

# Desarrollo en comunidad con eXtreme Programming

José Dapena Paz  
jdapena@igalia.com

Igalia, C/Gutenberg 34B, 2º, A Coruña 15008, Spain,  
info@igalia.com,  
WWW home page: <http://www.igalia.com>

**Resumen** El objetivo de esta contribución es la descripción de buenas prácticas y herramientas para el desarrollo de software libre, centrándonos en aquellas que facilitan la participación de la comunidad. Dentro de estas técnicas, se profundizará en las metodologías ligeras, y en particular, en *eXtreme Programming*. A continuación se detallarán herramientas y técnicas que permiten mejorar la comunicación en un proyecto de software libre, tanto con desarrolladores, como con usuarios, así como mejorar su visibilidad.

## 1. Introducción

El modelo de Bazar, documentado por Eric S. Raymond en su libro *La catedral y el Bazar* (ver [1]), y de enorme popularidad en el mundo del software libre, describe cómo un modelo iterativo de desarrollo, llevado por una comunidad descentralizada, podía llevar adelante un proyecto de software de gran complejidad.

El texto de Raymond se centra en dos ejemplos: el núcleo del sistema operativo Linux, y la utilidad para obtener correos remotos Fetchmail. El primero es un proyecto de enorme complejidad y tamaño, que ha sido realizado con contribuciones de multitud de desarrolladores, bajo la dirección (a veces ejecutiva, a veces con menor implicación) de Linus Torvalds. El segundo, Fetchmail, fue desarrollado por él mismo, como prueba experimental de cómo funciona el desarrollo en comunidad, y por tanto se utiliza de ejemplo principal.

El modelo de Bazar, tiene un conjunto de características especiales, dentro de los procedimientos de desarrollo de software. La principal es la enorme dispersión de los desarrolladores, y la dedicación impredecible de éstos al proyecto. Veremos durante el artículo cómo se puede aprovechar este hecho para facilitar la creación de buenos proyectos, mediante la mejora de las infraestructuras y la comunicación.

### 1.1. Mitos del desarrollo en comunidad

El planteamiento de Bazar, en el que un proyecto se desarrolla a partir de las contribuciones (voluntarias o no) de un conjunto suficiente de personas, da lugar a varios mitos y falacias:

- El software libre está desarrollado por aficionados.
- Desarrollo *hacker*, y soluciones *ad hoc*. Ausencia de metodologías.
- El éxito de un proyecto es más casualidad que algo predecible.
- Es imposible implantar medidas para el control de la calidad de un proyecto.

De forma resumida, todos estos puntos se resumen en uno: el mito de que es imposible hacer *Ingeniería del software libre*, y productos de calidad profesional.

En los siguientes puntos expondré los motivos para considerar que estos mitos no tienen un fundamento real.

### 1.2. ¿Cosas de aficionados?

Si bien es cierto que los proyectos de software libre que tienen un cierto éxito, tienen una participación significativa de voluntarios y aficionados, los productos de más éxito de software libre no se hacen por amor al arte.

La gestión y elaboración de un proyecto es una tarea extremadamente compleja, que exige un alto nivel de dedicación y conocimientos. Un proyecto puede llegar a ser complejo y grande en la medida en que existe suficiente gente que puede dedicar su tiempo completo a él. Los núcleos de los principales y más conocidos proyectos de software libre son llevados por profesionales, financiados por universidades o empresas. Estos profesionales se dedican a sus tareas a tiempo completo.

Por tanto, el desarrollo de un proyecto puede tener un coste económico significativo, debido a la necesidad de remunerar a un conjunto de desarrolladores. La dedicación a tiempo completo suele exigirlo (¡de algo tiene que vivir el desarrollador!).

La búsqueda de financiación es un aspecto muy importante a tener en cuenta para el desarrollo de un proyecto de software libre, por tanto. Fundaciones y organizaciones sin ánimo de lucro, pueden actuar como mediadoras para captar la financiación de diferentes entidades.

Proyectos como Gnome y Apache son gestionados por fundaciones. Y el peso de la dirección y gestión de estos proyectos es llevado por profesionales que trabajan para diferentes entidades.

### 1.3. ¿Sin metodología?

Si observamos los principales proyectos de software libre, nos encontramos que a medida que van madurando se incorporan técnicas y métodos altamente considerados en la ingeniería del software. Esto no es casual.

El desarrollo en comunidad ha permitido que el software libre incorpore como normales técnicas y herramientas de ingeniería de software como la gestión de requisitos, control de versiones, repositorios de incidencia, utilización de patrones de diseño y análisis, y múltiples formas de mejorar la comunicación entre los desarrolladores.

Si bien es cierto que existe una fuerte componente *ad hoc* en las soluciones iniciales, las comunidades de desarrolladores con necesidades heterogéneas llevan a refactorizaciones excelentes, para poder cubrir las necesidades de todos.

#### 1.4. El éxito no es casual

El éxito de un proyecto de software libre se basa principalmente en conseguir una comunidad activa de desarrolladores y usuarios. Este éxito no es casual, y existen métodos para facilitar que suceda.

La técnicas que facilitan el incremento de la base de usuarios y desarrolladores son una mezcla de *Marketing* y la gestión de una infraestructura adecuada. Hay que saber *vender* el proyecto, y es fundamental cuidar a los posibles interesados.

#### 1.5. Se puede controlar la calidad

Es un hecho que hay muchos proyectos de software libre que responden mejor a los requisitos de calidad que productos propietarios validados.

Esto se debe a que la comunidad del software libre ha desarrollado técnicas que gestionan de forma transparente la calidad, a través de la gestión de errores, control de versiones, métricas, autodocumentación o pruebas.

En los principales proyectos de software libre podemos encontrar la implantación de las mejores técnicas de control de calidad.

#### 1.6. A continuación

Durante este artículo describiré algunas técnicas y métodos que permiten obtener un proyecto de software libre de éxito, y de gran calidad.

## 2. eXtreme Programming

En esta sección describiré un enfoque metodológico que encaja muy bien en el desarrollo de proyectos de software libre en comunidad muy popular, denominado *eXtreme Programming*.

*eXtreme Programming* es un conjunto de técnicas y prácticas para el desarrollo de software. Se encuadra dentro de la familia de *metodologías ligeras*, tratando de obtener métodos sencillos de obtener software de calidad.

Sus principios básicos son dos: la mejora de la comunicación con los usuarios, para retroalimentar el proceso de desarrollo; y obtener cuanto antes un programa que haga algo, partiendo de esto para ir añadiendo incrementalmente nuevas características.

Estos objetivos encajan exactamente con los de el desarrollo de software libre en comunidad. La mejora de los mecanismos de comunicación es una forma excelente de cuidar a la comunidad de usuarios y desarrolladores. Y el segundo punto coincide con el *release early, release often* del modelo de Bazar: es más fácil conseguir adeptos cuando hay algo que se pueda usar e instalar. Y lanzar versiones nuevas con frecuencia aumenta la visibilidad pública del proyecto.

Estas técnicas se aplican a proyectos con un equipo de desarrollo medio-grande, para solucionar un problema no trivial. Algunas de las medidas que proponen no tienen sentido para proyectos pequeños. La propia formulación de ésta recomienda que no se apliquen aquellas que van a entorpecer el proyecto.

Las técnicas de *eXtreme Programming* (*XP* a partir de ahora) se dividen en cuatro ámbitos: planificación, diseño, codificación y pruebas.

## 2.1. Planificación

En el ámbito de planificación de un proyecto, las técnicas que se sugieren son las siguientes:

1. *Historias de usuarios*. El desarrollo es dirigido por una descripción informal de las necesidades de usuarios (dos o tres frases), que se denominan *historias*. Orientan el proceso de desarrollo (los objetivos que se planifican son el cumplimiento de estas historias), y las pruebas de aceptación (se pueden establecer métodos más o menos formales para verificar que ya se cubre una *historia*). Una historia se divide en varias tareas planificables y medibles en su estado de realización.
2. *Sacar nuevas versiones con frecuencia*. Es necesario definir un plan de versiones, con una estricta planificación temporal. Una vez que se decide cuándo se lanzarán versiones, se decide qué historias se van a implementar en cada versión. Este plan indicará, por tanto, diferentes iteraciones del proyecto y qué se debe implementar en cada una de ellas. La realización de este plan debe tener en cuenta en la medida de lo posible, las prioridades de los usuarios, para satisfacerles en mayor medida.
3. *Iteraciones*. Se recomienda un proceso de desarrollo iterativo de ciclo corto. El primer objetivo es obtener algo funcional cuanto antes. Una vez que se consigue, se comienza a implementar de forma incremental nuevas historias de usuario en las sucesivas iteraciones, tal y como indica el plan. Un aspecto importante es que al desarrollar una iteración, los desarrolladores se deben centrar en los objetivos de esa iteración, y no en los de las próximas. Las especificaciones de iteraciones futuras pueden cambiar, y por tanto podemos llegar a hacer un trabajo innecesario. Por tanto, mejor es centrarse en implementar estrictamente lo que se pide en la iteración actual, y confiar en la refactorización para incluir características en el futuro. Es importante hacer notar que los objetivos de las iteraciones se replanifican al principio de cada una, para permitir ajustar en mayor medida éstas a la realidad del proyecto.
4. *Trabajo en equipo*. Es deseable cambiar a los desarrolladores de tareas en el proyecto, de tal forma que hayan realizado tareas en la mayor parte posible de módulos. Esta estrategia favorece la cohesión del equipo de desarrollo, y el aprendizaje. El objetivo es disminuir las dependencias necesarias del proyecto (nunca debe haber una sola persona que sepa hacer algo).

En el desarrollo de proyectos de software libre, es muy recomendable el lanzamiento frecuente de versiones. Pero la aportación principal de estas técnicas es la forma de orientar los objetivos de las versiones. Para implementar esto, se puede utilizar el repositorio de fallos e incidencias (por ejemplo, Bugzilla), etiquetando las tareas como "Peticiones de mejora". Estos informes se etiquetarán como objetivos de cada versión. También se pueden utilizar herramientas de gestión de tareas específicas de *XP*, como *XPtracker*, que veremos más tarde.

¿Cómo se encajan las contribuciones voluntarias en esta estrategia? Simplemente no se hace. Generalmente un voluntario va a hacer lo que él quiere o le interesa, y por tanto, no podemos imponer tareas. En caso de solicitar él mismo alguna tarea, se le deben asignar historias de iteraciones futuras, para no interferir en el cumplimiento de los objetivos actuales. La no previsibilidad del cumplimiento de tareas por voluntarios hace que sea poco recomendable implicarlos en objetivos estrictos.

## 2.2. Diseño

1. *Regla KISS.* KISS es *Keep It Simple, Stupid!*. En *XP* se huye de las soluciones complejas. Si existe algo excesivamente complejo en el proyecto, y se puede sustituir por algo más simple, debe hacerse. Se desea obtener una solución exactamente tan eficaz como necesitemos, ni más ni menos.
2. *Nomenclatura.* La comunicación del equipo necesita un vocabulario común. Medidas para conseguir esto son: mantener una buena documentación de los conceptos que se manejen, actualizar frecuentemente un diccionario de datos, buscar un espacio de nombres que facilite la comprensión rápida de los conceptos involucrados, o el uso de patrones, que nos proporcionan un lenguaje común para manipular conceptos abstractos de diseño.
3. *Seguir estrictamente el plan.* Solo implementaremos lo que se haya decidido hacer en el plan para la iteración actual, y nada más que eso. Evita que nos influya tanto los cambios de especificaciones (no haremos nada que pueda ser tirado abajo por un cambio de especificaciones posterior). Si sobra tiempo, se dedicará a mejorar la solución a los objetivos actuales.
4. *Refactorización.* Aunque debemos huir de las soluciones complejas, genéricas, que cubran los casos que no nos interesan, buscamos siempre la solución más simple. Por tanto, cuando encontremos puntos en común con otras historias ya implementadas, debemos refactorizar. Se obtendrá una solución común que cubra ambos casos, partiendo de la historia ya implementada. Así obtendremos una solución más general a partir de requisitos reales. ¿Cuándo refactorizar? Cuando la solución refactorizada sea más simple que las dos soluciones por separado.

En software libre, debemos tener en cuenta que quizá haya otras soluciones en otros proyectos que se adapten bien a nuestros problemas. Es importante realizar un pequeño trabajo de investigación, y reutilizar y refactorizar código e incluso proyectos. Un ejemplo de esto puede ser FreeDesktop, que aglutina la refactorización de esfuerzos para cumplir objetivos comunes a diferentes entornos de escritorio libres.

## 2.3. Codificación

1. *Comunicación con el usuario.* Debe favorecerse la comunicación con el usuario, e intentar involucrarlo lo más posible en las etapas de desarrollo, proporcionándole la capacidad de probar cómo se van implementando sus necesidades.

2. *Estándares de codificación.* Debe mantenerse un estándar claro sobre cómo programar, para facilitar la colaboración de múltiples personas sobre una base de código común. Estos estándares deben ser públicos.
3. *Programación en pareja.* Solo aplicable cuando hay desarrolladores próximos geográficamente. Consiste en realizar sesiones de programación en las cuales dos personas comparten un equipo. Es una de las ideas más llamativas de XP. Una de las personas programa, manteniendo una visión más centrada en el nivel de codificación, mientras que otra observa y asesora al que programa, manteniendo una visión de conjunto. Facilita la refactorización, la detección de errores de diseño, y en general se basa en la idea de que son más productivos dos programadores de esta forma que trabajando por separado.
4. *Pruebas de unidad.* Es muy recomendable añadir pruebas de unidad para todo el código desarrollado. Estas pruebas deben ser automatizables. El objetivo es poder determinar en cualquier momento si una característica sigue funcionando.
5. *Integración continua.* Es recomendable establecer una infraestructura que compile y pruebe automáticamente todo el proyecto, y alerte cuando no suceda así. De esta forma, se consigue detectar rápidamente cuándo un árbol de compilación no es utilizable, e impide que otros desarrolladores se queden bloqueados esperando a que se solucione el fallo introducido.
6. *Propiedad colectiva del código.* Cualquier miembro del equipo puede aportar ideas, corregir errores, o refactorizar el código de otras personas, impidiendo que una persona sea un cuello de botella. La integración continua y las pruebas de unidad facilitan que estas modificaciones no introduzcan fallos en las funcionalidades ya implementadas.
7. *Sólo optimizar cuando sea necesario.* Mejor centrarse en cumplir objetivos reales. Si un objetivo necesita una mayor eficiencia, se miden los cuellos de botella y se optimizan. Nunca se debe hacer una solución más compleja si no es necesario, y esto cubre también las tareas de optimización.
8. *Cuidado con hacer muchas horas.* El desarrollo de software es una tarea compleja, así que el desarrollador debe estar en buenas condiciones físicas. Los excesos de horas perjudican a la calidad del proyecto. Si una iteración requiere horas extra para ser acabada, está mal planteada y se debe replanificar. Si esto afecta a la planificación global del proyecto, se deben revisar los objetivos de éste. La hora extra es algo excepcional y de lo que se debe huir.

#### **2.4. Pruebas**

Indicamos antes que las historias dan lugar a unas pruebas de aceptación formales en XP. Estas pruebas se hacen de acuerdo con el cliente, y sólo se considerará una historia acabada cuando se cumplan. Por tanto, en una relación comercial se debe formalizar en qué van a consistir estas pruebas. En un proyecto de software libre, los objetivos los marca una comunidad, así que no es tan

interesante formalizarlos. Interesa más facilitar a los usuarios la aportación de opiniones sobre cómo se están cubriendo sus necesidades.

Sin embargo, las pruebas de unidad sí son útiles. Cada módulo que se desarrolle puede tener una batería de pruebas de unidad que comprueben que funciona correctamente a lo largo del tiempo. Esto combinado con un proceso de integración continua permite conocer el grado de funcionamiento del proyecto completo. Si las pruebas de unidad se crean antes de desarrollar las funcionalidades que prueban, también valen como indicativo de cuánto falta para completar una historia.

Existe multitud de herramientas para realizar pruebas de unidad en software libre. Por ejemplo, *test* en *C*, o *phpunit* en *PHP*.

### 3. Comunicación con la comunidad

Debemos cuidar a nuestros desarrolladores y usuarios. Para ello la mejor herramienta es la mejora de la comunicación en ambas direcciones.

#### 3.1. Relación con el usuario

Un proyecto no sólo debe desarrollarse, debe parecerlo también. Por tanto es fundamental informar continuamente de qué se está haciendo y cómo. Esta información debe ser personalizable, y por tanto, son personas concretas con nombres y apellidos las que hacen las aportaciones, aún manteniendo la afiliación con alguna empresa o entidad.

También es importante lanzar nuevas versiones con frecuencia, así como anunciar los planes para versiones futuras y cumplir los plazos (incluso más importante que cumplir los objetivos). En cualquier caso el usuario debe enterarse de la actividad del proyecto, así que es importante anunciar las nuevas versiones por los canales adecuados.

Es fundamental cuidar la comunicación con el usuario también cuando la iniciativa es suya, principalmente porque eso demuestra que es un usuario ya interesado en el proyecto. Por tanto, es recomendable esforzarse en responder rápidamente a las preguntas, peticiones, informes de bugs, . . . Deben establecerse herramientas para facilitar esta comunicación, y priorizar adecuadamente las tareas que éstos soliciten. También es bueno dar medios adecuados para que la base de usuarios financie parte del proyecto mediante donaciones.

#### 3.2. Relación con el desarrollador

Alguien que desee colaborar en el proyecto, debe tenerlo muy fácil para empezar. Para ello, y para la relación posterior dentro de él, es necesario mantener documentación amplia y actualizada en los niveles de análisis, diseño e implementación del proyecto.

Debe favorecerse la contribución de opiniones, y no ocultar los debates sobre las decisiones del proyecto. Realizar este tipo de debates públicamente permite

que otros desarrolladores participen y aporten ideas. Dos cabezas piensan mejor que una. Sin embargo se debe tener bastante tacto para evitar guerras entre desarrolladores.

## 4. Herramientas

El software libre proporciona ya herramientas adecuadas para implementar estos métodos. En esta sección se exponen algunas de ellas:

**CVS** Repositorio de código con control de versiones, módulos y ramas.

**Bugzilla** Base de datos de informes de error, clasificados por módulos. Permite asignarles una prioridad y una importancia.

**Bonsai** Herramienta para realizar consultas o navegar por un árbol de CVS, desde web. Facilita la consulta del estado de módulos concretos del código sin tener que acceder al CVS.

**Tinderbox** Herramienta de integración continua distribuida.

**tWiki y Docbook** Sistemas de documentación concurrente.

**xPtracker** Es un módulo de tWiki que permite documentar un proceso de XP tal y como se ha descrito en este artículo.

**Listas de correo** Principal forma de comunicación de un equipo de desarrollo, caracterizadas por ser asíncronas y catalogables.

**Web del proyecto** Fachada pública del proyecto, y primer punto de entrada a él.

### 4.1. CVS

*CVS*, o *Concurrent Versions System*, es el sistema de control de versiones dominante en el desarrollo de aplicaciones *Open Source*. Es un repositorio central de código, que permite el trabajo de múltiples desarrolladores sobre un árbol de código, manteniendo información histórica sobre la evolución de los ficheros.

Permite el acceso remoto y autenticado de los desarrolladores a través de internet, y tiene un mecanismo simple de resolución de conflictos en el caso de manipulación de un mismo fichero por varios desarrolladores. Además, permite mantener varias ramas de un proyecto, así como mantener información sobre versiones concretas publicadas. En él se puede almacenar además del código, documentación como manuales de usuario, ayuda en línea, o manuales de desarrollo.

*CVS* es viejo, y tiene defectos graves, como puede ser la carencia de información sobre el estado de los directorios de un proyecto o la imposibilidad de renombrar ficheros sin perder su histórico de modificaciones. Sin embargo, también tiene un amplio conjunto de herramientas, que permiten dar soporte a integración continua (*Tinderbox*, navegación y consultas a través de web (*Bonsai*), e integración de estas herramientas con el repositorio de incidencias *Bugzilla*.

## 4.2. Bugzilla

*Bugzilla* es una aplicación web que permite almacenar y gestionar incidencias sobre productos software (e incluso otro tipo de proyectos y actividades). Almacena descripciones de las incidencias, así como cierta metainformación, como puede ser el estado de resolución, prioridad, o gravedad. Esto sirve para poder gestionar los errores que tiene el proyecto software, así como poder realizar un seguimiento de estas. También se puede explotar la información contenida, para poder calcular tiempos medios de resolución de incidencias, número de incidencias pendientes, estadísticas de actividad de los desarrolladores, etc.

Es el principal sistema de gestión de incidencias del panorama *open source*, siendo utilizado por proyectos tan conocidos como *Mozilla* (a partir del cual surgió el proyecto), o *Gnome*.

Es bastante flexible, permitiendo definir nueva metainformación asociada a cada incidencia. Por ejemplo, podemos decidir almacenar no solo las incidencias, si no las tareas pendientes de un proyecto. Además, se puede clasificar mediante etiquetas las incidencias por diferentes aspectos. Así, también podríamos almacenar las historias de usuario, y asignarlas a diferentes iteraciones mediante etiquetas.

## 4.3. Bonsai

*Bonsai* es otra de las herramientas de apoyo al desarrollo surgidas dentro de *Mozilla*. Consiste en una interfaz web que permite realizar consultas sobre un repositorio CVS, y los cambios que se han producido en él. Además, permite la consulta de los ficheros de CVS, pudiendo consultar las diferencias entre versiones y la gestión de diferentes ramas.

Utilizando una herramienta como Bonsai, se pueden hacer consultas concretas sobre el código, sin tener que realizar el proceso de descargarse una copia del repositorio.

## 4.4. Tinderbox

*Tinderbox* es un sistema cliente servidor de integración continua. Los clientes se dedican a realizar compilaciones en diferentes plataformas, y el servidor agrega la información de las compilaciones y las presenta mediante una interfaz web. De esta forma, podemos compilar periódicamente un proyecto, y consultar el resultado de esa compilación.

Se integra con CVS y Bonsai, de forma que podemos averiguar los cambios que han sucedido entre compilaciones. Es importante esto, ya que nos permite determinar qué cambios han provocado un error en la compilación. Además, también alerta cuándo se ha producido un fallo en la compilación, pudiendo así los desarrolladores saber cuándo se ha introducido un error de este tipo, y qué cambios han sido los causantes. También se integra con Bugzilla, pudiendo saber qué bugs se han marcado como resueltos en cada compilación.

La primera ventaja importante es que se puede mantener con más facilidad el repositorio de código funcional y utilizable. Los problemas de compilación que no causa uno mismo son causa de grandes retrasos en la integración del código. La segunda ventaja es que podemos integrar en el proceso de compilación las pruebas de unidad, y así podemos detectar inmediatamente cuándo se introduce un error que provoca que estas fallen. La tercera ventaja, debida a la arquitectura cliente servidor que establece *Tinderbox* es que podemos mantener el proyecto en diferentes plataformas y detectar errores en plataformas concretas más rápido.

#### 4.5. Documentación con TWiki y Docbook

Un wiki permite elaborar documentación en un entorno distribuido de forma rápida y cómoda, y con cierta estructuración. Su interfaz web hace a un wiki idóneo para el desarrollo de documentación rápida, y sin grandes procesos de compilación y formateado. Por tanto, es muy adecuado para almacenar documentos en pleno desarrollo y evolución. Por ejemplo propuestas de análisis, información de tareas, etc. *TWiki* en particular, es una de las implementaciones más flexibles de la idea de wiki.

Sin embargo, el uso de los wikis permite también una excesiva anarquización de los contenidos, además de ser difícil la integración de documentación almacenada en wiki en otros sistemas, como una ayuda en línea, o libros impresos. Para estas otras tareas, podemos utilizar, dentro del repositorio de código, estándares de documentación estructurada como puede ser *Docbook*. Estos documentos, elaborados de forma más cuidadosa normalmente, tienen siempre el carácter intencional de ser formales. *Docbook* está basado en *XML*, y nos permite almacenar documentación de forma que es fácil su exportación a web, o a papel escrito. También muchos sistemas de ayuda en línea están integrados con *docbook*, como los del proyecto *Gnome*.

#### 4.6. xPTracker

*xPTracker* es un módulo de *TWiki* que permite realizar la planificación y gestión de un proyecto realizado mediante *eXtreme Programming*. Se organiza mediante el almacenamiento de proyectos, equipos de desarrollo, iteraciones, historias de usuario y tareas. Para cada proyecto existe un conjunto de equipos de desarrollo, y cada uno de los equipos cuenta con su planificación en base a iteraciones.

Cada una de las iteraciones tiene un plazo establecido de ejecución, y varias historias a completar, que a su vez se dividen en tareas. Con *xPTracker* es bastante sencillo realizar el seguimiento de cómo se está ejecutando una iteración, el estado de completación de cada una de sus historias, así como las desviaciones sobre la planificación. Además, al estar dentro del wiki, se puede añadir abundante documentación sobre cómo se ha realizado cada actividad. Todo esto permite la coordinación de un equipo de desarrollo grande, de forma cómoda, algo ideal en software libre.

#### 4.7. Listas de correo

Mediante el uso de listas de correo, se puede organizar la discusión de diferentes asuntos del proyecto, así como la atención a los usuarios no desarrolladores. Al ser una forma de comunicación asíncrona y abierta, permite una comunicación cómoda y no muy exigente entre los miembros de la comunidad. No muy exigente ya que uno lee el correo cuando puede, y sobre todo en el caso de voluntarios, tiene que ser fácil que puedan participar en debates, sin una exigencia de planificación temporal, que sí puede ser necesaria en otras formas de comunicación, como reuniones presenciales o chats.

La otra gran ventaja de las listas de correo, es que son en sí una forma de documentación de incidencias, y debates. Estableciendo una interfaz web para acceder al archivo de los correos existentes, se puede consultar los debates que ya han sucedido, y hacer búsquedas en ellos, facilitando la entrada en el proyecto a nuevos desarrolladores.

#### 4.8. Web del proyecto

Elemento fundamental. Debe servir de punta de partida para acceder a los demás recursos del proyecto, facilitando en especial saber qué hace el proyecto y poder seguir el estado actual de desarrollo. También debe facilitar la información necesaria para poder comenzar a contribuir en el proyecto.

Por tanto, todas las herramientas que hemos mencionado, deben estar ampliamente documentadas en la web. También debe ser dinámica, y dar información *al minuto* del estado del proyecto. Así, un desarrollador nuevo, podrá conocer en poco tiempo los entresijos del proyecto, un desarrollador habitual podrá consultar rápidamente cómo están yendo las tareas e incidencias, y un usuario podrá leer cómo se usa, y realizar fácilmente consultas e informes de errores.

Una buena web del proyecto, además, debe cumplir con la función de *convencer* a nuevos usuarios y desarrolladores. Es la fachada de un proyecto, y si es atractiva, cómoda y funcional, facilitará que haya una mayor comunidad de desarrolladores en el proyecto.

### 5. Conclusión

Las técnicas de *eXtreme Programming* nos proporcionan un camino para obtener productos de calidad medible en el desarrollo de proyectos de software libre. El hecho de que sea una metodología ligera, la hace especialmente idónea para un entorno heterogéneo con un grupo grande de desarrolladores. Como vemos, son diferentes técnicas que se caracterizan por tratar de minimizar su mantenimiento funcional (nada de grandes tareas de análisis ciego, nada de sistemas complejos de gestión de requisitos, nada de grandes arquitecturas para pequeños proyectos).

Estas técnicas, y los procedimientos adecuados de comunicación entre los desarrolladores y usuarios, facilitan por un lado aumentar el conjunto de personas

interesadas en el proyecto, y por otro lado, la calidad de la comunicación y colaboración entre estas personas.

## Referencias

1. Raymond, E.: The Cathedral and the Bazaar  
<http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/>
2. Wells, D.: Extreme Programming: A Gentle Introduction  
<http://www.extremeprogramming.org>
3. The Mozilla Organization: Development Tools  
<http://www.mozilla.org/tools.html>
4. Concurrent Versioning System: The open standard for version control  
<http://www.cvshome.org>
5. Wikipedia: The free encyclopedia  
<http://www.wikipedia.org>
6. Peter Thoeny and others: TWiki - An Enterprise Collaboration Platform  
<http://twiki.org>