

La catedral y el bazar



Las versiones HTML, texto y PostScript de este archivo se encuentran en el servidor de Linux Resources. El resto de los enlaces apuntan a la página personal de Eric.

Mi análisis de cómo y porqué funciona el modelo de desarrollo de Linux, tal como se expuso en el Linux Kongress 97, el Atlanta Linux Showcase, la primera Perl Conference, y en LinuxPro 97 (y llamada a ser en breve una presentación por invitación en Usenix 1998!). Si estás usando un navegador gráfico, verás a la izquierda una foto mía hablando en el Kongress.

Existen grabaciones RealAudio en la red de la presentación en el Kongress como comunicación oral. Yo dispongo de unas digitalizadas en frecuencia ISDN (9 megas). Solía tener unas a 28.8K, pero resultó que no era raro que se truncaran en la transferencia.

También puedes acceder a versiones completas (en inglés) de la comunicación en formato PostScript o incluso en texto ASCII.

Este documento ha tenido ya consecuencias en el mundo real. El día en que se anunció que [el código fuente de Netscape Communicator pasaba a dominio público](#), la gente de Netscape me informó de que había tenido una gran influencia en su decisión. Eric Hahn, vicepresidente ejecutivo y jefe de la oficina técnica (Chief Technology Officer) en Netscape, me escribió cuatro días más tarde confirmándolo: "En nombre de todo el personal de Netscape, quiero darle las gracias ante todo por ayudarnos a llegar a este punto. Sus reflexiones y sus escritos fueron una inspiración fundamental para tomar esta decisión."

Nota: este documento (al igual que el tema) está aún en evolución como consecuencia de las opiniones e informaciones que recibo. Los cambios fundamentales más recientes los hice el 1 de Junio de 1997 y añadí una anécdota tomada de la primera conferencia sobre Perl el 18 de Noviembre de 1997. Añadí la sección "Lecturas adicionales recomendadas" el 10 de Julio de 1997. Hasta en el caso de hayas asistido a la presentación oral, puedes querer volver a leerlo.

Puedes leer una [traducción al Alemán](#) de una versión anterior del documento (tras Würzburg, antes de Atlanta).

Si te gusta este documento, probablemente te gustará también mi [How To Become A Hacker](#) FAQ.

La catedral y el bazar

por [Eric S. Raymond](#)

[Eric's Home Page](#)

\$Fecha: 1998/01/29 05:47:17 \$

Analizo un proyecto de software de dominio público desarrollado con éxito, fetchmail, que se realizó como una prueba deliberada de algunas teorías sorprendentes sobre ingeniería de software sugeridas por la historia de Linux. Presento estas teorías en términos de dos estilos de desarrollo completamente distintos, el modelo "catedral", aplicable a la mayor parte de los desarrollos realizados en el mundo del software comercial, frente al modelo "bazar", más propio del mundo Linux. Muestro que estos modelos se derivan de puntos de partida opuestos sobre la naturaleza del proceso de depuración del software. Prosigo manteniendo a partir de la experiencia Linux la hipótesis de que "Dado un número suficiente de ojos, todos los errores son irrelevantes", sugiriendo analogías productivas con otros sistemas auto-correctores integrados por agentes autónomos, y concluyo realizando alguna exploración de las consecuencias de este punto de vista sobre el futuro del software.

Nota del traductor al castellano:

El término "Free Software" se ha traducido como "software abierto". Se ha preferido esta denominación a la de "software de dominio público" (que correspondería estrictamente al término "public domain software") para enfatizar lo que se cree es una característica necesaria de este tipo de software: disponer del código fuente del programa. El término "hacker" resulta de difícil traducción. En ocasiones se ha traducido como "programador aficionado", si bien ha sido más frecuente que se haya dejado sin traducir. Según el libro "Léeme.ya", ISBN 84-605-7033-9, pg. 734, la definición de "hacker" vendría a ser la siguiente: "Hacker: Citando literalmente el libro 'Linux. Edición especial', pg. 30: '... la definición popular de hacker tiene una connotación negativa en la sociedad actual... la actividad de los hackers tiene que ver básicamente con el aprendizaje de todo lo que hay que conocer de un sistema, la posibilidad de sumergirse en éste al grado de abstraerse, y la capacidad de repararlo cuando se cae. En general a los hackers les interesa conocer el funcionamiento de los sistemas que encuentran interesantes. ...algunos hackers transponen esa línea y se convierten en lo que la comunidad hacker ha denominado cracker. A los hackers de la computación les molesta bastante que los comparen con estos vándalos y delincuentes que los medios de difusión popular llaman equivocadamente hackers en vez de crackers...' . Pues eso, no se confunda. A pesar de todo, un colega bastante enterado mantiene que esa sería la definición de "wizard" (brujo o mago) y no la de "hacker", siempre con alguna connotación en el límite de la legalidad. Lo dejo a su elección.'

1. [La catedral y el bazar](#)
2. [El correo debe pasar](#)
3. [La importancia de tener usuarios](#)
4. [Lánzalo pronto, lánzalo a menudo](#)
5. [¿Cuándo una rosa deja de serlo?](#)
6. [Popclient se convierte en Fetchmail](#)
7. [Fetchmail se hace mayor](#)
8. [Algunas lecciones más a partir de Fetchmail](#)
9. [Condiciones previas necesarias para el modelo bazar](#)
10. [El contexto social del software abierto](#)
11. [Agradecimientos](#)
12. [Lecturas adicionales](#)
13. [Historial de versiones y cambios](#)

1. La catedral y el bazar

Linux es subversivo. ¿Quién hubiera pensado, tan solo cinco años atrás, que un sistema operativo de gran calidad pudiera concretarse como por ensalmo a partir del trabajo aficionado y a tiempo parcial de varios miles de programadores esparcidos por todo el planeta y conectados tan solo por las ténues hebras de Internet?.

Desde luego, yo no. En el momento en que Linux surgió en la pantalla de mi radar a principio de 1993, había estado ya involucrado en el desarrollo de Unix y de software abierto durante diez años. Era uno de los que primero contribuyó al desarrollo de GNU a mediados de los ochenta. Había lanzado en la red una cantidad respetable de software abierto, desarrollando o co-desarrollando varios programas (nethack, los modos VC y GUD de Emacs, xlife y algunos más) que aún se emplean ampliamente hoy en día. Creía saber como se hacía.

Linux puso patas arriba mucho de lo que yo creía que sabía. Había estado predicando durante años el evangelio Unix consistente en herramientas pequeñas, rápido desarrollo de prototipos y programación evolutiva. Pero también creía que existía una cierta complejidad crítica por encima de la cual era preciso recurrir a un enfoque más centralizado y planificado desde el principio. Creía que el software más importante (los sistemas operativos o las herramientas realmente grandes tales como Emacs) necesitaban ser construidas al modo de las catedrales, ser cuidadosamente ensamblados por magos o pequeñas bandas de hechiceros trabajando en un espléndido aislamiento, sin que hubiera lugar al lanzamiento de versiones de prueba antes de que hubiera llegado el momento.

El estilo de desarrollo de Linus Torvalds - lanzar versiones de prueba enseguida y a menudo, delegar cuanto sea posible, estar abierto hasta el punto de resultar promiscuo - resultó una verdadera sorpresa. Nada que ver con la silenciosa y reverente construcción de una catedral -- la comunidad Linux, por contra, parecía semejarse a un gran bazar bullicioso con diferentes agendas y enfoques (adecuadamente reflejado por los depósitos de software Linux, que admitían contribuciones de *cualquiera*) del cual solo parecía posible que emergiera un sistema coherente y estable mediante una sucesión de milagros.

El hecho de que este estilo bazar parecía funcionar, y bien, me produjo una auténtica conmoción. Mientras lo iba aprendiendo trabajé, además de en proyectos individuales, para intentar comprender porqué el mundo Linux no solo no se desmoronaba en medio de una colosal confusión sino que parecía ir de logro en logro a una velocidad difícil de imaginar para los constructores de catedrales.

A mediados de 1996 creí que lo comenazaba a entender. La casualidad me proporcionó una forma perfecta de poner a prueba my teoría, mediante un proyecto de software abierto que podía intentar conducir de forma deliberada según el estilo bazar. Lo hice así -- y resultó un éxito incuestionable.

En el resto de este artículo, contaré la historia de ese proyecto, y lo emplearé para proponer algunas reglas sobre desarrollo eficaz de software abierto. No todas son cosas que aprendí en el mundo Linux, pero veremos que este les aporta un brillo especial. Si estoy en lo cierto, le permitirán entender exactamente qué es lo que hace de la comunidad Linux una fuente tal de buen software -- y le ayudarán a Vd. a ser más productivo.

2. El correo debe pasar

Desde 1993, he estado manteniendo la parte técnica de un pequeño proveedor de Internet de acceso libre llamado Chester County InterLink (CCIL) en West Chester, Pennsylvania (soy cofundador de CCIL y escribí nuestro único software BBS multiusuario -- como puedes comprobar realizando un "telnet" a locke.ccil.org. En la actualidad da soporte a casi tres mil usuarios a través de diecinueve líneas.) Este trabajo me permitía tener un acceso a la red durante 24 horas diarias a través de la línea de 56K del CCIL -- de hecho, ¡casi fui yo el que exigió tal cosa!.

En consecuencia, me he acostumbrado bastante a tener un acceso instantáneo al email en Internet. Por una serie de complejas razones, era difícil conseguir que SLIP funcionara entre la máquina de mi casa (snark.thyrus.com) y CCIL. Cuando finalmente lo conseguí, me encontré teniendo que realizar periódicamente sesiones de "telnet" en "locke" para poder comprobar mi correo, lo que me resultó molesto. Lo que yo quería era que mi correo llegara a "snark", de modo que biff(1) pudiera informarme de su llegada.

Una simple redirección mediante el "sendmail forwarding" no hubiera funcionado, ya que "snark" no está siempre en red y carece de dirección IP estática. Lo que yo necesitaba era un programa que pudiera recogerlo a través de mi conexión SLIP y se trajera mi correo para entregarlo en mi casa. Sabía que tales cosas existían, y que la mayor parte de ellas empleaban un sencillo protocolo denominado POP (Post Office Protocol). Y estaba bastante seguro de que el sistema operativo BSD/OS con que funcionaba "locke" contaba ya con un servidor POP3.

Necesitaba un cliente POP3. Por lo tanto me dí unas vueltas por la red y encontré uno. De hecho, encontré tres o cuatro. Empleé pop-perl durante una temporada, pero echaba en falta lo que parecía ser una capacidad obvia, la habilidad de capturar las direcciones del correo recogido de modo que las contestaciones funcionaran correctamente.

El problema era éste: supongamos que alguien llamado 'joe' en "locke" me envía un mensaje. Si lo recojo y lo envío a "snark" e intento contestar, mi programa de correo intentará enviarlo alegremente a un inexistente 'joe' en "snark". Editar a mano las direcciones de las respuestas para restaurar el original '@ccil.org' se convirtió pronto en una considerable molestia.

Era claramente algo que el ordenador debía hacer en mi lugar. (De hecho, según RFC1123 sección 5.2.18, sendmail debería estar haciéndolo.) ¡Pero ninguno de los clientes POP sabía como!. Y esto nos lleva a la primera lección:

1. Todos los trabajos buenos en software comienzan tratando de paliar un problema personal del que los programa.

Quizá esto hubiera debido resultar evidente (se sabe desde hace tiempo que "La necesidad es la madre de la invención") pero los programadores desperdician demasiado a menudo sus días peleando por dinero con programas que no necesitan y a los que no aman. No así en el mundo Linux -- lo que puede explicar porqué la calidad media del software desarrollado por dicha comunidad es tan alta.

¿Me entregué de inmediato, por tanto, a un frenético intento de programación de un nuevo cliente POP3 para competir con los que ya había?. Jamás de los jamases. Busqué cuidadosamente entre las utilidades POP que tenía a mano preguntándome "¿cual es la que más se acerca a lo que yo quiero?". Porque ...

2. Los buenos programadores saben qué escribir. Los grandes saben qué reescribir (y reutilizar).

Aunque no pretendo ser un gran programador, trato de imitarlos. Una característica importante de los grandes de verdad es la vagancia constructiva. Saben que te dan un diez no por tu esfuerzo, sino por los resultados, y es casi siempre más fácil empezar a partir de una buena solución parcial que desde la nada más absoluta.

Linus, sin ir más lejos, no intentó en realidad escribir Linux partiendo de cero. En vez de eso, comenzó reutilizando código e ideas de Minix, un minúsculo simil de Unix para máquinas 386. Al final, todo el código Minix fue desechado o se reescribió por completo -- pero mientras estuvo allí proporcionó el andamiaje necesario para la criatura que eventualmente llegaría a ser Linux.

En la misma línea, comencé a buscar una herramienta POP ya existente que estuviera razonablemente bien programada para utilizarla como punto de partida.

La tradición Unix de compartir el código fuente ha facilitado la reutilización del código (es la razón por la que el proyecto GNU eligió Unix como sistema operativo de partida, a pesar de tener serias reservas sobre el mismo). El mundo Linux ha llevado esta tradición muy cerca de su límite desde el punto de vista tecnológico; pues cuenta con terabytes de código fuente de uso libre. Es por eso por lo que es más probable que invertir tiempo en la búsqueda de algo casi lo suficientemente bueno realizado por otro de mejores resultados que en ningun otro lugar.

Y a mí me lo dió. Con los que ya había encontrado anteriormente, mi segunda búsqueda elevó el total a nueve candidatos -- fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail y upop. Al principio me quedé con 'fetchpop' de Seung-Hong Oh. Le añadí mi característica de reescribir las cabeceras, y le hice algunas otras mejoras que fueron aceptadas por el autor e incorporadas a la versión 1.9.

Sin embargo, algunas semanas después cayó en mis manos el código de 'popclient' de Carl Harris, y supe que tenía un problema. Aunque 'fetchpop' contaba con algunas ideas originales (su modo demon, sin ir más lejos), sólo era capaz de manejar POP3 y estaba codificado de una forma no demasiado seria (Seung-Hong era un programador brillante pero poco experimentado, y ambas características quedaban claramente de manifiesto). El código de Carl era mejor, bastante profesional y sólido, pero su programa carecía de varias posibilidades importantes y bastante puñeteras de implementar con las que contaba 'fetchpop' (entre ellas las que yo le había añadido).

¿Seguir o cambiar?. Si cambiaba, tenía que renunciar al código que ya había desarrollado a cambio de una mejor base para el futuro.

Una razón práctica para realizar el cambio era la presencia de soporte para protocolos múltiples. POP3 es el que se usa más comunmente, pero no el único. 'Fetchpop' y los demás competidores no soportaban POP2, RPOP o APOP, y yo albergaba ya alguna remota intención de añadir tal vez

IMAP (Internet Message Access Protocol, el protocolo más reciente y potente para oficinas de correo) aunque solo fuera como diversión.

Pero tenía una razón teórica adicional para pensar que cambiar podía ser también una buena idea, algo que aprendí mucho antes de llegar a Linux.

3. "Piensa en desechar al menos uno: lo terminarás haciendo de todos modos." (Fred Brooks, "The Mythical Man-Month", Capítulo 11)

O, por decirlo de otra manera, a menudo no entiendes un problema hasta después de haber implementado una primera solución. La segunda vez, tal vez sepas lo suficiente como para hacerlo bien. Por tanto, si lo quieres hacer bien, debes estar preparado para empezar al menos dos veces.

Bueno (me dije) es posible que 'fetchpop' haya sido mi primer intento. Y cambié.

Tras enviar mi primera remesa de modificaciones a 'popclient' a Carl Harris el 25 de Junio de 1996, me di cuenta de que hacía ya algún tiempo que él había perdido interés en 'popclient'. El código se veía un tanto polvoriento, con algunos errores menores dando vueltas por ahí. Tenía muchas modificaciones que hacer, y nos pusimos de acuerdo con rapidez en que lo más lógico era que me hiciera cargo del programa.

Sin darme cuenta, la escala del proyecto había cambiado. Ya no se trataba de realizar pequeñas modificaciones a un cliente POP ya existente, sino de mantener uno por completo, y sabía que por si fuera poco algunas de las ideas que bullían en mi cabeza conducían a cambios drásticos.

En una cultura que promueve compartir código esta es la forma natural en que evoluciona un proyecto. La cosa funcionaba del siguiente modo:

4. Si tienes la actitud adecuada, los problemas interesantes te encontrarán

Pero la actitud de Carl Harris era incluso más importante. Entendía que...

5. Cuando un programa deja de interesarte, tu último deber es pasarlo a un sucesor competente.

Sin tener que entrar en discusiones, Carl y yo sabíamos que compartíamos el propósito de desarrollar la mejor solución de todas al problema que nos ocupaba. El único punto a aclarar era si yo era un sucesor digno. Una vez que quedó claro, actuó con gracia y determinación. Espero ser capaz de actuar de la misma forma cuando llegue mi turno.

3. La importancia de tener usuarios

O sea, que heredé 'popclient'. Y lo que es casi igual de importante, heredé sus usuarios. Los usuarios son una gran cosa, y no sólo porque demuestren que estás satisfaciendo una necesidad, que estás haciendo algo bien. Si los cuidas bien, pueden convertirse en tus ayudantes, en programadores asociados.

Otro de los puntos fuertes de la tradición Unix, y de nuevo uno que Linux lleva hasta un feliz extremo, es que muchos usuarios son a la vez 'hackers'. -- y puesto que el código fuente está disponible pueden ser 'hackers' productivos. Esto puede resultar enormemente eficaz para reducir el tiempo necesario para la depuración. Con un poco de motivación, tus usuarios diagnosticarán problemas, sugerirán correcciones, y ayudarán a mejorar el código mucho más rápidamente de lo que tú serías capaz de lograr sin ayuda.

6. Tratar a tus usuarios como colaboradores es el camino menos complicado para mejorar con rapidez y depurar eficazmente un programa.

Resulta fácil subestimar la potencia de este efecto. En realidad, muchos de los que estamos involucrados en el desarrollo de software abierto infravalorábamos drásticamente lo bien que puede aumentarse de escala sin más que recurrir a un número aún mayor de usuarios y contribuir a reducir la complejidad del sistema, hasta que Linus demostró lo contrario.

Llego a creer que el logro más brillante y trascendental de Linus no fue la construcción del núcleo de Linux en sí mismo, sino más bien la invención del modelo de desarrollo Linux. Una vez que me atreví a decir tal cosa en su presencia, se limitó a sonreír y repitió en voz baja algo que ha dicho a menudo: "Soy básicamente una persona muy perezosa a la que le gusta recibir los laureles de lo que otros han hecho". Perezoso como un zorro. O, como diría Robert Heinlein, demasiado perezoso como para equivocarse.

Mirando hacia atrás, puede verse un claro precedente de los métodos y el éxito de Linux en el desarrollo de la librería Lisp de GNU Emacs y en los archivos de código Lisp. Al contrario de lo ocurrido con el desarrollo en plan construcción de catedrales del núcleo en C de Emacs y de la mayor parte de las herramientas de la FSF (Free Software Foundation), la evolución del depósito de código en Lisp fué fluida y estuvo sumamente orientada al usuario final. Las ideas y los modos adicionales en desarrollo se escribieron a menudo tres o cuatro veces antes de alcanzar un estado final estable. Y fueron frecuentes las colaboraciones ocasionales mediante Internet, en plan Linux.

Sin ir más lejos, el desarrollo concreto de mayor éxito que yo había realizado antes de 'fetchmail' fue probablemente el modo VC de Emacs, una colaboración en la línea de Linux con otras tres personas a través de e-mail, de las cuales sólo he llegado a conocer hasta el momento a una de ellas (Richard Stallman). Se trataba de un interfase de acceso a SCCS, RCS y posteriormente CVS desde Emacs que permitía llevar a cabo operaciones de control de versiones de manera directa. El punto de partida fue un minúsculo y bastante crudo `scs.el.mode` que algún otro había escrito. Y el desarrollo de VC tuvo éxito debido a que, al contrario que el mismo Emacs, los programas en Lisp de Emacs permitían llevar a cabo ciclos de lanzamiento/prueba/mejora muy rápidamente.

(Un efecto colateral inesperado de la política de la FSF de mantener legalmente el código bajo la licencia GPL consiste en que su gestión dificulta la utilización del modelo bazar, ya que creen que deben conseguir una transferencia del copyright para cada contribución que supera las veinte líneas de código con el fin de blindar el código bajo esta licencia de cualquier amenaza proveniente de las leyes de propiedad intelectual. Aquellos que recurren a licencias del tipo BSD o del consorcio MIT X no sufren este problema por cuanto no tratan de reservar unos derechos que difícilmente alguien pudiera tener interés en poner en peligro).

4. Lánzalo pronto, lánzalo a menudo

Los lanzamientos tempranos y frecuentes de versiones de prueba son una parte crítica del modelo de desarrollo Linux. La mayoría de los programadores (incluyéndome a mí) solían creer que era una práctica perjudicial para el desarrollo de cualquier proyecto que no fuera simplemente trivial, por cuanto las versiones más tempranas están casi por definición condenadas a ser defectuosas y no parece conveniente agotar la paciencia de los usuarios.

Esta creencia reforzaba la adhesión generalizada al modelo de construcción de catedrales para el desarrollo de un programa. Si el objetivo prioritario consistía en exponer al usuario final al menor número posible de errores, no había razón para lanzar versiones de prueba en menos de seis meses (o incluso mucho menos a menudo) y tener que trabajar como un perro para depurarlas entre lanzamiento y lanzamiento. El núcleo en C de Emacs se desarrolló así. La librería en Lisp, sin embargo, no -- por cuanto existían depósitos activos de código en Lisp fuera del control de la FSF a los que se podía acudir para encontrar versiones nuevas y en desarrollo saltándose los ciclos de lanzamiento de Emacs.

El más importante de todos ellos, el depósito de 'elisp' de la Universidad Ohio State, anticipó el espíritu y muchas de las características de los grandes depósitos Linux de la actualidad. Pero pocos de nosotros prestamos atención a lo que estábamos haciendo, o a los problemas relacionados con el estilo de construcción de catedrales empleado por la FSF que la mera existencia de tales archivos implicaba. Hacia 1992, traté seriamente de incorporar formalmente una buena parte del código Ohio a la librería oficial de Emacs Lisp. Topé con problemas políticos y fue prácticamente un desastre...

Pero cosa de un año después, conforme Linux se hacía cada vez más visible, quedó claro que algo distinto y mucho más saludable estaba ocurriendo. La política de desarrollo abierto de Linus era justo lo contrario del estilo catedral. Los depósitos de 'sunsite' y 'tsx-11' estaban a rebosar, y aparecieron varias distribuciones. Y todo ello era el fruto de lanzamientos del núcleo del sistema a una frecuencia insólita.

Linus estaba tratando a sus usuarios como colaboradores de la forma más eficaz que imaginarse pueda:

7. Lánzalo pronto. Lánzalo a menudo. Y escucha a tus usuarios.

La innovación de Linus no consistió tanto en cómo hacer las cosas (algo parecido había sido ya tradición en el mundo Unix durante mucho tiempo), sino en llevarlo a una escala del mismo nivel que la complejidad del proyecto que estaba acometiendo. No era raro que en aquellos tiempos ¡lanzara más de una versión nueva del núcleo *al día*!. Y, debido a que cuidaba su base de colaboradores y usaba los recursos ofrecidos por Internet para la colaboración mejor que nadie, funcionó.

Pero *¿cómo funcionó?*. ¿Era algo que yo pudiera reproducir o dependía de alguna extraña habilidad de Linus?.

No lo creía así. Por supuesto que Linus es un hacker condenadamente capaz (¿cuantos de nosotros seríamos capaces de construir el núcleo completo de un sistema operativo con calidad a nivel de

producción?). Pero Linux no representaba ningún salto conceptual aterrador. Linus no es (o al menos no lo es todavía) un genio innovador del diseño a la manera en que lo son, por poner un par de ejemplos, Richard Stallman o James Gosling. Linus me parece más bien un genio de la ingeniería, con un sexto sentido para evitar errores y vías muertas en el desarrollo y una maravillosa capacidad para encontrar el camino de menor resistencia que conecta el punto A con el B. De hecho, el propio diseño de Linux respira esta cualidad y refleja esa aproximación conservadora y simplificadora tan propia de Linus.

Por tanto, si los lanzamientos rápidos y el empleo de Internet no eran accidentes sino partes integrales del genio ingenieril de Linus que le llevaba a localizar el camino de mínimo esfuerzo ¿qué era lo que estaba maximizando?. ¿Qué producía su maquinaria?.

Así planteada, la pregunta se contesta sola. Linus estaba manteniendo a sus colaboradores/usuarios en un continuo estímulo y una recompensa constante -- estimulados por la perspectiva de tener un trozo de la acción a su disposición para satisfacer su ego, y recompensados por la visión de una mejora continua (incluso *diaria*) de su trabajo.

Linus estaba intentando maximizar directamente el número de horas por persona dedicadas a la depuración y el desarrollo aún a costa de aumentar la inestabilidad del código y arriesgarse a quemar a su base de usuarios si algún error serio terminaba por ser imposible de solucionar. Linus se estaba comportando como si creyera algo parecido a esto:

8. Dada una base lo suficientemente amplia de probadores y colaboradores, casi todos los problemas se identificarán con rapidez y su solución será obvia para alguien.

O, expresado con menor formalidad, "Con un número de ojos suficiente, todos los errores son naderías". Yo lo llamo "La ley de Linus".

Al principio yo lo planteaba diciendo que todo problema "sería transparente para alguien". Linus observó que aquel que entiende y soluciona un problema no necesariamente, y ni siquiera habitualmente, es quien primero lo caracteriza. "Alguien encuentra un problema", dice, "y *algún otro* lo entiende. Y me atrevo a decir que la parte más difícil es encontrarlo". Pero lo importante es que ambas cosas tienden a ocurrir con rapidez.

Esta es, creo, la diferencia fundamental entre los estilos catedral y bazar. De acuerdo con la forma en que contempla la programación quien lo hace al modo en que se construye una catedral, los errores y los problemas de desarrollo son fenómenos insidiosos, profundos y retorcidos. Hacen falta meses de escrutinio por un pequeño número de gente dedicada a ello para poder confiar en que se hayan eliminado. De ahí los periodos largos requeridos para el lanzamiento de nuevas versiones, y el inevitable disgusto experimentado cuando las que tanto tiempo se han esperado no resultan perfectas.

A la luz del modelo bazar, en cambio, se supone que los errores son normalmente asuntos leves -- o, al menos, que se convertirán en tales con bastante rapidez en cuanto se expongan a los ojos ansiosos de algunos miles de colaboradores dedicados a poner del derecho y del revés cada nueva versión. Así que te dedicas a lanzar versiones con frecuencia para conseguir aún más correcciones, y como un beneficioso efecto colateral logras tener menos que perder si haces alguna chapuza de vez en cuando.

Y esto es todo. Y ya basta. Si la "ley de Linus" es falsa, entonces cualquier sistema tan complejo como el núcleo de Linux, producido artesanalmente y de manera aficionada, debería haberse colapsado en algún momento bajo el peso de interacciones perjudiciales no previstas y de errores profundos no descubiertos. Por otra parte, si es cierta, basta para explicar la relativa falta de defectos en Linux.

Y quizá no debiera haber sido una sorpresa. Hace algunos años que los sociólogos descubrieron que el promedio de las opiniones de un grupo de observadores igualmente experto (o igualmente ignorante) es un estimador bastante más fiable que el obtenido a partir de un grupo elegido aleatoriamente. Lo llaman "efecto Delphi". Parece que lo que Linus ha puesto de manifiesto es que lo anterior se aplica también a la depuración de un sistema operativo -- que el efecto Delphi puede mitigar la complejidad del desarrollo incluso al nivel de complejidad asociado al núcleo de un sistema operativo.

Tengo una deuda con Jeff Dutky <dutky@wam.umd.edu> por apuntar que la ley de Linus podía reformularse diciendo que "la depuración es paralelizable". Jeff hace notar que aunque la depuración requiere que los que la realizan se comuniquen a través de alguien que coordine el proceso, no precisa de una gran coordinación entre ellos. No está sujeta por tanto a ese aumento cuadrático de complejidad y costes que convierten en un problema la simple adición de programadores.

En la práctica, la pérdida teórica de eficacia debida a la duplicación del trabajo por los depuradores casi nunca parece ser un problema en el mundo Linux. Otro efecto de la política de lanzamientos tempranos y frecuentes consiste en minimizar tal redundancia de esfuerzos mediante la difusión rápida de correcciones ya realizadas.

Brooks incluso realizó una observación original relacionada con la de Jeff: ``El coste total de mantenimiento de un programa de uso generalizado viene a ser de un 40 por ciento o más del coste necesario para su desarrollo. Resulta sorprendente, sin embargo, que este coste depende fuertemente del número de usuarios. *Más usuarios implica que se encuentran mas errores.*" (El énfasis es mío).

Un número mayor de usuarios encuentra más errores debido a que añade muchas más formas diferentes de forzar el programa. Este efecto se amplifica cuando los usuarios colaboran en el desarrollo. Cada uno enfoca la caracterización de los errores con un esquema de percepción y un conjunto de herramientas ligeramente distinto, mirando el problema desde otro punto de vista. El "efecto Delphi" parece funcionar precisamente por causa de esta diversidad. En el contexto de la depuración de un programa, esta variación tiende también a reducir la duplicación de esfuerzos.

En consecuencia, añadir más personas al ciclo de prueba y depuración puede no reducir la magnitud del error más profundo existente en un momento dado desde el punto de vista *del que desarrolla el programa*, pero aumenta la probabilidad de que las herramientas de alguien resulten adecuadas a su detección de tal manera que dicho error resulte una nadería *para esa persona*.

Linus toma también algunas precauciones adicionales. Si es probable que existan errores importantes, las versiones del núcleo se numeran de tal forma que los usuarios potenciales puedan elegir entre emplear la última versión etiquetada como "estable" o pasear por el filo de la navaja exponiéndose a errores si desean nuevas posibilidades. La mayoría de los 'hackers' del mundo Linux no imitan aún formalmente esta táctica, pero quizá debieran hacerlo pues el hecho de disponer de ambas opciones las hace más atractivas.

5. ¿Cuándo una rosa deja de serlo?

Tras estudiar el comportamiento de Linus y desarrollar una teoría sobre los motivos de su éxito, decidí conscientemente comprobar su validez sobre mi nuevo (desde luego mucho menos complejo y ambicioso) proyecto.

Pero lo primero que hice fue reorganizar y simplificar 'popclient' en buena medida. La implementación de Carl Harris era muy brillante, pero mostraba cierta complejidad innecesaria bastante generalizada entre muchos programadores en C. Trataba el código como la parte más importante y las estructuras de datos como un mero apoyo. Como resultado, el código era magnífico pero las estructuras de datos se habían diseñado con descuido y resultaban bastante espantosas (al menos para los severos criterios impuestos por el antiguo 'hacker' en Lisp que les está narrando esto).

Por si no bastara, tenía otro propósito en mente para reescribirlo además de mejorar el código y el diseño de las estructuras de datos. Consistía en conducirlo a algo que yo entendiera por completo. No es divertido ser el responsable de la depuración de un programa que no comprendes.

Por tanto, más o menos durante el primer mes, me limité a aceptar las consecuencias derivadas del diseño original de Carl Harris. El primer cambio importante que hice fue añadir soporte para 'IMAP'. Para ello reorganicé los módulos relacionados con la gestión de protocolos (para POP2, POP3 e IMAP) convirtiéndolos en un controlador genérico y tres tablas para los métodos de gestión. Esto, y los cambios anteriores, ilustran un principio general que conviene tener en mente, especialmente en lenguajes que, como C, no conducen de forma natural al concepto de estructuras de datos dinámicas:

9. Estructuras de datos inteligentes asociadas a un código torpe funcionan mucho mejor que la alternativa opuesta.

Otra vez Fred Brooks, en el Capítulo 11: "Enséñame tu código y mantén ocultas tus estructuras de datos, y me seguirás engañando. Muéstrame tus estructuras de datos y normalmente no necesitaré que me enseñes tu código: resultará evidente."

En realidad, él decía "organigramas" y "tablas". Pero pasándolo por el filtro de treinta años de terminología y evolución cultural, viene a ser lo mismo.

En este momento (a principios de Septiembre de 1996, unos seis meses después de haber empezado) empecé a pesar en la conveniencia de un cambio de nombre -- al fin y al cabo, ya no era sólo un cliente POP. Pero dudaba, pues todavía no había nada realmente nuevo en el diseño. Mi versión de 'popclient' aún tenía que desarrollar su propia personalidad.

La situación cambió radicalmente cuando 'fetchmail' aprendió a enviar el correo recogido a una puerta SMTP. Pronto veremos eso. Pero, en primer lugar, ocupémonos de lo siguiente: dije antes que decidí usar este proyecto para poner a prueba mi teoría sobre qué había hecho bien Linus Torvalds. Puedes estarte preguntando ¿cómo lo llevé a cabo?. De las siguientes maneras:

1. Lancé versiones pronto y con frecuencia (casi nunca menos de una versión cada diez días y, en periodos de desarrollo intenso, una diaria).
2. Hice crecer mi lista de probadores de las versiones beta añadiendo a ella a todo aquel que se puso en contacto conmigo interesándose por 'fetchmail'.
3. Envié anuncios seductores a la lista de personas incluidas en la prueba de las betas siempre que lancé una nueva versión, animando a la gente a que participara.
4. Y escuché lo que ellos tuvieron que decir, manteniéndoles al corriente de las decisiones tomadas sobre el diseño y agradeciendo su colaboración siempre que enviaban correcciones o informes sobre el funcionamiento del programa.

La recompensa obtenida a cambio de acciones tan sencillas fue inmediata. Desde que el proyecto comenzó, dispuse de informes sobre errores de una calidad tal que la mayoría de los desarrolladores matarían a alguien por conseguirla, a menudo con correcciones propuestas para solucionarlos. Logré críticas razonadas, conseguí correo animándome a proseguir, me llegaron un montón de sugerencias sobre cosas a añadir. Lo que nos lleva a proponer que:

10. Si tratas a la gente que te ayuda a depurar un programa como si fueran tu recurso más valioso, responderán convirtiéndose en eso precisamente.

Una forma interesante de medir el éxito de 'fetchmail' consiste simplemente en mirar el tamaño de la lista de personas que participaron en las pruebas en la fase beta del desarrollo, de los que apoyaron 'fetchmail'. Cuando escribo esto, tiene 249 miembros y sigue creciendo al ritmo de dos o tres por semana.

De hecho, al revisar esto a principios de Mayo de 1997, la lista está perdiendo miembros por una razón interesante. Algunos me han pedido que los retire de ella por cuanto 'fetchmail' les funciona tan bien ¡que no necesitan seguir ya su evolución!. Quizá esta sea la forma natural en que evoluciona un proyecto maduro en el estilo bazar.

6. Popclient se convierte en Fetchmail

El punto a partir del cual el programa cambió de verdad fue cuando Harry Hochheiser me envió su versión inicial para redirigir el correo a una puerta SMTP. Me di cuenta casi inmediatamente de que la implementación fiable de esta posibilidad hacía de los demás modos de reenvío algo casi obsoleto.

Había estado modificando 'fetchmail' durante varias semanas de forma progresiva pero no dejaba de notar que el diseño de la interfase era utilizable pero poco presentable -- muy poco elegante y con demasiadas opciones de uso poco frecuente dando vueltas por ahí. Las opciones para volcar el correo recogido a un fichero 'mailbox' o a la salida estándar me preocupaban de manera especial, pero no terminaba de entender la razón.

Lo que ví cuando pensé sobre el reenvío a un servidor SMTP era que 'popclient' había intentado hacer demasiadas cosas. Había sido diseñado para ser a la vez un agente de transporte de correo ("mail transport agent" MTA) y un agente de envío de correo local ("local delivery agent" MDA). Con el reenvío a un servidor SMTP, podía salir del mundo MDA y convertirse en un MTA auténtico, encargándose de remitir el correo a otros programas para su entrega local del mismo modo en que lo hace 'sendmail'.

¿Qué necesidad hay de lidiar con la complejidad de configurar un agente de envío de correo o con el bloqueo y la adición a un fichero 'mailbox' si existe una garantía casi total de que la puerta 25 se encuentre disponible desde el principio en cualquier plataforma que soporte TCP/IP?. En especial cuando eso significa que el correo que se recoja tiene la garantía de aparecer como un correo normal SMTP remitido desde el lugar de origen, que es lo que en verdad andamos buscando.

Hay varias lecciones que extraer de aquí. En primer lugar, que esta idea de realizar un reenvío al servidor SMTP fue la mayor contribución individual que obtuve al intentar imitar los métodos de Linus. Un usuario me dió una idea tan brillante -- y todo lo que tuve que hacer por mi parte fue comprender sus implicaciones.

11. La siguiente cosa mejor que tener buenas ideas consiste en reconocer las buenas ideas de tus usuarios. Y en ocasiones ésta última es la mejor en términos absolutos.

Resulta bastante interesante observar que uno se da cuenta enseguida de que si se es completa y generosamente verídico acerca de cuanto le debes a los demás, el resto de la humanidad te tratará como si fueras el responsable de hasta la última porción de un invento y considerará que estás siendo simplemente modesto respecto a tu indiscutible genialidad. ¡No hay más que ver lo bien que este efecto funcionó con Linus!

(Cuando presenté esta comunicación en la conferencia Perl en Agosto de 1997, Larry Wall estaba sentado en la primera fila. Al llegar a la última línea citada anteriormente, exclamó: "Dí que sí, dí que sí, tío". Todos los presentes rieron, porque sabían que también le había ocurrido exactamente igual al inventor de Perl).

Y después de unas pocas semanas de conducir el proyecto de esta forma, comencé a recibir un reconocimiento similar no solo de mis usuarios, sino también de otra gente a la que le habían

llegado noticias indirectamente. Guardé a buen recaudo algunos de aquellos mensajes de e-mail; y los leeré de nuevo si alguna vez me pregunto si mi vida ha servido para algo útil.

Pero hay aquí dos lecciones fundamentales más, nada políticas, que pueden aplicarse a cualquier clase de diseño.

12. A menudo, las soluciones más sorprendentes e innovadoras surgen al darte cuenta de que la idea que se tenía del problema estaba equivocada.

Había tratado de resolver el problema equivocado al continuar desarrollando 'popclient' como una combinación MTA/MDA capaz de realizar cualquier suerte de entrega local del correo por extraña que pudiera resultar. El diseño de 'fetchmail' debía ser rehacerse por completo, planteándolo como un MTA puro, como una parte más del camino habitual del correo en Internet regulado mediante diálogos SMTP.

Cuando topas con una pared en un desarrollo dado -- cuando te ves obligado a pensar más allá de cual va a ser el próximo parche -- suele ser el momento de plantearte no si tienes la respuesta adecuada, sino si estás respondiendo la respuesta correcta. Quizá el problema necesita ser replanteado.

Bueno, había replanteado mi problema. Estaba claro que lo más adecuado era (1) transformar el soporte para redireccionar el correo mediante SMTP en el driver genérico, (2) fijarlo como el modo de funcionamiento por defecto, y (3) eliminar cuando fuera posible las demás formas de entrega de correo, en especial las opciones de entrega a fichero y a la salida estándar.

Estuve dudando algún tiempo sobre la conveniencia de poner en práctica la tercera etapa, temiendo perjudicar a aquellos antiguos usuarios de 'pop-client' que dependían de los modos alternativos de entrega de correo. En teoría, podrían pasar inmediatamente a emplear ficheros '.forward' o sus equivalencias alternativas al empleo de 'sendmail' para conseguir los mismos resultados. Pero en la práctica, la transición hubiera podido crearles un cierto lío.

Pero cuando terminé por hacerlo, los beneficios fueron enormes. La parte más problemática del driver desapareció. La configuración se hizo muchísimo más simple -- ya no había necesidad de andar buscando el MDA del sistema o el fichero 'mailbox' del usuario, ni era preciso preocuparse de si el sistema operativo sobre el que se ejecutaba soportaba o no el bloqueo de ficheros.

Además, desapareció la única posibilidad de que se perdiera el correo. Si intentabas entregar el correo en un fichero y el espacio en disco se acababa, lo perdías. Algo que no puede pasar mediante la redirección en SMTP por cuanto el receptor SMTP no devolverá un OK a menos que el mensaje se haya entregado o al menos se haya puesto en lista de espera para entregarlo en cuanto sea posible.

Por otra parte, la velocidad aumentó (aunque no tanto como para poderlo notar en un uso ocasional). Otro beneficio nada insignificante del cambio consistió en que la página 'man' del programa se hizo mucho más sencilla.

Más adelante, hube de reintroducir la posibilidad de realizar la entrega a través de un MDA local para poder trabajar en algunas oscuras situaciones relacionadas con el uso de SLIP dinámico. Pero encontré una manera mucho más sencilla de hacerlo.

La moraleja?. No dudes en eliminar posibilidades ya caducas cuando sea posible hacerlo sin pérdida de efectividad. Antoine de Saint-Exupery (que fue aviador y diseñador de aeroplanos antes que autor de libros clásicos para niños) decía que:

13. "La perfección (de un diseño) no se consigue cuando no queda nada por añadir, sino más bien cuando no resta nada por eliminar."

Cuando tu código se hace mejor y más sencillo, es cuando *notas* que es correcto. El diseño de 'fetchmail' adquirió su propia identidad en este proceso, una distinta a la de su antepasado 'popclient'

Había llegado el momento de cambiarle el nombre. El nuevo diseño se parecía más a un asistente de 'sendmail' de lo que lo había sido el antiguo 'popclient'; ambos son del tipo MTA, pero mientras 'sendmail' remite los mensajes y luego los envía, el nuevo 'popclient' los recoge y luego los envía. Tras dos meses de esfuerzos, le cambié el nombre a 'fetchmail'.

7. Fetchmail se hace mayor

Aquí estaba yo con un diseño nuevo e innovador, un código que sabía que funcionaba bien porque lo usaba todos los días, y una lista de desarrolladores desbordante. Me fui dando cuenta poco a poco de que ya no se trataba sólo de un proyectito personal que quizá les resultara útil a unas pocas personas más. Tenía en mis manos un programa que todo 'hacker' con un ordenador Unix y una conexión de correo del tipo SLIP/PPP necesitaba de verdad.

Con la posibilidad de realizar el reenvío a través de SMTP, se adelantó lo suficiente a la competencia como para poder llegar a ser el programa dominante de este tipo, uno de esos clásicos que cumplen su cometido tan bien que las demás alternativas no sólo se descartan sino que casi caen en el olvido.

Creo que no es posible hacer planes para llegar a un resultado así. Te ves conducido a él por ideas para el diseño tan potentes que el resultado posterior parece simplemente inevitable, natural, incluso premeditado. La única forma de encontrar este tipo de conceptos es disponer de un montón de ideas -- o teniendo el juicio y la habilidad ingenieril para llevar las ideas de los demás más lejos de lo que ellos creían posible.

Andrew Tanenbaum tuvo la idea original de construir un Unix nativo sencillo para el 386, con el fin de usarlo en enseñanza. Linus Torvalds llevó el concepto Minix probablemente mucho más lejos de lo que Andrew creía que podía llevarse -- y llegó a ser algo admirable. Del mismo modo (aunque a menor escala), cogí algunas ideas de Carl Harris y Harry Hochheiser y las desarrollé considerablemente. Ninguno de nosotros fue 'original' en el sentido romántico que mucha gente asocia a un genio. Sin embargo, la mayor parte de la ciencia, la ingeniería o el desarrollo del software no es realizada por genios, aunque la mitología 'hacker' mantenga lo contrario.

El resultado fue de todas formas bastante trascendental -- de hecho, fue el tipo de éxito que todo 'hacker' ansía lograr. Y eso hacía que tuviera que hacer mis estándares aún más exigentes. Para hacer 'fetchmail' tan bueno como ahora sabía que podía llegar a ser, tenía que escribir no solo para cubrir mis necesidades, sino también incluir y dar soporte a características necesarias para otros pero que quedaban fuera de mi ámbito. Y hacerlo al tiempo que mantenía la sencillez y robustez del programa.

La primera característica, y con diferencia la más importante, que añadí a continuación, fue el soporte para entregas múltiples -- la posibilidad de recoger el correo de puestos que habían acumulado todo el correo de un grupo de usuarios y enviar cada uno de los mensajes a su destinatario particular.

Decidí añadirlo en parte porque algunos lo estaban pidiendo con insistencia, pero ante todo porque creí que contribuiría a sacar a la luz los errores que pudieran quedar en el modo normal de entrega simple al obligarme a poner en funcionamiento un sistema de entrega completamente general. Y así ocurrió. Conseguir implementar un intérprete de órdenes acorde a la norma RFC822 me llevó una considerable cantidad de tiempo, no porque hubiera algún trozo especialmente problemático, sino por un auténtico montón de detalles confusos e interdependientes.

Pero la decisión de soportar entregas múltiples resultó al final excelente. Me topé con lo siguiente:

*14. Toda herramienta debe resultar útil en la forma prevista, pero una *gran herramienta* te lleva a usarla para realizar cosas jamás pensadas.*

Este uso imprevisto para la capacidad de realizar entregas múltiples consiste en gestionar listas de correo manteniéndolas y realizando la conversión de los alias en el lado *cliente* de una conexión SLIP/PPP. Esto permite que cualquiera que esté empleando una máquina personal conectada a Internet a través de una cuenta en un proveedor de las más corrientes pueda gestionar una lista de correo sin tener que acceder de manera continua a las listas de alias del proveedor de acceso a Internet.

Otro cambio importante que mis colaboradores pedían era el soporte para la codificación de mensajes MIME 8-bits. Resultó bastante fácil debido a que había tenido bastante cuidado en que el programa no interfiriera con su utilización. No es que me anticipara a esta petición, sino porque trataba de seguir otra regla importante:

*15. Cuando escribas programas que actúen como pasarelas de datos ('gateway software'), ten cuidado de modificarlos lo menos posible -- y *nunca* elimines información a menos que su destinatario te fuerce a hacerlo.*

Si no hubiera seguido esta regla, dar soporte a mensajes en código MIME 8-bits hubiera resultado difícil y propenso a errores. En cambio, lo único que hizo falta fue leer la norma RFC 1652 y añadir un poco más de lógica a la generación de cabeceras. Resultó bastante trivial.

Algunos usuarios europeos insistieron en que añadiera una opción para limitar el número de mensajes recogidos en cada sesión (de modo que pudieran controlar los costes ocasionados por sus carísimas líneas telefónicas). Me resistí durante mucho tiempo, y aún no estoy demasiado contento con el resultado. Pero si se escribe para todo el mundo, tienes la obligación de escuchar a tus clientes -- nada cambia a este respecto aunque no te estén pagando con dinero.

8. Algunas lecciones más a partir de 'fetchmail'.

Antes de que volvamos a asuntos generales de diseño de software, no estaría de más valorar un par de lecciones puntuales.

La sintaxis de los ficheros 'rc' incluye una serie de órdenes opcionales que pueden ser consideradas como 'ruido' y son ignoradas por el intérprete. Permiten el empleo de una sintaxis más próxima al inglés que resulta mucho más legible que la tradicional, formada por pares orden-dato, que es a lo que se reduce si se eliminan.

Empezaron como un experimento a altas horas de la noche cuando caí en la cuenta de que las declaraciones de ficheros en los archivos 'rc' comenzaban a parecer un pequeño lenguaje en tono imperativo. (Es también la razón por la que cambié la orden 'server' original de 'popclient' a 'poll') *(Nota del traductor: No se han traducido por cuanto las órdenes de 'fetchmail' siguen en inglés).*

Me pareció que aproximar ese minilenguaje imperativo al inglés lo haría de uso más sencillo. Ahora, aunque soy un francotirador convencido de la corrección de la escuela que predica el "crea un lenguaje" al modo de Emacs, HTML y muchos de los motores de las bases de datos, no soy tan entusiasta de las sintaxis que se aproximan al inglés.

Los programadores han tendido tradicionalmente a favorecer sintaxis de control que eran muy precisas y compactas y carecían de redundancias. Es una herencia cultural de los tiempos en que los recursos de computación eran caros, lo que aconsejaba que las etapas de interpretación de órdenes fueran lo más eficaces y simples que se pudiera lograr. El inglés, con un 50% aproximadamente de redundancia, parecía un modelo poco apropiado para estos lenguajes.

No es esta la razón de mi modesta oposición a las sintaxis que tratan de asemejarse al inglés; la cito sólo para negarla. Con los costes de computación de hoy en día, la simplicidad no debe ser un fin en sí mismo. En este momento es más importante que un lenguaje resulte fácil de usar para los humanos que eficaz para el ordenador.

Sin embargo, hay razones para ser prudente. Una es el coste en términos de complejidad de la etapa de interpretación -- no es deseable llevarlo a un punto en que sean una causa importante de errores y confusión del usuario. Otra consiste en que intentar aproximar un lenguaje al inglés suele requerir que el inglés que se termine utilizando se vea seriamente deformado, de modo que esa semejanza superficial al lenguaje natural acaba por resultar tan confusa como lo hubiera sido la sintaxis original. (Es algo que puede verse en muchos '4GL' y lenguajes comerciales para acceder a bases de datos).

La sintaxis de control de 'fetchmail' parece evitar estos problemas debido a que el dominio en que se aplica es extremadamente restringido. No se parece en nada a un lenguaje de propósito general; como las cosas que dice no son nada complicadas hay poco lugar para la confusión al pasar mentalmente de un pequeño subconjunto del inglés al lenguaje de control real. Me parece que puede haber una lección general a extraer de eso.

16. Si el lenguaje de tu programa no es Turing-completo ni por asomo, puede venir bien endulzar su sintaxis.

Otra lección trata sobre el intento de llegar a la seguridad mediante la oscuridad. Algunos usuarios de 'fetchmail' me pidieron que cambiara el programa para que las contraseñas se guardaran encriptadas en el fichero 'rc', de modo que los mirones no fueran capaces de leerlas.

No lo hice, por cuanto no añade protección alguna. Cualquiera que consiga los permisos necesarios para leer tu fichero 'rc' podrá ejecutar 'fetchmail' en tu lugar -- y si lo que busca es tu contraseña, será capaz de extraer el decodificador del código de 'fetchmail' para lograrla.

Todo lo que se hubiera logrado al codificar las contraseñas en '.fethcmailrc' hubiera sido proporcionar una falsa sensación de seguridad a aquellos que no piensan demasiado. La regla general es:

17. Un sistema es sólo tan seguro como su secreto. Cuidado con los falsos secretos.

9. Condiciones previas necesarias para el modelo bazar

Los primeros jueces y audiencias de este trabajo plantearon constantemente preguntas sobre las condiciones previas necesarias para el éxito de un desarrollo que empleara el modelo bazar, relativas tanto a la calificación del líder del proyecto como al estado en que debe encontrarse el código al ponerlo a disposición del público para intentar formar un grupo de colaboradores.

Está bastante claro que uno no puede empezar desde cero en el modelo bazar. Se puede poner a prueba, depurar y mejorar en el estilo bazar, pero parece difícil *iniciar* un proyecto con este modelo. Linus no lo intentó. Yo tampoco. Tu naciente grupo de colaboradores necesita algo que pueda ejecutar y probar para jugar con él.

Cuando empiezas a reclutar un grupo, hace falta que puedas presentarle una *promesa plausible*. Tu programa no tiene que trabajar particularmente bien. Puede ser tosco, incompleto, estar lleno de errores y escasamente documentado. Pero no puede dejar de convencer a los potenciales colaboradores sobre su capacidad de evolucionar hasta convertirse en algo brillante en un futuro no demasiado lejano.

Tanto Linux como 'fetchmail' se pusieron a disposición del público con diseños básicos robustos y atractivos. Mucha gente que ha pensado sobre el modelo bazar como yo lo he expuesto ha considerado acertadamente que esto era crítico y ha pasado de un salto a concluir que el líder del proyecto necesita contar con un alto grado de inteligencia e intuición para el diseño.

Pero Linus tomó su diseño de Unix. Yo cogí el mío inicialmente del ancestral 'popmail' (aunque luego cambió en buena medida, proporcionalmente mucho más que Linux). Por tanto ¿hace falta de verdad que el líder/coordinador de un esfuerzo en el modelo bazar tenga un talento excepcional para el diseño, o puede bastar con aprovechar el talento de los demás?.

No creo que la capacidad del coordinador para crear diseños de excepcional brillantez resulte crítica, pero sí lo es su capacidad para *reconocer las buenas ideas de diseño de los demás*.

Tanto Linux como 'fetchmail' muestran evidencias de esto último. Mientras que Linus no es (como ya discutimos) un programador espectacularmente original, ha exhibido un potente don para reconocer buenos diseños e incorporarlos al núcleo de Linux. Y yo ya he contado como la idea más potente en el diseño de 'fetchmail' (el reenvío SMTP) la proporcionó otra persona.

Los primeros oyentes de este trabajo me alagaron al sugerir que tendía a infravalorar la originalidad de los diseños en el modelo bazar porque yo tengo mucha, y en consecuencia la daba por descontada. Puede ser cierto; el diseño (en lugar de la programación o la depuración) es desde luego mi punto fuerte.

Pero el problema de ser listo y original a la hora de diseñar software consiste en que se convierte en un hábito -- empiezas deliberadamente a hacer las cosas brillantes y complicadas cuando deberías tratar de mantenerlas robustas y sencillas. He llevado bastantes proyectos a la ruina al cometer esta equivocación, pero conseguí que no me ocurriera con 'fetchmail'.

Así que creo que el proyecto 'fetchmail' tuvo éxito en parte porque contuve mi tendencia a exhibir mi brillantez, y eso (al menos) parece oponerse a que para que un proyecto al estilo bazar tenga éxito sea preciso un alto grado de originalidad en el diseño. Y pensemos en Linus. Supongamos que Linus Torvalds hubiera intentado introducir innovaciones fundamentales en el diseño de sistemas operativos durante el desarrollo; ¿es plausible que el núcleo resultante fuera tan estable y exitoso como el que tenemos?.

Hace falta un cierto nivel de capacidad de diseño y programación, por supuesto, pero creo que casi cualquiera que piense seriamente en lanzar un proyecto siguiendo el estilo bazar estará por encima de ese mínimo. El mercado de reputaciones ligado a la comunidad del software abierto ejerce una sutil presión sobre la gente para no embarcarse en esfuerzos que no sean capaces de llevar a cabo. Hasta el momento, esto parece haber funcionado bastante bien.

Para tener éxito en el estilo bazar, hay otro tipo de habilidad que no se asocia normalmente con el desarrollo de software que creo es tan importante como la lucidez para el diseño -- y quizá lo sea más, incluso. El que coordina un proyecto en el modelo bazar debe ser capaz de comunicarse con la gente.

Esto último debería resultar obvio. Para formar una comunidad de desarrollo, es preciso atraer a la gente, interesarlos en lo que estás haciendo, y mantenerlos contentos con el trabajo que están haciendo. Tu habilidad técnica cuenta bastante para lograrlo, pero no es desde luego toda la historia. Tu personalidad también cuenta.

No es casualidad que Linus sea un tipo agradable que le cae bien a la gente y al que no les importa ayudar. No es coincidencia que yo sea un extrovertido enérgico al que le encantan las multitudes y que posee ciertos instintos más propios de un cómico que otra cosa. Para que el modelo bazar funcione, ayuda enormemente que seas un poquito habil cuando se trata de agradar a la gente.

10. El contexto social del software abierto

Lo que se dijo era cierto: los mejores desarrollos surgen al abordar soluciones personales a los problemas cotidianos del autor, y se difunden porque el problema afecta también a un amplio grupo de usuarios. Esto nos devuelve a la regla 1, que ahora podemos reformular de una forma quizá más útil:

18. Para resolver un problema interesante, comienza por encontrar uno que lo sea para tí.

Fue el caso de Carl Harris y el 'popclient' original, y también el mío con 'fetchmail'. Pero es algo que se sabía de antiguo. La parte importante, aquella que los casos de Linux y 'fetchmail' ponen de manifiesto es la etapa siguiente -- la evolución del software cuando hay una comunidad amplia y activa de usuarios y colaboradores.

En el libro ``The Mythical Man-Month" (*Nota del traductor: Se mantiene el título en inglés de los libros citados a continuación porque se cree que no han sido traducidos y editados en España. Al no disponer de datos que permitan acceder a la versión castellana en caso de que exista, se prefiere conservar el título original para facilitar al menos el acceso a la versión inglesa*), Fred Brooks observó que el tiempo que alguien dedica a programar no es un consumible más; añadir más programadores a un proyecto que se desarrolla con retraso lo retrasa aún más. Lo justificó diciendo que la complejidad y los costes de comunicación crecían con el cuadrado del número de personas implicadas, en tanto el trabajo realizado lo hacía linealmente. Este enunciado pasó a ser la "ley de Brook" y suele considerarse un axioma. Pero si la ley de Brooks fuera la verdad absoluta, Linux sería imposible.

Algunos años más tarde el clásico de Gerald Weinberg ``The Psychology Of Computer Programming" proporcionó lo que puede verse como una corrección fundamental a Brooks. En su discusión sobre "la programación sin ego", Weinberg hizo notar que en aquellos lugares en que los programadores no desarrollan un sentido de la propiedad sobre su propio código, y animan a los demás a buscar errores y posibles mejoras, el desarrollo progresa a una velocidad dramáticamente superior a la habitual.

La terminología empleada por Weinberg ha dificultado quizá que su análisis se acepte a la escala que merecía -- uno no puede sino sonreír cuando se describe a los hackers de Internet como desprovistos de ego. Pero creo que sus argumentos suenan mucho más plausibles hoy que nunca.

La historia de Unix nos debería haber preparado para lo que estamos aprendiendo con Linux (y que yo he comprobado a menor escala al copiar deliberadamente sus métodos). Es decir, que aunque la programación sea una actividad solitaria, los auténticos logros surgen de la puesta en común de la atención y la capacidad intelectual de comunidades enteras. Aquel que dependa tan sólo de su cerebro al desarrollar un sistema va estar siempre en desventaja frente al que sepa cómo crear un ambiente abierto y en evolución en el cual la búsqueda de errores y las mejoras se confían a cientos de personas.

El mundo Unix tradicional se vió imposibilitado de llevar este enfoque hasta el extremos debido a varias razones. Una fue la limitación impuesta por los diversos tipos de licencia de uso, y los

secretos e intereses comerciales. Otra (podemos pensarlo ahora) fue que Internet todavía no estaba lo bastante desarrollada.

Antes de que Internet fuera ampliamente accesible, había algunas comunidades geográficamente compactas en las que el ambiente cultural alentaba esa programación sin ego propuesta por Weinberg, donde un programador podía atraer con facilidad a un montón de colaboradores y virtuosos del bit. Los laboratorios Bell, el laboratorio de inteligencia artificial del MIT, la universidad de Berkeley -- se convirtieron en el hogar de innovaciones que aún son legendarias y potentes.

Linux fue el primer proyecto que se esforzó de manera consciente y exitosa en usar el mundo *entero* como su depósito intelectual. No creo que sea una mera coincidencia que el periodo de gestación de Linux viera al tiempo el nacimiento de la World Wide Web, y que Linux dejara atrás su infancia en el mismo periodo 1993-1994 en que despegó la industria de los proveedores de acceso a Internet y se produjo la explosión del interés generalizado por Internet. Linus fue la primera persona que aprendió a jugar con las nuevas reglas que la disponibilidad de Internet ponía en juego.

Una Internet accesible era condición necesaria para la evolución de Linux, pero no creo que fuera suficiente. Otro factor vital fue el desarrollo de un estilo de liderazgo y unos hábitos de cooperación que fueran capaces de atraer colaboradores para extraer los máximos frutos del nuevo medio.

Pero, ¿en qué consisten?. No pueden basarse en relaciones de poder -- y aunque así fuera, el liderazgo por coacción no produciría los resultados que vemos. Weinberg cita la autobiografía del anarquista ruso del siglo diecinueve Kropotkin "Memorias de un revolucionario" de manera adecuada a este tema:

``Al crecer en una familia que disponía de siervos, empecé mi vida activa, como todos los hombres de mi tiempo, teniendo una gran confianza en la necesidad de ejercer el mando, administrar el castigo o la coacción y demás. Pero cuando, poco después, hube de hacerme cargo de empresas importantes y tratar con hombres [libres], y cuando cada error tenía consecuencias graves, comencé a apreciar la diferencia entre actuar bajo el principio del mando y la disciplina o el del entendimiento común. El primero funciona admirablemente en un desfile militar, pero carece de valor en la vida real, donde un objetivo sólo puede lograrse a través del esfuerzo concertado de muchas voluntades".

El "esfuerzo concertado de muchas voluntades" era precisamente lo que necesitaba un proyecto como Linux -- y el "principio de mando" resulta imposible de utilizar en el contexto de un grupo de voluntarios en el paraíso anarquista que denominamos Internet. Para trabajar y competir con eficacia, los hackers que quieren desarrollar un proyecto en colaboración deben aprender a reclutar y motivar a la gente en base a intereses comunes al modo vagamente sugerido por Kropotkin con su "principio de comprensión mutua". Deben aprender a usar la ley de Linus.

Anteriormente me referí al "efecto Delphi" como una posible explicación de la ley de Linus. Analogías más potentes con los sistemas adaptativos en biología y economía surgen irresistiblemente por sí solas. El mundo Linux se asemeja en muchos aspectos a un mercado libre o un sistema ecológico, a una colección de agentes autónomos que intentan maximizar la utilidad en un proceso que termina conduciendo a un orden derivado de la autocorrección espontánea mucho más eficiente y elaborado de lo que hubiera podido lograr cualquier cantidad de planificación. Es en estos términos en los que hay que hablar del "principio de comprensión mutua".

La "función de utilidad" que los hackers de Linux están maximizando no es económica en sentido clásico, sino un intangible derivado de su grado de satisfacción personal y de la reputación adquirida ante los demás hackers. (Podríamos hablar de altruismo, pero sería ignorar que éste es en sí mismo una forma a través de la cual el altruista satisface su ego personal). Las culturas que funcionan voluntariamente mediante este esquema no son infrecuentes; otra en la que he participado con asiduidad es el colectivo de los entusiastas de la ciencia ficción, en la cual, al contrario de lo que ocurre en el mundo hacker, se reconoce explícitamente el culto al ego (el aumento de tu reputación dentro del colectivo) como la motivación fundamental que justifica la actividad de tipo voluntario.

Linus, al saber colocarse como administrador de un proyecto en que el desarrollo era principalmente realizado por los demás, y alimentando el interés en él hasta que fué capaz de automantenerse, mostró un notable grado de percepción del "principio de comprensión mutua" de Kropotkin. Esta visión cuasi-económica del mundo Linux nos permite ver el modo en que se aplica este conocimiento.

Podemos ver el método de Linus como una forma de crear un mercado eficaz movido en torno del culto al ego -- una forma de conectar la individualidad de los hackers tan firmemente como fuera posible para llevarla a culminar objetivos difíciles solo alcanzables mediante una colaboración sostenida. Con el proyecto 'fetchmail' he mostrado (aunque a menor escala) que su método se puede reproducir con buenos resultados. Quizá yo lo he hecho de una forma un poco más consciente y sistemática que él.

Mucha gente (en particular aquellos que por razones políticas no confían en el libre mercado) esperaría que la cultura de un grupo de egoístas autónomos resultara fragmentada, territorial, despilfarradora, secretista y hostil. Pero estas predicciones son inequívocamente contradichas por (sólo por poner un ejemplo) la asombrosa variedad, calidad y profundidad de la documentación de Linux. Nadie en su sano juicio niega que los programadores *odian* documentar sus obras; ¿como es posible, entonces, que los hackers en Linux produzcan tanta?. Resulta evidente que el libre mercado de culto al ego en Linux funciona mejor en este aspecto que la financiación masiva para la producción comercial de títulos de referencia de los fabricantes de software.

Tanto 'fetchmail' como el núcleo Linux demuestran que recompensando adecuadamente los egos de muchos otros hackers, un desarrollador/coordinador puede usar Internet para capturar los beneficios derivados de tener muchos colaboradores sin que el proyecto colapse en un desorden caótico. Por lo cual propongo lo siguiente, en contra de la ley de Brooks:

19: Si el coordinador de un proyecto tiene a su disposición un medio de comunicación al menos tan potente como Internet, y sabe como conducir a la gente sin coaccionarla, muchas cabezas son inevitablemente mejor que una.

Creo que el futuro del software abierto pertenecerá cada vez más a gente que sepa como jugar al modo de Linus, a gente que deje atrás la catedral y abrace el bazar. Esto no quiere decir que la visión y la brillantez individual ya no importen; al contrario, creo que los proyectos más trascendentales en el mundo del software abierto serán los de aquellos que comiencen a partir de la visión y la brillantez individual y amplifiquen su importancia mediante la construcción eficaz de grupos con intereses comunes.

Y tal vez no sólo el futuro del software *abierto*. Ningún fabricante comercial de software puede igualar el depósito de inteligencia que la comunidad Linux puede movilizar sobre un problema dado. ¡Muy pocos podrían permitirse siquiera alquilar a las más de doscientas personas que colaboraron en el desarrollo de 'fetchmail'!

Quizá al final la cultura del software abierto triunfe no porque la colaboración sea moralmente "buena" y la posesión avara del software "mala" (suponiendo que creas esto último, algo que ni yo ni Linus hacemos), sino simplemente porque el mundo comercial no puede ganar una carrera de armamentos frente a comunidades de software abierto capaces de movilizar cantidades de tiempo de personal cualificado algunos órdenes de magnitud superiores a la hora de resolver un problema.

11. Agradecimientos

Este trabajo se mejoró a través de conversaciones con un gran número de personas que ayudaron a perfeccionarlo. Mi especial agradecimiento a Jeff Dutky <dutky@wam.umd.edu>, que sugirió el planteamiento de que "la depuración es paralelizable", y ayudó a desarrollar el análisis que de ella se deriva. También a Nancy Lebovitz <nancyl@universe.digex.net> por su sugerencia de que imitara a Weinberg citando a Kropotkin. Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> también me proporcionó críticas penetrantes junto a Marty Franz <marty@net-link.net> de la lista de General Technics. Paul Eggert <eggert@twinsun.com> hizo notar el conflicto entre GPL y el modelo bazar. Agradezco también a los miembros del PLUG, el grupo de usuarios de Linux de Philadelphia, el haberme proporcionado la primera audiencia que escuchó la versión pública de este trabajo. Y para terminar, la ayuda proporcionada por los comentarios de Linus Torvalds y su apoyo inicial que tanto me animó.

12. Lecturas adicionales

Se ha citado varias veces el clásico de Frederick P. Brooks's *The Mythical Man-Month* debido a que, en muchos aspectos, sus observaciones aún no han sido superadas. Recomiendo ardientemente la edición del 25 aniversario realizada por Addison-Wesley (ISBN 0-201-83595-9), que le añade su trabajo de 1986 ``*No Silver Bullet*".

La nueva edición se ve completada por una inestimable retrospectiva del tema 20 años después de haber sido propuesto en la que Brooks aborda directamente los pocos juicios del texto original que no han soportado el paso del tiempo. Leí por primera vez dicha retrospectiva tras haber completado prácticamente este trabajo, y me sorprendió que Brooks ¡atribuya prácticas similares al estilo bazar a Microsoft!.

El libro *The Psychology Of Computer Programming* (New York, Van Nostrand Reinhold 1971) de Gerald P. Weinberg introdujo el desafortunado término de "programación sin ego". Aunque no fuera desde luego la primera persona en darse cuenta de la futilidad del "principio de mando", sí fue probablemente la primera en reconocer su importancia y discutirlo en relación con el desarrollo del software.

Richard P. Gabriel, al estudiar la cultura Unix en la época anterior a Linux, se vió forzado a argumentar sobre la superioridad de un primitivo modelo bazar en su trabajo de 1989 *Lisp: Good News, Bad News, and How To Win Big*. Aunque haya quedado desfasado en algunos aspectos, este ensayo es todavía muy celebrado entre los incondicionales de Lisp (en los que me incluyo). Un corresponsal me recordó que la sección titulada "Peor es lo mejor" puede considerarse casi como una anticipación de Linux. Este trabajo puede conseguirse a través de Internet en la dirección siguiente <http://alpha-bits.ai.mit.edu/articles/good-news/good-news.html>.

El libro de De Marco y Lister *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) es una joya menospreciada que me encantó ver citada por Fred Brooks en su retrospectiva. Aunque poco de lo que proponen los autores pueda aplicarse a las comunidades Linux o a la del software abierto, la visión del autor al estudiar las condiciones precisas para realizar un trabajo creativo resulta aguda y valiosa para cualquiera que intente tomar algunas de las virtudes del modelo bazar con el fin de aplicarlas en un contexto más comercial.

Finalmente, debo admitir que estuve muy cerca de titular este trabajo "La catedral y el ágora", un título en el que "ágora" indica el vocablo griego para designar un mercado abierto o una plaza pública. Los trabajos seminales de Mark Miller y Eric Drexler sobre lo que ellos denominaban "sistemas agóricos", en los que describían las propiedades emergentes de ecologías computacionales similares a las de un mercado, me ayudaron a pensar con claridad sobre fenómenos análogos en la cultura del software abierto cuando, cinco años más tarde, Linux me metió de cabeza en él. Estos trabajos se pueden conseguir asimismo a través de Internet en <http://www.agorics.com/agorpapers.html>.

13. Historial de versiones y cambios:\$Id: cathedral-paper.sgml,v 1.28 1998/01/29 05:47:17 esr Exp \$I 1.16 presentada en el Linux Kongress, 21 Mayo 1997.

La bibliografía se añadió en la versión 1.20 el 7 de Julio de 1997.

La anécdota de la conferencia Perl se añadió a la versión 1.27 el 18 de Noviembre de 1997.

Otras versiones incluyen pequeñas correcciones de errores tipográficos y de formato.