo está haciendo *así* para tomar los pasos que sugeriste, entonces tu correo ha cerrado el hilo sin aparentar hacerlo. Por supuesto, no hay una manera infalibe de cerrar un hilo, e incluso si la hubiera, no querrías usarla. Pero preguntando a los participantes a crear progresos visibles o parar de escrbiri correos es perfectamente defendible, si se hace diplomáticamente. Sin embargo, se cauteloso de anular los hilos de correo prematuramente. Alguna cantidad de charla especulativa puede llegar a ser productiva, dependiendo del tema, y preguntando para que se resuelva demasiado rápida apagará el proceso creativo, así como tambien te hara parecer impaciente.

Don't expect any thread to stop on a dime. Probablemente habrá todavía unos pocos correos despues del tuyo, ya sea porque los mails se cruzan en la red, o porque la gente quiere tener la última palabra. Esto no es nada por lo que preocuparse, y no necesitas escribir otro correo otra vez. Simplemente deja que el hilo se vaya esfumando o que no se esfume como puede ser el caso. No puedes tener control completo; por otra parte, puedes esperar tener estadísticamente un efecto significativo a través de varios hilos de correo.

Cuanto más blando sea el tema, más largo será el debate

Aunque las discusiones pueden extenderse a cualquier topico, la probabilidad de que se vayan extendiendo va conforme la dificultad tecnica del tema disminuye. Despues de todo cuanta mas sea la dificultad tecnica, menor sera el numero de participantes que realmente podran seguirla. Aquellos quienes pueden ser los desarrolladores mas experimentados, quienes ya han tomado parte en esas discusiones antes cientos de veces, y conocen el tipo de comportamiento es el que va a llevar a un consenso con el cual todo el mundo este de acuerdo.

De esta manera, en cuestiones tecnicas que son simples de comprender y faciles de tener una opinion sobre ellas, es dificil llegar a un consenso, y en temas "blandos" como organizacion, publicidad, ingresos, etc. La gente puede participar en aquellos argumentos siempre, porque no es necesario ninguna cualificacion para hacerlo, no hay un camino claro para decidir (incluso despues de todo) si una decision fue buena o mala, y porque simplemente esperar a que otros discutan es a veces una tactica correcta.

El principio de que la cantidad de discusion es inversamente proporcional a la complejidad del tema tra-

tado, ha estado ahi durante algun tiempo, y es conocido informalmente como el *Efecto Bikeshed* . Aqui esta la explicacion de Poul-Henning Kamp's, de un correo ahora famoso, hecho en la lista de desarrolladores de BSD:

> Es una larga historia o mas bien es una vieja historia, pero es bastante escasa actualmente. C. Northcote Parkinson escribio un libro en los comienzoa de 1960 titulado "La ley de Parkinson", la cual contenia mucho entendimiento sobre la dinamica de la gestion.
>
> [...]
>
> En el ejemplo especifico cubriendo el refugio de bicicletas, el otro componente vital es una planta de energia atomica, supongo que ilustra la epoca de el libro. .
>
> Parkinson nos muestra que puedes ir a un consejo de direccion y conseguir la aprobacion de un edificio multi millonario o incluso de billones de dolares de una planta de energia atomica, pero si quieres construir un refugio de bicicletas te veras implicado en discusiones sin fin.
>
> Parkinson explica que esto es porque una planta de energia atomica es tan enorme, tan cara y tan complicada que la gente no tendra conocimiento de ello, y en lugar de intentarlo, recurriran al supuesto de que alguien revisara todos los detalles antes de ir mas alla. Richard P. Feynmann dio un par de interesantes, y muy cercanos a esto ejemplos en relacion a los Alamos en sus libros.
>
> Por otra parte, un refugio para bicletas. Cualquiera puede construir uno de esos en un fin de semana, y todavia tendra tiempo para ver la TV. Asi que no importa lo bien preparado que estes, tampoco importa lo razonable que seas en tu proposicion, alguien se hara con la oportunidad para demostrar que esta haciendo su trabajo, que esta atento, que esta *ahi*.
>
> En Dinamarca lo llamamos "deja tu huella". Trata sobre el orgullo personal y el prestigio, va sobre ser capaz de señalar en algun sitio y decir "Hay! esto lo hice *Yo*." Es fuerte, simplemente piensa en las pisadas del semento mojado.

(Su post completo es una lectura de mucho valor. Miralo.Apéndice C, *Why Should I Care What Color the Bikeshed Is?*; see also http://bikeshed.com.)

Cualquiera que regularmente tome parte en decisiones hechas en grupo reconocera sobre que es lo que esta hablando Kamp. Sin embargo, normalmente es imposible persuadir a *todo el mundo* a evitar pintar un cobijo de bicis. Lo mejor que puedes hacer es señalar que el fenomeno existe, y cuando veas que esta ocurriendo, persuadir al desarrollador senior; las personas cuyos mails llevan todo el peso; a soltar sus brochas pronto, asi al menos no contribuiran al ruido. Las fiestas para pintar bicis nunca se esfumaran enteramente, pero puedes hacerlas mas cortas y menos frecuentes extendiendo una concienciacion del fenomeno en la cultura del proyecto.

Evitando las Guerras Santas

Una*Guerra Santa* es una disputa, a menudo pero no siempre sobre un tema relativamente menor el cual no se puede resolver con los meritos de los argumentos, pero donde la gente se siente demasiado apasionada para continuar discutiendo de cualquier manera con la esperanza de que su lado prevalecera. Las Guerras Santas no son lo mismo que la pintura de un garaje de bicicletas. La gente de la pintura de bicicletas normalmente salen rapido con una opinion (porque pueden), pero ellos, necesariamente no se sentiran demasiado apasionados sobre ello, y por lo tanto, otras veces, expresaran opiniones incompatibles para mostrar que ellos comprenden todas las caras del tema tratado. Por otra parte, en una Guerra Santa, comprender a las otras partes es un signo de debilidad. En una Guerra Santa, todo el mundo sabe que hay UNA Respuesta Correcta; Per ellos no estan de acuerdo con esta.

Una vez que una Guerra Santa ha empezado, generalmente no se puede resolver con la satisfaccion de todo el mundo. No es bueno mostrar, en el medio de una Guerra Santa, que esta esta teniendo lugar. Todo el mundo ya lo sabe. Desafortunadamente una caracteristica comun de las Guerra Santa es el desacuerdo en cada cuestion *si* la disputa se puede resolver continuando la discusion. Visto desde fuera, esta claro que ninguna parte va a cambiar la opinion de los otros. Visto desde dentro, la otra parte esta siendo obtusa y no esta pensando claramente, pero pueden cambiar de opinion si las cosas se vuelven feas. Ahora,*no* estoy diciendo que no haya una parte con razon en una guerra santa. A veces la hay en las Guerras Santas que yo he participado, siempre ha sido mi bando, por supuesto. Pero no importa porque no hay algoritmo para demostrar convencidamente que una parte o la otra estan en lo cierto.

Un comun, pero insatisfactorio modo de intentar solucinar una Guerra Santa es decir "Ya hemos gastado bastante tiempo y energia de lo que vale discutiendo esto! Por favor, ¿podemos dejarlo? Hay dos problemas en esto. Primero, que el tiempo y la energia ya se han gastado y ya no se pueden recuperar; la unica cuestion ahora es, cuanto esfuerzo *mas* permanecera? Si todavia alguno siente que un poco mas de discusion resolvera la cuestion pronto, entonces todavia tiene sentido (desde su punto de vista) continuar.

El otro problema en preguntar para que la cuestion sea zanjada es que esto es a menudo equivalente a permitir a una parte el status quo, a declarar la victoria por inaccion. Y en algunos casos, el status quo es conocido por ser de cualquier forma inaceptable: todo el mundo esta de acuerdo en que se debe llegar a una decision, se debe tomar alguna accion. Dejar el tema seria peor para todo el mundo que simplemente apoyando el argumento que daria alguien. Pero dado que el dilema se aplica igualmente a todo el mundo, todavia es posible terminar discutiendo por siempre sobre que hacer.

¿Como deberias manejar una Guerra Santa?

Puedes anticipar ciertas Guerras Santa estandar: tienden a tratar sobre lenguajes de programacion, licencias (mira “La GPL y compatibilidad entre licencias” in Capítulo 9, *Licencias, Copyrights y Patentes*), en respuesta a munging (mira “El gran debate del Reply-To” en Capítulo 3, *Infraestructura Técnica*), y algunos otros topicos. Normalmente cada proyecto tiene una o dos Guerras Santas, tambien, las cuales los desarrolladores mas experimentados estaran ya familiarizados. Las tecnicas para frenar las Guerras Santas, o al menos limitar su daño, son casi las mismas en cualquier lugar. Incluso si eres positivo y tu parte es correcta, intenta encontrar *alguna* manera de expresar simpatia y comprension hacia los puntos de vista que los otros hacen. A menudo el problema en una Guerra Santa es porque cada parte ha construido sus muros lo mas alto posible, y dejan claro que cualquier otra opinion es totalmente idiota, el acto de rendirse o cambiar el pensamiento de alguien se hace psicologicamente insostenible: sería un reconocimiento no solamente siendo erróneo, pero habiendo sido *ciertamente*y todavia siendo erróneo. La manera en que puedes hacer este reconocimiento aceptable por la otra parte es expresar alguna duda tu mismo; precisamente mostrando que comprendes sus argumentos y al menos eres sensible a ellos, si no persuasivo finalmente. Haz un gesto que proporcione espacio para un gesto recíproco, y normalmente la situación mejorará. No es ni más ni menos probable que consigas el resultado técnico que querías, pero al menos puedes evitar el daño colateral innecesario a la moral del proyecto.

Cuando una Guerra Santa no se puede evitar, decide pronto cuanto la apoyas, y entonces estáte dispuesto públicamente a ceder. Cuando hagas esto, puedes decir que no estás respaldándola porque la Guerra Santa no lo vale, pero no expreses ningún rencor y *no* tomes la oportunidad para una despedida disparando contra los argumentos de la otra parte. Darse por vencido es efectivo sólo cuando se hace con elegancia.

Las Guerras Santas de lenguajes de programación son un caso especial, porque a menudo son mayormente técnicas, todavía mucha gente se siente cualificada para tomar parte en ellas, y el interes es muy alto, ya que el resultado puede determinar en gran medida en que lenguaje se va a escribir el proyecto. La mejor solución es elegir el lenguaje pronto, con la influencia de los desarrolladores iniciales, y entonces defenderlo en los terrenos en los que eres comfortable escribiendo, *no* en el terreno que sería mejor en el que otro lenguaje se pudiera utilizar. Nunca dejes que la conversación en una comparación académica de lenguajes de programación (esto parece ocurrir especialmente cuando alguien menciona Perl, por alguna razón); éste es un tópico muerto en el que simplemente debes evitar caer.

Para consultar más fondo histórico de las Guerras Santas, mira

http://catb.org/~esr/jargon/html/H/holy-wars.html, y el artículo de Danny Cohen que popularizó el término, http://www.ietf.org/rfc/ien/ien137.txt.

El efecto "Ruido Minoritario"

En cualquier discusion de una lista de correo, es fácil para una pequeña minoría dar la impresión de que hay un gran acuerdo de contrariead, esto es inundando la lista con numerosos y largos emails. Es igual a hacer una maniobra obstruccionista, excepto que la ilusión de la disensión general es incluso más poderosa, porque está dividida entre un número arbitrario de posts discretos y a la mayoría de la gente no le importa seguir la pista de quién ha dicho que, cuando. Sólo tienen una impresión instintiva de que el tema es muy controvertido, y esperan a que el escándalo disminuya.

La mejor manera de contrarrestar este efecto es indicarlo muy claramente y proporcionar pistas respaldadas mostrando cómo de pequeño es el número actual de disidentes comparado a los que están en acuerdo. Para incrementar la disparidad, puedes querer encuestar de manera privada a la gente que ha estado la mayor parte del tiempo en silencio, pero que sospechas que estarán de acuerdo con la mayoría. been mostly silent, but who you suspect would agree with the majority. No digas nada que sugiera que los disidentes estaban intentando deliberadamente inflar la impresión que estaban creando. Oportunidades que no tuvieron, e incluso si las tuvieron no había una ventaja estratégica para señalarla. Todo lo que necesitas es mostrar el número actual en una comparación cara-a-cara, y la gente se dara cuenta que su intuición de la situación no coincidía con la realidad.

Este consejo no sólo se aplica a temas con una clara posición a-favor-en-contra. Se aplica a cualquier discusión donde hay un alboroto, pero no esta claro que la mayoría de la gente considere ese tema bajo discusión que sea un problema real. Despues de todo, si estas de acuerdo en que el tema no es digno de acción, y puedes ver que ha fallado en atraer la atención (incluso si ha generado muchos mails), puedes observar públicamente que no está teniendo tracción. Si el efecto "ruido minoritario" ha funcionado, tu post parecerá un soplo de aire fresco. La mayoría de la impresión de la gente de la discusión se dará cuenta de que ese punto habrá sido algo turbio: Huh, seguro que sienten como que hay un gran acuerdo aqui, porque seguramente hay un montón de posts, pero no puedo ver que esté habiendo ningún progreso claro." Explicando como la manera en que la discusión se hizo parezca más turbulenta de lo que realmente es, tú retrospectivamente le darás una nueva forma, a través de la cual la gente pueda recapitular su comprensión del resultado.

Gente difícil

No es tan fácil tratar con gente díficl en foros electrónicos como lo sería en persona. Con "díficil" no me refiero a "maleducados". La gente maleducada es molesta, pero no son necesariamente díficiles. En este libro ya se ha discutido como manejarlos:comenta la grsoería la primera vez, y desde entonces ignórales o tratalos como otro cualquiera. Si continuan siendo maleducados, ellos mismos se haran tan impopulares que no tendrán influencia en nadie del proyecto, por lo que serán un problema de ellos mismos.

Los casos realmente difíciles son la gente que no son manifiestamente groseros, pero que manipulan o abusan en los procesos del proyecto de una manera que termina costando el tiempo y la energía de otras personas, y todo ello sin traer ningún beneficio al proyecto. Tales personas a menudo buscan puntos de presión en los procedimientos del proyecto, para darse a sí mismos más influencia que de otra manera no tendrían. Esto es mucho más insidioso que la grosería meramente, porque ni el comportamiento ni el daño que causa es aparente a los observadores casuales. Un ejemplo clásico es aquellos que realizan maniobras obstruccionistas, en la que alguien (siempre sonando tan razonable como sea posible, por supuesto) viene demandando que la cuestión bajo discusión no esta lista para una solución, y ofrece más y más posibles soluciones, o nuevos puntos de vista de viejas soluciones, cuando lo que realmente está pasando es que el sentido de un consenso o votación está a punto de ocurrir, y no le gusta por donde va encaminado. Otro ejemplo es cuando hay un debate que no converge en consenso, pero el grupo al menos intenta clarificar los puntos en desacuerdo y produce un sumario para que todo el mundo se refiera a partir de el. El obstruccionista, que sabe que el sumario puede llevar a un punto que a el no le va a gustar, a menudo intentará retrasar el sumario, implacablemente mediante complicadas cuestiones que deberían estar ahí, u objetando sugerencias razonables, o mediante la introducción de nuevos asuntos.

Tratando con gente difícil

Para contrarrestar tal comportamiento, ayuda el comprender la mentalidad de aquellos que caen en él. La gente generalmente no lo hara conscientemente. Nadie se levanta por la mañana y se dice a sí mismo: "Hoy voy a manipular cínicamente las formas y procedimientos para ser así un irritante obstruccionista." En cambio, tales acciones están a menudo precedidas por un sentimiento de semi-paranoia de estar fuera de las interacciones y decisiones del grupo. La persona piensa que no se le toma seriamente, o (en casos más severos), que existe una conspiración contra él;y que los otros miembros del proyecto han decidido formar un club exclusivo, del cual el no es miembro. Esto entonces justifica en su mente, a tomar las reglas literalmente y encargándose de una manipulación formal de los procedimientos del proyecto, para así *hacer* que todo el mundo le tome en serio. En casos extremos, la persona puede incluso pensar que está luchando una batalla solo para salvar el proyecto de sí mismo.

Es la naturaleza del propio ataque la que hara que nadie se percate de él al mismo tiempo, y mucha gente no lo notará, a menos que se presente con evidencias muy fuertes. Esto significa que neutralizarlo puede llevar algo de trabajo. No basta con persuadirse a sí mismo de que está ocurriendo; tendrás que organizar muy bien las evidencias para persuadir a los demás de lo que está ocurriendo, y entonces tendrás que distribuir estas evidencias de una manera atenta.

Dado que hay mucho por lo que luchar, a menudo la mejor opción es tolerarlo de vez en cuando. Piensa en esto como un parásito, esto es una dolencia suave: si no es muy debilitante, el proyecto podrá afrontar el permanecer infectado, y la medicina podría tener efectos perjudiciales. Sin embargo, si consigue más daño del que se pueda tolerar, entonces es tiempo de entrar en acción. Empieza reuniendo notas de los patrones que observas. Asegurate de incluir referencias a archivos públicos; esta es una de las razones por la que los proyectos mantiene históricos, para que puedas usarlos tambien. Una vez que tengas una buena recopilación, empieza a entablar conversaciones privadas con otros participantes del proyecto. No les digas lo que has observado; en vez de eso, pregúntales primero que es lo que observan ellos. Esta puede ser tu última oportunidad de conseguir feedback sin filtrar sobre como los demás observan el comportamiento de los que crean problemas; una vez que has empezado a hablar abiertamente, la opinión se polarizará y nadie será capaz de recordar que es lo que anteriormente opinaba sobre el tema en cuestión.

Si las discusiones privadas indican que tambien hay otros que perciben el problema, entonces es hora de hacer algo. Aquí es donde tienes que ser *realmente* cauteloso, porque sera muy fácil para este tipo de persona hacer parecer como que tú eres el que actua injustamente. Hagas lo que hagas, nunca acuses de abusar maliciosamente de los procedimientos del proyecto, o de ser paranoico, o, en general, de cualquier otra cosa que sospeches que probablemente sea cierta. Tu estrategia deberá mostrarse tanto razonable como consciente del bienestar global del proyecto. Con el objetivo de reformar la actitud de la persona, o de expulsarla del proyecto permanentemente. Dependiendo de los otros desarrolladores, y de tu relación con ellos, puede ser ventajoso conseguir aliados de manera privada primero. O puede que no; ya que puede dificultar el ambiente interno, si la gente piensa que te estas dedicando a una campaña de falsos e impropios rumores.

Recuerda que aunque la otra persona sea la que se este portando destructivamente *tu* seras la que parezca destructiva si le culpas públicamente y no lo puedes probar. Asegurate de tener varios ejemplos y demostrar lo que estas diciendo, y dilo tan suave como puedes pero siendo directo. Puede que no persuadas a la persona en cuestión, pero estará bien mientras puedas persuadir a los demás.

Estudio del caso

Recuerdo sólo una situación, en más de 10 años trabajando en Software Libre, donde las cosas fueron tan mal, que nosotros tuvimos que preguntar a alguien para que parase de postear completamente. Como era tan a menudo el caso, el no era maleducado y quería sinceramente ser de utilidad. Simplemente no sabía cuando escribir a la lista y cuando no hacerlo. Nuestras listas estaban abiertas al público, y él escribía muy a menudo, preguntando cuestiones de diferentes temas, que empezó a ser un problema de ruido para la comunidad. Nosotros habíamos intentado preguntarle de buenas maneras para que hiciera un poco más de investigación para las respuestas antes de escribir a la lista, pero no hizo efecto.

La estrategia que al final funcionó es un ejemplo perfecto de como construir una situación neutral, y con datos cuantitativos. Uno de los cuatro desarrolladores hizo una exploración en los archivos, y envío entonces el siguiente mensaje de manera privada a unos pocos desarrolladores. El ofendido (el tercer nombre en la lista de abajo, se muestra aquí como "J. Random") tenía muy poca historia con el proyecto, y no había contribuido ni con código ni documentación. Y aún así era el tercero más activo en escribir mensajes en la lista de correo:

```
From: "Brian W. Fitzpatrick" <fitz@collab.net>
To: [... recipient list omitted for anonymity ...]
Subject: The Subversion Energy Sink
Date: Wed, 12 Nov 2003 23:37:47 -0600

En los últimos 25 días, el top de los 6 que más han escrito en la lista de svn [de

    294  kfogel@collab.net
    236  "C. Michael Pilato" <cmpilato@collab.net>
    220  "J. Random" <jrandom@problematic-poster.com>
    176  Branko #ibej <brane@xbc.nu>
    130  Philip Martin <philip@codematters.co.uk>
    126  Ben Collins-Sussman <sussman@collab.net>

Diría que cinco de esas personas están contribuyendo con éxito al desarrollo
de la versión 1.0 de subversión en un futuro cercano.

Tambien diría que una de esas personas está constantemente atrayendo tiempo y ener
otras cinco, sin mencionar a la lista como un todo, así, (aunque no intencionadame
el desarrollo de Subversion. No hice un análisis de los hilos de correo, pero haci
en mi archivo de correo me muestra que a cada correo de esta persona le responde a
dos de los otros cinco de la lista anterior.

Creo que algún tipo de intervención radical es necesaria en esto, incluso si nos a
susodicho se marche. Se ha comprobado que la finura y amabilidad aquí no tienen ef

dev@subversion es una lista de correo para facilitar el desarrollo de un sistema d
no una sesión de terapia de grupo.

-Fitz, intentando abrir camino con dificultad por el correo de svn de tres días qu
```

Aunque no pueda parecerlo al principio, el comportamiento de J. Random's era un clásico de abuso de los procedimientos del proyecto. El no estaba haciendo nada obvio más que intentando obstruccionar en los votos, y estaba aprovechándose de la ventaja de la política de la lista de correo de depender en la propia moderación de sus miembros. Dejamos al juicio de cada individuo en lo que escribe y sobre que materias. De esta manera, no teníamos recursos de procedimiento para tratar con aquellos que no tenían buen juicio, o que no lo practicaban. No había ninguna regla que pudieras apuntar e indicar que se estaba violando, aunque todo el mundo ya sabía que sus frecuentes correos se estaban convirtiendo en un problema serio.

La estrategia de Fitz era retrospectivamente maestra. El recopilo una cantidad de evidencia irrefutable, y entonces la distribuyo discretamente, enviándola primero a unas pocas personas cuyo soporte sería clave en una acción drástica. Ellos estuvieron de acuerdo en que era necesaria algún tipo de acción, y al final llamamos a J. Random por teléfono, le describimos el problema directamente, y le preguntamos para que simplemente parase de escribir correos a la lista. El nunca comprendio realmente las razones de ello; si hubiera sido capaza de comprenderlo, probablemente hubiera ejercido un juicio apropiado en primer lugar. Pero el acordó en parar de escribir correos, y la lista de correo se convirtio en útil de nuevo. Una de las razones por las que esta estrategia funcionó fue quizás, la amenaza implícita con la que hubieramos empezado a restringir sus posts vía el software de moderación que normalmente se utiliza para prevenir el spam (consulta “Prevenir el Spam” en Capítulo 3, *Infraestructura Técnica*). Pero la razón por la que fuimos capaces de aquella opción en reserva fue que Fitz había recopilado el apoyo necesario de la gente clave en primer lugar.

Manejando el crecimiento

El precio del éxito es muy pesado en el mundo del Open Source. Conforme tu software se hace más popular, el número de gente que empieza a buscar información sobre él, se incrementa dramaticamente, mientras el número de gente capaza de proporcionar información se incrementa mucho más despacio. Además, incluso si el ratio fuera uniformemente balanceado, todavía existiría un problema de escalabilidad en la forma en que la mayoría de los proyectos Open source manejan las comunicaciones. Considera por ejemplo las listas de correo. La mayoría de los proyectos tienen una lista de correo para cuestiones generales de los usuarios; a veces los nombres de estas listas son "usuarios", "discusiones", o "ayuda" o algo similar. Cualquiera que sea su nombre, el propósito de esas listas es el mismo: proporcionar un lugar donde la gente pueda resolver sus cuestiones, mientras otros observan y (presumiblemente) absorben conocimiento de la observación de ese intercambio de conocimiento.

Estas listas de correo funcionan muy bien hasta unos pocos miles de usuarios y/o un par de cientos de posts al día. Pero más o menos, a partir de ahí el sistema empieza a romperse, porque cada suscriptor vee cada post; si el número de post a la lista empieza a exceder lo que cualquier lector individual puede procesar en un día, la lista se convierte en una carga para sus miembros. Imagina por ejemplo, si Microsoft tuviera tal lista de correo para Windows XP. Windows XP tiene cientos de millones de usuarios; aún incluso si el uno por ciento de ellos tuviera cuestiones en un periodo de veinticuatro horas, entonces esta lista hipotética cientos de miles de posts al día! Por supuesto, tal lista de correo no podría existir, porque nadie permanecería subscrito. Este problema no esta limitado a las listas de correo; la misma lógica se aplica a los canales del IRC, los foros de discusión online y por ende, a cualquier sistema en el cual un grupo escuche preguntas de individuos. Las implicaciones son siniestras: el modelo usual del Open Source del soporte masivamente paralelizado simplemente no escala los niveles necesarios para la dominación mundial.

No habrá una explosión cuando los foros alcancen su punto de ruptura. Se trata simplemente de un efecto silencioso de feedback negativo: la gente se borrará de las listas, o saldrán de los canales del IRC, o a cualquier ritmo cesaran de preocuparse en preguntar cuestiones, porque verán que no se les escuchará entre tanta gente. Así cuanta más gente haga de estas su principal elección racional, la actividad de los foros empezará a permanecer a un nivel inmanejable precisamente porque la gente racional o (al menos experimentada), empezará a buscar información por otros medios, mientras la gente sin experiencia permanecerá y continuará preguntando en foros y listas de correo. En otras palabras, uno de los efectos de continuar con el uso de modelos de comunicación que no son escalables mientras que el proyecto crece es que la calidad media tanto de preguntas y respuestas tiene a disminuir, lo cual hace que los nuevos usuarios parezcan más tontos de lo que son, cuando de hecho probablemente no lo sean. Se trata simplemente de que el ratio beneficio/costo de el uso de esos foros masificados, disminuye, por lo que de manera natural, aquellos con experiencia, empezarán a buscar respuestas en otros sitios. Ajustar los mecanismos de comunicación para poder con el crecimiento del proyecto, implicará dos estrategias relacionadas:

1. Reconociendo cuando partes especiales de un foro *no* sufren un crecimiento desmesurado, incluso si el foro se trata como un todo, y separando aquellas partes creando otras nuevas, en foros más especializados (ejem., no dejes que los buenos se arrastren por los malos).

2. Asegurando que existen muchas fuentes de información automáticas disponibles, y que se mantienen organizadas, actualizadas y fáciles de localizar.

La estrategia (1) normalmente no es muy dura. La mayoría de los proyectos empiezan con un foro principal: y una lista de correo para discusiones generales, en las cuales las ideas de características, cuestiones de diseño y problemas de codificación puedan ser discutidos. Todo el mundo involucrado en el proyecto está en la lista. Despues de un tiempo, se comprueba que la lista ha evolucionado en varias sublistas basadas en diferentes temáticas. Por ejemplo, algunos hilos son claramente sobre desarrollo y diseño; otros son dudad de usuarios del tipo ¿"Cómo hago tal cosa"?; quizá exista una tercera temática centrada en el registro de procesar los informes de bugs y peticiones de mejora; y así. Un individuo dado, por supuesto puede participar en varios tipos diferentes de hilos, pero lo más importante de todo es que no hay

mucho solapamiento entre los diferentes tipos mismos. Pueden ser divididos en listas separadas sin causar ningún perjuicio en el proyecto, porque los hilos se mantienen repartidos por temáticas.

Actualmente, realizar esta división es un proceso de dos pasos. Creas la nueva lista (o el canal IRC, o lo que vaya a ser), y entonces gastas el tiempo necesario de manera educada pero insistiendo y recordando a la gente a *usar* los nuevos foros apropiadamente. Este paso puede llevar semanas pero finalmente la gente captará la idea. Simplemente tienes que hacer ver a alguien que envía un post al destino equivocado, cual es el nuevo camino y hacerlo de manera visible, animando a que otras personas ayuden tambien en los nuevos usuos. Es tambien muy útil tener una página web proporcionando una guía hacía todas las listas disponibles; tus respuestas simplemente pueden referenciar esta página y, como gratificación, el destinatario puede aprender sobre las pautas a seguir antes de escribir un correo.

La estrategia (2) es un proceso en curso, dura durante todo el tiempo de vida del proyecto e involucra a muchos participantes. Por supuesto es en parte cuestión de tener una documentación actualizada (mira “Documentación” en Capítulo 2, *Primeros Pasos*) y asegurándote que la gente vaya ahí. Pero es tambien mucho más que eso; las secciones que siguen discuten esta estrategia en detalle.

Sobresaliente uso de los archivos

Tipicamente, todas las comunicaciones de un proyecto Open Source (excepto algunas veces conversaciones en el IRC), son archivadas. Los archivos son públicos y se pueden buscar, y tienen una estabilidad referencial: que significa, una vez que una pieza de información se ha grabado en una dirección particular, permanece en esa dirección para siempre.

Usa estos archivos tanto como puedas, y tan visiblemente como sea posible. Incluso cuando sepas la respuesta a alguna pregunta, si piensas que existe una referencia en los archivos que contiene la respuestas, gasta el tiempo necesario para buscarla y presentarla. Cada vez que hagas esto de una manera públicamente visible, algunas personas aprenderan la primera vez que significan esos archivos, y que buscando en ellos pueden encontrar respuestas. Tambien, refiriéndose a los archivos en vez de reescribir la respuesta, refuerzas la norma social contra la duplicación de información. ¿Por qué obtenemos la misma respuesta en dos sitios diferentes? Cuando el número de sitios que se puede encontrar es mantenido a un mínimo, la gente que lo ha encontrado antes están más predispuestos a recordar qué y donde buscarlo para las próximas veces. Las referencias bien situadas tambien contribuyen a la calidad de los resultados de búsqueda en general, porque ellos refuerzan los recursos del objetivo en los rankings de los motores de búsqueda en Internet.

Sin embargo, hay veces en las que duplicar la información tiene sentido. Por ejemplo, supon que hay una respuesta en los archivos, que no es de tí, diciendo:

```
Parece que los índices Scanley indexes han sido corrompidos. Para devolverlos a
su estado original, ejecuta estos pasos:

1. Apaga el servidor Scanley.
2. Ejecuta el programa 'descorromper' que viene con Scanley.
3. Inicia el servidor.
```

Entonces, meses después, ves otro mail indicando que algunos indices han sido corrompidos. Buscas los archivos y presentas la vieja respuesta anterior, pero te das cuenta que faltan algunos pasos (quizás por error, o quizá porque el software ha cambiado desde que se escribió ese post). La clásica manera para manejar esto, es escribir un nuevo mail, con un conjunto de instrucciones más completo, y explicitamente dar como obsoleto el anterior post mencionándolo así:

```
Parece que tus índices Scanley han sido corrompidos. Vimos este problem allá por J
y J. Random publicó una solución en http://blahblahblah/blah. Abajo hay una descri
más completa de como recuperar tus índices, basado en las instrucciones de J. Rand
pero extendiéndolo un poco más:
```

```
1. Para el servidor Scanley.
2. Cambiate al usuario con el que se ejecuta el servidor Scanley.
3. Como este usuario, ejecuta el programa 'recuperar' en los índices.
4. Ejecuta Scanley a mano para ver si los índices funcionan ahora.
5. Reinicia el servidor.
```

(En un mundo ideal, sería posible poner una nota en el viejo post, indicando que existe información más actualizada y apuntando al nuevo post que la contiene. Sin embargo, no conozco ningún software de archivación que ofrezca una característica "obsoleto por", quizá porque sería muy difícil de implementar de una manera en que no viole la integridad de los archivos. Esta es otra razón de porqué es buena idea crear páginas web con respuestas a cuestiones comunes.

Los archivos probablemente son buscados más a menudo para respuestas a cuestiones técnicas, pero su importancia para el proyecto va más allá de eso. Si una pauta formal del proyecto son sus leyes establecidas, los archivos son su ley común: una grabación de todas las decisiones hechas y como se llegó hasta ellas. En cualquier discusión recurrente, actualmente es casi obligatorio empezar con una búsqueda en los archivos. Esto permite empezar la discusión con un sumario del estado actual de las cosas, anticipandose a objeciones, preparando refutaciones y posiblemente descubriendo ángulos que no habías imaginado. También los otros participantes *esperan* de ti que hayas hecho una búsqueda en los archivos. Incluso si las discusiones previas no llevaron a ninguna parte, tú deberías incluir sugerencias cuando vuelvas al téma, para que la gente pueda ver por si mismos a) que no llegaron a ningun consenso, y b) que tú hiciste tu trabajo, y por tanto que probablemente se este diciendo algo ahora que no se dijo anteriormente.

Treat all resources like archives

Todos los consejos anteriores son extensibles más allá de los archivos de las listas de mail. Tener piezas particulares de información de manera estable, y en direcciones que se puedan encontrar convenientemente debería ser un principio de organización para toda la información de un proyecto. Vamos a ver la FAQ como un caso de estudio.

¿Cómo usa la gente una FAQ?

1. Buscan palabras y frases específicas.
2. Quieren poder navegarla, disfrutando de la información sin buscar necesariamente respuestas a cuestiones específicas.
3. Esperan que motores de búsqueda como google conozcan el contenido de la FAQ, de manera que las búsquedas puedan ser entradas en la FAQ.
4. Quieren ser capaces de dirigirse directamente a otra gente en temas específicos en la FAQ.
5. Quieren ser capaces de añadir nuevo material a la FAQ, pero hay que ver que esto ocurre menos a menudo que la búsqueda de respuestas —Las FAQs son de lejos mucho más leidas que escritas.

El punto 1 implica que la FAQ debería estar disponible en algún tipo de formato textual. Los puntos 2 y 3 implican que la FAQ debería estar disponible en forma de página HTML, con el punto 2 indicando adicionalmnente que el HTML debería ser diseñado con legibilidad (ejem., necesitaras algún tipo de control sobre su apariencia), y debería tener una tabla de contenidos. El punto 4 significa que cada entrada individual en la FAQ debería ser asignada como un HTML *named anchor*, un tag que permite a la gente alcanzar un sitio particular en la página. El punto 5 significa que los ficheros fuente de la FAQ deberían estar disponibles de una manera conveniente (ver “Versiones de todo” en Capítulo 3, *Infraestructura Técnica*), un formato que sea fácil de editar.

Named Anchors and ID Attributes

There are two ways to get a browser to jump to a specific location within a web page: named anchors and id attributes.

A *named anchor* is just a normal HTML anchor element (`<a>...</a>`), but with a "name" attribute:

```
<a name="mylabel">...</a>
```

More recent versions of HTML support a generic *id attribute*, which can be attached to any HTML element, not just to `<a>`. For example:

```
<p id="mylabel">...</p>
```

Both named anchors and id attributes are used in the same way. One appends a hash mark and the label to a URL, to cause the browser to jump straight to that spot in the page:

```
http://myproject.example.com/faq.html#mylabel
```

Virtually all browsers support named anchors; most modern browsers support the id attribute. To play it safe, I would recommend using either named anchors alone, or named anchors *and* id attributes together (with the same label for both in a given pair, of course). Named anchors cannot be self-closing—even if there's no text inside the element, you must still write it in two-sided form:

```
<a name="mylabel"></a>
```

...though normally there would be some text, such as the title of a section.

Whether you use a named anchor, or an id attribute, or both, remember that the label will not be visible to someone who browses to that location without using the label. But such a person might want to discover the label for a particular location, so they can mail the URL for a FAQ answer to a friend, for example. To help them do this, add a *title attribute* to the same element(s) where you added the "name" and/or "id" attribute, for example:

```
<a name="mylabel" title="#mylabel">...</a>
```

When the mouse pointer is held over the text inside the title-attributed element, most browsers will pop up a tiny box showing the title. I usually include the hash-sign, to remind the user that this is what she would put at the end of the URL to jump straight to this location next time.

Formatting the FAQ like this is just one example of how to make a resource presentable. The same properties—direct searchability, availability to major Internet search engines, browsability, referential stability, and (where applicable) editability—apply to other web pages, the source code tree, the bug tracker, etc. It just happens that most mailing list archiving software long ago recognized the importance of these properties, which is why mailing lists tend to have this functionality natively, while other formats may require some extra effort on the maintainer's part (Capítulo 8, *Coordinando a los Voluntarios* discusses how to spread that maintenance burden across many volunteers).

Codifying Tradition

As a project acquires history and complexity, the amount of data each incoming participant must absorb

increases. Those who have been with the project a long time were able to learn, and invent, the project's conventions as they went along. They will often not be consciously aware of what a huge body of tradition has accumulated, and may be surprised at how many missteps recent newcomers seem to make. Of course, the issue is not that the newcomers are of any lower quality than before; it's that they face a bigger acculturation burden than newcomers did in the past.

The traditions a project accumulates are as much about how to communicate and preserve information as they are about coding standards and other technical minutae. We've already looked at both sorts of standards, in “Documentación para Desarrolladores” in Capítulo 2, *Primeros Pasos* and “Tomando Nota de Todo” in Capítulo 4, *Infraestructura Social y Política* respectively, and examples are given there. What this section is about is how to keep such guidelines up-to-date as the project evolves, especially guidelines about how communications are managed, because those are the ones that change the most as the project grows in size and complexity.

First, watch for patterns in how people get confused. If you see the same situations coming up over and over, especially with new participants, chances are there is a guideline that needs to be documented but isn't. Second, don't get tired of saying the same things over and over again, and don't *sound* like you're tired of saying them. You and other project veterans will have to repeat yourselves often; this is an inevitable side effect of the arrival of newcomers.

Every web page, every mailing list message, and every IRC channel should be considered advertising space—not for commercial advertisements, but for ads about your project's own resources. What you put in that space depends on the demographics of those likely to read it. An IRC channel for user questions, for example, is likely to get people who have never interacted with the project before—often someone who has just installed the software, and has a question he'd like answered immediately (after all, if it could wait, he'd have sent it to a mailing list instead, which would probably use less of his total time, although it would take longer for an answer to come back). People usually don't make a permanent investment in the IRC channel; they'll show up, ask their question, and leave.

Therefore, the channel topic should be aimed at people looking for technical answers about the software *right now*, rather than at, say, people who might get involved with the project in a long term way and for whom community interaction guidelines might be more appropriate. Here's how a really busy channel handles it (compare this with the earlier example in “IRC / Sistemas de Chat en Tiempo Real” in Capítulo 3, *Infraestructura Técnica*):

```
You are now talking on #linuxhelp

Topic for #linuxhelp is Please READ
http://www.catb.org/~esr/faqs/smart-questions.html &&
http://www.tldp.org/docs.html#howto BEFORE asking questions | Channel
rules are at http://www.nerdfest.org/lh_rules.html | Please consult
http://kerneltrap.org/node/view/799 before asking about upgrading to a
2.6.x kernel | memory read possible: http://tinyurl.com/4s6mc ->
update to 2.6.8.1 or 2.4.27 | hash algo disaster: http://tinyurl.com/6w8rf
| reiser4 out
```

With mailing lists, the "ad space" is a tiny footer appended to every message. Most projects put subscription/unsubscription instructions there, and perhaps a pointer to the project's home page or FAQ page as well. You might think that anyone subscribed to the list would know where to find those things, and they probably do—but many more people than just subscribers see those mailing list messages. An archived post may be linked to from many places; indeed, some posts become so widely known that they eventually have more readers off the list than on it.

Formatting can make a big difference. For example, in the Subversion project, we were having limited success using the bug-filtering technique described in “Pre-filtrado del gestor de fallos” in Capítulo 3, *Infraestructura Técnica*. Many bogus bug reports were still being filed by inexperienced people, and each time it happened, the filer had to be educated in exactly the same way as the 500 people before

him. One day, after one of our developers had finally gotten to the end of his rope and flamed some poor user who didn't read the issue tracker guidelines carefully enough, another developer decided this pattern had gone on long enough. He suggested that we reformat the issue tracker front page so that the most important part, the injunction to discuss the bug on the mailing lists or IRC channels before filing, would stand out in huge, bold red letters, on a bright yellow background, centered prominently above everything else on the page. We did so (you can see the results at http://subversion.tigris.org/project_issues.html), and the result was a noticeable drop in the rate of bogus issue filings. We still get them, of course—we always will—but the rate has slowed considerably, even as the number of users increases. The outcome is not only that the bug database contains less junk, but that those who respond to issue filings stay in a better mood, and are more likely to remain friendly when responding to one of the now-rare bogus filings. This improves both the project's image and the mental health of its volunteers.

The lesson for us was that merely writing up the guidelines was not enough. We also had to put them where they'd be seen by those who need them most, and format them in such a way that their status as introductory material would be immediately clear to people unfamiliar with the project.

Static web pages are not the only venue for advertising the project's customs. A certain amount of interactive policing (in the friendly-reminder sense, not the handcuffs-and-jail sense) is also required. All peer review, even the commit reviews described in “Practicad revisiones visibles del código” in Capítulo 2, *Primeros Pasos*, should include review of people's conformance or non-conformance with project norms, especially with regard to communications conventions.

Another example from the Subversion project: we settled on a convention of "r12908" to mean "revision 12908 in the version control repository." The lower-case "r" prefix is easy to type, and because it's half the height of the digits, it makes an easily-recognizable block of text when combined with the digits. Of course, settling on the convention doesn't mean that everyone will begin using it consistently right away. Thus, when a commit mail comes in with a log message like this:

```
------------------------------------------------------------------------
r12908 | qsimon | 2005-02-02 14:15:06 -0600 (Wed, 02 Feb 2005) | 4 lines

Patch from J. Random Contributor <jrcontrib@gmail.com>

* trunk/contrib/client-side/psvn/psvn.el:
  Fixed some typos from revision 12828.
------------------------------------------------------------------------
```

...part of reviewing that commit is to say "By the way, please use 'r12828', not 'revision 12828' when referring to past changes." This isn't just pedantry; it's important as much for automatic parsability as for human readership.

By following the general principle that there should be canonical referral methods for common entities, and that these referral methods should be used consistently everywhere, the project in effect exports certain standards. Those standards enable people to write tools that present the project's communications in more useable ways—for example, a revision formatted as "r12828" could be transformed into a live link into the repository browsing system. This would be harder to do if the revision were written as "revision 12828", both because that form could be divided across a line break, and because it's less distinct (the word "revision" will often appear alone, and groups of numbers will often appear alone, whereas the combination "r12828" can only mean a revision number). Similar concerns apply to issue numbers, FAQ items (hint: use a URL with a named anchor, as described in Named Anchors and ID Attributes), etc.

Even for entities where there is not an obvious short, canonical form, people should still be encouraged to provide key pieces of information consistently. For example, when referring to a mailing list message, don't just give the sender and subject; also give the archive URL *and* the Message-ID header. The last allows people who have their own copy of the mailing list (people sometimes keep offline copies, for example to use on a laptop while traveling) to unambiguously identify the right message even if they don't have access to the archives. The sender and subject wouldn't be enough, because the same person

might make several posts in the same thread, even on the same day.

The more a project grows, the more important this sort of consistency becomes. Consistency means that everywhere people look, they see the same patterns being followed, so they know to follow those patterns themselves. This, in turn, reduces the number of questions they need to ask. The burden of having a million readers is no greater than that of having one; scalability problems start to arise only when a certain percentage of those readers ask questions. As a project grows, therefore, it must reduce that percentage by increasing the density and accessibility of information, so that any given person is more likely to find what he needs without having to ask.

No Conversations in the Bug Tracker

In any project that's making active use of its bug tracker, there is always a danger of the tracker turning into a discussion forum itself, even though the mailing lists would really be better. Usually it starts off innocently enough: someone annotates an issue with, say, a proposed solution, or a partial patch. Someone else sees this, realizes there are problems with the solution, and attaches another annotation pointing out the problems. The first person responds, again by appending to the issue...and so it goes.

The problem with this is, first, that the bug tracker is a pretty cumbersome place to have a discussion, and second, that other people may not be paying attention—after all, they expect development discussion to happen on the development mailing list, so that's where they look for it. They may not be subscribed to the issue changes list at all, and even if they are, they may not follow it very closely.

But exactly where in the process did something go wrong? Was it when the original person attached her solution to the issue—should she have posted it to the list instead? Or was it when the second person responded in the issue, instead of on the list?

There isn't one right answer, but there is a general principle: if you're just adding data to an issue, then do it in the tracker, but if you're starting a *conversation*, then do it on the mailing list. You may not always be able to tell which is the case, but just use your best judgement. For example, when attaching a patch that contains a potentially controversial solution, you might be able to anticipate that people are going to have questions about it. So even though you would normally attach the patch to the issue (assuming you don't want to or can't commit the change directly), in this case you might choose to post it to a mailing list instead. At any rate, there eventually will come a point in the exchange where one party or the other can tell that it is about to go from mere appending of data to an actual conversation—in the example that started this section, that would be the second respondent, who on realizing that there were problems with the patch, could predict that a real conversation is about to ensue, and therefore that it should be held in the appropriate medium.

To use a mathematical analogy, if the information looks like it will be quickly convergent, then put it directly in the bug tracker; if it looks like it will be divergent, then a mailing list or IRC channel would be a better place.

This doesn't mean there should never be any exchanges in the bug tracker. Asking for more details of the reproduction recipe from the original reporter tends to be a highly convergent process, for instance. The person's response is unlikely to raise new issues; it's simply going to flesh out information already filed. There's no need to distract the mailing list with that process; by all means, take care of it with a series of comments in the tracker. Likewise, if you're fairly sure that the bug has been misreported (i.e., is not a bug), then you can simply say so right in the issue. Even pointing out a minor problem with a proposed solution is fine, assuming the problem is not a showstopper for the entire solution.

On the other hand, if you're raising philosophical issues about the bug's scope or the software's proper behavior, you can be pretty sure other developers will want to be involved. The discussion is likely to diverge for a while before it converges, so do it on the mailing list.

Always link to the mailing list thread from the issue, when you choose to post to the mailing list. It's still important for someone following the issue to be able to reach the discussion, even if the issue itself isn't the forum of discussion. The person who starts the thread may find this laborious, but open source is

fundamentally a writer-responsible culture: it's much more important to make things easy for the tens or hundreds of people who may read the bug than for the three or five people writing about it.

It's fine to take important conclusions or summaries from the list discussion and paste them into the issue, if that will make things convenient for readers. A common idiom is to start a list discussion, put a link to the thread in the issue, and then when the discussion finishes, paste the final summary into the issue (along with a link to the message containing that summary), so someone browsing the issue can easily see what conclusion was reached without having to click to somewhere else. Note that the usual "two masters" data duplication problem does not exist here, because both archives and issue comments are usually static, unchangeable data anyway.

Publicity

In free software, there is a fairly smooth continuum between purely internal discussions and public relations statements. This is partly because the target audience is always ill-defined: given that most or all posts are publicly accessible, the project doesn't have full control over the impression the world gets. Someone—say, a slashdot.org editor—may draw millions of readers' attention to a post that no one ever expected to be seen outside the project. This is a fact of life that all open source projects live with, but in practice, the risk is usually small. In general, the announcements that the project most wants publicized are the ones that will be most publicized, assuming you use the right mechanisms to indicate relative newsworthiness to the outside world.

For major announcements, there tend to be four or five main channels of distribution, on which announcements should be made as nearly simultaneously as possible:

1. Your project's front page is probably seen by more people than any other part of the project. If you have a really major announcement, put a blurb there. The blurb should be a very brief synopsis that links to the press release (see below) for more information.

2. At the same time, you should also have a "News" or "Press Releases" area of the web site, where the announcement can be written up in detail. Part of the purpose of a press release is to provide a single, canonical "announcement object" that other sites can link to, so make sure it is structured accordingly: either as one web page per release, as a discrete blog entry, or as some other kind of entity that can be linked to while still being kept distinct from other press releases in the same area.

3. If your project has an RSS feed, make sure the announcement goes out there too. This may happen automatically when you create the press release, depending on how things are set up at your site. (*RSS* is a mechanism for distributing meta-data-rich news summaries to "subscribers", that is, people who have indicated an interest in receiving those summaries. See http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html for more information about RSS.)

4. If the announcement is about a new release of the software, then update your project's entry on http://freshmeat.net/ (see “Anunciar” about creating the entry in the first place). Every time you update a Freshmeat entry, that entry goes onto the Freshmeat change list for the day. The change list is updated not only on Freshmeat itself, but on various portal sites (including slashdot.org) which are watched eagerly by hordes of people. Freshmeat also offers the same data via RSS feed, so people who are not subscribed to your project's own RSS feed might still see the announcement via Freshmeat's.

5. Send a mail to your project's announcement mailing list. This list's name should actually be "announce", that is, `announce@yourprojectdomain.org`, because that's a fairly standard convention now, and the list's charter should make it clear that it is very low-traffic, reserved for major project announcements. Most of those announcements will be about new releases of the software, but occasionally other events, such as a fundraising drive, the discovery of a security vulnerability (see “Announcing Security Vulnerabilities”) later in this chapter, or a major shift in project direction may be posted there as well. Because it is low traffic and used only for important things, the `announce` list typically has the highest subscribership of any mailing list in the project (of course, this means

you shouldn't abuse it—consider carefully before posting). To avoid random people making announcements, or worse, spam getting through, the `announce` list must always be moderated.

Try to make the announcements in all these places at the same time, as nearly as possible. People might get confused if they see an announcement on the mailing list but then don't see it reflected on the project's home page or in its press releases area. If you get the various changes (emails, web page edits, etc.) queued up and then send them all in a row, you can keep the window of inconsistency very small.

For a less important event, you can eliminate some or all of the above outlets. The event will still be noticed by the outside world in direct proportion to its importance. For example, while a new release of the software is a major event, merely setting the date of the next release, while still somewhat newsworthy, is not nearly as important as the release itself. Setting a date is worth an email to the daily mailing lists (not the announce list), and an update of the project's timeline or status web page, but no more.

However, you might still see that date appearing in discussions elsewhere on the Internet, wherever there are people interested in the project. People who are lurkers on your mailing lists, just listening and never saying anything, are not necessarily silent elsewhere. Word of mouth gives very broad distribution; you should count on it, and construct even minor announcements in such a way as to encourage accurate informal transmission. Specifically, posts that you expect to be quoted should have a clearly meant-to-be-quoted portion, just as though you were writing a formal press release. For example:

> *Just a progress update: we're planning to release version 2.0 of Scanley in mid-August 2005. You can always check http://www.scanley.org/status.html for updates. The major new feature will be regular-expression searches.*
>
> *Other new features include: ... There will also be various bugfixes, including: ...*

The first paragraph is short, gives the two most important pieces of information (the release date and the major new feature), and a URL to visit for further news. If that paragraph is the only thing that crosses someone's screen, you're still doing pretty well. The rest of the mail could be lost without affecting the gist of the content. Of course, sometimes people will link to the entire mail anyway, but just as often, they'll quote only a small part. Given that the latter is a possibility, you might as well make it easy for them, and in the bargain get some influence over what gets quoted.

Announcing Security Vulnerabilities

Handling a security vulnerability is different from handling any other kind of bug report. In free software, doing things openly and transparently is normally almost a religious credo. Every step of the standard bug-handling process is visible to all who care to watch: the arrival of the initial report, the ensuing discussion, and the eventual fix.

Security bugs are different. They can compromise users' data, and possibly users' entire computers. To discuss such a problem openly would be to advertise its existence to the entire world—including to all the parties who might make malicious use of the bug. Even merely committing a fix effectively announces the bug's existence (there are potential attackers who watch the commit logs of public projects, systematically looking for changes that indicate security problems in the pre-change code). Most open source projects have settled on approximately the same set of steps to handle this conflict between openness and secrecy, based on the these basic guidelines:

1. Don't talk about the bug publicly until a fix is available; then supply the fix at exactly the same moment you announce the bug.

2. Come up with that fix as fast as you can—especially if someone outside the project reported the bug, because then you know there's at least one person outside the project who is able to exploit the vulnerability.

In practice, those principles lead to a fairly standardized series of steps, which are described in the sections below.

Receive the report

Obviously, a project needs the ability to receive security bug reports from anyone. But the regular bug reporting address won't do, because it can be watched by anyone too. Therefore, have a separate mailing list for receiving security bug reports. That mailing list must not have publicly readable archives, and its subscribership must be strictly controlled—only long-time, trusted developers can be on the list. If you need a formal definition of "trusted", you can use "anyone who has had commit access for two years or more" or something like that, to avoid favoritism. This is the group that will handle security bugs.

Ideally, the security list should not be spam-protected or moderated, since you don't want an important report to get filtered out or delayed just because no moderators happened to be online that weekend. If you do use automated spam-protection software, try to configure it with high-tolerance settings; it's better to let a few spams through than to miss a report. For the list to be effective, you must advertise its address, of course; but given that it will be unmoderated and, at most, lightly spam-protected, try to never to post its address without some sort of address hiding transformation, as described in “Ocultar las direcciones en los archivos” in Capítulo 3, *Infraestructura Técnica*. Fortunately, address-hiding need not make the address illegible; see http://subversion.tigris.org/security.html, and view that page's HTML source, for an example.

Develop the fix quietly

So what does the security list do when it receives a report? The first task is to evaluate the problem's severity and urgency:

1. How serious is the vulnerability? Does it allow a malicious attacker to take over the computer of someone who uses your software? Or does it, say, merely leak information about the sizes of some of their files?

2. How easy is it to exploit the vulnerability? Can an attack be scripted, or does it require circumstantial knowledge, educated guessing, and luck?

3. *Who* reported the problem to you? The answer to this question doesn't change the nature of the vulnerability, of course, but it does give you an idea of how many other people might know about it. If the report comes from one of the project's own developers, you can breathe a little easier (but only a little), because you can trust them not to have told anyone else about it. On the other hand, if it came in an email from `anonymous14@globalhackerz.net`, then you'd better act as fast as you can. The person did you a favor by informing you of the problem at all, but you have no idea how many other people she's told, or how long she'll wait before exploiting the vulnerability on live installations.

Note that the difference we're talking about here is often just a narrow range between *urgent* and *extremely urgent*. Even when the report comes from a known, friendly source, there could be other people on the Net who discovered the bug long ago and just haven't reported it. The only time things aren't urgent is when the bug inherently does not compromise security very severely.

The "`anonymous14@globalhackerz.net`" example is not facetious, by the way. You really may get bug reports from identity-cloaked people who, by their words and behavior, never quite clarify whether they're on your side or not. It doesn't matter: if they've reported the security hole to you, they'll feel they've done you a good turn, and you should respond in kind. Thank them for the report, give them a date on or before which you plan to release a public fix, and keep them in the loop. Sometimes they may give *you* a date—that is, an implicit threat to publicize the bug on a certain date, whether you're ready or not. This may feel like a bullying power play, but it's more likely a preëmptive action resulting from past disappointment with unresponsive software producers who didn't take security reports seriously enough.

Either way, you can't afford to tick this person off. After all, if the bug is severe, he has knowledge that could cause your users big problems. Treat such reporters well, and hope that they treat you well.

Another frequent reporter of security bugs is the security professional, someone who audits code for a living and keeps up on the latest news of software vulnerabilities. These people usually have experience on both sides of the fence—they've both received and sent reports, probably more than most developers in your project have. They too will usually give a deadline for fixing a vulnerability before going public. The deadline may be somewhat negotiable, but that's up to the reporter; deadlines have become recognized among security professionals as pretty much the only reliable way to get organizations to address security problems promptly. So don't treat the deadline as rude; it's a time-honored tradition, and there are good reasons for it.

Once you know the severity and urgency, you can start working on a fix. There is sometimes a tradeoff between doing a fix elegantly and doing it speedily; this is why you must agree on the urgency before you start. Keep discussion of the fix restricted to the security list members, of course, plus the original reporter (if she wants to be involved) and any developers who need to be brought in for technical reasons.

Do not commit the fix to the repository. Keep it in patch form until the go-public date. If you were to commit it, even with an innocent-looking log message, someone might notice and understand the change. You never know who is watching your repository and why they might be interested. Turning off commit emails wouldn't help; first of all, the gap in the commit mail sequence would itself look suspicious, and anyway, the data would still be in the repository. Just do all development in a patch and keep the patch in some private place, perhaps a separate, private repository known only to the people already aware of the bug. (If you use a decentralized version control system like Arch or SVK, you can do the work under full version control, and just keep that repository inaccessible to outsiders.)

CAN/CVE numbers

You may have seen a *CAN number* or a *CVE number* associated with security problems. These numbers usually look like "CAN-2004-0397" or "CVE-2002-0092", for example.

Both kinds of numbers represent the same type of entity: an entry in the list of "Common Vulnerabilities and Exposures" list maintained at http://cve.mitre.org/. The purpose of the list is to provide standardized names for all known security problems, so that everyone has a unique, canonical name to use when discussing one, and a central place to go to find out more information. The only difference between a "CAN" number and a "CVE" number is that the former represents a candidate entry, not yet approved for inclusion in the official list by the CVE Editorial Board, and the latter represents an approved entry. However, both types of entries are visible to the public, and an entry's number does not change when it is approved—the "CAN" prefix is simply replaced with "CVE".

A CAN/CVE entry does not itself contain a full description of the bug and how to protect against it. Instead, it contains a brief summary, and a list of references to external resources (such as mailing list archives) where people can go to get more detailed information. The real purpose of http://cve.mitre.org/ is to provide a well-organized space in which every vulnerability can have a name and a clear route to more data. See http://cve.mitre.org/cgi-bin/cvename.cgi?name=2002-0092 for an example of an entry. Note that the references can be very terse, with sources appearing as cryptic abbreviations. A key to those abbreviations is at http://cve.mitre.org/cve/refs/refkey.html.

If your vulnerability meets the CVE criteria, you may wish to acquire it a CAN number. The process for doing so is deliberately gated: basically, you have to know someone, or know someone who knows someone. This is not as crazy as it might sound. In order for the CVE Editorial Board to avoid being overwhelmed with spurious or poorly written submissions, they take submissions only from sources they already know and trust. In order to get your vulnerability listed, therefore, you need to find a path of acquaintance from your project to the CVE Editorial Board. Ask around among your developers; one of them will probably know someone else who has either done the CAN process before, or knows someone who has, etc. The advantage of doing it this way is also that somewhere along the chain, someone may know enough to tell you that a) it wouldn't count as a vulnerability or exposure according to MITRE's criteria, so there is no point submitting it, or b) the vulnerability already *has* a CAN or CVE number.

The latter can happen if the bug has already been published on another security advisory list, for example at http://www.cert.org/ or on the BugTraq mailing list at http://www.securityfocus.com/. (If that happened without your project hearing about it, then you should worry what else might be going on that you don't know about.)

If you get a CAN/CVE number at all, you usually want to get it in the early stages of your bug investigation, so that all further communications can refer to that number. CAN entries are embargoed until the go-public date; the entry will exist as an empty placeholder (so you don't lose the name), but it won't reveal any information about the vulnerability until the date on which you will be announcing the bug and the fix.

More information about the CAN/CVE process may be found at http://cve.mitre.org/about/candidates.html, and a particularly clear exposition of one open source project's use of CAN/CVE numbers is at http://www.debian.org/security/cve-compatibility.

Pre-notification

Once your security response team (that is, those developers who are on the security mailing list, or who have been brought in to deal with a particular report) has a fix ready, you need to decide how to distribute it.

If you simply commit the fix to your repository, or otherwise announce it to the world, you effectively force everyone using your software to upgrade immediately or risk being hacked. It is sometimes appropriate, therefore, to do *pre-notification* for certain important users. This is particularly true with client/server software, where there may be well-known servers that are tempting targets for attackers. Those servers' administrators would appreciate having an extra day or two to do the upgrade, so that they are already protected by the time the exploit becomes public knowledge.

Pre-notification simply means sending mails to those administrators before the go-public date, telling them of the vulnerability and how to fix it. You should send pre-notification only to people you trust to be discreet with the information. That is, the qualification for receiving pre-notification is twofold: the recipient must run a large, important server where a compromise would be a serious matter, *and* the recipient must be known to be someone who won't blab about the security problem before the go-public date.

Send each pre-notification mail individually (one at a time) to each recipient. Do *not* send to the entire list of recipients at once, because then they would see each others' names—meaning that you would essentially be alerting each recipient to the fact that each *other* recipient may have a security hole in her server. Sending it to them all via blind CC (BCC) isn't a good solution either, because some admins protect their inboxes with spam filters that either block or reduce the priority of BCC'd mail, since so much spam is sent via BCC these days.

Here's a sample pre-notification mail:

```
From: Your Name Here
To: admin@large-famous-server.com
Reply-to: Your Name Here (not the security list's address)
Subject: Confidential Scanley vulnerability notification.


This email is a confidential pre-notification of a security alert
in the Scanley server.

Please *do not forward* any part of this mail to anyone.  The public
announcement is not until May 19th, and we'd like to keep the
information embargoed until then.

You are receiving this mail because (we think) you run a Scanley
server, and would want to have it patched before this security hole is
```

```
made public on May 19th.

References:
===========

   CAN-2004-1771: Scanley stack overflow in queries

Vulnerability:
==============

   The server can be made to run arbitrary commands if the server's
   locale is misconfigured and the client sends a malformed query.

Severity:
=========

   Very severe, can involve arbitrary code execution on the server.

Workarounds:
============

   Setting the 'natural-language-processing' option to 'off' in
   scanley.conf closes this vulnerability.

Patch:
======

   The patch below applies to Scanley 3.0, 3.1, and 3.2.

   A new public release (Scanley 3.2.1) will be made on or just before
   May 19th, so that it is available at the same time as this
   vulnerability is made public.  You can patch now, or just wait for
   the public release.  The only difference between 3.2 and 3.2.1 will
   be this patch.

[...patch goes here...]
```

If you have a CAN number, include it in the pre-notification (as shown above), even though the information is still embargoed and therefore the MITRE page will show nothing. Including the CAN number allows the recipient to know with certainty that the bug they were pre-notified about is the same one they later hear about through public channels, so they don't have to worry whether further action is necessary or not, which is precisely the point of CAN/CVE numbers.

Distribute the fix publicly

The last step in handling a security bug is to distribute the fix publicly. In a single, comprehensive announcement, you should describe the problem, give the CAN/CVE number if any, describe how to work around it, and how to permanently fix it. Usually "fix" means upgrading to a new version of the software, though sometimes it can mean applying a patch, particularly if the software is normally run in source form anyway. If you do make a new release, it should differ from some existing release by exactly the security patch. That way, conservative admins can upgrade without worrying about what else they might be affecting; they also don't have to worry about future upgrades, because the security fix will be in all future releases as a matter of course. (Details of release procedures are discussed in “Security Releases” in Capítulo 7, *Packaging, Releasing, and Daily Development*.)

Whether or not the public fix involves a new release, do the announcement with roughly the same priority as you would a new release: send a mail to the project's `announce` list, make a new press release, update the Freshmeat entry, etc. While you should never try to play down the existence of a security bug out of concern for the project's reputation, you may certainly set the tone and prominence of a security announcement to match the actual severity of the problem. If the security hole is just a minor informa-

tion exposure, not an exploit that allows the user's entire computer to be taken over, then it may not warrant a lot of fuss. You may even decide not to distract the announce list with it. After all, if the project cries wolf every time, users might end up thinking the software is less secure than it actually is, and also might not believe you when you have a really big problem to announce. See http://cve.mitre.org/about/terminology.html for a good introduction to the problem of judging severity.

In general, if you're unsure how to treat a security problem, find someone with experience and talk to them about it. Assessing and handling vulnerabilities is very much an acquired skill, and it's easy to make missteps the first few times.

Capítulo 7. Packaging, Releasing, and Daily Development

This chapter is about how free software projects package and release their software, and how overall development patterns organize around those goals.

A major difference between open source projects and proprietary ones is the lack of centralized control over the development team. When a new release is being prepared, this difference is especially stark: a corporation can ask its entire development team to focus on an upcoming release, putting aside new feature development and non-critical bug fixing until the release is done. Volunteer groups are not so monolithic. People work on the project for all sorts of reasons, and those not interested in helping with a given release still want to continue regular development work while the release is going on. Because development doesn't stop, open source release processes tend to take longer, but be less disruptive, than commercial release processes. It's a bit like highway repair. There are two ways to fix a road: you can shut it down completely, so that a repair crew can swarm all over it at full capacity until the problem is solved, or you can work on a couple of lanes at a time, while leaving the others open to traffic. The first way is very efficient *for the repair crew*, but not for anyone else—the road is entirely shut down until the job is done. The second way involves much more time and trouble for the repair crew (now they have to work with fewer people and less equipment, in cramped conditions, with flaggers to slow and direct traffic, etc.), but at least the road remains useable, albeit not at full capacity.

Open source projects tend to work the second way. In fact, for a mature piece of software with several different release lines being maintained simultaneously, the project is sort of in a permanent state of minor road repair. There are always a couple of lanes closed; a consistent but low level of background inconvenience is always being tolerated by the development group as a whole, so that releases get made on a regular schedule.

The model that makes this possible generalizes to more than just releases. It's the principle of parallelizing tasks that are not mutually interdependent—a principle that is by no means unique to open source development, of course, but one which open source projects implement in their own particular way. They cannot afford to annoy either the roadwork crew or the regular traffic too much, but they also cannot afford to have people dedicated to standing by the orange cones and flagging traffic along. Thus they gravitate toward processes that have flat, constant levels of administrative overhead, rather than peaks and valleys. Volunteers are generally willing to work with small but consistent amounts of inconvenience; the predictability allows them to come and go without worrying about whether their schedule will clash with what's happening in the project. But if the project were subject to a master schedule in which some activities excluded other activities, the result would be a lot of developers sitting idle a lot of the time—which would be not only inefficient but boring, and therefore dangerous, in that a bored developer is likely to soon be an ex-developer.

Release work is usually the most noticeable non-development task that happens in parallel with development, so the methods described in the following sections are geared mostly toward enabling releases. However, note that they also apply to other parallelizable tasks, such as translations and internationalization, broad API changes made gradually across the entire code base, etc.

Release Numbering

Before we talk about how to make a release, let's look at how to name releases, which requires knowing what releases actually mean to users. A release means that:

- Old bugs have been fixed. This is probably the one thing users can count on being true of every release.

- New bugs have been added. This too can usually be counted on, except sometimes in the case of security releases or other one-offs (see “Security Releases” later in this chapter).
- New features may have been added.
- New configuration options may have been added, or the meanings of old options may have changed subtly. The installation procedures may have changed slightly since the last release too, though one always hopes not.
- Incompatible changes may have been introduced, such that the data formats used by older versions of the software are no longer useable without undergoing some sort of (possibly manual) one-way conversion step.

As you can see, not all of these are good things. This is why experienced users approach new releases with some trepidation, especially when the software is mature and was already mostly doing what they wanted (or thought they wanted). Even the arrival of new features is a mixed blessing, in that it may mean the software will now behave in unexpected ways.

The purpose of release numbering, therefore, is twofold: obviously the numbers should unambiguously communicate the ordering of releases (i.e., by looking at any two releases' numbers, one can know which came later), but also they should indicate as compactly as possible the degree and nature of the changes in the release.

All that in a number? Well, more or less, yes. Release numbering strategies are one of the oldest bikeshed discussions around (see “Cuanto más blando sea el tema, más largo será el debate” in Capítulo 6, *Communications*), and the world is unlikely to settle on a single, complete standard anytime soon. However, a few good strategies have emerged, along with one universally agreed-on principle: *be consistent*. Pick a numbering scheme, document it, and stick with it. Your users will thank you.

Release Number Components

This section describes the formal conventions of release numbering in detail, and assumes very little prior knowledge. It is intended mainly as a reference. If you're already familiar with these conventions, you can skip this section.

Release numbers are groups of digits separated by dots:

Scanley 2.3
Singer 5.11.4

...and so on. The dots are *not* decimal points, they are merely separators; "5.3.9" would be followed by "5.3.10". A few projects have occasionally hinted otherwise, most famously the Linux kernel with its "0.95", "0.96"... "0.99" sequence leading up to Linux 1.0, but the convention that the dots are not decimals is now firmly established and should be considered a standard. There is no limit to the number of components (digit portions containing no dots), but most projects do not go beyond three or four. The reasons why will become clear later.

In addition to the numeric components, projects sometimes tack on a descriptive label such as "Alpha" or "Beta" (see Alfa y Beta), for example:

Scanley 2.3.0 (Alpha)
Singer 5.11.4 (Beta)

An Alpha or Beta qualifier means that this release *precedes* a future release that will have the same number without the qualifier. Thus, "2.3.0 (Alpha)" leads eventually to "2.3.0". In order to allow several such candidate releases in a row, the qualifiers themselves can have meta-qualifiers. For example, here is a series of releases in the order that they would be made available to the public:

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Notice that when it has the "Alpha" qualifier, Scanley "2.3" is written as "2.3.0". The two numbers are equivalent—trailing all-zero components can always be dropped for brevity—but when a qualifier is present, brevity is out the window anyway, so one might as well go for completeness instead.

Other qualifiers in semi-regular use include "Stable", "Unstable", "Development", and "RC" (for "Release Candidate"). The most widely used ones are still "Alpha" and "Beta", with "RC" running a close third place, but note that "RC" always includes a numeric meta-qualifier. That is, you don't release "Scanley 2.3.0 (RC)", you release "Scanley 2.3.0 (RC 1)", followed by RC2, etc.

Those three labels, "Alpha", "Beta", and "RC", are pretty widely known now, and I don't recommend using any of the others, even though the others might at first glance seem like better choices because they are normal words, not jargon. But people who install software from releases are already familiar with the big three, and there's no reason to do things gratuitously differently from the way everyone else does them.

Although the dots in release numbers are not decimal points, they do indicate place-value significance. All "0.X.Y" releases precede "1.0" (which is equivalent to "1.0.0", of course). "3.14.158" immediately precedes "3.14.159", and non-immediately precedes "3.14.160" as well as "3.15.anything", and so.

A consistent release numbering policy enables a user to look at two release numbers for the same piece of software and tell, just from the numbers, the important differences between those two releases. In a typical three-component system, the first component is the *major number*, the second is the *minor number*, and the third is the *micro number*. For example, release "2.10.17" is the seventeenth micro release in the tenth minor release line within the second major release series. The words "line" and "series" are used informally here, but they mean what one would expect. A major series is simply all the releases that share the same major number, and a minor series (or minor line) consists of all the releases that share the same minor *and* major number. That is, "2.4.0" and "3.4.1" are not in the same minor series, even though they both have "4" for their minor number; on the other hand, "2.4.0" and "2.4.2" are in the same minor line, though they are not adjacent if "2.4.1" was released between them.

The meanings of these numbers are exactly what you'd expect: an increment of the major number indicates that major changes happened; an increment of the minor number indicates minor changes; and an increment of the micro number indicates really trivial changes. Some projects add a fourth component, usually called the *patch number*, for especially fine-grained control over the differences between their releases (confusingly, other projects use "patch" as a synonym for "micro" in a three-component system). There are also projects that use the last component as a *build number*, incremented every time the software is built and representing no change other than that build. This helps the project link every bug report with a specific build, and is probably most useful when binary packages are the default method of distribution.

Although there are many different conventions for how many components to use, and what the components mean, the differences tend to be minor—you get a little leeway, but not a lot. The next two sections discuss some of the most widely used conventions.

The Simple Strategy

Most projects have rules about what kinds of changes are allowed into a release if one is only incrementing the micro number, different rules for the minor number, and still different ones for the major number. There is no set standard for these rules yet, but here I will describe a policy that has been used successfully by multiple projects. You may want to just adopt this policy in your own project, but even if you don't, it's still a good example of the kind of information release numbers should convey. This policy is adapted from the numbering system used by the APR project, see http://apr.apache.org/versioning.html.

1. Changes to the micro number only (that is, changes within the same minor line) must be both forward- and backward-compatible. That is, the changes should be bug fixes only, or very small enhancements to existing features. New features should not be introduced in a micro release.

2. Changes to the minor number (that is, within the same major line) must be backward-compatible, but not necessarily forward-compatible. It's normal to introduce new features in a minor release, but usually not too many new features at once.

3. Changes to the major number mark compatibility boundaries. A new major release can be forward- and backward-incompatible. A major release is expected to have new features, and may even have entire new feature sets.

What *backward-compatible* and *forward-compatible* mean, exactly, depends on what your software does, but in context they are usually not open to much interpretation. For example, if your project is a client/server application, then "backward-compatible" means that upgrading the server to 2.6.0 should not cause any existing 2.5.4 clients to lose functionality or behave differently than they did before (except for bugs that were fixed, of course). On the other hand, upgrading one of those clients to 2.6.0, along with the server, might make *new* functionality available for that client, functionality that 2.5.4 clients don't know how to take advantage of. If that happens, then the upgrade is *not* "forward-compatible": clearly you can't now downgrade that client back to 2.5.4 and keep all the functionality it had at 2.6.0, since some of that functionality was new in 2.6.0.

This is why micro releases are essentially for bug fixes only. They must remain compatible in both directions: if you upgrade from 2.5.3 to 2.5.4, then change your mind and downgrade back to 2.5.3, no functionality should be lost. Of course, the bugs fixed in 2.5.4 would reappear after the downgrade, but you wouldn't lose any features, except insofar as the restored bugs prevent the use of some existing features.

Client/server protocols are just one of many possible compatibility domains. Another is data formats: does the software write data to permanent storage? If so, the formats it reads and writes need to follow the compatibility guidelines promised by the release number policy. Version 2.6.0 needs to be able to read the files written by 2.5.4, but may silently upgrade the format to something that 2.5.4 cannot read, because the ability to downgrade is not required across a minor number boundary. If your project distributes code libraries for other programs to use, then APIs are a compatibility domain too: you must make sure that source and binary compatibility rules are spelled out in such a way that the informed user need never wonder whether or not it's safe to upgrade in place. She will be able to look at the numbers and know instantly.

In this system, you don't get a chance for a fresh start until you increment the major number. This can often be a real inconvenience: there may be features you wish to add, or protocols that you wish to redesign, that simply cannot be done while maintaining compatibility. There's no magic solution to this, except to try to design things in an extensible way in the first place (a topic easily worth its own book, and certainly outside the scope of this one). But publishing a release compatibility policy, and adhering to it, is an inescapable part of distributing software. One nasty surprise can alienate a lot of users. The policy just described is good partly because it's already quite widespread, but also because it's easy to explain and to remember, even for those not already familiar with it.

It is generally understood that these rules do not apply to pre-1.0 releases (although your release policy should probably state so explicitly, just to be clear). A project that is still in initial development can release 0.1, 0.2, 0.3, and so on in sequence, until it's ready for 1.0, and the differences between those releases can be arbitrarily large. Micro numbers in pre-1.0 releases are optional. Depending on the nature of your project and the differences between the releases, you might find it useful to have 0.1.0, 0.1.1, etc., or you might not. Conventions for pre-1.0 release numbers are fairly loose, mainly because people understand that strong compatibility constraints would hamper early development too much, and because early adopters tend to be forgiving anyway.

Remember that all these injunctions only apply to this particular three-component system. Your project could easily come up with a different three-component system, or even decide it doesn't need such fine granularity and use a two-component system instead. The important thing is to decide early, publish exactly what the components mean, and stick to it.

The Even/Odd Strategy

Some projects use the parity of the minor number component to indicate the stability of the software: even means stable, odd means unstable. This applies only to the minor number, not the major and micro numbers. Increments in the micro number still indicate bug fixes (no new features), and increments in the major number still indicate big changes, new feature sets, etc.

The advantage of the even/odd system, which has been used by the Linux kernel project among others, is that it offers a way to release new functionality for testing without subjecting production users to potentially unstable code. People can see from the numbers that "2.4.21" is okay to install on their live web server, but that "2.5.1" should probably stay confined to home workstation experiments. The development team handles the bug reports that come in from the unstable (odd-minor-numbered) series, and when things start to settle down after some number of micro releases in that series, they increment the minor number (thus making it even), reset the micro number back to "0", and release a presumably stable package.

This system preserves, or at least, does not conflict with, the compatibility guidelines given earlier. It simply overloads the minor number with some extra information. This forces the minor number to be incremented about twice as often as would otherwise be necessary, but there's no great harm in that. The even/odd system is probably best for projects that have very long release cycles, and which by their nature have a high proportion of conservative users who value stability above new features. It is not the only way to get new functionality tested in the wild, however. “Stabilizing a Release” later in this chapter describes another, perhaps more common, method of releasing potentially unstable code to the public, marked so that people have an idea of the risk/benefit trade-offs immediately on seeing the release's name.

Release Branches

From a developer's point of view, a free software project is in a state of continuous release. Developers usually run the latest available code at all times, because they want to spot bugs, and because they follow the project closely enough to be able to stay away from currently unstable areas of the feature space. They often update their copy of the software every day, sometimes more than once a day, and when they check in a change, they can reasonably expect that every other developer will have it within 24 hours.

How, then, should the project make a formal release? Should it simply take a snapshot of the tree at a moment in time, package it up, and hand it to the world as, say, version "3.5.0"? Common sense says no. First, there may be no moment in time when the entire development tree is clean and ready for release. Newly-started features could be lying around in various states of completion. Someone might have checked in a major change to fix a bug, but the change could be controversial and under debate at the moment the snapshot is taken. If so, it wouldn't work to simply delay the snapshot until the debate ends, because another, unrelated debate could start in the meantime, and then you'd have wait for *that* one to end

too. This process is not guaranteed to halt.

In any case, using full-tree snapshots for releases would interfere with ongoing development work, even if the tree could be put into a releasable state. Say this snapshot is going to be "3.5.0"; presumably, the next snapshot would be "3.5.1", and would contain mostly fixes for bugs found in the 3.5.0 release. But if both are snapshots from the same tree, what are the developers supposed to do in the time between the two releases? They can't be adding new features; the compatibility guidelines prevent that. But not everyone will be enthusiastic about fixing bugs in the 3.5.0 code. Some people may have new features they're trying to complete, and will become irate if they are forced to choose between sitting idle and working on things they're not interested in, just because the project's release processes demand that the development tree remain unnaturally quiescent.

The solution to these problems is to always use a *release branch*. A release branch is just a branch in the version control system (see *rama (branch)*), on which the code destined for this release can be isolated from mainline development. The concept of release branches is certainly not original to free software; many commercial development organizations use them too. However, in commercial environments, release branches are sometimes considered a luxury—a kind of formal "best practice" that can, in the heat of a major deadline, be dispensed with while everyone on the team scrambles to stabilize the main tree.

Release branches are pretty much required in open source projects, however. I have seen projects do releases without them, but it has always resulted in some developers sitting idle while others—usually a minority—work on getting the release out the door. The result is usually bad in several ways. First, overall development momentum is slowed. Second, the release is of poorer quality than it needed to be, because there were only a few people working on it, and they were hurrying to finish so everyone else could get back to work. Third, it divides the development team psychologically, by setting up a situation in which different types of work interfere with each other unnecessarily. The developers sitting idle would probably be happy to contribute *some* of their attention to a release branch, as long as that were a choice they could make according to their own schedules and interests. But without the branch, their choice becomes "Do I participate in the project today or not?" instead of "Do I work on the release today, or work on that new feature I've been developing in the mainline code?"

Mechanics of Release Branches

The exact mechanics of creating a release branch depend on your version control system, of course, but the general concepts are the same in most systems. A branch usually sprouts from another branch or from the trunk. Traditionally, the trunk is where mainline development goes on, unfettered by release constraints. The first release branch, the one leading to the "1.0" release, sprouts off the trunk. In CVS, the branch command would be something like this

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_0_X
```

or in Subversion, like this:

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.0.x
```

(All these examples assume a three-component release numbering system. While I can't show the exact commands for every version control system, I'll give examples in CVS and Subversion and hope that the corresponding commands in other systems can be deduced from those two.)

Notice that we created branch "1.0.x" (with a literal "x") instead of "1.0.0". This is because the same minor line—i.e., the same branch—will be used for all the micro releases in that line. The actual process of stabilizing the branch for release is covered in “Stabilizing a Release” later in this chapter. Here we are concerned just with the interaction between the version control system and the release process. When the release branch is stabilized and ready, it is time to tag a snapshot from the branch:

```
$ cd RELEASE_1_0_X-working-copy
$ cvs tag RELEASE_1_0_0
```

or

```
$ svn copy http://.../repos/branches/1.0.x http://.../repos/tags/1.0.0
```

That tag now represents the exact state of the project's source tree in the 1.0.0 release (this is useful in case anyone ever needs to get an old version after the packaged distributions and binaries have been taken down). The next micro release in the same line is likewise prepared on the 1.0.x branch, and when it is ready, a tag is made for 1.0.1. Lather, rinse, repeat for 1.0.2, and so on. When it's time to start thinking about a 1.1.x release, make a new branch from trunk:

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_1_X
```

or

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.1.x
```

Maintenance can continue in parallel along both 1.0.x and 1.1.x, and releases can be made independently from both lines. In fact, it is not unusual to publish near-simultaneous releases from two different lines. The older series is recommended for more conservative site administrators, who may not want to make the big jump to (say) 1.1 without careful preparation. Meanwhile, more adventurous people usually take the most recent release on the highest line, to make sure they're getting the latest features, even at the risk of greater instability.

This is not the only release branch strategy, of course. In some circumstances it may not even be the best, though it's worked out pretty well for projects I've been involved in. Use any strategy that seems to work, but remember the main points: the purpose of a release branch is to isolate release work from the fluctuations of daily development, and to give the project a physical entity around which to organize its release process. That process is described in detail in the next section.

Stabilizing a Release

Stabilization is the process of getting a release branch into a releasable state; that is, of deciding which changes will be in the release, which will not, and shaping the branch content accordingly.

There's a lot of potential grief contained in that word, "deciding". The last-minute feature rush is a familiar phenomenon in collaborative software projects: as soon as developers see that a release is about to happen, they scramble to finish their current changes, in order not to miss the boat. This, of course, is the exact opposite of what you want at release time. It would be much better for people to work on features at a comfortable pace, and not worry too much about whether their changes make it into this release or the next one. The more changes one tries to cram into a release at the last minute, the more the code is destabilized, and (usually) the more new bugs are created.

Most software engineers agree in theory on rough criteria for what changes should be allowed into a release line during its stabilization period. Obviously, fixes for severe bugs can go in, especially for bugs without workarounds. Documentation updates are fine, as are fixes to error messages (except when they are considered part of the interface and must remain stable). Many projects also allow certain kinds of low-risk or non-core changes to go in during stabilization, and may have formal guidelines for measu-

ring risk. But no amount of formalization can obviate the need for human judgement. There will always be cases where the project simply has to make a decision about whether a given change can go into a release. The danger is that since each person wants to see their own favorite changes admitted into the release, then there will be plenty of people motivated to allow changes, and not enough people motivated to bar them.

Thus, the process of stabilizing a release is mostly about creating mechanisms for saying "no". The trick for open source projects, in particular, is to come up with ways of saying "no" that won't result in too many hurt feelings or disappointed developers, and also won't prevent deserving changes from getting into the release. There are many different ways to do this. It's pretty easy to design systems that satisfy these criteria, once the team has focused on them as the important criteria. Here I'll briefly describe two of the most popular systems, at the extreme ends of the spectrum, but don't let that discourage your project from being creative. Plenty of other arrangements are possible; these are just two that I've seen work in practice.

Dictatorship by Release Owner

The group agrees to let one person be the *release owner*. This person has final say over what changes make it into the release. Of course, it is normal and expected for there to be discussions and arguments, but in the end the group must grant the release owner sufficient authority to make final decisions. For this system to work, it is necessary to choose someone with the technical competence to understand all the changes, and the social standing and people skills to navigate the discussions leading up to the release without causing too many hurt feelings.

A common pattern is for the release owner to say "I don't think there's anything wrong with this change, but we haven't had enough time to test it yet, so it shouldn't go into this release." It helps a lot if the release owner has broad technical knowledge of the project, and can give reasons why the change could be potentially destabilizing (for example, its interactions with other parts of the software, or portability concerns). People will sometimes ask such decisions to be justified, or will argue that a change is not as risky as it looks. These conversations need not be confrontational, as long as the release owner is able to consider all the arguments objectively and not reflexively dig in his heels.

Note that the release owner need not be the same person as the project leader (in cases where there is a project leader at all; see “Dictadores Benevolentes” in Capítulo 4, *Infraestructura Social y Política*). In fact, sometimes it's good to make sure they're *not* the same person. The skills that make a good development leader are not necessarily the same as those that make a good release owner. In something as important as the release process, it may be wise to have someone provide a counterbalance to the project leader's judgement.

Contrast the release owner role with the less dictatorial role described in “Release manager” later in this chapter.

Change Voting

At the opposite extreme from dictatorship by release owner, developers can simply vote on which changes to include in the release. However, since the most important function of release stabilization is to *exclude* changes, it's important to design the voting system in such a way that getting a change into the release involves positive action by multiple developers. Including a change should need more than just a simple majority (see “¿Quién Vota?” in Capítulo 4, *Infraestructura Social y Política*). Otherwise, one vote for and none against a given change would suffice to get it into the release, and an unfortunate dynamic would be set up whereby each developer would vote for her own changes, yet would be reluctant to vote against others' changes, for fear of possible retaliation. To avoid this, the system should be arranged such that subgroups of developers must act in cooperation to get any change into the release. This not only means that more people review each change, it also makes any individual developer less hesitant to vote against a change, because she knows that no particular one among those who voted for it would take her vote against as a personal affront. The greater the number of people involved, the more the discussion becomes about the change and less about the individuals.

The system we use in the Subversion project seems to have struck a good balance, so I'll recommend it here. In order for a change to be applied to the release branch, at least three developers must vote in favor of it, and none against. A single "no" vote is enough to stop the change from being included; that is, a "no" vote in a release context is equivalent to a veto (see “Vetos”). Naturally, any such vote must be accompanied by a justification, and in theory the veto could be overridden if enough people feel it is unreasonable and force a special vote over it. In practice, this has never happened, and I don't expect that it ever will. People are conservative around releases anyway, and when someone feels strongly enough to veto the inclusion of a change, there's usually a good reason for it.

Because the release procedure is deliberately biased toward conservatism, the justifications offered for vetoes are sometimes procedural rather than technical. For example, a person may feel that a change is well-written and unlikely to cause any new bugs, but vote against its inclusion in a micro release simply because it's too big—perhaps it adds a new feature, or in some subtle way fails to fully follow the compatibility guidelines. I've occasionally even seen developers veto something because they simply had a gut feeling that the change needed more testing, even though they couldn't spot any bugs in it by inspection. People grumbled a little bit, but the vetoes stood and the change was not included in the release (I don't remember if any bugs were found in later testing or not, though).

Managing collaborative release stabilization

If your project chooses a change voting system, it is imperative that the physical mechanics of setting up ballots and casting votes be as convenient as possible. Although there is plenty of open source electronic voting software available, in practice the easiest thing to do is just to set up a text file in the release branch, called `STATUS` or `VOTES` or something like that. This file lists each proposed change—any developer can propose a change for inclusion—along with all the votes for and against it, plus any notes or comments. (Proposing a change doesn't necessarily mean voting for it, by the way, although the two often go together.) An entry in such a file might look like this:

```
* r2401 (issue #49)
  Prevent client/server handshake from happening twice.
  Justification:
    Avoids extra network turnaround; small change and easy to review.
  Notes:
    This was discussed in http://.../mailing-lists/message-7777.html
    and other messages in that thread.
  Votes:
    +1: jsmith, kimf
    -1: tmartin (breaks compatibility with some pre-1.0 servers;
                 admittedly, those servers are buggy, but why be
                 incompatible if we don't have to?)
```

In this case, the change acquired two positive votes, but was vetoed by tmartin, who gave the reason for the veto in a parenthetical note. The exact format of the entry doesn't matter; whatever your project settles on is fine—perhaps tmartin's explanation for the veto should go up in the "Notes:" section, or perhaps the change description should get a "Description:" header to match the other sections. The important thing is that all the information needed to evaluate the change be reachable, and that the mechanism for casting votes be as lightweight as possible. The proposed change is referred to by its revision number in the repository (in this case a single revision, r2401, although a proposed change could just as easily consist of multiple revisions). The revision is assumed to refer to a change made on the trunk; if the change were already on the release branch, there would be no need to vote on it. If your version control system doesn't have an obvious syntax for referring to individual changes, then the project should make one up. For voting to be practical, each change under consideration must be unambiguously identifiable.

Those proposing or voting for a change are responsible for making sure it applies cleanly to the release branch, that is, applies without conflicts (see *conflicto*). If there are conflicts, then the entry should either point to an adjusted patch that does apply cleanly, or to a temporary branch that holds an adjusted version of the change, for example:

```
* r13222, r13223, r13232
  Rewrite libsvn_fs_fs's auto-merge algorithm
  Justification:
    unacceptable performance (>50 minutes for a small commit) in
    a repository with 300,000 revisions
  Branch:
    1.1.x-r13222@13517
  Votes:
    +1: epg, ghudson
```

That example is taken from real life; it comes from the STATUS file for the Subversion 1.1.4 release process. Notice how it uses the original revisions as canonical handles on the change, even though there is also a branch with a conflict-adjusted version of the change (the branch also combines the three trunk revisions into one, r13517, to make it easier to merge the change into the release, should it get approval). The original revisions are provided because they're still the easiest entity to review, since they have the original log messages. The temporary branch wouldn't have those log messages; in order to avoid duplication of information (see “Singularidad de la información” in Capítulo 3, *Infraestructura Técnica*), the branch's log message for r13517 should simply say "Adjust r13222, r13223, and r13232 for backport to 1.1.x branch." All other information about the changes can be chased down at their original revisions.

Release manager

The actual process of merging (see *merge*) approved changes into the release branch can be performed by any developer. There does not need to be one person whose job it is to merge changes; if there are a lot of changes, it can be better to spread the burden around.

However, although both voting and merging happen in a decentralized fashion, in practice there are usually one or two people driving the release process. This role is sometimes formally blessed as *release manager*, but it is quite different from a release owner (see “Dictatorship by Release Owner” earlier in this chapter) who has final say over the changes. Release managers keep track of how many changes are currently under consideration, how many have been approved, how many seem likely to be approved, etc. If they sense that important changes are not getting enough attention, and might be left out of the release for lack of votes, they will gently nag other developers to review and vote. When a batch of changes are approved, these people will often take it upon themselves to merge them into the release branch; it's fine if others leave that task to them, as long as everyone understands that they are not obligated to do all the work unless they have explicitly committed to it. When the time comes to put the release out the door (see “Testing and Releasing” later in this chapter), the release managers also take care of the logistics of creating the final release packages, collecting digital signatures, uploading the packages, and making the public announcement.

Packaging

The canonical form for distribution of free software is as source code. This is true regardless of whether the software normally runs in source form (i.e., can be interpreted, like Perl, Python, PHP, etc.) or needs to be compiled first (like C, C++, Java, etc.). With compiled software, most users will probably not compile the sources themselves, but will instead install from pre-built binary packages (see “Binary Packages” later in this chapter). However, those binary packages are still derived from a master source distribution. The point of the source package is to unambiguously define the release. When the project distributes "Scanley 2.5.0", what it means, specifically, is "The tree of source code files that, when compiled (if necessary) and installed, produces Scanley 2.5.0."

There is a fairly strict standard for how source releases should look. One will occasionally see deviations from this standard, but they are the exception, not the rule. Unless there is a compelling reason to do otherwise, your project should follow this standard too.

Format

The source code should be shipped in the standard formats for transporting directory trees. For Unix and Unix-like operating systems, the convention is to use TAR format, compressed by **compress**, **gzip**, **bzip** or **bzip2**. For MS Windows, the standard method for distributing directory trees is *zip* format, which happens to do compression as well, so there is no need to compress the archive after creating it.

> **TAR Files**
>
> *TAR* stands for "Tape ARchive", because tar format represents a directory tree as a linear data stream, which makes it ideal for saving directory trees to tape. The same property also makes it the standard for distributing directory trees as a single file. Producing compressed tar files (or *tarballs*) is pretty easy. On some systems, the **tar** command can produce a compressed archive itself; on others, a separate compression program is used.

Name and Layout

The name of the package should consist of the software's name plus the release number, plus the format suffixes appropriate for the archive type. For example, Scanley 2.5.0, packaged for Unix using GNU Zip (gzip) compression, would look like this:

scanley-2.5.0.tar.gz

or for Windows using zip compression:

scanley-2.5.0.zip

Either of these archives, when unpacked, should create a single new directory tree named `scanley-2.5.0` in the current directory. Underneath the new directory, the source code should be arranged in a layout ready for compilation (if compilation is needed) and installation. In the top level of new directory tree, there should be a plain text `README` file explaining what the software does and what release this is, and giving pointers to other resources, such as the project's web site, other files of interest, etc. Among those other files should be an `INSTALL` file, sibling to the `README` file, giving instructions on how to build and install the software for all the operating systems it supports. As mentioned in “Cómo aplicar una licencia a nuestro software” in Capítulo 2, *Primeros Pasos*, there should also be a `COPYING` or `LICENSE` file, giving the software's terms of distribution.

There should also be a `CHANGES` file (sometimes called `NEWS`), explaining what's new in this release. The `CHANGES` file accumulates changelists for all releases, in reverse chronological order, so that the list for this release appears at the top of the file. Completing that list is usually the last thing done on a stabilizing release branch; some projects write the list piecemeal as they're developing, others prefer to save it all up for the end and have one person write it, getting information by combing the version control logs. The list looks something like this:

```
Version 2.5.0
(20 December 2004, from /branches/2.5.x)
http://svn.scanley.org/repos/svn/tags/2.5.0/

 New features, enhancements:
    * Added regular expression queries (issue #53)
```

```
    * Added support for UTF-8 and UTF-16 documents
    * Documentation translated into Polish, Russian, Malagasy
    * ...

Bugfixes:
    * fixed reindexing bug (issue #945)
    * fixed some query bugs (issues #815, #1007, #1008)
    * ...
```

The list can be as long as necessary, but don't bother to include every little bugfix and feature enhancement. Its purpose is simply to give users an overview of what they would gain by upgrading to the new release. In fact, the changelist is customarily included in the announcement email (see “Testing and Releasing” later in this chapter), so write it with that audience in mind.

CHANGES Versus ChangeLog

Traditionally, a file named *ChangeLog* lists every change ever made to a project—that is, every revision committed to the version control system. There are various formats for ChangeLog files; the details of the formats aren't important here, as they all contain the same information: the date of the change, its author, and a brief summary (or just the log message for that change).

A `CHANGES` file is different. It too is a list of changes, but only the ones thought important for a certain audience to see, and often with metadata like the exact date and author stripped off. To avoid confusion, don't use the terms interchangeably. Some projects use "NEWS" instead of "CHANGES"; although this avoids the potential for confusion with "ChangeLog", it is a bit of a misnomer, since the CHANGES file retains change information for all releases, and thus has a lot of old news in addition to the new news at the top.

ChangeLog files may be slowly disappearing anyway. They were helpful in the days when CVS was the only choice of version control system, because change data was not easy to extract from CVS. However, with more recent version control systems, the data that used to be kept in the ChangeLog can be requested from the version control repository at any time, making it pointless for the project to keep a static file containing that data—in fact, worse than pointless, since the ChangeLog would merely duplicate the log messages already stored in the repository.

The actual layout of the source code inside the tree should be the same as, or as similar as possible to, the source code layout one would get by checking out the project directly from its version control repository. Usually there are a few differences, for example because the package contains some generated files needed for configuration and compilation (see “Compilation and Installation” later in this chapter), or because it includes third-party software that is not maintained by the project, but that is required and that users are not likely to already have. But even if the distributed tree corresponds exactly to some development tree in the version control repository, the distribution itself should not be a working copy (see *copia funcional*). The release is supposed to represent a static reference point—a particular, unchangeable configuration of source files. If it were a working copy, the danger would be that the user might update it, and afterward think that he still has the release when in fact he has something different.

Remember that the package is the same regardless of the packaging. The release—that is, the precise entity referred to when someone says "Scanley 2.5.0"—is the tree created by unpacking a zip file or tarball. So the project might offer all of these for download:

scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip

...but the source tree created by unpacking them must be the same. That source tree is the distribution; the form in which it is downloaded is merely a matter of convenience. Certain trivial differences between source packages are allowable: for example, in the Windows package, text files should have lines ending with CRLF (Carriage Return and Line Feed), while Unix packages should use just LF. The trees may be arranged slightly differently between source packages destined for different operating systems, too, if those operating systems require different sorts of layouts for compilation. However, these are all basically trivial transformations. The basic source files should be the same across all the packagings of a given release.

To capitalize or not to capitalize

When referring to a project by name, people generally capitalize it as a proper noun, and capitalize acronyms if there are any: "MySQL 5.0", "Scanley 2.5.0", etc. Whether this capitalization is reproduced in the package name is up to the project. Either `Scanley-2.5.0.tar.gz` or `scanley-2.5.0.tar.gz` would be fine, for example (I personally prefer the latter, because I don't like to make people hit the shift key, but plenty of projects ship capitalized packages). The important thing is that the directory created by unpacking the tarball use the same capitalization. There should be no surprises: the user must be able to predict with perfect accuracy the name of the directory that will be created when she unpacks a distribution.

Pre-releases

When shipping a pre-release or candidate release, the qualifier is truly a part of the release number, so include it in the name of the package's name. For example, the ordered sequence of alpha and beta releases given earlier in “Release Number Components” would result in package names like this:

scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz

The first would unpack into a directory named `scanley-2.3.0-alpha1`, the second into `scanley-2.3.0-alpha2`, and so on.

Compilation and Installation

For software requiring compilation or installation from source, there are usually standard procedures that experienced users expect to be able to follow. For example, for programs written in C, C++, or certain other compiled languages, the standard under Unix-like systems is for the user to type:

```
$ ./configure
$ make
# make install
```

The first command autodetects as much about the environment as it can and prepares for the build process, the second command builds the software in place (but does not install it), and the last command installs it on the system. The first two commands are done as a regular user, the third as root. For more details about setting up this system, see the excellent *GNU Autoconf, Automake, and Libtool* book by Vaughan, Elliston, Tromey, and Taylor. It is published as treeware by New Riders, and its content is also freely available online at http://sources.redhat.com/autobook/.

This is not the only standard, though it is one of the most widespread. The Ant (http://ant.apache.org/) build system is gaining popularity, especially with projects written in Java, and it has its own standard procedures for building and installing. Also, certain programming languages, such as Perl and Python, recommend that the same method be used for most programs written in that language (for example, Perl modules use the command **perl Makefile.pl**). If it's not obvious to you what the applicable standards are for your project, ask an experienced developer; you can safely assume that *some* standard applies, even if you don't know what it is at first.

Whatever the appropriate standards for you project are, don't deviate from them unless you absolutely must. Standard installation procedures are practically spinal reflexes for a lot of system administrators now. If they see familiar invocations documented in your project's `INSTALL` file, that instantly raises their faith that your project is generally aware of conventions, and that it is likely to have gotten other things right as well. Also, as discussed in “Descargas” in Capítulo 2, *Primeros Pasos*, having a standard build procedure pleases potential developers.

On Windows, the standards for building and installing are a bit less settled. For projects requiring compilation, the general convention seems to be to ship a tree that can fit into the workspace/project model of the standard Microsoft development environments (Developer Studio, Visual Studio, VS.NET, MSVC++, etc.). Depending on the nature of your software, it may be possible to offer a Unix-like build option on Windows via the Cygwin (http://www.cygwin.com/) environment. And of course, if you're using a language or programming framework that comes with its own build and install conventions—e.g., Perl or Python—you should simply use whatever the standard method is for that framework, whether on Windows, Unix, Mac OS X, or any other operating system.

Be willing to put in a lot of extra effort in order to make your project conform to the relevant build or installation standards. Building and installing is an entry point: it's okay for things to get harder after that, if they absolutely must, but it would be a shame for the user's or developer's very first interaction with the software to require unexpected steps.

Binary Packages

Although the formal release is a source code package, most users will install from binary packages, either provided by their operating system's software distribution mechanism, or obtained manually from the project web site or from some third party. Here "binary" doesn't necessarily mean "compiled"; it just means any pre-configured form of the package that allows a user to install it on his computer without going through the usual source-based build and install procedures. On RedHat GNU/Linux, it is the RPM system; on Debian GNU/Linux, it is the APT (`.deb`) system; on MS Windows, it's usually `.MSI` files or self-installing `.exe` files.

Whether these binary packages are assembled by people closely associated with the project, or by distant third parties, users are going to *treat* them as equivalent to the project's official releases, and will file issues in the project's bug tracker based on the behavior of the binary packages. Therefore, it is in the project's interest to provide packagers with clear guidelines, and work closely with them to see to it that what they produce represents the software fairly and accurately.

The main thing packagers need to know is that they should always base their binary packages on an official source release. Sometimes packagers are tempted to pull a later incarnation of the code from the repository, or include selected changes that were committed after the release was made, in order to provide users with certain bug fixes or other improvements. The packager thinks he is doing his users a favor by giving them the more recent code, but actually this practice can cause a great deal of confusion. Projects are prepared to receive reports of bugs found in released versions, and bugs found in recent trunk and major branch code (that is, found by people who deliberately run bleeding edge code). When a bug report comes in from these sources, the responder will often be able to confirm that the bug is known to be present in that snapshot, and perhaps that it has since been fixed and that the user should upgrade or wait for the next release. If it is a previously unknown bug, having the precise release makes it easier to reproduce and easier to categorize in the tracker.

Projects are not prepared, however, to receive bug reports based on unspecified intermediate or hybrid versions. Such bugs can be hard to reproduce; also, they may be due to unexpected interactions in isolated changes pulled in from later development, and thereby cause misbehaviors that the project's developers should not have to take the blame for. I have even seen dismayingly large amounts of time wasted because a bug was *absent* when it should have been present: someone was running a slightly patched up version, based on (but not identical to) an official release, and when the predicted bug did not happen, everyone had to dig around a lot to figure out why.

Still, there will sometimes be circumstances when a packager insists that modifications to the source release are necessary. Packagers should be encouraged to bring this up with the project's developers and describe their plans. They may get approval, but failing that, they will at least have notified the project of their intentions, so the project can watch out for unusual bug reports. The developers may respond by putting a disclaimer on the project's web site, and may ask that the packager do the same thing in the appropriate place, so that users of that binary package know what they are getting is not exactly the same as what the project officially released. There need be no animosity in such a situation, though unfortunately there often is. It's just that packagers have a slightly different set of goals from developers. The packagers mainly want the best out-of-the-box experience for their users. The developers want that too, of course, but they also need to ensure that they know what versions of the software are out there, so they can receive coherent bug reports and make compatibility guarantees. Sometimes these goals conflict. When they do, it's good to keep in mind that the project has no control over the packagers, and that the bonds of obligation run both ways. It's true that the project is doing the packagers a favor simply by producing the software. But the packagers are also doing the project a favor, by taking on a mostly unglamorous job in order to make the software more widely available, often by orders of magnitude. It's fine to disagree with packagers, but don't flame them; just try to work things out as best you can.

Testing and Releasing

Once the source tarball is produced from the stabilized release branch, the public part of the release process begins. But before the tarball is made available to the world at large, it should be tested and approved by some minimum number of developers, usually three or more. Approval is not simply a matter of inspecting the release for obvious flaws; ideally, the developers download the tarball, build and install it onto a clean system, run the regression test suite (see “Comprobaciones Automáticas”) in Capítulo 8, *Coordinando a los Voluntarios*, and do some manual testing. Assuming it passes these checks, as well as any other release checklist criteria the project may have, the developers then digitally sign the tarball using GnuPG (http://www.gnupg.org/), PGP (http://www.pgpi.org/), or some other program capable of producing PGP-compatible signatures.

In most projects, the developers just use their personal digital signatures, instead of a shared project key, and as many developers as want to may sign (i.e., there is a minimum number, but not a maximum). The more developers sign, the more testing the release undergoes, and also the greater the likelihood that a security-conscious user can find a digital trust path from herself to the tarball.

Once approved, the release (that is, all tarballs, zip files, and whatever other formats are being distributed) should be placed into the project's download area, accompanied by the digital signatures, and by MD5/SHA1 checksums (see http://en.wikipedia.org/wiki/Cryptographic_hash_function). There are various standards for doing this. One way is to accompany each released package with a file giving the corresponding digital signatures, and another file giving the checksum. For example, if one of the released packages is `scanley-2.5.0.tar.gz`, place in the same directory a file `scanley-2.5.0.tar.gz.asc` containing the digital signature for that tarball, another file `scanley-2.5.0.tar.gz.md5` containing its MD5 checksum, and optionally another, `scanley-2.5.0.tar.gz.sha1`, containing the SHA1 checksum. A different way to provide checking is to collect all the signatures for all the released packages into a single file, `scanley-2.5.0.sigs`; the same may be done with the checksums.

It doesn't really matter which way you do it. Just keep to a simple scheme, describe it clearly, and be consistent from release to release. The purpose of all this signing and checksumming is to give users a way to verify that the copy they receive has not been maliciously tampered with. Users are about to run

this code on their computers—if the code has been tampered with, an attacker could suddenly have a back door to all their data. See “Security Releases” later in this chapter for more about paranoia.

Candidate Releases

For important releases containing many changes, many projects prefer to put out *release candidates* first, e.g., `scanley-2.5.0-beta1` before `scanley-2.5.0`. The purpose of a candidate is to subject the code to wide testing before blessing it as an official release. If problems are found, they are fixed on the release branch and a new candidate release is rolled out (`scanley-2.5.0-beta2`). The cycle continues until no unacceptable bugs are left, at which point the last candidate release becomes the official release—that is, the only difference between the last candidate release and the real release is the removal of the qualifier from the version number.

In most other respects, a candidate release should be treated the same as a real release. The *alpha*, *beta*, or *rc* qualifier is enough to warn conservative users to wait until the real release, and of course the announcement emails for the candidate releases should point out that their purpose is to solicit feedback. Other than that, give candidate releases the same amount of care as regular releases. After all, you want people to use the candidates, because exposure is the best way to uncover bugs, and also because you never know which candidate release will end up becoming the official release.

Announcing Releases

Announcing a release is like announcing any other event, and should use the procedures described in “Publicity” in Capítulo 6, *Communications*. There are a few specific things to do for releases, though.

Whenever you give the URL to the downloadable release tarball, make sure to also give the MD5/SHA1 checksums and pointers to the digital signatures file. Since the announcement happens in multiple forums (mailing list, news page, etc.), this means users can get the checksums from multiple sources, which gives the most security-conscious among them extra assurance that the checksums themselves have not been tampered with. Giving the link to the digital signature files multiple times doesn't make those signatures more secure, but it does reassure people (especially those who don't follow the project closely) that the project takes security seriously.

In the announcement email, and on news pages that contain more than just a blurb about the release, make sure to include the relevant portion of the CHANGES file, so people can see why it might be in their interests to upgrade. This is as important with candidate releases as with final releases; the presence of bugfixes and new features is important in tempting people to try out a candidate release.

Finally, don't forget to thank the development team, the testers, and all the people who took the time to file good bug reports. Don't single out anyone by name, though, unless there's someone who is individually responsible for a huge piece of work, the value of which is widely recognized by everyone in the project. Just be wary of sliding down the slippery slope of credit inflation (see “Credit” in Capítulo 8, *Coordinando a los Voluntarios*).

Maintaining Multiple Release Lines

Most mature projects maintain multiple release lines in parallel. For example, after 1.0.0 comes out, that line should continue with micro (bugfix) releases 1.0.1, 1.0.2, etc., until the project explicitly decides to end the line. Note that merely releasing 1.1.0 is not sufficient reason to end the 1.0.x line. For example, some users make it a policy never to upgrade to the first release in a new minor or major series—they let others shake the bugs out of, say 1.1.0, and wait until 1.1.1. This isn't necessarily selfish (remember, they're forgoing the bugfixes and new features too); it's just that, for whatever reason, they've decided to be very careful with upgrades. Accordingly, if the project learns of a major bug in 1.0.3 right before it's about to release 1.1.0, it would be a bit severe to just put the bugfix in 1.1.0 and tell all the old 1.0.x users they should upgrade. Why not release both 1.1.0 and 1.0.4, so everyone can be happy?

After the 1.1.x line is well under way, you can declare 1.0.x to be at *end of life*. This should be announced officially. The announcement could stand alone, or it could be mentioned as part of a 1.1.x release announcement; however you do it, users need to know that the old line is being phased out, so they can make upgrade decisions accordingly.

Some projects set a window of time during which they pledge to support the previous release line. In an open source context, "support" means accepting bug reports against that line, and making maintenance releases when significant bugs are found. Other projects don't give a definite amount of time, but watch incoming bug reports to gauge how many people are still using the older line. When the percentage drops below a certain point, they declare end of life for the line and stop supporting it.

For each release, make sure to have a *target version* or *target milestone* available in the bug tracker, so people filing bugs will be able to do so against the proper release. Don't forget to also have a target called "development" or "latest" for the most recent development sources, since some people—not only active developers—will often stay ahead of the official releases.

Security Releases

Most of the details of handling security bugs were covered in “Announcing Security Vulnerabilities” in Capítulo 6, *Communications*, but there are some special details to discuss for doing security releases.

A *security release* is a release made solely to close a security vulnerability. The code that fixes the bug cannot be made public until the release is available, which means not only that the fixes cannot be committed to the repository until the day of the release, but also that the release cannot be publicly tested before it goes out the door. Obviously, the developers can examine the fix among themselves, and test the release privately, but widespread real-world testing is not possible.

Because of this lack of testing, a security release should always consist of some existing release plus the fixes for the security bug, with *no other changes*. This is because the more changes you ship without testing, the more likely that one of them will cause a new bug, perhaps even a new security bug! This conservatism is also friendly to administrators who may need to deploy the security fix, but whose upgrade policy prefers that they not deploy any other changes at the same time.

Making a security release sometimes involves some minor deception. For example, the project may have been working on a 1.1.3 release, with certain bug fixes to 1.1.2 already publicly declared, when a security report comes in. Naturally, the developers cannot talk about the security problem until they make the fix available; until then, they must continue to talk publicly as though 1.1.3 will be what it's always been planned to be. But when 1.1.3 actually comes out, it will differ from 1.1.2 only in the security fixes, and all those other fixes will have been deferred to 1.1.4 (which, of course, will now *also* contain the security fix, as will all other future releases).

You could add an extra component to an existing release to indicate that it contains security changes only. For example, people would be able to tell just from the numbers that 1.1.2.1 is a security release against 1.1.2, and they would know that any release "higher" than that (e.g., 1.1.3, 1.2.0, etc.) contains the same security fixes. For those in the know, this system conveys a lot of information. On the other hand, for those not following the project closely, it can be a bit confusing to see a three-component release number most of the time with an occasional four-component one thrown in seemingly at random. Most projects I've looked at choose consistency and simply use the next regularly scheduled number for security releases, even when it means shifting other planned releases by one.

Releases and Daily Development

Maintaining parallel releases simultaneously has implications for how daily development is done. In particular, it makes practically mandatory a discipline that would be recommended anyway: have each commit be a single logical change, and never mix unrelated changes in the same commit. If a change is too big or too disruptive to do in one commit, break it across N commits, where each commit is a well-

partitioned subset of the overall change, and includes nothing unrelated to the overall change.

Here's an example of an ill-thought-out commit:

```
------------------------------------------------------------------------
r6228 | jrandom | 2004-06-30 22:13:07 -0500 (Wed, 30 Jun 2004) | 8 lines

Fix Issue #1729: Make indexing gracefully warn the user when a file
is changing as it is being indexed.

* ui/repl.py
  (ChangingFile): New exception class.
  (DoIndex): Handle new exception.

* indexer/index.py
  (FollowStream): Raise new exception if file changes during indexing.
  (BuildDir): Unrelatedly, remove some obsolete comments, reformat
  some code, and fix the error check when creating a directory.

Other unrelated cleanups:

* www/index.html: Fix some typos, set next release date.
------------------------------------------------------------------------
```

The problem with it becomes apparent as soon as someone needs to port the `BuildDir` error check fix over to a branch for an upcoming maintenance release. The porter doesn't want any of the other changes—for example, perhaps the fix to issue #1729 wasn't approved for the maintenance branch at all, and the `index.html` tweaks would simply be irrelevant there. But she cannot easily grab just the `BuildDir` change via the version control tool's merge functionality, because the version control system was told that that change is logically grouped with all these other unrelated things. In fact, the problem would become apparent even before the merge. Merely listing the change for voting would become problematic: instead of just giving the revision number, the proposer would have to make a special patch or change branch just to isolate the portion of the commit being proposed. That would be a lot of work for others to suffer through, and all because the original committer couldn't be bothered to break things into logical groups.

In fact, that commit really should have been *four* separate commits: one to fix issue #1729, another to remove obsolete comments and reformat code in `BuildDir`, another to fix the error check in `BuildDir`, and finally, one to tweak `index.html`. The third of those commits would be the one proposed for the maintenance release branch.

Of course, release stabilization is not the only reason why having each commit be one logical change is desirable. Psychologically, a semantically unified commit is easier to review, and easier to revert if necessary (in some version control systems, reversion is really a special kind of merge anyway). A little up-front discipline on everyone's part can save the project a lot of headache later.

Planning Releases

One area where open source projects have historically differed from proprietary projects is in release planning. Proprietary projects usually have firmer deadlines. Sometimes it's because customers were promised that an upgrade would be available by a certain date, because the new release needs to be coordinated with some other effort for marketing purposes, or because the venture capitalists who invested in the whole thing need to see some results before they put in any more funding. Free software projects, on the other hand, were until recently mostly motivated by amateurism in the most literal sense: they were written for the love of it. No one felt the need to ship before all the features were ready, and why should they? It wasn't as if anyone's job was on the line.

Nowadays, many open source projects are funded by corporations, and are correspondingly more and

more influenced by deadline-conscious corporate culture. This is in many ways a good thing, but it can cause conflicts between the priorities of those developers who are being paid and those who are volunteering their time. These conflicts often happen around the issue of when and how to schedule releases. The salaried developers who are under pressure will naturally want to just pick a date when the releases will occur, and have everyone's activities fall into line. But the volunteers may have other agendas—perhaps features they want to complete, or some testing they want to have done—that they feel the release should wait on.

There is no general solution to this problem except discussion and compromise, of course. But you can minimize the frequency and degree of friction caused, by decoupling the proposed *existence* of a given release from the date when it would go out the door. That is, try to steer discussion toward the subject of which releases the project will be making in the near- to medium-term future, and what features will be in them, without at first mentioning anything about dates, except for rough guesses with wide margins of error. By nailing down feature sets early, you reduce the complexity of the discussion centered on any individual release, and therefore improve predictability. This also creates a kind of inertial bias against anyone who proposes to expand the definition of a release by adding new features or other complications. If the release's contents are fairly well defined, the onus is on the proposer to justify the expansion, even though the date of the release may not have been set yet.

In his multi-volume biography of Thomas Jefferson, *Jefferson and His Time*, Dumas Malone tells the story of how Jefferson handled the first meeting held to decide the organization of the future University of Virginia. The University had been Jefferson's idea in the first place, but (as is the case everywhere, not just in open source projects) many other parties had climbed on board quickly, each with their own interests and agendas. When they gathered at that first meeting to hash things out, Jefferson made sure to show up with meticulously prepared architectural drawings, detailed budgets for construction and operation, a proposed curriculum, and the names of specific faculty he wanted to import from Europe. No one else in the room was even remotely as prepared; the group essentially had to capitulate to Jefferson's vision, and the University was eventually founded more or less in accordance with his plans. The facts that construction went far over budget, and that many of his ideas did not, for various reasons, work out in the end, were all things Jefferson probably knew perfectly well would happen. His purpose was strategic: to show up at the meeting with something so substantive that everyone else would have to fall into the role of simply proposing modifications to it, so that the overall shape, and therefore schedule, of the project would be roughly as he wanted.

In the case of a free software project, there is no single "meeting", but instead a series of small proposals made mostly by means of the issue tracker. But if you have some credibility in the project to start with, and you start assigning various features, enhancements, and bugs to target releases in the issue tracker, according to some announced overall plan, people will mostly go along with you. Once you've got things laid out more or less as you want them, the conversations about actual release *dates* will go much more smoothly.

It is crucial, of course, to never present any individual decision as written in stone. In the comments associated with each assignment of an issue to a specific future release, invite discussion, dissent, and be genuinely willing to be persuaded whenever possible. Never exercise control merely for the sake of exercising control: the more deeply others participate in the release planning process (see “Share Management Tasks as Well as Technical Tasks” in Capítulo 8, *Coordinando a los Voluntarios*), the easier it will be to persuade them to share your priorities on the issues that really count for you.

The other way the project can lower tensions around release planning is to make releases fairly often. When there's a long time between releases, the importance of any individual release is magnified in everyone's minds; people are that much more crushed when their code doesn't make it in, because they know how long it might be until the next chance. Depending on the complexity of the release process and the nature of your project, somewhere between every three and six months is usually about the right gap between releases, though maintenance lines may put out micro releases a bit faster, if there is demand for them.

Capítulo 8. Coordinando a los Voluntarios

Conseguir que la gente se ponga de acuerdo sobre cuales son las necesidades de un proyecto, y trabajar en la misma dirección para llevarlas a cabo, requiere de algo más que un ambiente de trabajo genial sin disfunciones visibles. Requiere una, o varias personas, que coordinen diligentemente a toda la gente involucrada en el proyecto. Es posible que coordinar voluntarios no sea un arte tecnológico como lo pueda ser la programación, pero como cualquier arte puede mejorarse a través del estudio y la práctica.

Este capítulo es un cajón de sastre de técnicas para coordinar voluntarios. Se nutre, quizá más intensamente que capítulos anteriores, en el proyecto Subversion, en parte porque yo trabajaba en dicho proyecto mientras escribía esto y tenía información de primera mano, y en parte porque es más aceptable tirar piedras sobre mi propio tejado que sobre el tejado de otro. No obstante, también he sido testigo de las consecuencias que ha tenido aplicar sobre otros proyectos (y las consecuencias de no hacerlo) las recomendaciones que se exponen a continuación. Siempre que sea políticamente correcto dar detalles de otros proyectos, así lo haré.

Hablando de política, este momento es tan bueno como cualquier otro para inspeccionar en detalle la susodicha palabra maldita. Muchos ingenieros hablan de política como si no fuera con ellos; "*Yo* me centro en la mejor para el proyecto, pero *Ella* no deja de poner pegas por razones políticas." Pienso que dicho distanciamiento de la política (o por lo que se piensa que es la política) es especialmente significativo entre los ingenieros, porque se los educa para pensar que siempre hay razones objetivas para determinar que unas soluciones son superiores a otras. Por tanto, cuando alguien actúa con motivaciones aparentemente externas al proyecto (p.ej. solidificar la propia posición de influencia en el proyecto, disminuir la influencia de otros, chalaneos descarados, o evitar herir los sentimientos de alguien) otros participantes del proyecto pueden molestarse. Por supuesto, esto raramente evita que ellos se comporten de la misma manera cuando sus propios intereses se ven amenazados.

Si consideras que la "política" es una palabra sucia, y tienes esperanzas de mantener tu proyecto libre de ella, mejor que abandones ahora mismo. La política es algo que surge inevitablemente cada vez que un grupo de personas han de cooperar en la gestión de recursos compartidos. Es de lógica que, cada vez que una persona toma una decisión, ésta se vea influenciada por cómo va a afectar a su futuro personal en el proyecto. Después de todo, si tienes confianza en tu juicio y habilidades, como sucede con la mayoría de programadores, entonces una posible pérdida de influencia sobre el proyecto podría ser considerada, en cierta manera, como un aspecto técnico a tener en cuenta. Razonamientos similares se pueden aplicar a comportamientos que, en apariencia, sean "pura" política. De hecho, no existe tal cosa como la "pura política"; para empezar, porque como todas las acciones tienen múltiples consecuencias en la vida real la gente adquiere conciencia política. La política, al final, simplemente reconoce que *todas* las consecuencias de nuestras decisiones han de tenerse en cuenta. Si decisiones concretas llevan el proyecto por un camino que la mayoría considera técnicamente correcto, pero dicha decisión conlleva un cambio en el equilibrio de influencias del proyecto aislando a figuras clave del mismo, entonces hemos de dar igual grado de importancia a ambas consecuencias. No hacerlo así, no sólo no sería juicioso, sino que sería una actitud corta de miras.

Por tanto, mientras leas los consejos que siguen a continuación, y mientras trabajas en tu propio proyecto, recuerda que no hay *nadie* que esté por encima de la política. Dar la impresión que se está por encima de ella es simplemente otra estrategia política más y, por cierto, en ocasiones una muy útil, pero nunca refleja la realidad. La política es sencillamente lo que sucede cuando la gente no está de acuerdo en algo, y los proyectos que acaban con éxito son aquellos capaces de desarrollar un mecanismo político para gestionar los desacuerdos de forma constructiva.

Conseguir el Máximo de los Voluntarios

Por qué los voluntarios trabajan en proyectos de software libre? [1]

Cuando se les pregunta, muchos dicen que lo hace por que quieren producir buen software, o por que quieren involucrarse en arreglar errores que son importantes para ellos. Pero estas razones no acostumbran a ser toda la verdad. Después de todo ¿podrías imaginarte a un voluntario permaneciendo en un proyecto a pesar de que nadie, nunca, le dirigiera un palabra de aprecio por su trabajo, o se le escuchara en las discusiones? Claro que no. Claramente, la gente gasta su tiempo en el software libre por razones que van más allá del deseo abstracto de producir buen código. Comprender las verdaderas motivaciones de los voluntarios te ayudará a organizar las cosas de manera que puedas atraerlos y que no se vayan. El deseo de producir buen código puede estar entre esas motivaciones, junto con el desafío y valor educativo que supone trabajar en problemas difíciles, pero los seres humanos tienen el deseo natural de trabajar con otros humanos, y el de dar y recibir respeto a través de actividades colectivas. Grupos cooperando en tareas colectivas deben desarrollar normas de comportamiento tales que el prestigio sea adquirido ,y mantenido, a través de acciones que ayuden a la consecución de las metas del grupo.

Esas normas no aparecen por sí solas. Por ejemplo, en algunos proyectos (desarrolladores con experiencia en código abierto podrían mencionar varias de memoria) la gente piensa que se gana prestigio por enviar largos mensajes frecuentemente. No llegan a esta conclusión por casualidad, llegan a ella porque son recompensados con respeto por crear largos e intrincados argumentos, independientemente de que estos ayuden al proyecto. A continuación se explican algunas técnicas para crear una atmósfera en la que se pueda adquirir prestigio a través de acciones constructivas.

Delegar

Delegar no simplemente una forma de distribuir la carga de trabajo, sino que también es una herramienta política y social. Piensa en todas las consecuencias que tiene pedir a alguien que haga algo. El efecto más evidente es que, si acepta, esa persona hace el trabajo y tú no. Pero otro efecto es que esa persona toma consciencia de que tú confías en él para realizar la tarea. Además, si haces la petición en un foro público, también aprecia que el resto de participantes son conscientes de la confianza que ha sido depositada en él. También es posible que se sienta presionado a aceptar, por tanto has de preguntar de manera que le sea fácil declinar la oferta si realmente no quiere aceptarla. Si la tarea requiere coordinarse con otros participantes del proyecto, entonces le estás pidiendo un mayor grado de compromiso con el proyecto, crear lazos que no se hubieran creado de otra forma y, quizá, convertirse en una fuente de autoridad y algunos subdominios del proyecto. La responsabilidad adquirida puede ser agobiante, o puede llevarle a involucrarse en más áreas del proyecto gracias a un renovado sentimiento de compromiso.

Debido a todos estos posibles efectos, a menudo tiene sentido pedir a otro que haga algo incluso cuando sabes que tú lo podrías hacer mejor o más rápido. Por supuesto, siempre hay momentos en que delegarás únicamente por motivos de eficiencia; quizá el coste de oportunidad de hacer una tarea tú mismo es demasiado alto debido a que podrías dedicarte a hacer algo más importante para el proyecto. Pero incluso cuando el argumento del coste de oportunidad no existe, *aún así* es posible que pidas a otro hacer una tarea simplemente porque quieres involucrar más a una determinada persona en el proyecto, incluso si eso significa que has de supervisar su trabajo al principio. Esta técnica también funciona al revés; si ocasionalmente te presentas voluntario para hacer un trabajo que alguien no quiere, o no tiene tiempo de hacer, te ganarás su confianza y su respeto. Delegar y sustituir no tiene que ver simplemente con finalizar tareas; tiene mucho que ver también con aumentar el grado de compromiso de la gente con el proyecto.

Distingue claramente entre pedir y asignar

En ocasiones es lógico esperar que una persona aceptará de buen grado una tarea en particular. Por ejemplo, si alguien introduce un error en el código, o envía código que incumple de forma clara con las directrices del proyecto, debería ser suficiente con mencionar el problema y esperar que el responsable de dicho código se haga cargo. Pero hay otras situaciones donde no está nada claro que puedas esperar que alguien haga cargo. La persona podría hacer lo que le pides, o quizá no. Como a nadie le gusta que se dé por hecho que va a obedecer, has de ser consciente en todo momento de cual de estas dos situacio-

[1]Esta cuestión fue estudiada en detalle, arrojando interesantes resultados, en un ensayo de Karim Lakhani y Robert G. Wolf, titulado*Por qué los Hackers Hacen lo que Hacen: Comprensión de la Motivación y el Esfuerzo en Proyectos de Código Libre/Abierto*. Ved http://freesoftware.mit.edu/papers/lakhaniwolf.pdf.

nes estás tratando y, en función de la misma, medir tus palabras a la hora de solicitar una tarea.

Algo que casi siempre sienta mal es que te pidan hacer algo como si fuera tu responsabilidad cuanto tu piensas que no es así. Por ejemplo, asignar nuevos problemas en el código es terreno abonado para este tipo de situaciones. Los participantes de un proyecto normalmente saben quien es experto en cada área, así que cuando un error en el código aparece, habrá normalmente una o dos personas en las que todo el mundo piensa que podrían arreglar el problema rápidamente. No obstante, si asignas la tarea a una de estas personas sin pedirles permiso, dicha persona podría sentirse incómoda y forzada a encargarse de la tarea, podría sentir que se espera mucho de ella y que, de alguna forma, está siendo castigada por sus conocimientos y experiencia. Después de todo, la forma en la que uno adquiere dichas habilidades es corrigiendo código, así que ¡quizá debería ser otro el que lo corrija esta vez! (Cabe mencionar que los gestores automáticos de errores que asignan tareas en función del tipo de error, reducen este tipo de conflictos porque todo el mundo sabe que las tareas se asignan de forma automática y que nadie espera nada de la persona a la que se ha asignado la tarea.)

Aunque lo deseable sería distribuir la carga de trabajo tan equitativamente como fuera posible, hay ocasiones en las que simplemente quieres que puede arreglar el problema más rápidamente haga el trabajo. Dado que no puedes permitirte establecer un diálogo para todas y cada una de las tareas que se han de asignar ("¿Te importaría echarle un vistazo a este error?" "Sí." "Vale, entonces te asigno la tarea a ti." "Vale."), deberías siempre asignar tareas en forma de petición, sin ejercer presión alguna. Casi todos los gestores de errores permiten asociar comentarios a la tarea asignada. En dicho comentario podrías decir algo así:

> Tarea asignada a ti, jaleatorio, porque tu eres la persona más familiarizada con este código. No te preocupes si no puedes hacerte cargo de la tarea por el motivo que sea. (En cualquier caso hazme saber si preferirías no recibir más tareas en el futuro.)

Así se distingue claramente la *petición* de la tarea de la *aceptación* de la misma. El mensaje no sólo va dirigido a la persona a la que se pide la tarea; informa a todo el grupo del área de experiencia de dicho miembro y, además, deja claro a todos que dicha persona es libre de aceptar o rechazar la tarea.

Supervisar después de delegar

Cuando le pidas a alguien que haga algo, recuerda que los has hecho y supervisa con él la tarea pase lo que pase. La mayoría de peticiones se hace en foros públicos más o menos de la siguiente manera "¿Podrías encargarte de X? Dinos algo en cualquier caso. En caso de que no puedas no pasa nada, pero háznoslo saber." Pueden responder, o no, a tu petición, pero si te responden, y la respuesta es negativa, el proceso se cierra y tendrás que buscar alternativas para llevar a cabo la tarea X. Si la respuesta es positiva entonces vigila como progresa el trabajo y haz comentarios sobre el mismo (todo el mundo trabaja mejor cuando sabe que hay alguien que aprecia su trabajo). Si no hay respuesta después de unos días, pregunta de nuevo, o comenta en el foro que nadie a respondido y que buscas a alguien que ses encargue de la tarea, o simplemente hazlo tú pero, en cualquier caso, asegúrate de comentar que no recibiste respuesta alguna a tu petición"

El motivo de hacer pública la falta de respuesta *no* es para humillar a la persona a la que se le hizo la petición, y cuando menciones el tema lo has de hacer de forma que esto quede claro. El propósito es de dejar claro que sigues la pista de todo lo que pides, y que reaccionas ante las respuestas que recibes, o no, a tus peticiones. Esta actitud hace que sea más probable que la gente acepta la tarea la próxima vez, y esto es así porque se darán cuenta (aunque sea inconscientemente) que es probable que aprecies cualquier trabajo que hagan debido a que prestas atención a detalles tan pequeños como el que alguien no responda a una petición.

Fíjate en lo que se interesa la gente

Otra cosa que pone a la gente contenta es que te fijes en sus intereses; en general, cuantos más aspectos recuerdas de la personalidad de alguien, más a gusto se encontrará dicha persona, y se interesará más por

trabajar en grupos de los que tú seas parte.

Por ejemplo, había una clara diferencia en el proyecto Subversion entre la gente que quería llegar pronto a una versión final 1.0 (cosa que finalmente hicimos), y la gente que principalmente quería añadir nuevas funcionalidades y trabajar en problemas interesantes sin importarles mucho cuando estaría lista la versión 1.0. Ninguna de estas dos posturas es mejor o peor que la otra; simplemente reflejan dos tipos distintos de desarrolladores, y ambos realizan mucho trabajo en el proyecto. Pero aprendimos rápidamente que era vital *no* asumir que el interés por tener una versión 1.0 era compartido por todos. Las comunicaciones electrónicas pueden ser muy engañosos; puedes pensar que hay un ambiente de propósito común cuando, en realidad, dicho propósito sólo es compartido por la gente con la que tú has estado hablando, mientras otras personas tienen prioridades completamente distintas

Cuanto más consciente seas de lo que la gente quiere sacar del proyecto, más eficientemente podrás pedirles cosas. Simplemente demostrando comprensión por lo que quieren, sin demandar nada por ello, tiene una utilidad per se, puesto que confirma a cada uno que no es una simple partícula de una masa informe.

Halagos y Críticas

Halago y crítica no son antónimos, de hecho, en muchos aspectos son muy similares. Ambas son muestras de atención, y son más efectivas cuando son específicas que cuando son genéricas. Ambas deben hacerse con metas específicas en mente. Ambas pueden diluirse por el abuso; halaga mucho o demasiado y tus halagos perderán valor, y lo mismo sirve para las críticas aunque, en la práctica, las críticas provocan reacciones que las hacen mucho más resistentes a la devaluación.

Un importante aspecto de la cultura tecnológica es que la detallada y desapasionada crítica a menudo se toma como una especia de alago (como se vio en “Reconociendo la grosería” en Capítulo 6, *Communications*), esto se debe a la implicación de que el trabajo en cuestión vale la pena ser analizado. En cualquier caso, ambos aspectos; *detallada* y *desapasionada* han de cumplirse para que esto se cumpla. Por ejemplo, si alguien hace algún cambio chapucero en el código, es inútil (y de hecho perjudicial) comentar el asunto simplemente diciendo "Eso es una chapuza". El ser un chapuzas es, al final, una característica de la *persona*, no de su trabajo, y es importante mantener tu atención enfocada en el trabajo hecho. Es mucho más eficiente describir todas las cosas equivocadas que se han introducido al realizar el cambio, y hay que hacerlo con tacto y sin malicia. Si fuera el tercer o cuarto cambio descuidado de la misma persona, entonces lo más apropiado mencionar el hecho, después de la crítica sobre el trabajo realizado, y de nuevo sin ningún signo de enfado, para así dejar claro que dicho patrón de comportamiento es evidente.

Si alguien no mejora después de las críticas, la solución no es criticar más, o hacerlo más duramente. La solución consiste en que el grupo retire a dicha persona del trabajo en el que es incompetente de forma que se hiera lo menos posible los sentimientos de la misma, leed “Transitions” al final del capítulo unos ejemplos. En cualquier caso, este no es un hecho frecuente. La mayoría de gente responde muy bien a las críticas que son específicas, detalladas, y que contienen una clara (aunque sea entre líneas) indicación de que se espera una mejora.

Las alabanzas no herirá la sensibilidad de nadie, por supuesto, pero eso no significa que se deba usar con menos cuidado que las críticas. Las alabanzas son una herramienta; antes de usarla pregúntate por *qué* quieres usarla. Como regla, no es una buena idea alabar a alguien por hacer lo que normalmente hace, o por acciones que son habituales y esperadas de alguien que trabaja en grupo. Si hicieras eso, no será fácil saber cuando parar; ¿deberías alabar a *todo el mundo* por hacer lo habitual? Al fin y al cabo, si te dejas a alguien se preguntarán por qué. Es mucho mejor expresar alabanzas y gratitud ocasionalmente en respuesta a un esfuerzo inusual, o inesperado, con la intención de fomentar dichos esfuerzos. Cuando un participante parece haberse trasladado permanentemente a un estado de alta productividad, debes ajustar tu nivel de alabanzas consecuentemente. Repetidas alabanzas se acaban convirtiendo en algo sin significado alguno. En su lugar, dicha persona deber sentir que su alto nivel de productividad se considera normal y natural, y sólo trabajo que sobrepasa ese nivel debe ser significado.

Por supuesto, esto no quiere decir que las contribuciones de dicha persona no deban ser reconocidas. Pero recuerda que si el proyecto se organiza correctamente, todo lo que hace una persona es visible de todas formas, y por tanto el grupo verá (y la persona implicada sabrá que el resto de miembros lo ven también) todo lo que ella hace. También hay otras maneras de reconocer el trabajo de alguien además de las alabanzas. Podrías mencionar de pasada, al debatir sobre un tema, que dicha persona ha trabajado mucho en ese área y que es experta en la misma. Podrías realizar consultas públicas a dicha persona sobre el código o, quizá mejor, podrías utilizar su trabajo de forma ostensible para que la persona pueda apreciar que la gente tiene confianza en los resultados de su trabajo. Probablemente no es necesario hacer todas estas cosas de forma calculada. Las personas que contribuyen notablemente lo saben y ocuparán una posición de influencia de forma natural. Normalmente no hay que tomar medidas explícitas para asegurar esto, a menos que sientas que, por cualquier motivo, un miembro del grupo es poco valorado.

Prevén la Territorialidad

Ten cuidado con los participantes que intentan apropiarse la exclusividad en ciertas áreas del proyecto, y con aquellos que parecen querer hacer todo el trabajo en esas áreas hasta el punto de apropiarse del trabajo que otros han comenzado. Dicho comportamiento puede parecer saludable al principio, después de todo, a primera vista parece como si el individuo en cuestión simplemente está tomando más responsabilidad, y mostrando una mayor actividad en dichas áreas. A la larga, no obstante, dicho comportamiento es destructivo. Cuando la gente ve señales de "no pasar" se apartan del proyecto. Esto conlleva una visión más estrecha de esa área, y una mayor fragilidad de la misma puesto se depende de la disponibilidad e un único desarrollador. Todavía peor, rompe el espíritu de cooperación igualitaria del proyecto. En teoría, la ayuda de cualquier desarrollador debe ser bienvenida en cualquier momento, y sobre cualquier área. Por supuesto, en la práctica las cosas funciona de manera algo diferente; la gente tiene áreas donde es más o menos influyente, y los inexpertos habitualmente dejan que los expertos tomen las riendas en ciertos dominios del proyecto. Pero la clave es que todo esto es algo voluntario; la autoridad se gana mediante la competencia y conocimiento probado, y nunca debe ser *conquistada* activamente. Incluso si la persona deseando la autoridad es realmente competente, aún así es crucial que maneje esa autoridad de manera informal, a través del consenso del grupo, y sin apartar a nadie de colaborar en su área de influencia.

Por supuesto, rechazar o editar el trabajo de otro por motivos técnicos es un asusto totalmente distinto. En este caso, el factor decisivo es el contenido del trabajo, no quien actúa como portero. Podría suceder que la misma persona realice la mayor parte de revisiones para un área particular, pero mientras no evite que otros hagan su trabajo, las cosas deberían ir bien.

Para combatir cualquier territorialismo incipiente, o incluso la mera apariencia del mismo, muchos proyectos han tomado medidas como la de prohibir la inclusión del nombre del autor, o de los encargados elegidos, en el código fuente. Yo estoy de acuerdo de todo corazón con esta práctica; la utilizamos en el proyecto Subversion y es, más o menos, la política oficial en la Apache Software Foundation. El miembro del ASF Sander Striker lo explica de esta forma:

> *en la Apache Software Foundation desaconsejamos el uso de entradas con el nombre del autor en el código fuente. Hay varias razones para esto además de motivos legales. En el desarrollo en equipo se trata de trabajar en grupo y tratar al proyecto en grupo. Dar crédito es bueno, y debe hacerse, pero de alguna manera esta forma de actuar evita falsas atribuciones, incluso cuando sólo son implícitas. No hay una directriz clara de cuando se ha de añadir o quitar entradas con el nombre del autor; ¿Añades tu nombre si cambias un comentario? ¿Cuando arreglas una sola línea de código? ¿Has de borrar el nombre de alguien cuando reestructuras el código y es diferente al anterior en un 95%? ¿Qué haces con la gente que va tocando cada archivo, cambiando lo mínimo para que su nombre aparezca en todas partes?*
>
> *Hay mejores formas de dar crédito, y nosotros preferimos usar esas. Desde un punto de vista técnico las entradas con el nombre del autor son innecesarias; si quieres saber quién escribió una determinada línea de código se puede consultar el sistema de control de versiones para averiguarlo. Además, las entradas de autor acostumbran a*

> *estar caducadas; ¿Realmente quieres que se pongan en contacto contigo por unas líneas de código que programaste hace cinco años y estás contento de haberlas olvidado?*

El corazón de la identidad de un proyecto lo forman los archivos con el código fuente. Estos deben reflejar que la comunidad, como un todo, es responsable de los mismos, y no deben dividirse en pequeños feudos.

La gente a veces defiende las entradas de autor en el código fuente argumentando que dan crédito de forma visible a aquellos que han realizado más trabajo. Hay dos problemas con este argumento. Primero, las entradas de autor traen consigo la incómoda pregunta de cuánto trabajo se ha de realizar para que tu nombre también aparezca en el archivo. Segundo, las entradas fusiona la autoridad en un área con el crédito en la misma; haber hecho la mayor parte del trabajo en un área no implica que se posea dicha área, pero es difícil, sino imposible, evitar dicha implicación cuando hay nombres de personas al comienzo de un archivo de código fuente. En cualquier caso, se puede saber el autor del código a través del sistema de control de versiones u otros métodos alternativos como los archivos de las listas de correos, de esta forma no se pierde ninguna información al no permitir las entradas de autor en el código fuente.

Si en tu proyecto se decide no permitir incluir entradas de autores en el código fuente, asegúrate de no pasarte de la raya. Por ejemplo, muchos proyectos tienen un área `contrib/` donde se almacenan pequeñas herramientas y scripts de ayuda, a menudo escritos por gente que no están asociados con el proyecto de ninguna otra manera. Para ese tipo de archivos está bien que se introduzca los nombres de los autores porque ellos no están desarrollando el proyecto en sí. Por otro lado, si una de estas herramientas comienza a ser alterada por otros miembros del proyecto, puede que al final quieras trasladar dicha herramienta a un lugar menos aislado y, suponiendo que el autor original aceptara, borrar los nombres de los autores para que el archivo siga la política del resto de archivos del proyecto. Si el autor original no se siente a gusto con esta iniciativa se pueden alcanzar acuerdos, por ejemplo:

```
# indexclean.py: Borrar datos viejos del índice Scanley.
#
# Autor Original: K. Maru <kobayashi@yetanotheremailservice.com>
# Gestionado Ahora Por: The Scanley Project <http://www.scanley.org/>
#                       and K. Maru.
#
# ...
```

Pero, dentro de lo posible, es mejor evitar dichos compromisos, además, la mayoría de autores están dispuestos a ser persuadidos puesto que se sienten felices de que su contribución pase a ser parte integral del proyecto.

Lo importante es recordar que hay una continuidad entre el centro y la periferia de cualquier proyecto. Los archivos del código fuente del software son claramente centrales en el proyecto, y deben ser gestionados por la comunidad en su conjunto. Por otro lado, herramientas accesorias, o documentación, pueden ser el resultado del trabajo de un solo individuo, y lo puede gestionar él solo aunque su trabajo esté asociado, en incluso distribuido, con el proyecto. En cualquier caso, no hay necesidad de aplicar la misma regla a todos los archivos siempre y cuando los recursos comunitarios sean gestionados por todos, y no puedan convertirse en conto privado de nadie.

El Ratio de Automatización

Intenta evitar que los humanos hagan lo que pueden hacer las máquinas en su lugar. Como regla general, automatizar una tarea común supone un esfuerzo diez veces menor al esfuerzo que le supondría al desarrollador realizar la tarea a mano. Para tareas muy frecuentes, o muy complejas, el ratio puede ser veinte veces superior e incluso mayor.

Verte a ti mismo como a un "Director de Proyecto", en lugar de como a un desarrollador, puede llegar a ser una actitud positiva. A veces, algunos desarrolladores están demasiado enfrascados en su trabajo a bajo nivel, esto no les permite tener una visión general del proyecto y darse cuenta de que todo el mundo está desperdiciando mucho esfuerzo en realizar tareas manualmente que podrían muy bien automatizarse. Incluso aquellos que sí se dan cuenta, pueden no tomarse el tiempo para resolver el problema, al fin el al cabo, el rendimiento de cada individuo para realizar dicha tarea no es una carga demasiada onerosa, nadie se siente lo suficientemente molesto como para hacer algo al respecto. La automatización se hace atractiva cuando se tiene en cuenta que, esa pequeña carga, se multiplica por el número de veces que cada desarrollador ha de realizarla, y entonces *ese* número se múltiplica por el número de desarrolladores.

Aquí estoy utilizando la palabra "automatización" con un sentido muy amplio queriendo dar a entender, no sólo acciones repetitivas donde una o dos variables cambian cada vez, sino también cualquier tipo de infraestructura técnica que ayude a los humanos. La mínima automatización estándar necesaria para llevar a cabo un proyecto en nuestros días fue descrita en Capítulo 3, *Infraestructura Técnica*, pero cada proyecto puede tener sus propios problemas especiales. Por ejemplo, un grupo trabajando en la documentación quizá quiera una página web que muestre las versiones más actualizadas del documento en todo momento. Como la documentación se redacta normalmente en lenguajes del estilo de XML, puede haber un paso de compilación, a menudo bastante intrincado, relacionado con la creación de documentos para que puedan ser mostrados o bajados de la red. Crear un página web donde esa compilación tenga lugar automáticamente en cada envío puede ser bastante complejo y llevar mucho tiempo—pero vale la pena, incluso si te lleva uno o más días configurarla. Los beneficios globales de tener páginas actualizadas disponibles en todo momento son enormes, incluso si el coste de *no* tenerlas pueda parecer una simple molestia en un momento dado, para un desarrollador cualquiera.

Seguir estos pasos no sólo elimina las pérdidas de tiempo, sino también las obsesiones y frustraciones que aparecen cuando los humanos cometen errores (inevitablemente lo harán) al intentar realizar procedimientos complicados manualmente. Las tareas con múltiples pasos y operaciones deterministas son el tipo de cosas para las que se inventaron las computadoras, y así dejamos que los humanos hagan cosas más interesantes.

Comprobaciones Automáticas

Ejecutar pruebas automáticas es muy útil para cualquier proyecto de software, pero especialmente para proyectos de código abierto porque las pruebas automáticas (especialmente las pruebas de regresión) permiten que los desarrolladores se encuentren a gusto a la hora de cambiara código en áreas en las que no están familiarizados y, así, se favorece el desarrollo de exploración. Como es muy difícil detectar fallos a simple vista—básicamente has de adivinar dónde alguien puede haberse equivocado y realizar varios experimentos para asegurarte de que no lo hizo—tener métodos automáticos para detectar dichos errores ahorra *muchísimo* tiempo. También hace que la gente se relaje a la hora de refactorizar grandes franjas de código y, por lo tanto, contribuye a que el software pueda ser gestionado a largo plazo.

Pruebas de Regresión

Por *Pruebas de Regresión* se entienden las comprobaciones que se realizan para detectar errores que ya se han reparado. El objetivo de las pruebas de regresión es reducir las probabilidades de que los cambios en el código rompan el software de forma inesperada. Cuando un proyecto de software crece y se complica, las probabilidades de encontrarse dichos efectos secundarios crecen al mismo paso. Un buen diseño puede reducir el crecimiento del ratio de dicha probabilidad, pero no puede eliminar el problema completamente.

Como resultado de esta situación, muchos proyectos tienen una *colección de pruebas*, un programa aparte que ejecuta el software del proyecto en formas y maneras que se sabe producian errores específicos con anterioridad. Cuando la colección de pruebas consigue reproducir uno de estos errores, esto es conocido como *regresión*, dando a entender que las modificaciones de alguien han vuelto a recrear inesperadamente un error corregido con anterioridad.

Ved también http://en.wikipedia.org/wiki/Regression_testing.

Las pruebas de regresión no son un panacea. Por un lado funciona bien para programas que tienen un estilo de interfaz de lineas de comando. El software que se utiliza principalmente através de una interfaz gráfica es mucho más difícil de utilizar mediante otro programa. Otro problema radica en que la estructura misma de la colección de pruebas puede ser muy compleja, con una curva de aprendizaje y cargas de mantenimiento propias. Reducir esta complejidad es una de las cosas más útilies que puedes hacer, incluso si ello implica una cantidad de tiempo considerable. Cuanto más fácil sea añadir nuevas pruebas a la colección, más desarrolladores lo harán, y menos errores sobrevivirán a la versión final. Cualquier esfuerzo empleado en que la creación de pruebas sea sencilla redundará con intereses en el desarrollo del proyecto.

Muchos proyectos siguen la regla *"¡No rompas el código!"* , queriendo decir: no envíes cambios que hagan que sea imposible compilar o ejecutar el software. Ser la persona que rompe el código es normalmente causa de cierta vergüenza y burla. Proyectos con colecciones de pruebas de regresión tienen amenudo una nueva regla a modo de corolario: no envíes ningún cambio que hagan fallar las pruebas. Dichos fallos son más fáciles de identificar si se realizan ejecuciones automáticas cada noche de toda la colección de pruebas, con los resultados enviados por email a los desrrolladores o a listas de correo dedicadas al efecto; este es otro ejemplo de automatización que vale la pena.

La mayoría de los desarrolladores voluntarios están dispuestos a tomar el tiempo adicional para escribir pruebas de regresión, cuando el sistema de pruebas es comprensible y fácil de trabajar. Acompañar modificaciones con pruebas es entendido como una responsabilidad de hacerlo, y es también una oportunidad fácil para la colaboración: a menudo los desarrolladores dividen el trabajo para corregir un fallo, uno de ellos corrige, y el otro escribe el ejemplo de prueba. El segundo desarrollador puede terminar con más trabajo, y aunque escribir un ejemplo de prueba es menos satisfactorio que corregir realmente el fallo, es imprescindible que el conjunto de pruebas no haga la experiencia más dolorosa de lo que debe ser.

Some projects go even further, requiring that a new test accompany *every* bugfix or new feature. Whether this is a good idea or not depends on many factors: the nature of the software, the makeup of the development team, and the difficulty of writing new tests. The CVS (http://www.cvshome.org/) project has long had such a rule. It is a good policy in theory, since CVS is version control software and therefore very risk-averse about the possibility of munging or mishandling the user's data. The problem in practice is that CVS's regression test suite is a single huge shell script (amusingly named `sanity.sh`), hard to read and hard to modify or extend. The difficulty of adding new tests, combined with the requirement that patches be accompanied by new tests, means that CVS effectively discourages patches. When I used to work on CVS, I sometimes saw people start on and even complete a patch to CVS's own code, but give up when told of the requirement to add a new test to `sanity.sh`.

It is normal to spend more time writing a new regression test than on fixing the original bug. But CVS carried this phenomenon to an extreme: one might spend hours trying to design one's test properly, and still get it wrong, because there are just too many unpredictable complexities involved in changing a 35,000-line Bourne shell script. Even longtime CVS developers often grumbled when they had to add a new test.

This situation was due to a failure on all our parts to consider the automation ratio. It is true that switching to a real test framework—whether custom-built or off-the-shelf—would have been a major effort.[2] But neglecting to do so has cost the project much more, over the years. How many bugfixes and new features are *not* in CVS today, because of the impediment of an awkward test suite? We cannot know the exact number, but it is surely many times greater than the number of bugfixes or new features the developers might forgo in order to develop a new test system (or integrate an off-the-shelf system). That task would only take a finite amount of time, while the penalty of using the current test suite will conti-

[2]Note that there would be no need to convert all the existing tests to the new framework; the two could happily exist side by side, with old tests converted over only as they needed to be changed.

nue forever if nothing is done.

The point is not that having strict requirements to write tests is bad, nor that writing your test system as a Bourne shell script is necessarily bad. It might work fine, depending on how you design it and what it needs to test. The point is simply that when the test system becomes a significant impediment to development, something must be done. The same is true for any routine process that turns into a barrier or a bottleneck.

Treat Every User as a Potential Volunteer

Each interaction with a user is an opportunity to get a new volunteer. When a user takes the time to post to one of the project's mailing lists, or to file a bug report, he has already tagged himself as having more potential for involvement than most users (from whom the project will never hear at all). Follow up on that potential: if he described a bug, thank him for the report and ask him if he wants to try fixing it. If he wrote to say that an important question was missing from the FAQ, or that the program's documentation was deficient in some way, then freely acknowledge the problem (assuming it really exists) and ask if he's interested in writing the missing material himself. Naturally, much of the time the user will demur. But it doesn't cost much to ask, and every time you do, it reminds the other listeners in that forum that getting involved in the project is something anyone can do.

Don't limit your goals to acquiring new developers and documentation writers. For example, even training people to write good bug reports pays off in the long run, if you don't spend *too* much time per person, and if they go on to submit more bug reports in the future—which they are more likely to do if they got a constructive reaction to their first report. A constructive reaction need not be a fix for the bug, although that's always the ideal; it can also be a solicitation for more information, or even just a confirmation that the behavior *is* a bug. People want to be listened to. Secondarily, they want their bugs fixed. You may not always be able to give them the latter in a timely fashion, but you (or rather, the project as a whole) can give them the former.

A corollary of this is that developers should not express anger at people who file well-intended but vague bug reports. This is one of my personal pet peeves; I see developers do it all the time on various open source mailing lists, and the harm it does is palpable. Some hapless newbie will post a useless report:

> Hi, I can't get Scanley to run. Every time I start it up, it just errors. Is anyone else seeing this problem?

Some developer—who has seen this kind of report a thousand times, and hasn't stopped to think that the newbie has not—will respond like this:

> What are we supposed to do with so little information? Sheesh. Give us at least some details, like the version of Scanley, your operating system, and the error.

This developer has failed to see things from the user's point of view, and also failed to consider the effect such a reaction might have on all the *other* people watching the exchange. Naturally a user who has no programming experience, and no prior experience reporting bugs, will not know how to write a bug report. What is the right way to handle such a person? Educate them! And do it in such a way that they come back for more:

> Sorry you're having trouble. We'll need more information in order to figure out what's happening here. Please tell us the version of Scanley, your operating system, and the exact text of the error. The very best thing you can do is send a transcript showing the exact commands you ran, and the output they produced. See http://www.scanley.org/how_to_report_a_bug.html for more.

This way of responding is far more effective at extracting the needed information from the user, because

it is written to the user's point of view. First, it expresses sympathy: *You had a problem; we feel your pain*. (This is not necessary in every bug report response; it depends on the severity of the problem and how upset the user seemed.) Second, instead of belittling her for not knowing how to report a bug, it tells her how, and in enough detail to be actually useful—for example, many users don't realize that "show us the error" means "show us the exact text of the error, with no omissions or abridgements." The first time you work with such a user, you need to be specific about that. Finally, it offers a pointer to much more detailed and complete instructions for reporting bugs. If you have successfully engaged with the user, she will often take the time to read that document and do what it says. This means, of course, that you have to have the document prepared in advance. It should give clear instructions about what kind of information your development team wants to see in every bug report. Ideally, it should also evolve over time in response to the particular sorts of omissions and misreports users tend to make for your project.

The Subversion project's bug reporting instructions are a fairly standard example of the form (see Apéndice D, *Ejemplo de Instrucciones para Informar sobre Fallos*). Notice how they close with an invitation to provide a patch to fix the bug. This is not because such an invitation will lead to a greater patch/report ratio—most users who are capable of fixing bugs already know that a patch would be welcome, and don't need to be told. The invitation's real purpose is to emphasize to all readers, especially those new to the project or new to free software in general, that the project runs on volunteer contributions. In a sense, the project's current developers are no more responsible for fixing the bug than is the person who reported it. This is an important point that many new users will not be familiar with. Once they realize it, they're more likely to help make the fix happen, if not by contributing code then by providing a more thorough reproduction recipe, or by offering to test fixes that other people post. The goal is to make every user realize that there is no *innate* difference between herself and the people who work on the project—that it's a question of how much time and effort one puts in, not a question of who one is.

The admonition against responding angrily does not apply to rude users. Occasionally people post bug reports or complaints that, regardless of their informational content, show a sneering contempt at the project for some failing. Often such people are alternately insulting and flattering, such as the person who posted this to a Subversion mailing list:

> Why is it that after almost 6 days there still aren't any binaries posted for the windows platform?!? It's the same story every time and it's pretty frustrating. Why aren't these things automated so that they could be available immediately?? When you post an "RC" build, I think the idea is that you want users to test the build, but yet you don't provide any way of doing so. Why even have a soak period if you provide no means of testing??

Initial response to this rather inflammatory post was surprisingly restrained: people pointed out that the project had a published policy of not providing official binaries, and said, with varying degrees of annoyance, that he ought to volunteer to produce them himself if they were so important to him. Believe it or not, his next post started with these lines:

> First of all, let me say that I think Subversion is awesome and I really appreciate the efforts of everyone involved. [...]

...and then he went on to berate the project *again* for not providing binaries, while still not volunteering to do anything about it. After that, about 50 people just jumped all over him, and I can't say I really minded. The "zero-tolerance" policy toward rudeness advocated in “Echad a volar la mala educación” in Capítulo 2, *Primeros Pasos* applies to people with whom the project has (or would like to have) a sustained interaction. But when someone makes it clear from the start that he is going to be a fountain of bile, there is no point making him feel welcome.

Such situations are fortunately quite rare, and they are noticeably rarer in projects that make an effort to engage users constructively and courteously from their very first interaction.

Share Management Tasks as Well as Technical Tasks

Share the management burden as well as the technical burden of running the project. As a project becomes more complex, more and more of the work is about managing people and information flow. There is no reason not to share that burden, and sharing it does not necessarily require a top-down hierarchy either—what happens in practice tends to be more of a peer-to-peer network topology than a military-style command structure.

Sometimes management roles are formalized, and sometimes they happen spontaneously. In the Subversion project, we have a patch manager, a translation manager, documentation managers, issue managers (albeit unofficial), and a release manager. Some of these roles we made a conscious decision to initiate, others just happened by themselves; as the project grows, I expect more roles to be added. Below we'll examine these roles, and a couple of others, in detail (except for release manager, which was already covered in “Release manager” and “Dictatorship by Release Owner” earlier in this chapter).

As you read the role descriptions, notice that none of them requires exclusive control over the domain in question. The issue manager does not prevent other people from making changes in the issues database, the FAQ manager does not insist on being the only person to edit the FAQ, and so on. These roles are all about responsibility without monopoly. An important part of each domain manager's job is to notice when other people are working in that domain, and train them to do the things the way the manager does, so that the multiple efforts reinforce rather than conflict. Domain managers should also document the processes by which they do their work, so that when one leaves, someone else can pick up the slack right away.

Sometimes there is a conflict: two or more people want the same role. There is no one right way to handle this. You could suggest that each volunteer post a proposal (an "application") and have all the committers vote on which is best. But this is cumbersome and potentially awkward. I find that a better technique is just to ask the multiple candidates to settle it among themselves. They usually will, and will be more satisfied with the result than if a decision had been imposed on them from the outside.

Patch Manager

In a free software project that receives a lot of patches, keeping track of which patches have arrived and what has been decided about them can be a nightmare, especially if done in a decentralized way. Most patches arrive as posts to the project's development mailing list (though some may appear first in the issue tracker, or on external web sites), and there are a number of different routes a patch can take after arrival.

Sometimes someone reviews the patch, finds problems, and bounces it back to the original author for cleanup. This usually leads to an iterative process—all visible on the mailing list—in which the original author posts revised versions of the patch until the reviewer has nothing more to criticize. It is not always easy to tell when this process is done: if the reviewer commits the patch, then clearly the cycle is complete. But if she does not, it might be because she simply didn't have time, or doesn't have commit access herself and couldn't rope any of the other developers into doing it.

Another frequent response to a patch is a freewheeling discussion, not necessarily about the patch itself, but about whether the concept behind the patch is good. For example, the patch may fix a bug, but the project prefers to fix that bug in another way, as part of solving a more general class of problems. Often this is not known in advance, and it is the patch that stimulates the discovery.

Occasionally, a posted patch is met with utter silence. Usually this is due to no developer having time *at that moment* to review the patch, so each hopes that someone else will do it. Since there's no particular limit to how long each person waits for someone else to pick up the ball, and meanwhile other priorities are always coming up, it's very easy for a patch to slip through the cracks without any single person intending for that to happen. The project might miss out on a useful patch this way, and there are other

harmful side effects as well: it is discouraging to the author, who invested work in the patch, and it makes the project as a whole look a bit out of touch, especially to others considering writing patches.

The patch manager's job is to make sure that patches don't "slip through the cracks." This is done by following every patch through to some sort of stable state. The patch manager watches every mailing list thread that results from a patch posting. If it ends in a commit of the patch, he does nothing. If it goes into a review/revise iteration, ending with a final version of the patch but no commit, he files an issue pointing to the final version, and to the mailing list thread around it, so that there is a permanent record for developers to follow up on later. If the patch addresses an existing issue, he annotates that issue with the relevant information, instead of opening a new issue.

When a patch gets no reaction at all, the patch manager waits a few days, then follows up asking if anyone is going to review it. This usually gets a reaction: a developer may explain that she doesn't think the patch should be applied, and give the reasons why, or she may review it, in which case one of the previously described paths is taken. If there is still no response, the patch manager may or may not file an issue for the patch, at his discretion, but at least the original submitter got *some* reaction.

Having a patch manager has saved the Subversion development team a lot of time and mental energy. Without a designated person to take responsibility, every developer would constantly have to worry "If I don't have time to respond to this patch right now, can I count on someone else doing it? Should I try to keep an eye on it? But if other people are also keeping an eye on it, for the same reasons, then we'd have needlessly duplicated effort." The patch manager removes the second-guessing from the situation. Each developer can make the decision that is right for her at the moment she first sees the patch. If she wants to follow up with a review, she can do that—the patch manager will adjust his behavior accordingly. If she wants to ignore the patch completely, that's fine too; the patch manager will make sure it isn't forgotten.

Because this system works only if people can depend on the patch manager being there without fail, the role should be held formally. In Subversion, we advertised for it on the development and users mailing lists, got several volunteers, and took the first one who replied. When that person had to step down (see “Transitions” later in this chapter), we did the same thing again. We've never tried having multiple people share the role, because of the communications overhead that would be required between them; but perhaps at very high volumes of patch submission, a multiheaded patch manager might make sense.

Translation Manager

In software projects, "translation" can refer to two very different things. It can mean translating the software's documentation into other languages, or it can mean translating the software itself—that is, having the program display errors and help messages in the user's preferred language. Both are complex tasks, but once the right infrastructure is in place, they are largely separable from other development. Because the tasks are similar in some ways, it may make sense (depending on your project) to have a single translation manager handle both, or it may be better to have two different managers.

In the Subversion project, we have one translation manager handle both. He does not actually write the translations himself, of course—he may help out on one or two, but as of this writing, he would need to speak ten languages (twelve counting dialects) in order to work on all of them! Instead, he manages teams of volunteer translators: he helps them coordinate among each other, and he coordinates between the teams and the rest of the project.

Part of the reason the translation manager is necessary is that translators are a different demographic from developers. They sometimes have little or no experience working in a version control repository, or indeed with working as part of a distributed volunteer team at all. But in other respects they are often the best kind of volunteer: people with specific domain knowledge who saw a need and chose to get involved. They are usually willing to learn, and enthusiastic to get to work. All they need is someone to tell them how. The translation manager makes sure that the translations happen in a way that does not interfere unnecessarily with regular development. He also serves as a sort of representative of the translators as a unified body, whenever the developers must be informed of technical changes required to support the translation effort.

Thus, the position's most important skills are diplomatic, not technical. For example, in Subversion we have a policy that all translations should have at least two people working on them, because otherwise there is no way for the text to be reviewed. When a new volunteer shows up offering to translate Subversion to, say, Malagasy, the translation manager has to either hook him up with someone who posted six months ago expressing interest in doing a Malagasy translation, or else politely ask the volunteer to go find *another* Malagasy translator to work with as a partner. Once enough people are available, the manager sets them up with the proper kind of commit access, informs them of the project's conventions (such as how to write log messages), and then keeps an eye out to make sure they adhere to those conventions.

Conversations between the translation manager and the developers, or between the translation manager and translation teams, are usually held in the project's original language—that is, the language from which all the translations are being made. For most free software projects, this is English, but it doesn't matter what it is as long as the project agrees on it. (English is probably best for projects that want to attract a broad international development community, though.)

Conversations *within* a particular translation team usually happen in their shared language, however, and one of the other tasks of the translation manager is to set up a dedicated mailing list for each team. That way the translators can discuss their work freely, without distracting people on the project's main lists, most of whom would not be able to understand the translation language anyway.

Internationalization Versus Localization

Internationalization (*I18N*) and *localization* (*L10N*) both refer to the process of adapting a program to work in linguistic and cultural environments other than the one for which it was originally written. The terms are often treated as interchangeable, but in fact they are not quite the same thing. As http://en.wikipedia.org/wiki/G11n writes:

> The distinction between them is subtle but important: Internationalization is the adaptation of products for *potential* use virtually everywhere, while localization is the addition of special features for use in a *specific* locale.

For example, changing your software to losslessly handle Unicode (http://en.wikipedia.org/wiki/Unicode) text encodings is an internationalization move, since it's not about a particular language, but rather about accepting text from any of a number of languages. On the other hand, making your software print all error messages in Slovenian, when it detects that it is running in a Slovenian environment, is a localization move.

Thus, the translation manager's task is principally about localization, not internationalization.

Documentation Manager

Keeping software documentation up-to-date is a never-ending task. Every new feature or enhancement that goes into the code has the potential to cause a change in the documentation. Also, once the project's documentation reaches a certain level of completeness, you will find that a lot of the patches people send in are for the documentation, not for the code. This is because there are many more people competent to fix bugs in prose than in code: all users are readers, but only a few are programmers.

Documentation patches are usually much easier to review and apply than code patches. There is little or no testing to be done, and the quality of the change can be evaluated quickly just by review. Since the quantity is high, but the review burden fairly low, the ratio of administrative overhead to productive work is greater for documentation patches than for code patches. Furthermore, most of the patches will probably need some sort of adjustment, in order to maintain a consistent authorial voice in the documentation. In many cases, patches will overlap with or affect other patches, and need to be adjusted with respect to each other before being committed.

Given the exigencies of handling documentation patches, and the fact that the code base needs to be constantly monitored so the documentation can be kept up-to-date, it makes sense to have one person, or a small team, dedicated to the task. They can keep a record of exactly where and how the documentation lags behind the software, and they can have practiced procedures for handling large quantities of patches in an integrated way.

Of course, this does not preclude other people in the project from applying documentation patches on the fly, especially small ones, as time permits. And the same patch manager (see “Patch Manager” earlier in this chapter) can track both code and documentation patches, filing them wherever the development and documentation teams want them, respectively. (If the total quantity of patches ever exceeds one human's capacity to track, though, switching to separate patch managers for code and documentation is probably a good first step.) The point of a documentation team is to have people who think of themselves as responsible for keeping the documentation organized, up-to-date, and consistent with itself. In practice, this means knowing the documentation intimately, watching the code base, watching the changes *others* commit to the documentation, watching for incoming documentation patches, and using all these information sources to do whatever is necessary to keep the documentation healthy.

Issue Manager

The number of issues in a project's bug tracker grows in proportion to the number of people using the software. Therefore, even as you fix bugs and ship an increasingly robust program, you should still expect the number of open issues to grow essentially without bound. The frequency of duplicate issues will also increase, as will the frequency of incomplete or poorly described issues.

Issue managers help alleviate these problems by watching what goes into the database, and periodically sweeping through it looking for specific problems. Their most common action is probably to fix up incoming issues, either because the reporter didn't set some of the form fields correctly, or because the issue is a duplicate of one already in the database. Obviously, the more familiar an issue manager is with the project's bug database, the more efficiently she will be able to detect duplicate issues—this is one of the main advantages of having a few people specialize in the bug database, instead of everyone trying to do it *ad hoc*. When the group tries to do it in a decentralized manner, no single individual acquires a deep expertise in the content of the database.

Issue managers can also help map between issues and individual developers. When there are a lot of bug reports coming in, not every developer may read the issue notification mailing list with equal attention. However, if someone who knows the development team is keeping an eye on all incoming issues, then she can discreetly direct certain developers' attention to specific bugs when appropriate. Of course, this has to be done with a sensitivity to everything else going on in development, and to the recipient's desires and temperament. Therefore, it is often best for issue managers to be developers themselves.

Depending on how your project uses the issue tracker, issue managers can also shape the database to reflect the project's priorities. For example, in Subversion we schedule issues into specific future releases, so that when someone asks "When will bug X be fixed?" we can say "Two releases from now," even if we can't give an exact date. The releases are represented in the issue tracker as target milestones, a field available in IssueZilla.[3] As a rule, every Subversion release has one major new feature and a list of specific bug fixes. We assign the appropriate target milestone to all the issues planned for that release (including the new feature—it gets an issue too), so that people can view the bug database through the lens of release scheduling. These targets rarely remain static, however. As new bugs come in, priorities sometimes get shifted around, and issues must be moved from one milestone to another so that each release remains manageable. This, again, is best done by people who have an overall sense of what's in the database, and how various issues relate to each other.

Another thing issue managers do is notice when issues become obsolete. Sometimes a bug is fixed accidentally as part of an unrelated change to the software, or sometimes the project changes its mind about whether a certain behavior is buggy. Finding obsoleted issues is not easy: the only way to do it systema-

[3] IssueZilla is the issue tracker we use; it is a descendant of BugZilla.

tically is by making a sweep over all the issues in the database. Full sweeps become less and less feasible over time, however, as the number of issues grows. After a certain point, the only way to keep the database sane is to use a divide-and-conquer approach: categorize issues immediately on arrival and direct them to the appropriate developer's or team's attention. The recipient then takes charge of the issue for the rest of its lifetime, shepherding it to resolution or oblivion as necessary. When the database is that large, the issue manager becomes more of an overall coordinator, spending less time looking at each issue herself and more time getting it into the right person's hands.

FAQ Manager

FAQ maintenance is a surprisingly difficult problem. Unlike most other documents in a project, whose content is planned out in advance by the authors, a FAQ is a wholly reactive document (see Manteniendo un FAQ (Preguntas Más Frecuentes)). No matter how big it gets, you still never know what the next addition will be. And because it is always added to piecemeal, it is very easy for the document as a whole to become incoherent and disorganized, and even to contain duplicate or semi-duplicate entries. Even when it does not have any obvious problems like that, there are often unnoticed interdependencies between items—links that should be made but aren't—because the related items were added a year apart.

The role of a FAQ manager is twofold. First, she maintains the overall quality of the FAQ by staying familiar with at least the topics of all the questions in it, so that when people add new items that are duplicates of, or related to, existing items, the appropriate adjustments can be made. Second, she watches the project mailing lists and other forums for recurring problems or questions, and to write new FAQ entries based on this input. This latter task can be quite complex: one must be able to follow a thread, recognize the core questions raised in it, post a proposed FAQ entry, incorporate comments from others (since it's impossible for the FAQ manager to be an expert in every topic covered by the FAQ), and sense when the process is finished so the item can at last be added.

The FAQ manager usually also becomes the default expert in FAQ formatting. There are a lot of little details involved in keeping a FAQ in shape (see “Treat all resources like archives” in Capítulo 6, *Communications*); when random people edit the FAQ, they will sometimes forget some of these details. That's okay, as long as the FAQ manager is there to clean up after them.

Various free software is available to help with the process of FAQ maintenance. It's fine to use it, as long as it doesn't compromise the quality of the FAQ, but beware of over-automation. Some projects try to fully automate the process of FAQ maintenance, allowing everyone to contribute and edit FAQ items in a manner similar to a wiki (see “Wikis” in Capítulo 3, *Infraestructura Técnica*). I've seen this happen particularly with Faq-O-Matic (http://faqomatic.sourceforge.net/), though it may be that the cases I saw were simply abuses that went beyond what Faq-O-Matic was originally intended for. In any case, while complete decentralization of FAQ maintenance does reduce the workload for the project, it also results in a poorer FAQ. There's no one person with a broad view of the entire FAQ, no one to notice when certain items need updating or become obsolete entirely, and no one keeping watch for interdependencies between items. The result is a FAQ that often fails to provide users what they were looking for, and in the worst cases misleads them. Use whatever tools you need to to maintain your project's FAQ, but never let the convenience of the tools seduce you into compromising the quality of the FAQ.

See Sean Michael Kerner's article, *The FAQs on FAQs*, at http://osdir.com/Article1722.phtml, for descriptions and evaluations of open source FAQ maintenance tools.

Transitions

From time to time, a volunteer in a position of ongoing responsibility (e.g., patch manager, translation manager, etc.) will become unable to perform the duties of the position. It may be because the job turned out to be more work than he anticipated, or it may be due to completely external factors: marriage, a new baby, a new employer, or whatever.

When a volunteer gets swamped like this, he usually doesn't notice it right away. It happens by slow de-

grees, and there's no point at which he consciously realizes that he can no longer fulfill the duties of the role. Instead, the rest of the project just doesn't hear much from him for a while. Then there will suddenly be a flurry of activity, as he feels guilty for neglecting the project for so long and sets aside a night to catch up. Then you won't hear from him for a while longer, and then there might or might not be another flurry. But there's rarely an unsolicited formal resignation. The volunteer was doing the job in his spare time, so resigning would mean openly acknowledging to himself that his spare time is permanently reduced. People are often reluctant to do that.

Therefore, it's up to you and the others in the project to notice what's happening—or rather, not happening—and to ask the volunteer what's going on. The inquiry should be friendly and 100% guilt-free. Your purpose is to find out a piece of information, not to make the person feel bad. Generally, the inquiry should be visible to the rest of the project, but if you know of some special reason why a private inquiry would be better, that's fine too. The main reason to do it publicly is so that if the volunteer responds by saying that he won't be able to do the job anymore, there's a context established for your *next* public post: a request for a new volunteer to fill that role.

Sometimes, a volunteer is unable to do the job he's taken on, but is either unaware or unwilling to admit that fact. Of course, anyone may have trouble at first, especially if the responsibility is complex. However, if someone just isn't working out in the task he's taken on, even after everyone else has given all the help and suggestions they can, then the only solution is for him to step aside and let someone new have a try. And if the person doesn't see this himself, he'll need to be told. There's basically only one way to handle this, I think, but it's a multistep process and each step is important.

First, make sure you're not crazy. Privately talk to others in the project to see if they agree that the problem is as serious as you think it is. Even if you're already positive, this serves the purpose of letting others know that you're considering asking the person to step aside. Usually no one will object to that—they'll just be happy you're taking on the awkward task, so they don't have to!

Next, *privately* contact the volunteer in question and tell him, kindly but directly, about the problems you see. Be specific, giving as many examples as possible. Make sure to point out how people had tried to help, but that the problems persisted without improving. You should expect this email to take a long time to write, but with this sort of message, if you don't back up what you're saying, you shouldn't say it at all. Say that you would like to find a new volunteer to fill the role, but also point out that there are many other ways to contribute to the project. At this stage, don't say that you've talked to others about it; nobody likes to be told that people were conspiring behind his back.

There are a few different ways things can go after that. The most likely reaction is that he'll agree with you, or at any rate not want to argue, and be willing to step down. In that case, suggest that he make the announcement himself, and then you can follow up with a post seeking a replacement.

Or, he may agree that there have been problems, but ask for a little more time (or for one more chance, in the case of discrete-task roles like release manager). How you react to that is a judgement call, but whatever you do, don't agree to it just because you feel like you can't refuse such a reasonable request. That would prolong the agony, not lessen it. There is often a very good reason to refuse the request, namely, that there have already been plenty of chances, and that's how things got to where they are now. Here's how I put it in a mail to someone who was filling the release manager role but was not really suited for it:

```
> If you wish to replace me with some one else, I will gracefully
> pass on the role to who comes next.  I have one request, which
> I hope is not unreasonable.  I would like to attempt one more
>  release in an effort to prove myself.

I totally understand the desire (been there myself!), but in
this case, we shouldn't do the "one more try" thing.

This isn't the first or second release, it's the sixth or
seventh... And for all of those, I know you've been dissatisfied
```

```
with the results too (because we've talked about it before).  So
we've effectively already been down the one-more-try route.
Eventually, one of the tries has to be the last one... I think
[this past release] should be it.
```

In the worst case, the volunteer may disagree outright. Then you have to accept that things are going to be awkward and plow ahead anyway. Now is the time to say that you talked to other people about it (but still don't say who until you have their permission, since those conversations were confidential), and that you don't think it's good for the project to continue as things are. Be insistent, but never threatening. Keep in mind that with most roles, the transition really happens the moment someone new starts doing the job, *not* the moment the old person stops doing it. For example, if the contention is over the role of, say, issue manager, at any point you and other influential people in the project can solicit for a new issue manager. It's not actually necessary that the person who was previously doing it stop doing it, as long as he does not sabotage (deliberately or otherwise) the efforts of the new volunteer.

Which leads to a tempting thought: instead of asking the person to resign, why not just frame it as a matter of getting him some help? Why not just have two issue managers, or patch managers, or whatever the role is?

Although that may sound nice in theory, it is generally not a good idea. What makes the manager roles work—what makes them useful, in fact—is their centralization. Those things that can be done in a decentralized fashion are usually already being done that way. Having two people fill one managerial role introduces communications overhead between those two people, as well as the potential for slippery displacement of responsibility ("I thought you brought the first aid kit!" "Me? No, I thought *you* brought the first aid kit!"). Of course, there are exceptions. Sometimes two people work extremely well together, or the nature of the role is such that it can easily be spread across multiple people. But these are not likely to be of much use when you see someone flailing in a role he is not suited for. If he'd appreciated the problem in the first place, he would have sought such help before now. In any case, it would be disrespectful to let someone waste time continuing to do a job no one will pay attention to.

The most important factor in asking someone to step down is privacy: giving him the space to make a decision without feeling like others are watching and waiting. I once made the mistake—an obvious mistake, in retrospect—of mailing all three parties at once in order to ask Subversion's release manager to step aside in favor of two other volunteers. I'd already talked to the two new people privately, and knew that they were willing to take on the responsibility. So I thought, naïvely and somewhat insensitively, that I'd save some time and hassle by sending one mail to all of them to initiate the transition. I assumed that the current release manager was already fully aware of the problems and would see the reasonableness of my point immediately.

I was wrong. The current release manager was very offended, and rightly so. It's one thing to be asked to hand off the job; it's another thing to be asked that *in front of* the people you'll hand it off to. Once I got it through my head why he was offended, I apologized. He eventually did step aside gracefully, and continues to be involved with the project today. But his feelings were hurt, and needless to say, this was not the most auspicious of beginnings for the new volunteers either.

Committers

As the only formally distinct class of people found in all open source projects, committers deserve special attention here. Committers are an unavoidable concession to discrimination in a system which is otherwise as non-discriminatory as possible. But "discrimination" is not meant as a pejorative here. The function committers perform is utterly necessary, and I do not think a project could succeed without it. Quality control requires, well, control. There are always many people who feel competent to make changes to a program, and some smaller number who actually are. The project cannot rely on people's own judgement; it must impose standards and grant commit access only to those who meet them[4]. On the other hand, having people who can commit changes directly working side-by-side with people who cannot

sets up an obvious power dynamic. That dynamic must be managed so that it does not harm the project.

In “¿Quién Vota?” in Capítulo 4, *Infraestructura Social y Política*, we already discussed the mechanics of considering new committers. Here we will look at the standards by which potential new committers should be judged, and how this process should be presented to the larger community.

Choosing Committers

In the Subversion project, we choose committers primarily on the Hippocratic Principle: *first, do no harm*. Our main criterion is not technical skill or even knowledge of the code, but merely that the committer show good judgement. Judgement can mean simply knowing what not to take on. A person might post only small patches, fixing fairly simple problems in the code; but if the patches apply cleanly, do not contain bugs, and are mostly in accord with the project's log message and coding conventions, and there are enough patches to show a clear pattern, then an existing committer will usually propose that person for commit access. If at least three people say yes, and no one objects, then the offer is made. True, we might have no evidence that the person is able to solve complex problems in all areas of the code base, but that does not matter: the person has made it clear that he is capable of at least judging his own abilities. Technical skills can be learned (and taught), but judgement, for the most part, cannot. Therefore, it is the one thing you want to make sure a person has before you give him commit access.

When a new committer proposal does provoke a discussion, it is usually not about technical ability, but rather about the person's behavior on the mailing lists or in IRC. Sometimes someone shows technical skill and an ability to work within the project's formal guidelines, yet is also consistently belligerent or uncooperative in public forums. That's a serious concern; if the person doesn't seem to shape up over time, even in response to hints, then we won't add him as a committer no matter how skilled he is. In a volunteer group, social skills, or the ability to "play well in the sandbox", are as important as raw technical ability. Because everything is under version control, the penalty for adding a committer you shouldn't have is not so much the problems it could cause in the code (review would spot those quickly anyway), but that it might eventually force the project to revoke the person's commit access—an action that is never pleasant and can sometimes be confrontational.

Many projects insist that the potential committer demonstrate a certain level of technical expertise and persistence, by submitting some number of nontrivial patches—that is, not only do these projects want to know that the person will do no harm, they want to know that she is likely to do good across the code base. This is fine, but be careful that it doesn't start to turn committership into a matter of membership in an exclusive club. The question to keep in everyone's mind should be "What will bring the best results for the code?" not "Will we devalue the social status associated with committership by admitting this person?" The point of commit access is not to reinforce people's self-worth, it's to allow good changes to enter the code with a minimum of fuss. If you have 100 committers, 10 of whom make large changes on a regular basis, and the other 90 of whom just fix typos and small bugs a few times a year, that's still better than having only the 10.

Revoking Commit Access

The first thing to be said about revoking commit access is: try not to be in that situation in the first place. Depending on whose access is being revoked, and why, the discussions around such an action can be very divisive. Even when not divisive, they will be a time-consuming distraction from productive work.

However, if you must do it, the discussion should be had privately among the same people who would be in a position to vote for *granting* that person whatever flavor of commit access they currently have. The person herself should not be included. This contradicts the usual injunction against secrecy, but in this case it's necessary. First, no one would be able to speak freely otherwise. Second, if the motion fails,

[4]Note that the commit access means something a bit different in decentralized version control systems, where anyone can set up a repository that is linked into the project, and give themselves commit access to that repository. Nevertheless, the *concept* of commit access still applies: "commit access" is shorthand for "the right to make changes to the code that will ship in the group's next release of the software." In centralized version control systems, this means having direct commit access; in decentralized ones, it means having one's changes pulled into the main distribution by default. It is the same idea either way; the mechanics by which it is realized are not terribly important.

you don't necessarily want the person to know it was ever considered, because that could open up questions ("Who was on my side? Who was against me?") that lead to the worst sort of factionalism. In certain rare circumstances, the group may want someone to know that revocation of commit access is or was being considered, as a warning, but this openness should be a decision the group makes. No one should ever, on her own initiative, reveal information from a discussion and ballot that others assumed were secret.

Once someone's access is revoked, that fact is unavoidably public (see “Avoid Mystery” later in this chapter), so try to be as tactful as you can in how it is presented to the outside world.

Partial Commit Access

Some projects offer gradations of commit access. For example, there might be contributors whose commit access gives them free rein in the documentation, but who do not commit to the code itself. Common areas for partial commit access include documentation, translations, binding code to other programming languages, specification files for packaging (e.g., RedHat RPM spec files, etc.), and other places where a mistake will not result in a problem for the core project.

Since commit access is not only about committing, but about being part of an electorate (see “¿Quién Vota?” in Capítulo 4, *Infraestructura Social y Política*), the question naturally arises: what can the partial committers vote on? There is no one right answer; it depends on what sorts of partial commit domains your project has. In Subversion we've kept things fairly simple: a partial committer can vote on matters confined exclusively to that committer's domain, and not on anything else. Importantly, we do have a mechanism for casting advisory votes (essentially, the committer writes "+0" or "+1 (non-binding)" instead of just "+1" on the ballot). There's no reason to silence people entirely just because their vote isn't formally binding.

Full committers can vote on anything, just as they can commit anywhere, and only full committers vote on adding new committers of any kind. In practice, though, the ability to add new partial committers is usually delegated: any full committer can "sponsor" a new partial committer, and partial committers in a domain can often essentially choose new committers for that same domain (this is especially helpful in making translation work run smoothly).

Your project may need a slightly different arrangement, depending on the nature of the work, but the same general principles apply to all projects. Each committer should be able to vote on matters that fall within the scope of her commit access, and not on matters outside that, and votes on procedural questions should default to the full committers, unless there's some reason (as decided by the full committers) to widen the electorate.

Regarding enforcement of partial commit access: it's often best *not* to have the version control system enforce partial commit domains, even if it can. See “Autorizaciones” in Capítulo 3, *Infraestructura Técnica* for the reasons why.

Dormant Committers

Some projects automatically remove people's commit access if they go a certain amount of time (say, a year) without committing anything. I think this is usually unhelpful and even counterproductive, for two reasons.

First, it may tempt some people into committing acceptable but unnecessary changes, just to prevent their commit access from expiring. Second, it doesn't really serve any purpose. If the main criterion for granting commit access is good judgement, then why assume someone's judgement would deteriorate just because he's away from the project for a while? Even if he completely vanishes for years, not looking at the code or following development discussions, when he reappears he'll *know* how out of touch he is, and act accordingly. You trusted his judgement before, so why not trust it always? If high school diplomas do not expire, then commit access certainly shouldn't.

Sometimes a committer may ask to be removed, or to be explicitly marked as dormant in the list of com-

mitters (see “Avoid Mystery” below for more about that list). In these cases, the project should accede to the person's wishes, of course.

Avoid Mystery

Although the discussions around adding any particular new committer must be confidential, the rules and procedures themselves need not be secret. In fact, it's best to publish them, so people realize that the committers are not some mysterious Star Chamber, closed off to mere mortals, but that anyone can join simply by posting good patches and knowing how to handle herself in the community. In the Subversion project, we put this information right in the developer guidelines document, since the people most likely to be interested in how commit access is granted are those thinking of contributing code to the project.

In addition to publishing the procedures, publish the actual *list* of committers. The traditional place for this is a file called `MAINTAINERS` or `COMMITTERS` in the top level of the project's source code tree. It should list all the full committers first, followed by the various partial commit domains and the members of each domain. Each person should be listed by name and email address, though the address can be encoded to prevent spam (see “Ocultar las direcciones en los archivos” in Capítulo 3, *Infraestructura Técnica*) if the person prefers that.

Since the distinction between full commit and partial commit access is obvious and well defined, it is proper for the list to make that distinction too. Beyond that, the list should not try to indicate the informal distinctions that inevitably arise in a project, such as who is particularly influential and how. It is a public record, not an acknowledgments file. List committers either in alphabetical order, or in the order in which they arrived.

Credit

Credit is the primary currency of the free software world. Whatever people may say about their motivations for participating in a project, I don't know any developers who would be happy doing all their work anonymously, or under someone else's name. There are tangible reasons for this: one's reputation in a project roughly governs how much influence one has, and participation in an open source project can also indirectly have monetary value, because some employers now look for it on resumés. There are also intangible reasons, perhaps even more powerful: people simply want to be appreciated, and instinctively look for signs that their work was recognized by others. The promise of credit is therefore one of best motivators the project has. When small contributions are acknowledged, people come back to do more.

One of the most important features of collaborative development software (see Capítulo 3, *Infraestructura Técnica*) is that it keeps accurate records of who did what, when. Wherever possible, use these existing mechanisms to make sure that credit is distributed accurately, and be specific about the nature of the contribution. Don't just write "Thanks to J. Random <jrandom@example.com>" if instead you can write "Thanks to J. Random <jrandom@example.com> for the bug report and reproduction recipe" in a log message.

In Subversion, we have an informal but consistent policy of crediting the reporter of a bug in either the issue filed, if there is one, or the log message of the commit that fixes the bug, if not. A quick survey of Subversion commit logs up to commit number 14525 shows that about 10% of commits give credit to someone by name and email address, usually the person who reported or analyzed the bug fixed by that commit. Note that this person is different from the developer who actually made the commit, whose name is already recorded automatically by the version control system. Of the 80-odd full and partial committers Subversion has today, 55 were credited in the commit logs (usually multiple times) before they became committers themselves. This does not, of course, prove that being credited was a factor in their continued involvement, but it at least sets up an atmosphere in which people know they can count on their contributions being acknowledged.

It is important to distinguish between routine acknowledgment and special thanks. When discussing a particular piece of code, or some other contribution someone made, it is fine to acknowledge their work.

For example, saying "Daniel's recent changes to the delta code mean we can now implement feature X" simultaneously helps people identify which changes you're talking about and acknowledges Daniel's work. On the other hand, posting solely to thank Daniel for the delta code changes serves no immediate practical purpose. It doesn't add any information, since the version control system and other mechanisms have already recorded the fact that he made the changes. Thanking everyone for everything would be distracting and ultimately information-free, since thanks are effective largely by how much they stand out from the default, background level of favorable comment going on all the time. This does not mean, of course, that you should never thank people. Just make sure to do it in ways that tend not to lead to credit inflation. Following these guidelines will help:

- The more ephemeral the forum, the more free you should feel to express thanks there. For example, thanking someone for their bugfix in passing during an IRC conversation is fine, as is an aside in an email devoted mainly to other topics. But don't post an email solely to thank someone, unless it's for a truly unusual feat. Likewise, don't clutter the project's web pages with expressions of gratitude. Once you start that, it'll never be clear when or where to stop. And *never* put thanks into comments in the code; that would only be a distraction from the primary purpose of comments, which is to help the reader understand the code.

- The less involved someone is in the project, the more appropriate it is to thank her for something she did. This may sound counterintuitive, but it fits with the attitude that expressing thanks is something you do when someone contributes even more than you thought she would. Thus, to constantly thank regular contributors for doing what they normally do would be to express a lower expectation of them than they have of themselves. If anything, you want to aim for the opposite effect!

  There are occasional exceptions to this rule. It's acceptable to thank someone for fulfilling his expected role when that role involves temporary, intense efforts from time to time. The canonical example is the release manager, who goes into high gear around the time of each release, but otherwise lies dormant (dormant as a release manager, in any case—he may also be an active developer, but that's a different matter).

- As with criticism and crediting, gratitude should be specific. Don't thank people just for being great, even if they are. Thank them for something they did that was out of the ordinary, and for bonus points, say exactly why what they did was so great.

In general, there is always a tension between making sure that people's individual contributions are recognized, and making sure the project is a group effort rather than a collection of individual glories. Just remain aware of this tension and try to err on the side of group, and things won't get out of hand.

Forks

In “Forkability” in Capítulo 4, *Infraestructura Social y Política*, we saw how the *potential* to fork has important effects on how projects are governed. But what happens when a fork actually occurs? How should you handle it, and what effects can you expect it to have? Conversely, when should you *initiate* a fork?

The answers depend on what kind of fork it is. Some forks are due to amicable but irreconcilable disagreements about the direction of the project; perhaps more are due to both technical disagreements and interpersonal conflicts. Of course, it's not always possible to tell the difference between the two, as technical arguments may involve personal elements as well. What all forks have in common is that one group of developers (or sometimes even just one developer) has decided that the costs of working with some or all of the others now outweigh the benefits.

Once a project forks, there is no definitive answer to the question of which fork is the "true" or "original" project. People will colloquially talk of fork F coming out of project P, as though P is continuing unchanged down some natural path while F diverges into new territory, but this is, in effect, a declara-

tion of how that particular observer feels about it. It is fundamentally a matter of perception: when a large enough percentage of observers agree, the assertion starts to become true. It is not the case that there is an objective truth from the outset, one that we are only imperfectly able to perceive at first. Rather, the perceptions *are* the objective truth, since ultimately a project—or a fork—is an entity that exists only in people's minds anyway.

If those initiating the fork feel that they are sprouting a new branch off the main project, the perception question is resolved immediately and easily. Everyone, both developers and users, will treat the fork as a new project, with a new name (perhaps based on the old name, but easily distinguishable from it), a separate web site, and a separate philosophy or goal. Things get messier, however, when both sides feel they are the legitimate guardians of the original project and therefore have the right to continue using the original name. If there is some organization with trademark rights to the name, or legal control over the domain or web pages, that usually resolves the issue by fiat: that organization will decide who is the project and who is the fork, because it holds all the cards in a public relations war. Naturally, things rarely get that far: since everyone already knows what the power dynamics are, they will avoid fighting a battle whose outcome is known in advance, and just jump straight to the end.

Fortunately, in most cases there is little doubt as to which is the project and which is the fork, because a fork is, in essence, a vote of confidence. If more than half of the developers are in favor of whatever course the fork proposes to take, usually there is no need to fork—the project can simply go that way itself, unless it is run as a dictatorship with a particularly stubborn dictator. On the other hand, if fewer than half of the developers are in favor, the fork is a clearly minority rebellion, and both courtesy and common sense indicate that it should think of itself as the divergent branch rather than the main line.

Handling a Fork

If someone threatens a fork in your project, keep calm and remember your long-term goals. The mere *existence* of a fork isn't what hurts a project; rather, it's the loss of developers and users. Your real aim, therefore, is not to squelch the fork, but to minimize these harmful effects. You may be mad, you may feel that the fork was unjust and uncalled for, but expressing that publicly can only alienate undecided developers. Instead, don't force people to make exclusive choices, and be as cooperative as is practicable with the fork. To start with, don't remove someone's commit access in your project just because he decided to work on the fork. Work on the fork doesn't mean that person has suddenly lost his competence to work on the original project; committers before should remain committers afterward. Beyond that, you should express your desire to remain as compatible as possible with the fork, and say that you hope developers will port changes between the two whenever appropriate. If you have administrative access to the project's servers, publicly offer the forkers infrastructure help at startup time. For example, offer them a complete, deep-history copy of the version control repository, if there's no other way for them to get it, so that they don't have to start off without historical data (this may not be necessary depending on the version control system). Ask them if there's anything else they need, and provide it if you can. Bend over backward to show that you are not standing in the way, and that you want the fork to succeed or fail on its own merits and nothing else.

The reason to do all this—and do it publicly—is not to actually help the fork, but to persuade developers that your side is a safe bet, by appearing as non-vindictive as possible. In war it sometimes makes sense (strategic sense, if not human sense) to force people to choose sides, but in free software it almost never does. In fact, after a fork some developers often openly work on both projects, and do their best to keep the two compatible. These developers help keep the lines of communication open after the fork. They allow your project to benefit from interesting new features in the fork (yes, the fork may have things you want), and also increase the chances of a merger down the road.

Sometimes a fork becomes so successful that, even though it was regarded even by its own instigators as a fork at the outset, it becomes the version everybody prefers, and eventually supplants the original by popular demand. A famous instance of this was the GCC/EGCS fork. The *GNU Compiler Collection* (*GCC*, formerly the *GNU C Compiler*) is the most popular open source native-code compiler, and also one of the most portable compilers in the world. Due to disagreements between the GCC's official maintainers and Cygnus Software,[5] one of GCC's most active developer groups, Cygnus created a fork of

GCC called *EGCS*. The fork was deliberately non-adversarial: the EGCS developers did not, at any point, try to portray their version of GCC as a new official version. Instead, they concentrated on making EGCS as good as possible, incorporating patches at a faster rate than the official GCC maintainers. EGCS gained in popularity, and eventually some major operating system distributors decided to package EGCS as their default compiler instead of GCC. At this point, it became clear to the GCC maintainers that holding on to the "GCC" name while everyone switched to the EGCS fork would burden everyone with a needless name change, yet do nothing to prevent the switchover. So GCC adopted the EGCS codebase, and there is once again a single GCC, but greatly improved because of the fork.

This example shows why you cannot always regard a fork as an unadulteratedly bad thing. A fork may be painful and unwelcome at the time, but you cannot necessarily know whether it will succeed. Therefore, you and the rest of the project should keep an eye on it, and be prepared not only to absorb features and code where possible, but in the most extreme case to even join the fork if it gains the bulk of the project's mindshare. Of course, you will often be able to predict a fork's likelihood of success by seeing who joins it. If the fork is started by the project's biggest complainer and joined by a handful of disgruntled developers who weren't behaving constructively anyway, they've essentially solved a problem for you by forking, and you probably don't need to worry about the fork taking momentum away from the original project. But if you see influential and respected developers supporting the fork, you should ask yourself why. Perhaps the project was being overly restrictive, and the best solution is to adopt into the mainline project some or all of the actions contemplated by the fork—in essence, to avoid the fork by becoming it.

Initiating a Fork

All the advice here assumes that you are forking as a last resort. Exhaust all other possibilities before starting a fork. Forking almost always means losing developers, with only an uncertain promise of gaining new ones later. It also means starting out with competition for users' attention: everyone who's about to download the software has to ask themselves: "Hmm, do I want that one or the other one?" Whichever one you are, the situation is messy, because a question has been introduced that wasn't there before. Some people maintain that forks are healthy for the software ecosystem as a whole, by a standard natural selection argument: the fittest will survive, which means that, in the end, everyone gets better software. This may be true from the ecosystem's point of view, but it's not true from the point of view of any individual project. Most forks do not succeed, and most projects are not happy to be forked.

A corollary is that you should not use the threat of a fork as an extremist debating technique—"Do things my way or I'll fork the project!"—because everyone is aware that a fork that fails to attract developers away from the original project is unlikely to survive long. All observers—not just developers, but users and operating system packagers too—will make their own judgement about which side to choose. You should therefore appear extremely reluctant to fork, so that if you finally do it, you can credibly claim it was the only route left.

Do not neglect to take *all* factors into account in evaluating the potential success of your fork. For example, if many of the developers on a project have the same employer, then even if they are disgruntled and privately in favor of a fork, they are unlikely to say so out loud if they know that their employer is against it. Many free software programmers like to think that having a free license on the code means no one company can dominate development. It is true that the license is, in an ultimate sense, a guarantor of freedom—if others want badly enough to fork the project, and have the resources to do so, they can. But in practice, some projects' development teams are mostly funded by one entity, and there is no point pretending that that entity's support doesn't matter. If it is opposed to the fork, its developers are unlikely to take part, even if they secretly want to.

If you still conclude that you must fork, line up support privately first, then announce the fork in a non-hostile tone. Even if you are angry at, or disappointed with, the current maintainers, don't say that in the message. Just dispassionately state what led you to the decision to fork, and that you mean no ill will toward the project from which you're forking. Assuming that you do consider it a fork (as opposed to an emergency preservation of the original project), emphasize that you're forking the code and not the na-

[5]Now part of RedHat (http://www.redhat.com/).

me, and choose a name that does not conflict with the project's name. You can use a name that contains or refers to the original name, as long as it does not open the door to identity confusion. Of course it's fine to explain prominently on the fork's home page that it descends from the original program, and even that it hopes to supplant it. Just don't make users' lives harder by forcing them to untangle an identity dispute.

Finally, you can get things started on the right foot by automatically granting all committers of the original project commit access to the fork, including even those who openly disagreed with the need for a fork. Even if they never use the access, your message is clear: there are disagreements here, but no enemies, and you welcome code contributions from any competent source.

Capítulo 9. Licencias, Copyrights y Patentes

La licencia que elijas probablemente no tendrá un gran impacto en la adopción de tu proyecto, siempre que sea software libre. Los usuarios generalmente eligen software basándose en la calidad y las funcionalidades, no en los detalles de la licencia. No obstante, necesitas una comprensión básica de las implicaciones de las licencias libres, tanto para asegurar que la licencia del proyecto es compatible con sus objetivos, como para discutir sobre las decisiones acerca de la licencia con otros. Por favor, ten en cuenta que no soy abogado, y nada contenido en este capítulo debe ser tenido en cuenta como advertencia legal. Para ello, necesitarás contratar un abogado o serlo.

Terminología

En cualquier discusión acerca de las licencias de software libre, lo primero que encuentras es que parece que hay diferentes nomenclaturas para los mismos conceptos: *software libre (free software)*, *software de fuentes abiertas (open source)*, *FOSS*, *F/OSS*, y *FLOSS*. Empecemos por ordenar estos términos, además de algunos otros.

Software libre (free software)
: Software que puede ser compartido y modificado con libertad, incluyendo el código fuente. El término fue acuñado por Richard Stallman, quien lo utilizó en la GNU General Public License (GPL), y quien fundó la Free Software Foundation (http://www.fsf.org/) para promocionar el concepto.

  Aunque "software libre" cubre casi exactamente el mismo software que "software de fuentes abiertas", la FSF, entre otros, prefiere el término anterior porque hace hincapié en la idea de libertad, y el concepto de libre distribución del software principalmente como un movimiento social y no técnico. La FSF admite que el término es ambiguo—puede significar "free" como "de coste cero", en lugar de "free" como "libertad"— pero considera que aún es el mejor término, y que las otras alternativas en inglés tienen sus propias ambigüedades (a lo largo de este libro, "free" se utiliza con el sentido de libertad, y no con el concepto de gratuito).

Software de fuentes abiertas (open source)
: Software libre bajo otro nombre. Pero la diferencia en el nombre refleja una importante diferencia filosófica: "open source" fue acuñado por la Open Source Initiative (http://www.opensource.org/) como una alternativa deliberada a "free software", con el fin de hacer este tipo de software una opción más apetecible para empresas, mediante su presentación como una metodología de desarrollo en vez de un movimiento político. También podrían haber querido sobreponerse a otro estigma: lo "gratuíto" es de baja calidad.

  Mientras que cualquier licencia que sea "free software" es también "open source", y viceversa (con unas pocas excepciones), la gente tiende a elegir un término y aferrarse a él. En general, aquellos que prefieren "software libre" suelen tener una postura más filosófica o moral sobre el tema, mientras que los que prefieren "open source" o no lo ven como un asunto de libertad, o no están interesados en publicitar el hecho de ese modo. Veáse “#Libre# vs #Abierto#” en Capítulo 1, *Introducción* para una historia más detallada de este cisma.

  La Free Software Foundation tiene una excelente —carente de objetividad, pero con matices y muy justa —exégesis de los dos términos, en http://www.fsf.org/licensing/essays/free-software-for-freedom.html. La visión de la Open Source Initiative al respecto está en dos páginas: http://www.opensource.org/advocacy/case_for_hackers.php#marketing y http://www.opensource.org/advocacy/free-notfree.php.

FOSS, F/OSS, FLOSS
: Donde caben dos, caben tres, y eso es exactamente lo que está ocurriendo con los términos para el software libre. El mundo académico, quizá buscando precisión e incluso por encima de la elegancia, parece haber establecido FOSS, o a veces F/OSS, significando "Free / Open Source Software". Otra variantes ganando adeptos es FLOSS, que quiere decir "Free / Libre Open Source Software" (*libre* es común en varios idiomas y no sufre de la ambigüedad de "free"; veáse http://en.wikipedia.org/wiki/FLOSS para más detalles).

  Todos estos términos significan esencialmente lo mismo: software que puede ser modificado y redistribuido por cualquiera, algunas veces—pero no siempre— con el requisito de que los trabajos derivados deben ser distribuíbles libremente bajo los mismos términos.

Conforme con las DFSG
: Conforme con las Directrices de Software Libre de Debian (*Debian Free Software Guidelines* ,http://www.debian.org/social_contract#guidelines). Ésta es una prueba ampliamente usada para comprobar si una licencia dada es verdaderamente software libre. La misión del proyecto Debian es mantener un sistema operativo totalmente libre, de modo que cualquiera que lo instale nunca dude de que tiene el derecho a modificar y redistribuir cualquier parte del sistema. Las Directrices de Software Libre de Debian son los requisitos que la licencia de un paquete software debe cumplir para poder ser incluído en Debian. Debido a que el proyecto Debian invirtió gran cantidad de tiempo pensando en cómo realizar una prueba, las directrices que surgieron han demostrado ser muy robustas (veáse http://en.wikipedia.org/wiki/DFSG), hasta donde yo sé, ninguna objección seria se ha lanzado ni por la Free Software Foundation ni por la Open Source Initiative. Si sabes que una licencia es conforme con las DFSG, sabes que garantiza todas las libertades importantes (tales como la posibilidad de un *fork* incluso en contra de los deseos del autor) requeridas para mantener la dinámica de un proyecto libre. Todas las licencias tratadas en este capítulo son conformes a las DFSG.

Aprobadas por OSI
: Aprobadas por la Open Source Initiative. Este es otra prueba ampliamente usada para comprobar si una licencia cumple todas las libertades necesarias. La definición de la OSI del software libre está basada en las DFSG, y una licencia que se ajusta a una prueba casi siempre se ajusta a la otra. A través de los años han habido contadas excepciones, pero sólo relativas a un nicho de licencias de ninguna relevancia aquí. Al contrario que el proyecto Debian, la OSI mantiene una lista de todas las licencias que ha aprobado, en http://www.opensource.org/licenses/, por lo que ser aprobada por OSI es un estado sin ambigüedades: una licencia está o no está en la lista.

  La Free Software Foundation también mantiene una lista de licencias en http://www.fsf.org/licensing/licenses/license-list.html. La FSF clasifica las licencias no sólo por su libertad, sino también por su compatibilidad con la GNU General Public License. La compatibilidad con la GPL es un asunto important, que será tratado “La GPL y compatibilidad entre licencias” más tarde en este capítulo.

Propietario, Código cerrado
: Lo opuesto al software libre. Significa que el software se distribuye bajo términos tradicionales, con licencias basadas en derechos de autor, donde los usuarios pagan por copia, o bajo otros términos cualesquiera suficientemente restrictivos para prevenir la dinámica del software libre. Incluso el software distribuido gratuítamente puede ser propietario, si su licencia no permite la libre redistribución y derecho a modificación.

  Generalmente, software "proprietario" y software "de código cerrado" son sinónimos. Sin embargo, "código cerrado" adicionalmente implica que el código fuente no puede ser visto. Debido a que el código no puede verse en la mayoría del software privativo, esta diferencia generalmente no afecta. Sin embargo, ocasionalmente alguien publica software propietario bajo una licencia que permite a otros ver el código. Equivocadamente, algunas veces llaman a esto "código abierto" o "cercano al código abierto", etc., pero es engañoso. La *visibilidad* del código no es el problema, el tema importante es que estás autorizado a hacer con él. Así, la diferencia entre el código cerrado o propietario es en gran parte intrascendente, y ambos términos pueden tratarse como sinónimos.

Algunas veces *comercial* es usado como sinónimo de "propietario", pero hablando con propiedad, no es lo mismo. El software libre puede ser software comercial. Después de todo, el software libre puede venderse, siempre que los compradores no estén restringidos a distribuir copias. También puede comercializarse por otras vías, por ejemplo mediante la venta de soporte, servicios y certificaciones. Hoy existen empresas multimillonarias basadas en el software libre, por tanto claramente no es intrínsecamente anticomercial o contrario a las empresas. Por otro lado, *es* anti-proprietario por su naturaleza y ésta es la clave de la diferencia con los modelos tradicionales de licencia por copia.

Dominio público
: Que no tiene derechos de autor, significa que no hay nada que tenga el derecho a restringir la copia de la obra. Estar en el dominio público no es lo mismo que no tener autor. Todo tiene un autor, e incluso si el autor del trabajo o los autores eligen ponerlo en el dominio público, no cambia el hecho de que ellos lo escribieron.

  Cuando un trabajo está en el dominio público, material de éste puede ser incorporado en trabajos con derechos de autor, y entonces *esa copia* del material está cubierta bajo los mismos derechos de autor que el trabajo completo. Pero esto no afecta a la disponibilidad del trabajo original, que permanece en el dominio público. Así, publicar algo en el dominio público es técnicamente un modo de hacerlo "libre", de acuerdo con las directrices de la mayoría de organizaciones certificadoras de software libre. Sin embargo, normalmente hay buenas razones para usar una licencia en vez de publicar algo en el dominio público: incluso con el software libre, ciertas restricciones pueden ser útiles, no sólo para el poseedor de los derechos de autor sino también para los beneficiarios, como aclara la siguiente sección.

copyleft
: Una licencia que utiliza la ley de propiedad intelectual para lograr el resultado contrario al derecho de autor tradicional. Dependiendo de a quién preguntes, esto significa que cualquier licencia que permita las libertades bajo discusión aquí, o, más estrictamente, las licencias que no sólo permiten esas libertades sino que las *imponen*, estipulando que las libertades deben acompañar al trabajo. La Free Software Foundation usa la segunda definición exclusivamente; en otros sitios, es un complemento: mucha gente usa el término del mismo modo que la FSF, pero otros —incluyendo a algunos que escriben en los principales medios de comunicación— tienden a usar la primera definición. Está claro que no todo el mundo que usa el término es consciente de que hay distinciones que deben hacerse.

  El ejemplo clásico de la más limitada, una definición más estricta es la GNU General Public License, que estipula que todo trabajo derivado debe estar también bajo dicha licencia, veáse “La GPL y compatibilidad entre licencias” más tarde en este capítulo para más detalles.

Aspectos de las licencias

Aunque hay muchas licencias distintas de software libre disponibles, en los aspectos importantes, todas dicen lo mismo: que cualquier persona puede modificar el código, que cualquier persona puede redistriburlo tanto en la forma original como modificada, y que los titulares del copyright y los autores no ofrecen garantía alguna (evitar las responsabilidades es especialmente importante teniendo en cuenta que la gente puede ejecutar versiones modificadas incluso sin saberlo). Las diferencias entre las licencias se reducen a unas pocas cuestiones muy recurrentes:

Compatibilidad con licencias propietarias
: Algunas licencias libres permiten que el código que cubren se utilice en software propietario. Esto no afecta a los términos de la licencia del software propietario: sigue siendo propietario como siempre, sólo que contiene código de una fuente no propietaria. La Apache License, X Consortium License, licencias estilo BSD, y licencias estilo MIT son ejemplos de licencias compatibles con licencias propietarias.

Compatibilidad con otras licencias libres
: Muchas licencias libres son compatibles con las demás, significando que el código bajo una licencia puede combinarse con código bajo otra, y el resultado se distribuye bajo otra licencia sin violar los términos de las otras. La mayor excepción a esto es la GNU General Public License, que requiere que cualquier trabajo que utilice código GPL se distribuya bajo la GPL, y sin añadir ninguna restricción más aparte de las de la GPL. La GPL es compatible con algunas licencias libres, pero no con todas. Esto se trata con más detalle “La GPL y compatibilidad entre licencias”más tarde en este capítulo.

Obligación de acreditación
: Algunas licencias libres estipulan que cualquier uso del código que cubren debe ir acompañado de un aviso, cuya colocación y exhibición se suele especificar, dando crédito a los autores o titulares del copyright del código. Estas licencias son generalmente compatibles con licencias propietarias: no necesariamente requieren que el trabajo derivado sea libre, simplemente que se dé crédito al código libre.

Protección de la marca
: Una variante de la obligación de acreditación. Las licencias que protegen las marcas especifican que el nombre del software original (o los titulares del copyright, o su institución, etc.) *no* deben ser utilizados por trabajos derivados sin el permiso previo por escrito. Aunque la obligación de acreditación insiste en que se utilice cierto nombre, y la protección de la marca insiste en que no, son expresiones de un mismo deseo: que la reputación del código original se preserve y transmita, pero no sea empañada por asociación.

Protección de la "integridad artística"
: Algunas licencias (la Artistic License, usada para la implementación más popular del lenguaje de programación Perl, y la licencia TeX de Donald Knuth, por ejemplo) requieren que la modificación y redistribución se haga de modo que se distinga claramente entre la versión original del código y cualquier modificación. Permiten esencialmente las mismas libertades que las demás licencias libres, pero imponen una serie de requisitos que hacen fácil verificar la integridad del código original. Estas licencias no han despertado mucho interés más allá de los programas para los cuáles se hicieron, y no serán tratadas en este capítulo. Se mencionan aquí sólo con el propósito de exhaustividad.

La mayoría de estas disposiciones no son mutuamente excluyentes, y algunas licencias incluyen varias. El hilo común entre ellas es que imponen unas exigencias al beneficiario a cambio del derecho de los destinatarios a usar y/o redistribuir el código. Por ejemplo, algunos proyectos desean que su nombre y reputación se transmita con el código, y esto hace que impongan las claúsulas de crédito o de protección de la marca. Dependiendo de éstas, la carga añadida puede resultar en que algunos usuarios elijan un paquete con una licencia menos exigente.

La GPL y compatibilidad entre licencias

De largo la mayor línea divisoria en cuanto a licencias es entre las licencias compatibles y las incompatibles con las licencias propietarias, es decir, entre la GNU General Public License y todas las demás. Debido a que el objetivo primordial de los autores de la GPL es la promoción del software libre, deliberadamente crearon con sumo cuidado la licencia para hacer imposible la mezcla de código GPL en software propietario. Específicamente, entre las claúsulas de la GPL (veáse http://www.fsf.org/licensing/licenses/gpl.html para obtener el texto completo) están estos dos:

1. Todo trabajo derivado—es decir, todo trabajo que contenga una cantidad no trivial de código GPL —debe ser distribuído bajo la GPL.

2. Ninguna restricción adicional debe ser añadida a la redistribución del trabajo original o de un trabajo derivado (la frase literal es: "Usted no puede imponer ninguna restricción adicional a los beneficiarios en el ejercicio de los derechos otorgados en este documento.").

Con estas claúsulas, la GPL triunfa al hacer la libertad contagiosa. Una vez que un programa se licencia con la GPL, sus términos de distribución son *virales* —se pasan a cualquier sitio donde el código se incorpore, haciendo efectivamente imposible usar código GPL en programas de código cerrado. Sin embargo, estas mismas claúsulas hacen a la GPL incompatible con algunas otras licencias libres. La manera común de que esto ocurra es que la otra licencia impone un requisito —por ejemplo, una claúsula de crédito requiriendo que se mencione a los autores originales de algún modo— que es incompatible con el "Ninguna restricción adicional debe ser añadida..." de la GPL. Desde el punto de vista de la Free Software Foundation, estas consecuencias secundarias son deseables o, al menos, no lamentables. La GPL no sólo mantiene el software libre, sino que hace de tu software un agente para impulsar que *otro* software sea libre también.

La cuestión de si éste es o no un buen modo de promover el software libre es una de las guerras santas más persistentes en Internet (veáse “Evitando las Guerras Santas” en Capítulo 6, *Communications*), y no la vamos a tratar aquí. Lo que importante para nuestro objetivos es que la compatibilidad con la GPL es un problema importante cuando elegimos una licencia. La GPL es de lejos la licencia de software libre más popular; en http://freshmeat.net/stats/#license, tiene un 68%, y la siguiente en el ranking tiene un 6%. Si quieres que tu código se puede emplear libremente con código GPL — y hay mucho código GPL ahí fuera — debes elegir una licencia compatible con la GPL. Algunas de las licencias compatibles con la GPL son también compatibles con software propietario: es decir, código bajo esa licencia puede usarse en un programa GPL, y también en un programa propietario. Por supuesto, los *resultados* de estas mezclas no serán compatibles con la otra, ya que una estará bajo la GPL y otra estará bajo una licencia de código cerrado. Pero esa preocupación se aplica únicamente a las obras derivadas, y no al código que se distribuya en primer lugar.

Afortunadamente, la Free Software Foundation mantiene una lista que muestra qué licencias son compatibles con la GPL y cuáles no, en http://www.gnu.org/licenses/license-list.html. Todas las licencias tratadas en este capítulo están presentes en esa lista, en un lado u en otro.

Eligiendo una licencia

Cuando eliges una licencia para aplicarla a tu proyecto, si es posible usa una licencia existente en vez de crear una nueva. Hay dos razones por la que licencias existentes son una mejor opción:

- Familiaridad. Si utilizas una de las tres o cuatro licencias más populares, la gente no sentirá que debe leer textos legales para utilizar tu código, porque ya lo habrán hecho para esa licencia hace tiempo.
- Calidad. A menos que tengas un equipo de abogados a tu disposición, seguramente no consigas una licencia sólida legalmente. Las licencias mencionadas aquí son producto de mucho trabajo y experiencia. A menos que tu proyecto tenga necesidades poco comunes, es poco probable que lo hagas mejor.

Para aplicar una de estas licencias a tu proyecto, lee “Cómo aplicar una licencia a nuestro software” en Capítulo 2, *Primeros Pasos*.

La MIT / X Window System License

Si tu objetivo es que tu código sea accesible para el mayor número de desarrolladores y trabajos derivados posible, y no te importa que el código se pueda utilizar en software propietario, elije la MIT / X Window System license (llamada así debido a que es la licencia bajo la cual el Massachusetts Institute of Technology lanzó el código original del sistema de ventanas X). El mensaje básico de esta licencia es "Eres libre para usar este código como quieras.". Es compatible con la GNU GPL, y es corta, sencilla, y fácil de entender:

```
Copyright (c) <año> <propietarios del copyright>

Permission is hereby granted, free of charge, to any person obtaining
```

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

(Tomada de http://www.opensource.org/licenses/mit-license.php.)

La GNU General Public License

Si prefieres que tu código no sea utilizado en software propietario, o si, al menos, no te importa si puede o no usarse en éstos, elije la GNU General Public License (http://www.fsf.org/licensing/licenses/gpl.html). La GPL es probablemente la licencia de software libre más utilizada en el mundo a día de hoy; este capacidad de reconocerse en ella es una de las mayores ventajas de la GPL.

Cuando programamos una biblioteca cuyo fin es ser usada en otros programas, considera detenidamente si las restricciones que la GPL impone concuerdan con los objetivos de tu proyecto. En algunos casos —por ejemplo, si intentas desbancar una biblioteca propietaria competidora que realiza la misma función— tiene un sentido más estratégico el licenciar tu código de modo que pueda ser utilizada en software propietario, incluso aunque no lo desearas. La Free Software Foundation preparó una alternativa a la GPL para esas circunstancias: la *GNU Library GPL*, después renombrada como *GNU Lesser GPL* (la mayoría de la gente utiliza directamente el acrónimo *LGPL*, de todos modos). La LGPL tiene restricciones menos estrictas que la GPL, y puede mezclarse más fácilmente con código no libre. Sin embargo, también es más compleja y toma más tiempo entenderla, por lo que si no vas a utilizar la GPL, te recomiendo utilizar una licencia tipo MIT/X.

¿Es la GPL libre o no?

Una consecuencia de elegir la GPL es la posibilidad —pequeña, pero no infinitesimal— de encontrarte a ti o a tu proyecto envueltos en una disputa acerca de si la GPL es o no realmente libre, dado que exige ciertas restricciones en qué puedes hacer con el código—a saber, la restricción de que el código no puede ser redistribuído bajo ninguna otra licencia. Para algunos, la existencia de esta restricción significa que la GPL es "menos libre" que otras licencias más permisivas como la licencia MIT/X. El fin de este argumento generalmente es, por supuesto, que dado que "más libre" debe ser mejor que "menos libre" (después de todo, ¿quién no está a favor de la libertad?), esas licencias son mejores que la GPL.

Este debate es otra guerra santa (veáse “Evitando las Guerras Santas” en Capítulo 6, *Communications*) muy popular. Evita participar en ella, al menos en foros del proyecto. No intentes probar que la GPL es menos libre, tan libre o más libre que otras licencias. En vez de eso, explica las razones específicas por las que elegiste la GPL para tu proyecto. Si fue el conocimiento de la licencia, di eso. Si también fue por las restricciones de licencia libre para trabajos derivados, dilo también, pero niégate a discutir acerca de si esto hace al código más o menos libre. La libertad es un tema complejo, y no tiene mucho sentido hablar de ella si la terminología que va a ser utilizada como alimento para un caballo de acecho.

Dado que esto es un libro y no un hilo de una lista de correo, sin embargo, admitiré que nunca entendí el

argumento "la GPL no es libre". La única restricción que la GPL impone previene a la gente de imponer *mayores* restricciones. Decir que eso significa tener menos libertad siempre me ha parecido como decir que la abolición de la esclavitud reduce la libertad, porque previene que cierta gente posea esclavos.

(Oh, y si te ves inmerso en un debate sobre ello, no aumentes la apuesta haciendo analogías inflamatorias.)

¿Qué tal la licencia BSD?

Una gran cantidad de software libre se distribuye bajo la *BSD license* (o algunas veces una *licencia estilo BSD*). La licencia original BSD fue usada por la Berkeley Software Distribution, en la que la Universidad de California lanzó partes importantes de una implementación de Unix. Esta licencia (el texto exacto puede verse en la sección 2.2.2 de http://www.xfree86.org/3.3.6/COPYRIGHT2.html#6) era similar en esencia a la licencia MIT/X, excepto por una claúsula:

> *Todo material publicitado que mencione características o use este software debe mostrar la siguiente advertencia: "Este producto contiene software desarrollado por la Universidad de California, Lawrence Berkeley Laboratory.*

La presencia de esta claúsula no sólo hace a la BSD incompatible con la GPL, sino que también sienta un peligroso precedente: mientras otras organizaciones pongan claúsulas publicitarias similares en *su* software libre —sustituyendo su propio nombre en lugar de "la Universidad de California, Lawrence Berkeley Laboratory"— los redistribuidores del software se enfrentan a una creciente carga en cuanto a lo que se ven requeridos a mostrar. Afortunadamente, muchos de los proyectos que usaron esta licencia se percataron del problema, y simplemente eliminaron esa claúsula. En 1999, incluso la Universidad de California lo hizo.

El resultado es la licencia BSD revisada, que es simplemente la licencia BSD original sin la claúsula publicitaria. Sin embargo, la historia hace a la expresión "licencia BSD" un poco ambigua: ¿se refiere a la original, o a la versión revisada? Por esto es por lo que prefiero la licencia MIT/X, que es equivalente en esencia, y no sufre ninguna ambigüedad. Sin embargo, quizá hay una razón para preferir la BSD revisada frente a la licencia MIT/X, que es que la BSD incluye esta claúsula:

> *Ni el nombre de la <ORGANIZACIÓN> ni los nombres de sus contribuyentes debe usarse para apoyar o promocionar productos derivados de este software sin permiso previo por escrito explícito.*

No queda claro que sin esa claúsula, un receptor del software podría tener el derecho a usar el nombre del autor, pero esa claúsula borra cualquier tipo de duda. Para organizaciones preocupadas por el control de marcas registradas, por lo tanto, la licencia BSD puede ser preferible a la MIT/X. En general, sin embargo, una licencia de copyright liberal no implica que los receptores tengan ningún derecho a usar sus marcas — las leyes de copyright y las leyes de marcas son dos cosas diferentes.

Si quieres utilizar la licencia BSD revisada, una plantilla está disponible en http://www.opensource.org/licenses/bsd-license.php.

Asignación y propiedad del Copyright

Hay tres maneras de gestionar la propiedad del copyrighy en el software libre contribuído por mucha gente. La primera es ignorar totalmente el asunto del copyright (no recomiendo esta). La segunda es recoger un *acuerdo del contribuyente a la licencia*(*CLA*, de las iniciales de *contributor license agreement*) de cada persona que trabaja en el proyecto, garantizando explícitamente al proyecto el derecho de usar el código de esa persona. Generalmente esto es suficiente para la mayoría de proyectos, y algo positivo está en que en algunas jurisdicciones, los CLAs pueden ser enviados por correo electrónico. El tercer modo es obtener asignaciones de propiedad del copyright de parte de los contribuyentes, de modo que el

proyecto (por ejemplo, alguna entidad legal, generalmente sin ánimo de lucro) es la propietaria de todo el copyright. Esta es la manera más hermética legalmente, pero es también la más onerosa para los contribuyentes, sólo algunos proyectos insisten en ella.

Nótese que incluso bajo la propiedad centralizada del copyright, el código permanece libre, porque las licencias de software libre no dan al propietario del copyrifht el derecho a apropiarse retroactivamente de todas las copias del código. Por lo tanto, incluso si el proyecto, como entidad jurídica, de repente decide cambiar y empezar a distribuir el código bajo una licencia restrictiva, eso no causaría un problema a la comunidad. Los otros desarrolladores podrían simplemente comenzar un fork basado en la última copia libre del código, y continuar como si nada hubiera pasado. Debido a que saben que pueden hacer eso, muchos contribuyentes cooperan cuando se les pide firmar un CLA o asignar el copyright.

No hacer nada

Algunos proyectos nunca recogen CLAs o asignaciones de copyright de sus contribuyentes. En lugar de eso, aceptan el código siempre que quede razonablemente claro que el contribuyente pretendía que fuera incluído en el proyecto.
.

Bajo circunstancias normales, eso es suficiente. Pero de vez en cuando alguien puede decidir demandar por infringimiento del copyright, alegando que ellos son los verdaderos propietarios del código en cuestión y que nunca accedieron a que fuera distribuído por el proyecto bajo una licencia libre. Por ejemplo, el Grupo SCO hizo algo como esto con el proyecto Linux, veáse http://en.wikipedia.org/wiki/SCO-Linux_controversies para más detalles. Cuando esto ocurre, el proyecto no tiene documentación que demuestre que el contribuyente formalmente ha garantizado el derecho a utilizar el código, que puede hacer la defensa legal más complicada.

Contributor License Agreements

CLAs probably offer the best tradeoff between safety and convenience. A CLA is typically an electronic form that a developer fills out and sends in to the project. In many jurisdictions, email submission is enough. A secure digital signature may or may not be required; consult a lawyer to find out what method would be best for your project.

Most projects use two slightly different CLAs, one for individuals, and one for corporate contributors. But in both types, the core language is the same: the contributor grants the project *"...perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, sublicense, and distribute [the] Contributions and such derivative works."* Again, you should have a lawyer approve any CLA, but if you get all those adjectives into it, you're probably fine.

When you request CLAs from contributors, make sure to emphasize that you are *not* asking for actual copyright assignment. In fact, many CLAs start out by reminding the reader of this:

> *This is a license agreement only; it does not transfer copyright ownership and does not change your rights to use your own Contributions for any other purpose.*

Here are some examples:

- Individual contributor CLAs:
  - http://apache.org/licenses/icla.txt
  - http://code.google.com/legal/individual-cla-v1.0.html
- Corporate contributor CLAs:

- http://apache.org/licenses/cla-corporate.txt
- http://code.google.com/legal/corporate-cla-v1.0.html

Transfer of Copyright

Copyright transfer means that the contributor assigns to the project copyright ownership on her contributions. Ideally, this is done on paper and either faxed or snail-mailed to the project.

Some projects insist on full assignment because having a single legal entity own the copyright on the entire code base can be useful if the terms of the open source license ever need to be enforced in court. If no single entity has the right to do it, all the contributors might have to cooperate, but some might not have time or even be reachable when the issue arises.

Different organizations apply different amounts of rigor to the task of collecting assignments. Some simply get an informal statement from a contributor on a public list mailing list—something to the effect of "I hereby assign copyright in this code to the project, to be licensed under the same terms as the rest of the code." At least one lawyer I've talked to says that's really enough, presumably because it happens in a context where copyright assignment is normal and expected anyway, and because it represents a *bona fide* effort on the project's part to ascertain the developer's true intentions. On the other hand, the Free Software Foundation goes to the opposite extreme: they require contributors to physically sign and mail in a piece of paper containing a formal statement of copyright assignment, sometimes for just one contribution, sometimes for current and future contributions. If the developer is employed, the FSF asks that the employer sign it too.

The FSF's paranoia is understandable. If someone violates the terms of the GPL by incorporating some of their software into a proprietary program, the FSF will need to fight that in court, and they want their copyrights to be as airtight as possible when that happens. Since the FSF is copyright holder for a lot of popular software, they view this as a real possibility. Whether your organization needs to be similarly scrupulous is something only you can decide, in consultation with lawyers. In general, unless there's some specific reason why your project needs full copyright assignment, just go with CLAs; they're easier for everyone.

Dual Licensing Schemes

Some projects try to fund themselves by using a dual licensing scheme, in which proprietary derivative works may pay the copyright holder for the right to use the code, but the code still remains free for use by open source projects. This tends to work better with code libraries than with standalone applications, naturally. The exact terms differ from case to case. Often the license for the free side is the GNU GPL, since it already bars others from incorporating the covered code into their proprietary product without permission from the copyright holder, but sometimes it is a custom license that has the same effect. An example of the former is the MySQL license, described at http://www.mysql.com/company/legal/licensing/; an example of the latter is Sleepycat Software's licensing strategy, described at http://www.sleepycat.com/download/licensinginfo.shtml.

You might be wondering: how can the copyright holder offer proprietary licensing for a mandatory fee if the terms of the GNU GPL stipulate that the code must be available under less restrictive terms? The answer is that the GPL's terms are something the copyright holder imposes on everyone else; the owner is therefore free to decide *not* to apply those terms to itself. A good way to think of it is to imagine that the copyright owner has an infinite number of copies of the software stored in a bucket. Each time it takes one out of the bucket to send into the world, it can decide what license to put on it: GPL, proprietary, or something else. Its right to do this is not tied to the GPL or any other open source license; it is simply a power granted by copyright law.

The attractiveness of dual licensing is that, at its best, it provides a way for a free software project to get

a reliable income stream. Unfortunately, it can also interfere with the normal dynamics of open source projects. The problem is that any volunteer who makes a code contribution is now contributing to two distinct entities: the free version of the code and the proprietary version. While the contributor will be comfortable contributing to the free version, since that's the norm in open source projects, she may feel funny about contributing to someone else's semi-proprietary revenue stream. The awkwardness is exacerbated by the fact that in dual licensing, the copyright owner really needs to gather formal, signed copyright assignments from all contributors, in order to protect itself from a disgruntled contributor later claiming a percentage of royalties from the proprietary stream. The process of collecting these assignment papers means that contributors are starkly confronted with the fact that they are doing work that makes money for someone else.

Not all volunteers will be bothered by this; after all, their contributions go into the open source edition as well, and that may be where their main interest lies. Nevertheless, dual licensing is an instance of the copyright holder assigning itself a special right that others in the project do not have, and is thus bound to raise tensions at some point, at least with some volunteers.

What seems to happen in practice is that companies based on dual licensed software do not have truly egalitarian development communities. They get small-scale bug fixes and cleanup patches from external sources, but end up doing most of the hard work with internal resources. For example, Zack Urlocker, vice president of marketing at MySQL, told me that the company generally ends up hiring the most active volunteers anyway. Thus, although the product itself is open source, licensed under the GPL, its development is more or less controlled by the company, albeit with the (extremely unlikely) possibility that someone truly dissatisfied with the company's handling of the software could fork the project. To what degree this threat preëmptively shapes the company's policies I don't know, but at any rate, MySQL does not seem to be having acceptance problems either in the open source world or beyond.

Patents

Software patents are the lightning rod issue of the moment in free software, because they pose the only real threat against which the free software community cannot defend itself. Copyright and trademark problems can always be gotten around. If part of your code looks like it may infringe on someone else's copyright, you can just rewrite that part. If it turns out someone has a trademark on your project's name, at the very worst you can just rename the project. Although changing names would be a temporary inconvenience, it wouldn't matter in the long run, since the code itself would still do what it always did.

But a patent is a blanket injunction against implementing a certain idea. It doesn't matter who writes the code, nor even what programming language is used. Once someone has accused a free software project of infringing a patent, the project must either stop implementing that particular feature, or face an expensive and time-consuming lawsuit. Since the instigators of such lawsuits are usually corporations with deep pockets—that's who has the resources and inclination to acquire patents in the first place—most free software projects cannot afford the latter possibility, and must capitulate immediately even if they think it highly likely that the patent would be unenforceable in court. To avoid getting into such a situation in the first place, free software projects are starting to code defensively, avoiding patented algorithms in advance even when they are the best or only available solution to a programming problem.[1]

Surveys and anecdotal evidence show that not only the vast majority of open source programmers, but a majority of *all* programmers, think that software patents should be abolished entirely.[2] Open source programmers tend to feel particularly strongly about it, and may refuse to work on projects that are too closely associated with the collection or enforcement of software patents. If your organization collects software patents, then make it clear, in a public and irrevocable way, that the patents would never be enforced on open source projects, and that they are only to be used as a defense in case some other party initiates an infringement suit against your organization. This is not only the right thing to do, it's also good

[1] Sun Microsystems and IBM have also made at least a gesture at the problem from the other direction, by freeing large numbers of software patents—1600 and 500 respectively—for use by the open source community. I am not a lawyer and thus can't evaluate the real utility of these grants, but even if they are all important patents, and the terms of the grants make them truly free for use by any open source project, it would still be only a drop in the bucket.

[2] See http://lpf.ai.mit.edu/Whatsnew/survey.html for one such survey.

open source public relations.[3]

Unfortunately, collecting patents for defensive purposes is a rational action. The current patent system, at least in the United States, is by its nature an arms race: if your competitors have acquired a lot of patents, then your best defense is to acquire a lot of patents yourself, so that if you're ever hit with a patent infringement suit you can respond with a similar threat—then the two parties usually sit down and work out a cross-licensing deal so that neither of them has to pay anything, except to their intellectual property lawyers of course.

The harm done to free software by software patents is more insidious than just direct threats to code development, however. Software patents encourage an atmosphere of secrecy among firmware designers, who justifiably worry that by publishing details of their interfaces they will be giving technical help to competitors seeking to slap them with patent infringement suits. This is not just a theoretical danger; it has apparently been happening for a long time in the video card industry, for example. Many video card manufacturers are reluctant to release the detailed programming specifications needed to produce high-performance open source drivers for their cards, thus making it impossible for free operating systems to support those cards to their full potential. Why would the manufacturers do this? It doesn't make sense for them to work *against* software support; after all, compatibility with more operating systems can only mean more card sales. But it turns out that, behind the design room door, these shops are all violating one another's patents, sometimes knowingly and sometimes accidentally. The patents are so unpredictable and so potentially broad that no card manufacturer can ever be certain it's safe, even after doing a patent search. Thus, manufacturers dare not publish their full interface specifications, since that would make it much easier for competitors to figure out whether any patents are being infringed. (Of course, the nature of this situation is such that you will not find a written admission from a primary source that it is going on; I learned it through a personal communication.)

Some free software licenses have special clauses to combat, or at least discourage, software patents. The GNU GPL, for example, contains this language:

```
  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

[...]

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.
```

The Apache License, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0) also contains anti-patent requirements. First, it stipulates that anyone distributing code under the license must implicitly in-

[3]For example, RedHat has pledged that open source projects are safe from its patents, see http://www.redhat.com/legal/patent_policy.html.

clude a royalty-free patent license for any patents they might hold that could apply to the code. Second, and most ingeniously, it punishes anyone who initiates a patent infringement claim on the covered work, by automatically terminating their implicit patent license the moment such a claim is made:

```
3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except
as stated in this section) patent license to make, have made, use,
offer to sell, sell, import, and otherwise transfer the Work, where
such license applies only to those patent claims licensable by such
Contributor that are necessarily infringed by their Contribution(s)
alone or by combination of their Contribution(s) with the Work to
which such Contribution(s) was submitted. If You institute patent
litigation against any entity (including a cross-claim or counterclaim
in a lawsuit) alleging that the Work or a Contribution incorporated
within the Work constitutes direct or contributory patent
infringement, then any patent licenses granted to You under this
License for that Work shall terminate as of the date such litigation
is filed.
```

Although it is useful, both legally and politically, to build patent defenses into free software licenses this way, in the end these steps will not be enough to dispel the chilling effect that the threat of patent lawsuits has on free software. Only changes in the substance or interpretation of international patent law will do that. To learn more about the problem, and how it's being fought, go to http://www.nosoftwarepatents.com/. The Wikipedia article http://en.wikipedia.org/wiki/Software_patent also has a lot of useful information on software patents. I've also written a blog post summarizing the arguments against software patents, at http://www.rants.org/2007/05/01/how-to-tell-that-software-patents-are-a-bad-idea/.

Recursos adicionales

Este capítulo es sólo una introducción a las incidencias del licenciamiento de software libre. A pesar de que espero que contenga suficiente información para iniciarte en tu propio proyecto de software libre, cualquier investigación seria acerca de las licencias puede fácilmente aumentar la que este libro ofrece. Aquí hay una lista de recursos acerca de licencias de software libre:

- *Understanding Open Source and Free Software Licensing* por Andrew M. St. Laurent. Publicado por O'Reilly Media, primera edición Agosto 2004, ISBN: 0-596-00581-4.

  Se trata de un completo libro sobre licenciamiento de software libre en toda su complejidad, incluyendo muchos temas omitidos en este capítulo. Veáse http://www.oreilly.com/catalog/osfreesoft/ para más detalle.

- *Make Your Open Source Software GPL-Compatible. Or Else.* por David A. Wheeler, en http://www.dwheeler.com/essays/gpl-compatible.html.

  Se trata de un completo artículo bien escrito sobre por qué es importante usar una licencia compatible con la GPL incluso cuando no usamos la propia GPL. El artículo también trata otras muchas preguntas acerca de licencias de software, y tiene una gran cantidad de excelentes enlaces.

- http://creativecommons.org/

  Creative Commons es una organización que promociona una serie de copyrights más flexibles y liberales que las que la práctica más tradicional del copyright propone. Ofrecen licencias no sólo para software, sino también para textos, arte y música, todas accesibles mediante una selección fácil de usar. Algunas de estas licencias son copyleft, otras no son no-copyleft pero son libres, y otras son simplemente copyright con algunas restricciones relajadas. La página web de Creative Commons propor-

ciona claras instrucciones de sobre qué va. Si tuviera que elegir un sitio para demostrar las implicaciones filosóficas más amplias del movimiento del software libre, sería éste.

Apéndice A. Sistemas de Control de Versiones Libres

Estos son todos los sistemas de control de versiones de código abierto de los que soy consciente a mediados de 2007. El único que uso frecuentemente es Subversion. Poseo poca experiencia o ninguna con la mayoría de estos sistemas, excepto con Subversion y CVS; la información de aquí ha sido tomada de sus sitios web. Ver también http://en.wikipedia.org/wiki/List_of_revision_control_software.

CVS — http://www.nongnu.org/cvs/

CVS ha estado durante mucho tiempo, y muchos desarrolladores están ya familiarizados con él. En su día fue revolucionario: fue el primer sistema de control de versiones de código abierto con acceso a redes de área amplia para desarrolladores (que yo sepa), y el primero que ofreció ""checkouts"" anónimos de sólo lectura, los que dieron a los desarrolladores una manera fácil de implicarse en los proyectos. CVS sólo versiona ficheros, no directorios; ofrece ramificaciones, etiquetado, y un buen rendimiento en la parte del cliente, pero no maneja muy bien ficheros grandes ni ficheros binarios. Tampoco soporta cambios atómicos. *[Descargo: he estado activo en el desarrollo de CVS durante cinco años, antes de ayudar a empezar el proyecto Subversion para reemplazarlo.]*

Subversion — http://subversion.tigris.org/

Subversion fue escrito ante todo para reemplazar a CVS—es decir, para acceder al control de versiones aproximadamente de la misma manera que CVS lo hace, pero sin los problemas o falta de utilidades que más frecuentemente molestan a los usuarios de CVS. Uno de los objetivos de Subversion es encontrar la transición a Subversion relativamente suave para la gente que ya está acostumbrada a CVS. Aquí no hay sitio para entrar en detalles sobre las características de Subversion; acceda a su sitio web para más información. *[Descargo: Estoy implicado en el desarrollo de Subversion, y es el único de estos sistemas que uso habitualmente.]*

SVK — http://svk.elixus.org/

Aunque se ha construido sobre Subversion, probablemente SVK se parece más a algunos anteriores sistemas descentralizados más que Subversión. SVK soporta desarrollo distribuido, cambios locales, mezcla sofisticada de cambios, y la habiluidad de ""reflejar/clonar"" árboles desde sistemas de control de versiones que no son SVK. Vea su sitio web para más detalles.

Mercurial — http://www.selenic.com/mercurial/

Mercurial es un sistemas de control de versiones distribuido que ofrece, entre otras cosas, "una completa ""indexación cruzada"" de ficheros y conjutos de cambios; unos procotolos de sincronización SSH y HTTP eficientes respecto al uso de CPU y ancho de banda; una fusión arbitraria entre ramas de desarrolladores; una interfaz web autónoma integrada; [portabilidad a] UNIX, MacOS X, y Windows" y más (las anterior lista de características ha sido parafraseada del sitio web de Mercurial).

GIT — http://git.or.cz/

GIT es un proyecto empezado por Linus Torvalds para manejar el arbol fuente del ""kernel"" de Linux. Al principio GIT se enfoco bastante en las necesidades del desarrollo del ""kernel"", pero se ha expandido más allá que eso y ahora es usado por otros proyectos a parte del ""kernel"" de Linux. Su página web dice que está "... diseñado para manejar proyectos muy grandes eficaz y velozmente; se usa sobre todo en varios proyectos de código abierto, entre los cuales el más notable es el ""kernel"" de Linux. GIT cae

en la categoría de herramientas de administración de código abierto distribuído, similar al, por ejemplo, GNU Arch o Monotone (o bitKeeper en el mundo comercial). Cada directorio de trabajo de GIT es un repositorio completo con plenas capacidades de gestión de revisiones, sin depender del acceso a la red o de un servidor central."

Bazaar — http://bazaar.canonical.com/

Bazaar está todavía en desarrollo. Será una implementación del protocolo GNU Arch, mantendrá compatibilidad con el procotolo GNU Arch a medida que evolucione, y trabajará con el proceso de la comunidad GNU Arch para cualquier cambio de protocolo que fuera requerido a favor del agrado del usuario.

Bazaar-NG — http://bazaar-ng.org/

Bazaar-NG (o bzr) está actualmente siendo desarrollado por Canonical (http://canonical.com/). Ofrece la elección entre el trabajo centralizado y el descentralizado dentro de un mismo proyecto. Por ejemplo, cuando en la oficina uno puede trabajar en una rama central compartida; para cambios experimentales o trabajo desconectado, se puede crear una rama en el portatil y mezclarla más tarde.

Darcs — http://abridgegame.org/darcs/

"El Sistema Avanzado de Control de Revisión de David (David's Advanced Revision Control System) es aún otro reemplazo de CVS. Está escrito en Haskell, y se ha usado en Linux, MacOX, FreeBSD, Open BSD y Microsoft Windows. Darcs incluye un archivo de comandos cgi, el cual puede ser usado para ver los contenidos de su repositorio.

Arch — http://www.gnu.org/software/gnu-arch/

GNU Arch soporta tanto desarrollo distribuido como centralizado. Los desarrolladores suben sus cambios a un "archivo", que puede ser local, y los cambios pueden ser metidos y quitados de otros archivos tal y como los administradores de esos archivos vean conveniente. Como toda metodología implica, Arch posee más soporte de mezclado que CVS. Arch tambien permite a uno crear fácilmente ramas de archivos a las cuales uno no tiene acceso para subir cambios. Esto es sólo un breve sumario; para más detalles vea las páginas web de Arch.

monotone — http://www.venge.net/monotone/

"monotone es un sistema de control de versión distribuido libre. Ofrece un almacén simple de version transacional de un solo fichero, con una operación completa de desconexión y un protocolo de sincronización entre iguales ""peer-to-peer"" eficiente. Comprende el mezclado susceptible al historial, ramas ligeras, revisión de código integrado y pruebas de terceros. Usa nombrado criptográfico de versión y certificados de cliente RSA. Posee un buen soporte de internacionalización, no tiene dependencias externas, se ejecuta en Linux, Solaris, OSX, y windows, y está licenciado bajo la GNU GPL."

Codeville — http://codeville.org/

"¿Por qué otro sistema de control de versión? Todos los sistemas de control de versión requieren que se mantenga una gestión cuidadosa de las relaciones entre ramas con tal de no tener que mezclar repetidamente los mismos conflcitos. Codeville es mucho más anárquico. Permite actualizar o subir cambios a cualquier repositorio en cualquier momento sin innecesarias re-mezclas."

"Codeville funciona creando un identificador para cada cambio que sea hecho, y recordando la lista de todos los cambios que se han aplicado a cada fichero y el último cambio que modificó cada línea de cada fichero. Cuando hay un conflicto, lo comprueba para ver si una de las dos partes se ha aplicado ya a la otra, y si es así hace a la otra automáticamente ganadora. Cuando hay un conflicto de versión real que

no se puede mezclar automáticamente, Codeville se comporta de una manera casi exacta que CVS."

Vesta — http://www.vestasys.org/

"Vesta es un sistema portátil SCM [Administrador de Configuración de Software o Software Configuration Management] orientado a apoyar el desarrollo de sistemas de software de casi cualquier tamaño, desde bastante pequeños (menos de 10.000 líneas de código fuente) hasta muy grandes (10.000.000 líneas de código fuente)."

"Vesta es un sistema maduro. Es el resultado de más de 10 años de investigación y desarrollo en el Centro de Desarrollo desistemas Compaq/Digital, y ha estado en uso productivo por el grupo de microprocesador Alpha de Compaq durante más de dos años y medio. El grupo Alpha tuvo más de 150 desarrolladores activos en dos sitios a miles de millas de distancia el uno del otro, en la costa este y en la oeste de los Estados Unidos de América. El grupo usó Vesta para administrar ""builds"" con a lo sumo 130 MB de datos fuente, cada uno produciendo 1,5 GB de datos derivados. Las ""builds"" hechas en la parte oeste de media en un día producieron 10-15 GB de datos derivados, todos administrados por Vesta. Aunque Vesta se diseñó teniendo en mente el desarrollo de software, el grupo Alpha demostró la flexibilidad del sistema usándolo para el desarrollo de hardware, comprobando los ficheros en lenguaje de descripción de su hardware dentro de la facilidad de control de código fuente de Vesta y contruyendo simuladores y otros objetos derivados con el ""constructor"" de Vesta. Los miembros del ex-grupo Alpha, ahora parte de Intel, hoy continuan usado Vesta en el proyecto de un nuevo microprocesador."

Aegis — http://aegis.sourceforge.net/

"Aegis es un sistema de administración de configuración de software basado en transacciones. Proporciona un marco dentro del que un equipo de desarrolladores puede trabajar en muchos cambios de un programa independientemente, y Aegis coordina la integración de esos cambios dentro del código fuente maestro del programa, con el menor transtorno posible."

CVSNT — http://cvsnt.org/

"CVSNT es un sistema de control de versión multiplataforma avanzado. Compatible con el protocolo estándar CVS de industria ahora soporta muchas más características... CVSNT es Código Abierto, software libre licenciado bajo la Licencia Pública General GNU." Su lista de características incluye autentificación via todos los protocolos estándar de CVS, más SSPI y Directorio Activo específicos de Windows; soporte de transporte seguro, vía sserver o SSPI encriptada; plataformas cruzadas (corre en entornos Windows o Unix); la versión NT está totalmente integrada con el sistemas Win32; el proceso MergePoint significa no tener que mezclar etiquetas más; bajo desarrollo activo.

META-CVS — http://users.footprints.net/~kaz/mcvs.html

"Meta-CVS es un sistema de control de versión construído alrededor de CVS. Aunque conserva la mayoría de las características de CVS, incluyendo todo el soporte de red, tiene más capacidad que CVS, y es más fácil de usar." Las características listadas en el sitio web de META-CVS incluyen: versionado de estructura de directorio, manejo de tipos de ficheros mejorado, mezclado y ramificado más simple y más fácil de usar, soporte para enlaces simbólicos, listas de propiedad adjuntadas a los datos versionados, importación mejorada de datos de terceros, y un fácil actualización desde las existencias en CVS.

OpenCM — http://www.opencm.org/

"OpenCM está diseñado como un reemplazo seguro y de alta integridad de CVS. Se puede encontrar una lista de las características clave en la página de características. Aun cuando no es tan 'rico en características' como CVS, soporta algunas cosas útiles de las que carece CVS. En pocas palabras, OpenCM proporciona soporte de primera clase para renombrado y configuración, autentificación criptográfica y control de acceso, y ramas de primera clase."

Stellation — http://www.eclipse.org/stellation/

"Stellation es un sistema avanzado y extensible de administración de configuración de software, originalmente desarrollado por IBM Research. Mientras que Stellation proporciona todas las funciones estándar disponibles en cualquier sistema SCM, se distingue por un número de características avanzadas, tales como administración de cambios orientada a tarea, versionado consistente de proyectos y ramificación ligera, destinado para facilitar el desarrollo de sistemas software por grupos grandes de desarrolladores coordinados líbremente."

PRCS — http://prcs.sourceforge.net/

"PRCS, el Sistema de Control de Revisión de Proyecto (Project Revision Control System), es la interfaz de un conjunto de herramientas que (como CVS) proporcionan una manera de tratar con conjuntos de ficheros y directorios como una entidad, preservando versiones coherentes del conjunto entero... Su proposito es similar al de SCCS, RCS, y CVS, pero (al menos según sus autores), es mucho más simple que cualquiera de aquellos sistemas."

ArX — http://www.nongnu.org/arx/

ArX es un sistema de control de versión distribuído que ofrece características de ramificación y mezcla, verificarión criptográfica de integridad de datos, y la capacidad de publicar archivos fácilmente en cualquier servidor HTTP.

SourceJammer — http://sourcejammer.org/

"SourceJammer es un sistema de versionado y control de código fuente escrito en Java. Consiste en un componente en la parte del servidor que mantiene el historial de la versión y de los ficheros, y maneja subidas , descargas, etc, y otros comandos; y en un componente en la parte del cliente que hace peticiones al servidor y administra ficheros en el sistema de ficheros en la parte del cliente."

http://www.zedshaw.com/projects/fastcst/ind FastCST — ex.html

"Un sistema 'moderno' que usa conjuntos de cambios sobre revisiones de ficheros, y operacion distribuída más que control centralizado. Siempre y cuando se posea una cuenta de correo electrónico se puede usar FastCST. Para distribuciones más grandes sólo se necesita un servidor FTP y/o un servidor HTTP p usar el comando incorporado 'serve' para servir directamente las cosas. Todos los conjutos de cambios son únicos universalmente y tienen toneladas de meta-datos, por lo que se puede rechazar cualquier cosa que no se [quiera] antes de intentarlo. La mezcla es hecha comparando un conjunto de cambios mezclado con los contenidos del directorio actual, más que intentar mezclarlo con otro conjunto de cambios."

Superversion — http://www.superversion.org/

"Superversion es un sistema de control de versión distribuído multi-usuario basado en conjuntos de cambios. Apunta a ser una alternativa de código abierto con peso en la industria frente a soluciones comerciales, el cual es igual de fácil de usar (o inclusive más fácil) y con una potencia similar. De hecho, la utilización intuitiva y eficiente ha sido una de las máximas prioridades en el desarrollo de Superversion desde los comienzos."

Apéndice B. Gestor de fallos libres

Da igual que gestor de fallos usa un proyecto, a algunos desarrolladores siempre les gusta quejarse de ellos. Esto es inclusive más cierto para gestores de fallos que para cualquier otra herramienta estándar de desarrollo. Creo que es porque los gestores de fallos son tan visuales y tan interactivos que es fácil imaginar las mejoras que uno mismo haría (si solamente tuviera tiempo), y describirlas en voz alta. Toma las inevitables quejas con excepticismo— muchos de los gestores que se describen más tarde son bastante buenos.

A lo largo de estos listados, la palabra ejemplar se usa para referirse a elementos que el gestor gestiona. Pero recuerda que cada sistema puede tener su propia terminología, en la cual el término correspondiente podría ser "artefacto" o "fallo" o cualquier otra cosa.

Bugzilla — http://www.bugzilla.org/

Bugzilla es muy popular, es mantenido activamente, y parece que hace a sus usuarios muy felices. Yo he estado usando una variante modificada de él en mi trabajo desde hace cuatro años, y me gusta. No es altamente personalizable, pero esa puede ser, de una manera curiosa, una de sus características: Las instalaciones de Bugzilla tienden a parecer la misma sea donde sea que se encuentren, lo que significa que muchos desarrolladores estén ya acostumbrados a su interfaz y se sentirán en un territorio familiar.

GNATS — http://www.gnu.org/software/gnats/

GNU GNATS es uno de los gestores de fallos de código abierto más antiguos, y se usa extensamente. Su mayor fortaleza reside en la diversidad de interfaces (no solamente puede ser usado a través de un navegador WEB, sino que también a través de correo electrónico o utilidades de línea de comandos), y el almacenamiento de los ejemplares en texto plano. El hecho de que los datos de todos los ejemplares se almacenen en ficheros de texto en el disco hace que sea más fácil escribir herramientas a medida para buscar y analizar sintácticamente los datos (por ejemplo, para generar informes estadísticos). GNATS también puede absorber correos electrónicos de muchas maneras, y añadirlos a los ejemplares apropiados basados en patrones dentro de las cabeceras del correo electrónico, lo que hace que el registro de las conversaciones del usuario/desarrollador sean muy fáciles.

RequestTracker (RT) — http://www.bestpractical.com/rt/

El sitio Web de RT dice que "RT es un sistema de etiquetado de niveles de seguridad que permite a un grupo de gente manejar tareas, ejemplares, y peticiones enviadas por una comunidad de usuarios de una manera inteligente y eficiente, y todo eso en resumen. RT posee una interfaz web bastante pulida, y parece tener una base bastante ampliamente instalada. La interfaz es un poco compleja en términos visuales, pero que llega a ser menos molesto a medida que se utiliza. RT tiene una licencia GNU GPL (por alguna razón, su sitio web no deja esto claro).

Trac — http://trac.edgewall.com/

Trac es un poco más que un gestor de fallos: realmente es un sistema wiki y gestor de fallos integrado. Usa el enlace Wiki para conectar ejemplares, ficheros, grupos de cambios de control de versión, y simples páginas wiki. Es bastante simple de configurar, y se integra con Subversion (ver Apéndice A, *Sistemas de Control de Versiones Libres*).

Roundup — http://roundup.sourceforge.net/

Roundup es bastante fácil de instalar (sólo se necesita Python 2.1 o superior), y simple de usar. Tiene interfaces web, para correo electrónico y de línea de comandos. Las plantillas de datos de ejemplares y las

interfaces web son parametrizables, al igual que alguna de su lógica de transición de estados.

Mantis — http://www.mantisbt.org/

Mantis es un sistema de gestión de fallos basado en web, escrito en PHP, y que usa la base de datos MySQL como almacenaje. Posee las características que se esperarían de él. Personalmente, encuentro la interfaz web limpia, intuituva, y visualmente fácil.

Scarab — http://scarab.tigris.org/

Scarab está pensado para ser un gestor de fallos altamente parametizable y con todas las características, ofreciendo más o menos el conjunto total de las características ofrecidas por otros gestores de fallos: entradas de datos, consultas, informes, notificaciones a grupos interesados, acumulación colaborativa de comentarios, y gestión de dependencias.

Se parametriza a través de páginas web administrativas. Se puede tener múltiples "módulos" (proyectos) activos en una única instalación de Scarab. Dentro de un módulo dado, se puede crear nuevos tipos de ejemplares (defectos, mejoras, tareas, peticiones de apoyo, etc.), y añadir atributos arbitrarios, para afinar el gestor a los requisitos específicos de tu proyecto.

A últimos de 2004, Scarab se acercaba a su versión liberada 1.0.

Sistema de Gestión de Fallos de Debian (Debian Bug Tracking System http://www.chiark.greenend.org.uk/~ian/debb (DBTS)) — ugs/

El Sistema de Gestión de Fallos de Debian (Debian Bug Tracking System) es inusual en el sentido que todas las entradas y manipulaciones de ejemplares se hace vía correo electrónico: cada ejemplar obtiene su propia dirección de correo electrónico dedicada. El DBTS escala bastante bien: http://bugs.debian.org/ tiene 227.741 ejemplares, por ejemplo.

Ya que la interacción se hace vía clientes de correo normales, un entorno que es familiar y fácilmente accesible para la mayoría de gente, el DBTS es bueno para manejar grandes volúmenes de informes entrantes que necesitan una rápida clasificación y respuesta. Por supuesto, también existen desventajas. Los desarrolladores deben dedicar el tiempo necesario a aprender el sistema de comando del correo electrónico, y los usuarios deben escribir sus informes de fallos sin un formulario web que los guíe en la elección de la información que hay que escribir. Hay algunas herramientas disponibles para ayudar a los usuarios a enviar mejor sus informes de fallos, tales como el programa de línea de comandos **reportbug** o el paquete `debbugs-el` para Emacs. Pero la mayoría de la gente no usará estas herramientas; sólo escribirán correos electrónicos a mano, y podrán o no seguir las pautas para reportar fallos publicadas por su proyecto.

El DBTS tiene una interfaz web de sólo lectura, para ver y consultar ejemplares.

Gestores de Etiquetado de Problema

Estos están más orientados hacia la gestión de etiquetas del escritorio de ayuda que a la gestión de fallos de software. Probablemente sería mejor con un gestor de fallos habitual, pero estos se han listado por completitud, y porque posiblemente podría tener proyectos poco comunes para los cuales un sistema de etiquetado de problemas podría ser más apropiado que un gestor de fallos tradicional.

- **WebCall** — http://myrapid.com/webcall/
- **Teacup** — http://www.altara.org/teacup.html

  (Teacup no parece estar activo bajo desarrollo, pero los ficheros para descargar están todavía accesibles. Nota que tiene tanto interfaz web como por correo electrónico.)

Bluetail Ticket Tracker (BTT) — http://btt.sourceforge.net/

El BTT se situa en algún lugar entre un gestor de etiquetas de problema y un gestor de fallos. Ofrece características de privacidad que son algo inusuales entre los gestores de fallos de código abierto: los usuarios del sistema se clasifican como Plantilla (Staff), Amigo (Friend), Cliente (Customer), o Anónimo (Anonymous), y más o menos los datos son accesibles según la categoría de uno mismo. Ofrece algo de integración con el correo electrónico, una interfaz por línea de comandos, y un mecanismo para convertir correos electrónicos en etiquetas. También posee utilidades para mantener la información no asociada con una etiqueta específica, tal como documentación interna o FAQs. BTT is somewhere between a standard trouble-ticket tracker and a bug tracker. It offers privacy features that are somewhat unusual among open source bug trackers: users of the system are categorized as Staff, Friend, Customer, or Anonymous, and more or less data is available depending on one's category. It offers some email integration, a command-line interface, and mechanisms for converting emails into tickets. It also has features for maintaining information not associated with any specific ticket, such as internal documentation or FAQs.

Apéndice C. Why Should I Care What Color the Bikeshed Is?

You shouldn't; it doesn't really matter, and you have better things to spend your time on.

Poul-Henning Kamp's famous "bikeshed" post (an excerpt from which appears in Capítulo 6, *Communications*) is an eloquent disquisition on what tends to go wrong in group discussions. It is reprinted here with his permission. The orginal URL is http://www.freebsd.org/cgi/getmsg.cgi?fetch=506636+517178+/usr/local/www/db/text/1999/freebsd-hackers/19991003.freebsd-hackers.

```
Subject: A bike shed (any colour will do) on greener grass...
From: Poul-Henning Kamp <phk@freebsd.org>
Date: Sat, 02 Oct 1999 16:14:10 +0200
Message-ID: <18238.938873650@critter.freebsd.dk>
Sender: phk@critter.freebsd.dk
Bcc: Blind Distribution List: ;
MIME-Version: 1.0


[bcc'ed to committers, hackers]

My last pamphlet was sufficiently well received that I was not
scared away from sending another one, and today I have the time
and inclination to do so.

I've had a little trouble with deciding on the right distribution
of this kind of stuff, this time it is bcc'ed to committers and
hackers, that is probably the best I can do.  I'm not subscribed
to hackers myself but more on that later.

The thing which have triggered me this time is the "sleep(1) should
do fractional seconds" thread, which have pestered our lives for
many days now, it's probably already a couple of weeks, I can't
even be bothered to check.

To those of you who have missed this particular thread: Congratulations.

It was a proposal to make sleep(1) DTRT if given a non-integer
argument that set this particular grass-fire off.  I'm not going
to say anymore about it than that, because it is a much smaller
item than one would expect from the length of the thread, and it
has already received far more attention than some of the *problems*
we have around here.

The sleep(1) saga is the most blatant example of a bike shed
discussion we have had ever in FreeBSD.  The proposal was well
thought out, we would gain compatibility with OpenBSD and NetBSD,
and still be fully compatible with any code anyone ever wrote.

Yet so many objections, proposals and changes were raised and
launched that one would think the change would have plugged all
the holes in swiss cheese or changed the taste of Coca Cola or
something similar serious.

"What is it about this bike shed ?" Some of you have asked me.

It's a long story, or rather it's an old story, but it is quite
short actually.  C. Northcote Parkinson wrote a book in the early
```

1960'ies, called "Parkinson's Law", which contains a lot of insight into the dynamics of management.

You can find it on Amazon, and maybe also in your dads book-shelf, it is well worth its price and the time to read it either way, if you like Dilbert, you'll like Parkinson.

Somebody recently told me that he had read it and found that only about 50% of it applied these days. That is pretty darn good I would say, many of the modern management books have hit-rates a lot lower than that, and this one is 35+ years old.

In the specific example involving the bike shed, the other vital component is an atomic power-plant, I guess that illustrates the age of the book.

Parkinson shows how you can go in to the board of directors and get approval for building a multi-million or even billion dollar atomic power plant, but if you want to build a bike shed you will be tangled up in endless discussions.

Parkinson explains that this is because an atomic plant is so vast, so expensive and so complicated that people cannot grasp it, and rather than try, they fall back on the assumption that somebody else checked all the details before it got this far. Richard P. Feynmann gives a couple of interesting, and very much to the point, examples relating to Los Alamos in his books.

A bike shed on the other hand. Anyone can build one of those over a weekend, and still have time to watch the game on TV. So no matter how well prepared, no matter how reasonable you are with your proposal, somebody will seize the chance to show that he is doing his job, that he is paying attention, that he is *here*.

In Denmark we call it "setting your fingerprint". It is about personal pride and prestige, it is about being able to point somewhere and say "There! *I* did that." It is a strong trait in politicians, but present in most people given the chance. Just think about footsteps in wet cement.

I bow my head in respect to the original proposer because he stuck to his guns through this carpet blanking from the peanut gallery, and the change is in our tree today. I would have turned my back and walked away after less than a handful of messages in that thread.

And that brings me, as I promised earlier, to why I am not subscribed to -hackers:

I un-subscribed from -hackers several years ago, because I could not keep up with the email load. Since then I have dropped off several other lists as well for the very same reason.

And I still get a lot of email. A lot of it gets routed to /dev/null by filters: People like [omitted] will never make it onto my screen, commits to documents in languages I don't understand likewise, commits to ports as such. All these things and more go the winter way without me ever even knowing about it.

But despite these sharp teeth under my mailbox I still get too much email.

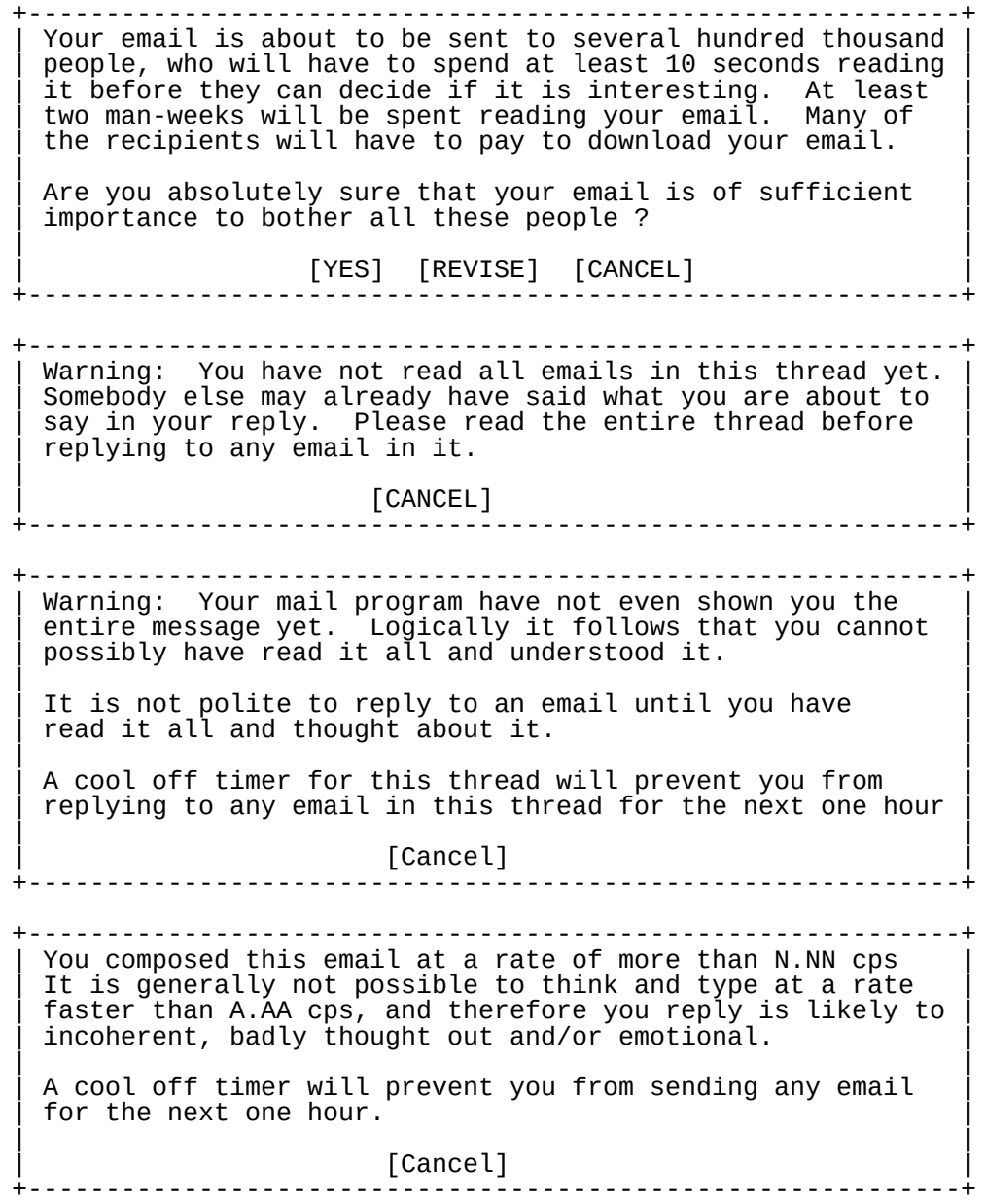
This is where the greener grass comes into the picture:

I wish we could reduce the amount of noise in our lists and I wish we could let people build a bike shed every so often, and I don't really care what colour they paint it.

The first of these wishes is about being civil, sensitive and intelligent in our use of email.

If I could concisely and precisely define a set of criteria for when one should and when one should not reply to an email so that everybody would agree and abide by it, I would be a happy man, but I am too wise to even attempt that.

But let me suggest a few pop-up windows I would like to see mail-programs implement whenever people send or reply to email to the lists they want me to subscribe to:



The second part of my wish is more emotional. Obviously, the capacities we had manning the unfriendly fire in the sleep(1) thread, despite their many years with the project, never cared enough to do this tiny deed, so why are they suddenly so enflamed by somebody else so much their junior doing it ?

I wish I knew.

I do know that reasoning will have no power to stop such "reactionaire conservatism". It may be that these people are frustrated about their own lack of tangible contribution lately or it may be a bad case of "we're old and grumpy, WE know how youth should behave".

Either way it is very unproductive for the project, but I have no suggestions for how to stop it. The best I can suggest is to refrain from fuelling the monsters that lurk in the mailing lists: Ignore them, don't answer them, forget they're there.

I hope we can get a stronger and broader base of contributors in FreeBSD, and I hope we together can prevent the grumpy old men and the [omitted]s of the world from chewing them up, spitting them out and scaring them away before they ever get a leg to the ground.

For the people who have been lurking out there, scared away from participating by the gargoyles: I can only apologise and encourage you to try anyway, this is not the way I want the environment in the project to be.

Poul-Henning

Apéndice D. Ejemplo de Instrucciones para Informar sobre Fallos

Esto es una copia editada ligeramente de las instrucciones en línea del proyecto Subversion para nuevos usuarios sobre cómo informar sobre fallos. Ver “Treat Every User as a Potential Volunteer” en Capítulo 8, *Coordinando a los Voluntarios* para saber porque es importante que un proyecto tenga tales instrucciones. El documento original se localiza en http://svn.collab.net/repos/svn/trunk/www/bugs.html.

```
                        Informar sobre Fallos en Subversion

Este documento explica cómo y dónde informar sobre fallos. (no es una lista
de todos los fallos pendientes - pero se puede conseguir eso aquí)

Dónde informar sobre un fallo
-----------------------------

    * Si el fallo está en el propio Subversion, mandar un correo a
      users@subversion.tigris.org. Una vez que se ha confirmado que es
      un fallo, alguién, posiblemente tú, puede introducirlo en el
      gestor de ""ejemplares"". (o si estás bastante seguro del fallo,
      sigue adelante y publicalo directamente en nuestra lista de desarrollo,
      dev@subversion.tigris.org. Preo si no estás seguro, es mejor que se
      publique primero en users@; ahí alguién puede decirte si el comportamiento
      que se encontró es el esperado o no.)

    * Si el fallo esta en la librería APR, por favor informálo en estás
      dos listas de correo: dev@apr.apache.org, dev@subversion.tigris.org.

    * Si el fallo está en la librería de HTTP Neon, por favor infórmalo en:
      neon@webdav.org, dev@subversion.tigris.org.

    * Si el fallo está en el Apache HTTPD 2.0, por favor infórmalo en
            estás dos listas de correo: dev@httpd.apache.org,
      dev@subversion.tigris.org. La lista de correo para el desarrollador de
      Apache httpd tiene mucho tráfico, así que tu publicación del informe
      del fallo puede que sea pasada por alto. También debes introducir
      un informe del fallo en http://httpd.apache.org/bug_report.html.

    * Si el fallo está en tu ""alfombra"", por favor dale un abrazo y
            déjalo ""cómodo"".

Cómo informar sobre un fallo
----------------------------

Primero, asegurate que es un fallo. Si Subversion no se comporta como esperas,
mira la documentación y los archivos de las listas de correo buscando alguna
evidencia que indique que debería comportarse como esperas. Por supuesto,
si es una cosa de sentido común, como que Subversion ha destruído tus datos
y ha hecho que salga humo de tu monitor, entonces puedes fiarte de tu juicio.
Pero si no estás seguro, sigue adelante y pregunta primero en las lista
de correo de usuarios, users@subversion.tigris.org, o pregunta en IRC,
irc.freenode.net, en el canal #svn.

Una vez que has demostrado que es un fallo, lo más importante que debes hacer
es proponer una descripción simple y una receta para reproducirlo. Por ejemplo,
si el fallo, como lo encontraste inicialmente, implica cinco ficheros
sobre diez cambios, intenta hacer que se reproduzca con solo un fichero
y un cambio. Cuanto más simple es la receta para reproducirlo, más
probable es que un desarrollador tenga éxito al reproducir el fallo
```

y en arreglarlo.

Cuando escribas la receta para reproducirlo, no escribas una descripción
inversa de lo que hiciste para que ocurriera el fallo. Por el contrario,
da una transcripción literal de la serie exacta de comandos que ejecutaste,
y sus salidas. Utiliza la función cortar-y-pegar para este fin. Si hay ficheros
implicados, asegurate incluir los nombres de los ficheros, e incluso su
contenido si piensas que podría ser relevante. Lo mejor es empaquetar
tu receta para reproducirlo en un archivo de comandos, lo cual ayuda mucho.

*Una sana comprobación rápida: *estás* utilizando la versión más
reciente de Subvesion, ¿no? :-) Posiblemente el fallo ya se ha corregido;
deberías probar tu receta para reproducir el fallo en el árbol de desarrollo
de Subversión más reciente.*

Además de la receta para reproducirlo, también necesitaremos una
descripción completa del entorno donde se reprodució el error.
Esto es:

* Tu sistema operativo
* La versión liberada y/o revisión de Subversion
* El compilador y las opciones de configuración con las que ""construíste"" Su
* Cualquier modificación personal que hiciste a Subversion
* La versión de la BD de Berkeley con la que estás corriendo Subversion, si u
* Cualquier otra cosa que podría ser posiblemente relevante. Mejor tener demas
  más que tener demasiada poca.

Una vez que tienes todo esto, estás listo para escribir el informe. Empieza
con una descripción clara del fallo en si. Es decir, di como esperabas que
Subversion se comportara, y contrástalo con como realmente se comportó.
Aunque el fallo puede parecerte obvio a ti, puede no ser tan obvio para
otra persona, por tanto es mejor evitar las adivinanzas. Sigue con la descripción
del entorno, y la receta para reproducirlo. Si también quieres incluir
alguna especulación sobre la causa, e inclusive un parche para arreglar
el problema, sería genial - ver http://svn.collab.net/repos/svn/trunk/www/hacking.
con las instrucciones para mandar parches.

Publica toda esta información en dev@subversion.tigris.org, o si ya lo has hecho
y has pedido que se publique un ""ejemplar"", entonces ve al Gestor de ""Ejemplare
y sigue las instrucciones de allí.

Gracias. Sabemos que es mucho trabajo publicar un informe de fallos efectivo,
pero a un buen informe puede ahorrar horas de tiempo a un desarrollador,
y hace que los fallos tengan más probabilidad de ser arreglados.

Apéndice E. Copyright

TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

   b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

   c. "Creative Commons Compatible License" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

   d. "Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

   e. "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

   f. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

   g. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights.

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant.

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be

made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You

may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted

under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the