# Pequeña guía de introducción al desarrollo colaborativo de software libre en red.

**Oscar Miras Ortiz**
oscar.miras@gmail.com

Abril 2010

**Resumen**

Presentamos en este documento una pequeña ayuda para empezar a colaborar en el desarrollo de software libre en red. Identificaremos los pasos básicos para encontrar un proyecto acorde a nuestros intereses, cómo obtener información de él, y averiguar las formas posibles de colaboración. Discutiremos como llevar a cabo nuestros objetivos dentro del proyecto seleccionado, y las diferentes aplicaciones existentes que nos ayudarán en su desarrollo. Esta guía está basada en la experiencia personal de su autor tras el desarrollo de su proyecto final de carrera, relacionado con la temática que aquí se trata.

## 1. Introducción y motivación

Cuando queremos empezar a crear software, es posible que algunas de las funcionalidades del programa que queremos realizar ya estén implementadas, en mayor o menor medida, por otros programas. ¿Por que realzar de nuevo un trabajo que ya está hecho?

Usando software libre, esto es, con acceso al código fuente, podemos encontrar un programa acorde a nuestra necesidades para usarlo; y ampliar o modificar alguna de sus funcionalidades; para suplir nuestras necesidades, y aportar nuestra creación de nuevo a la comunidad de desarrollo.

Tu equipo de trabajo no tienen por que ser tus compañeros de oficina. Están distribuidos por todo el mundo. Por ello, nos valdremos de varias herramientas para comunicarnos y coordinar el trabajo.

Esta guía pretende responder a las preguntas que toda persona interesada en el desarrollo de software libre puede hacerse alguna vez:

- ¿Por donde empiezo?
- ¿Que software de partida escoger?
- ¿Cual puede coordinarme con un equipo de desarrollo?

- ¿Qué herramientas voy a usar?

# 2. Software libre de partida

## 2.1. ¿Qué queremos hacer?

Ésta es la pregunta principal, ¿qué quieres hacer?. Y, ¿que te ha motivado a colaborar en el desarrollo de software libre?

1. **Ampliar las funcionalidades de un programa en concreto.** Conocemos un programa el cuál nos gusta tal como funciona, pero nos gustaría añadir algunas funcionalidades nuevas.
2. **Remodelar un programa por completo.** Es posible que los cambios que queremos realizar sobre un programa existente, sean tan profundos que cambiarían el funcionamiento del programa considerablemente.
3. **Diseñar un programa y ver si algún programa de software libre implementa sus funcionalidades o parte de ellas.** Tenemos en mente la idea de un programa, lo especificamos o incluso lo podemos diseñar. Entonces, podemos buscar si algún software libre implementa algunas de las funcionalidades de nuestro programa.
   *a*) Normalmente, ningún software implementará al 100 % lo que queremos realizar. Una vez encontremos un programa parecido al que habíamos diseñado, podremos ver que funcionalidades echamos a faltar de las que teníamos pensadas.
   *b*) Si diera la casualidad que hallaramos un software que hiciera **exactamente** lo que queremos realizar...¡magnífico!. Eso significa que nuestra idea es compartida por mucha gente, por lo que puede ser una buena idea. Una vez probado el programa, podemos pensar nuevas funcionalidades a incorporar.
4. **Ampliar las funcionalidades de una biblioteca.**. Es posible que hayamos estado usando una biblioteca para desarrollar un programa, pero echemos a faltar alguna característica. Podemos contactar con el equipo de desarrollo de dicha biblioteca para pedir que implementen la característica que deseamos, o implementarla nosotros mismos.

## 2.2. Buscando un software libre de partida

Éste apartado es necesario para el supuesto 3 de la sección anterior, en el que buscamos un software que se adapte a uno que previamente hemos especificado o diseñado. Si el lector ya dispone del software a ser ampliado, puede pasar a la sección siguiente.

Buscamos programas que implementen partes de las funcionalidades que hemos diseñado. Seguramente, varios programas y/o bibliotecas pueden ser buenos candidatos o candidatas como software de partida, con lo que en éste apartado aconsejaremos sobre como buscar y valorar el software candidato.

### 2.2.1. Granularidad y naturaleza de la solución

Podemos enfocar la búsqueda del software desde tres puntos de vista básicos:

1. Buscar un programa que implemente casi todas las funcionalidades que queremos. De éste modo, se espera que todas las herramientas (funciones, clases, estructuras, etc.) que ofrezca el programa, sean suficientes para implementar los cambios que deseemos. Imaginar que queremos crear un navegador gráfico de archivos pero que además, previsualice una fotografía cada vez que encuentra un archivo de imagen. Buscaremos entonces un explorador de ficheros de propósito general, e implementaremos en él la funcionalidad deseada.

2. Buscar diversos programas que implementen, modularmente, algunas de las funcionalidades que estamos buscando. Volviendo al ejemplo del navegador de archivos, esto equivaldría a buscar un software que se encargara de recorrer directorios y obtener los atributos de los ficheros; y por ejemplo, otro que nos permitiera mostrar gráficamente elementos por pantalla para poder reproducir el árbol de directorios.

3. Usar bibliotecas para desarrollar nuestro programa. Consideramos las bibliotecas como pequeñas herramientas que podemos usar para desarrollar nuestro programa; siendo así de hecho un software libre de partida; pues también las podemos modificar.

Comentar que estos puntos no tienen que ser una decisión inicial inamovible. Es posible que durante el desarrollo de nuestro software encontremos que debemos combinar varias soluciones para conseguir nuestro objetivo. Sin embargo, si deberíamos tener una inicial de qué vamos a usar. Lo ideal, es que con esta elección se cubra la mayoría de las funcionalidades que queremos implementar (a ser posible todas).

Si a medida que desarrollamos nuestro software van apareciendo nuevas necesidades que nuestra elección inicial no cubre, tenemos la posibilidad de usar otros programas. Sin embargo, no siempre es fácil adaptar estos programas a nuestra solución inicial; pues depende de varios factores. Esto también es aplicable si decidimos implementar el programa a partir de diversos programas, tal como comentábamos anteriormente:

1. ¿Las estructuras de datos que manejan los diferentes programas son adaptables entre si? Tendremos más posibilidades de combinar diferentes soluciones si los programas elegidos usan estructuras ampliamente usadas y soportadas. Por ejemplo, es más reusable una aplicación gráfica maneje las imágenes usando una estructura GDKPixbuff [1]; que no una propia que él mismo se haya creado.

2. ¿Es factible separar los módulos que nos interesan de nuestro programa inicial para combinarlo con el nuevo? Es posible que encontremos un programa completo del cual solo nos interesa una pequeña parte de sus funcionalidades. No siempre es fácil separar esa parte de resto de programa, ya que pueden existir muchas dependencias entre módulos para implementar dicha funcionalidad que nos interesa, lo que puede hacer muy difícil separar esa parte del resto del programa; y valdrá más la pena buscar la solución por otro lado.

3. ¿Causa algún tipo de conflicto el hecho de utilizar varios programas a la vez para conseguir nuestra solución? Imaginar que queremos implementar nuestro navegador de archivos a partir de otro ya creado, pero que queremos introducir alguna funcionalidad nueva, como por ejemplo, una previsualización de los archivos de imagen mientras navegamos. El navegador está implementado usando la biblioteca GTK; y hemos encontrado un programa implementado en Qt que es capaz de cargar una imagen a partir de un archivo y mostrarlo por pantalla. Seguramente, esta solución no sea la más adecuada, ya que usar estas dos bibliotecas juntas puede dar muchos problemas. Quizás será mas adecuado buscar otro programa, también implementado en GTK o usar otra biblioteca más adecuada, para implementar y adaptar la funcionalidad deseada.

Puntualicemos este ultimo caso: no estamos diciendo que no sea posible utilizar ambas bibliotecas a la vez, si no que es preferible no hacerlo a no ser que sea estrictamente necesario; y que para el ejemplo expuesto, existen soluciones que nos pueden dar un mejor resultado.

### 2.2.2. Portales y recursos

Especificada y definida nuestra solución, nos disponemos a buscar por la red.

**Sourceforge** Nuestra primera parada obligada es este portal de desarrollo de software libre [2]. En el podemos encontrar casi cualquier programa que deseemos, pues a fecha de Febrero del 2009, aglutina más de 230.000 proyectos diferentes. Además, ofrece herramientas y alojamiento para crear nuestros propios proyectos, o colaborar con los ya existentes.

Podemos ver en la figura 1 una captura de pantalla del portal. Hemos buscado programas relacionados con la *mensajería instantánea*. Nos devuelve una lista de programas relacionados con este concepto, y para cada uno de ellos algunos datos; tales como una descripción (en que consiste, que librerías usa, etc.), el rango de descargas que ocupa dicho proyecto dentro del portal o la actividad de sus colaboradores. Gracias al buscador, podemos ir afinando nuestra búsqueda, según los criterios que nos interesen. Por ejemplo, si quisiéramos especificar que el programa de mensajería instantánea que estamos interesados en utilizar nos gustaría que estuviera implementado en GTK; lo añadiríamos en los parámetros de búsqueda.

Mediante su buscador, podemos encontrar un software libre de partida (o varios de ellos) acordes con lo que queremos hacer. Si no lo encontramos en este portal, podemos probar de buscarlo utilizando el buscador de Google; pues algunos proyectos no siempre se encuentran en Sorceforge; y/o tienen una página web con información y recursos adicionales sobre el proyecto.

sourceforge FIND AND DEVELOP OPEN SOURCE SOFTWARE

Welcome, Guest! Log In | Create Account

instant messaging GTK+ Search

Find Software | Develop | Create Project | Blog | Site Support | About

SourceForge.net > Search

Search results in projects found for "instant messaging" Search Help

Results 1 - 10 of 1465 Display: Details Images Filters View: 10

| Name | Relevance | Activity | Rank | Registered | Latest File | Downloads |
|---|---|---|---|---|---|---|
| Pidgin | 67.17% | 99.97% | 82 | 1999-11-13 | 2009-06-29 | 34,075,461 |
| Pidgin is a GTK+ instant messaging application for Windows and Unix. It supports AIM, ICQ, Jabber/XMPP, MSN, Yahoo!, Bonjour, Gadu-Gadu, IRC, QQ, SILC, SIMPLE and more. See http://pidgin.im/about for more information. Topic: AOL Instant Messenger, MSN Messenger | | | | | | Download Now! |
| Miranda IM | 67.17% | 92.24% | 22,485 | 2003-11-04 | 2009-04-17 | 14,853,684 |
| Miranda IM is an open source, multi-protocol instant messaging client designed to be very light on system resources, extremely fast and customisable. A powerful plugin-based architecture make Miranda IM one of the most flexible clients on the planet. Topic: AOL Instant Messenger, ICQ, Internet Relay Chat, MSN Messenger | | | | | | Download Now! |

Figura 1: Portal Sourceforge

# 3. Obteniendo información del programa

## 3.1. Antes de contactar, nos vamos a informar.

Una vez hemos encontrado un software candidato a ser usado como software de partida, es posible que pensemos que ya podemos ofrecer todas nuestras ideas al equipo de desarrollo encargado de desarrollar dicho proyecto.

Esto es parcialmente cierto, pero hay algunas cosas deseables que deberíamos hacer antes de entrar en contacto con ellos.

### 3.1.1. Probar el programa y leer la documentación

En el caso que no conozcamos el programa, es bastante obvio darse cuenta que antes tenemos que ejecutarlo. Leernos la documentación también nos puede ayudar a saber como se maneja el programa; y a averiguar si existen funcionalidades del programa que generalmente no están activadas por defecto.

### 3.1.2. Compilar el programa

Puesto que vamos a usar el código fuente del programa para ampliarlo con nuestras funcionalidades, deberíamos tratar de compilarlo, para ver con que dificultades nos pode-

mos encontrar. Para conseguir este propósito, deberemos bajarnos el código fuente [1] del programa en vez de un ejecutable o un paquete pre-compilado.

Por una parte veremos que bibliotecas vamos a necesitar, o como vamos a tener que configurar nuestro entorno de desarrollo; y por otra, podemos valorar si compilar la última versión *en desarrollo* [2] del programa. Como comentamos, debemos conseguir el código fuente, que se puede obtener de varias formas:

**Desde la página web** Podemos descargar el código fuente de la página web si, el proyecto dispone de ella y el código fuente está disponible para su descarga. Normalmente, los equipos de desarrollo no cuelgan el código fuente de la versión actual en desarrollo, si no que suelen habilitar un archivo comprimido con el código fuente de la última versión estable.

**Usando clientes de repositorios** Las aportaciones al desarrollo del código del software, se suele realizar mediante el uso de repositorios. El lector debería buscar información de como funcionan antes de empezar a colaborar en el desarrollo de software libre, y su estructura organizativa habitual [4]. Gracias a ello, podemos acceder a la versión actual en desarrollo del programa que nos interese, y descargar el código fuente a nuestra máquina para su posterior compilación y estudio.

Algunos clientes conocidos son **svn** o **git**. Se puede encontrar amplia información sobre ellos en la red; y podemos descargarlos e instalarlos fácilmente mediante la aplicación *aptitude* o *apt-get* disponible en varias distribuciones Linux. Normalmente, no es necesario disponer de ningún tipo de registro para acceder a la descarga del código fuente de ningún proyecto de software libre; tan solo necesitamos conocer la dirección en donde se encuentra [3].

Podemos ver en la figura 2 la página Web del proyecto Geeqie; en la sección donde nos explican como obtener el código fuente desde el repositorio del programa.

Subversion Access

This project's SourceForge.net Subversion repository can be checked out through SVN with the following instruction set:

```
svn co https://geeqie.svn.sourceforge.net/svnroot/geeqie geeqie
```

Figura 2: Orden para obtener el código fuente de todo el proyecto Geeqie.

[1] usualmente lo encontramos como 'SOURCES', en la página web del proyecto.

[2] La última versión estable se ofrece de un proyecto, no es nunca la más actual. Puede existir una versión que está siendo desarrollada actualmente por el equipo que implemente nuevas funcionalidades que nos pueden interesar [3]

[3] Esto no es siempre cierto cuando queremos actualizar los archivos de un proyecto, pues algunas veces necesitamos registrarnos en el lugar donde se encuentren los repositorios.

### 3.1.3. Analizar mínimamente el código

Con esta tarea, junto con la lectura de la documentación, intentamos averiguar, aproximadamente, cuan difícil va a ser realizar las tareas que deseamos, y el tiempo aproximado que nos puede llevar. También podremos vislumbrar si el estilo de programación nos gusta, cuál es su calidad y otros detalles algo más subjetivos que dependen de cada programador.

### 3.1.4. Echar un vistazo a la lista de cosas que hacer (TODO list)

La mayoría de proyectos, ya sea en su página web como en el mismo código fuente, contienen un archivo llamado TODO. Dicho archivo da una buena idea aproximada de hacia donde tiran las lineas generales de desarrollo del proyecto, es decir, de lo que se pretende implementar y mejorar en próximas versiones. Nos puede ser considerablemente útil si queremos colaborar en dicho proyecto; sobretodo si no tenemos una idea clara de lo que queremos hacer.

### 3.1.5. Resumen

Todos estos pasos previos al contacto con el equipo de desarrollo, no son estrictamente necesarios, ni el orden tiene que ser el descrito. Puede ser que lo primero que queramos hacer es analizar el código para ver si nos interesa en vez de intentar compilarlo. Es posible que queramos contactar previamente con el equipo de programación para saber cual es su disponibilidad a adaptar nuestra idea a su software ya implementado. Pero la idea de realizar alguna de las tareas previamente explicadas, puede resumirse en los siguientes puntos:

1. Conocer el programa si no lo conocíamos previamente. Si ya lo conocíamos, intentar averiguar si existen funcionalidades no activadas por defecto que se acerquen a lo que queremos realizar.

2. Ver cuan fácil es compilar el programa con el cual estamos interesados en nuestra máquina actual. Casi cualquier código fuente se puede compilar en casi cualquier máquina, pero esto no tiene porque ser siempre así. Además, es posible que sea más fácil compilarlo en unos entornos que otros, y puede servirnos para valorar si instalar distribuciones y entornos de programación adicionales a los que ya dispongamos.Por ejemplo, es posible que por temas de rendimiento tengamos instalado una distribución Debian de 64 bits en nuestra máquina. Sin embargo, el entorno de compilación del proyecto el cual estamos interesados, solo funciona en 32 bits. Sabiendo esto, podremos valorar si instalar una distribución adicional Debian de 32 bits, o utilizar otros métodos para conseguir nuestro cometido; lo cual se traduce todo en un tiempo mayor a considerar en el desarrollo de nuestro programa.

3. Ver algunos de los aspectos básicos del código que estamos interesados en utilizar:

    *a*) ¿Es un diseño modular?

*b*) ¿Cuales son los archivos principales que nos interesan?

*c*) ¿Que estructuras de datos principales utiliza?

Finalmente, conseguir todo este tipo de información y estudiar el programa contextualiza nuestro punto de trabajo. Cuando contactemos con el equipo de desarrollo, no iremos con las manos vacuas: sabremos, aproximadamente, lo que hace o lo que no hace el programa por del cual pedimos información, y sabremos como funciona y sabremos hacerlo funcionar mínimamente. Si hacemos notar al equipo de desarrollo que hemos estudiado el software, verán que estamos realmente interesados en él, y nos tomarán más en serio, que si simplemente nos dedicamos a formular preguntas al aire sin haber estudiado el programa.

# 4. Coordinación con el equipo de desarrollo

## 4.1. Contactando (o no) con el equipo.

Una vez hemos recopilado suficiente información sobre el proyecto, podemos proceder a contactar con el equipo de desarrollo. Depende de la naturaleza de nuestro proyecto y de la complejidad de los cambios la decisión de ponernos en contacto con ellos o no, pero normalmente, establecer una comunicación con el equipo de desarrollo suele resultar ventajoso:

1. Podremos proponer los cambios e implementaciones que hemos pensado al equipo sobre el proyecto. Esto nos permite:

   *a*) Valorar la viabilidad y complejidad real de nuestra implementación.

   *b*) Averiguar si nuestra idea está ya en mente de otros desarrolladores.

   *c*) Pedir consejo sobre la mejor manera de llevar a cabo nuestra idea dentro del proyecto.

2. Informar e informarnos sobre posibles errores del programa que hemos escogido.

3. Estar al día de cambios importantes del proyecto y el lanzamiento de nuevas versiones con nuevas características.

### 4.1.1. Las listas de correo

La mayoría de proyectos de software libre usan listas de correo [5] para coordinase. Gestionan el correo de la siguiente manera: se envía un correo a una dirección de correo única, y el gestor reenvía el correo a todos los usuarios suscritos a dicha lista. De esta manera nos ahorramos en tener que lidiar con los cuentas de correo electrónico de todos los usuarios que colaboran en el proyecto.

Podemos ver un esquema de dicho funcionamiento en la figura . El dominio de la dirección de correo que vemos en dicha figura es *mail-archive.com*; y no es casual ni inventada, pues es uno de los gestores de correo más usados.

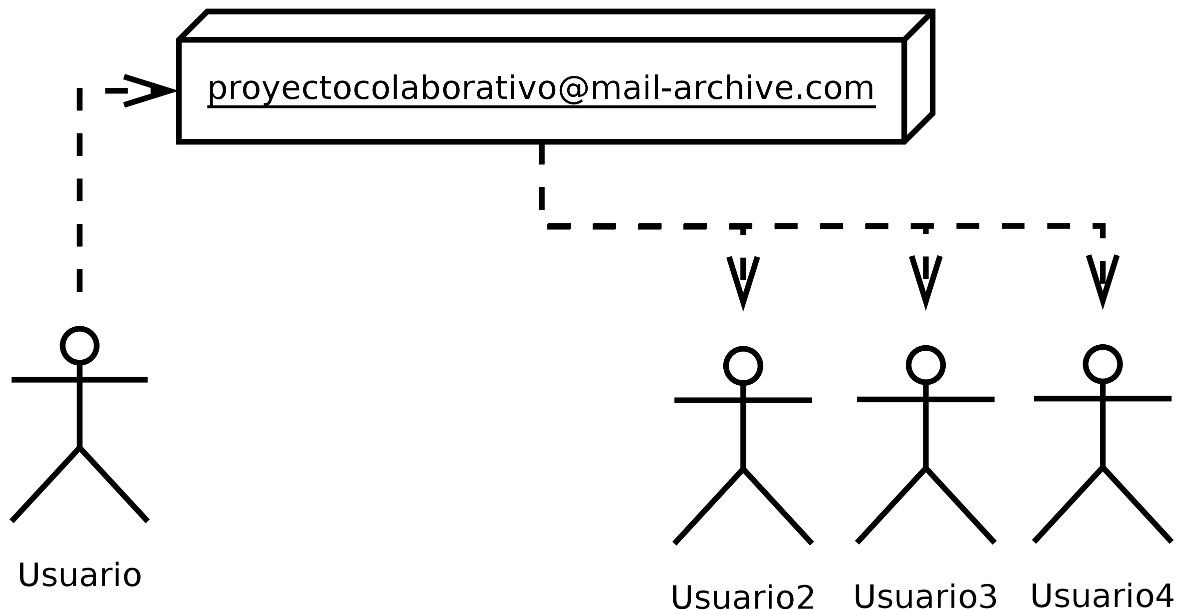


Figura 3: Esquema funcionamiento lista de correo.

Además de permitir las características que hemos explicado previamente, guarda permanentemente una copia de todos los correos enviados a esa dirección, de manera que una persona que se incorpora de nuevo en el proyecto; puede buscar información sobre temas ya tratados en el proyecto.

#### 4.1.2. Presentándonos

Ya sea a través de una lista de correo, una dirección de correo electrónico particular, o quizás a través de un foro; lo primero que deberíamos hacer es presentarnos. En este mensaje, son 3 los puntos que podemos tratar:

- Contextualizar nuestro interés en el desarrollo del software.
- Explicar que características nos gustarla introducir o cambiar.
- Obtener una primera respuesta de la viabilidad de nuestra propuesta.

Podemos ver un ejemplo de este correo en la figura 1

### 4.2. Propuestas y negociaciones

A partir de este punto, ya podemos empezar a negociar nuestra colaboración con el proyecto. Comentar que la forma de trabajar de cada equipo de desarrollo es intrinséco de ese equipo; y por lo tanto, dependerá de los integrantes del mismo y de quien está asumiendo

Hola a todos. Soy Óscar Miras y estoy realizando mi proyecto final de carrera. Mi tutor y yo estamos decidiendo con qué proyecto (de software libre) vamos a colaborar. Estamos buscando formas de hacer la navegación a través de los archivos más intuitiva. La razón es que GQView, y otros programas, muestran demasiadas fotos a la vez, y es difícil encontrar el subconjunto de fotos deseado. Un ejemplo de lo que queremos hacer es, que si estamos navegando por un mes en concreto del calendario, podríamos colorear los días donde hubiera fotos, y quizás mostrar una miniatura en cada uno de los días. También pensamos que podemos usar metadatos de carácter temporal para ordenar las fotos. Son solo algunas ideas, el proyecto acaba de empezar :).
Si finalmente decidimos añadir un nuevo escenario a GQWiew, ¿consideraríais la posibilidad de incluirlo en GQWiev como un modo más para mostrar las fotos?

Correo 1: Presentación al equipo de GQView. Autor: Óscar Miras. Fecha: 04/03/2009.

el liderazgo del proyecto, Pretendemos en esta sección dar una serie de pautas para hacer entender nuestras ideas a los colaboradores.

Comentar también que todos los proyectos suelen tener, lo que denominaremos, unas *lineas generales de desarrollo.*. Esto es, que los desarrolladores establecen como ha de ser el funcionamiento general del programa; así como las funcionalidades nuevas que se quieran incorporar. Cuanto más se asemeje nuestra idea a estos dos aspectos; más posibilidades de éxito tendrá. Esta información, la podemos encontrar tal como comentábamos en la sección 3.1.3, en el archivo TODO del proyecto; o informándonos a través de la lista de correo, foros, etc.

Para finalizar, recordar al lector que el hecho de que su idea no *cuaje* dentro del proyecto, no significa el fin de la colaboración con el mismo. Comentaremos al final que podemos hacer en estos casos.

### 4.2.1. Exponer nuestras ideas

Sencillamente, hablar de lo que queremos aportar al proyecto. Esto nos permite averiguar si ya se está trabajando en esta linea, o si ya está implementado pero no hemos activo la funcionalidad que nos interesa. También nos servirá para averiguar la viabilidad de implementarla dentro del proyecto, y si los demás desarrolladores la encuentran interesante.

Es de gran ayuda valernos de esquemas, dibujos, presentaciones, etc. Ello nos dará dos puntos a nuestro favor: tendremos más posibilidades de que entiendan lo que queremos hacer, y además daremos la impresión de que realmente estamos interesados en colaborar con el equipo.

### 4.2.2. Especificar nuestra propuesta

Una vez aceptada nuestra idea, debemos especificarla y diseñarla. Puesto que no se necesita ningún título para colaborar en un proyecto de desarrollo en red, debemos hablar en un lenguaje que sea entendido por todos los colaboradores, pues a priori, no sabemos el nivel de conocimientos sobre ingeniería informática que tienen los demás integrantes.

Los casos de uso simplificados son una buena forma de exponer la funcionalidad que queremos diseñar e implementar; porque explican lo que queremos hacer en un lenguaje conocido por todos. Podemos usar varios medios: tenerlos en una Wiki nos permitirá tener un registro de lo que vamos haciendo, además que permitirá a otros desarrolladores colaborar. Sin embargo, no a todo el mundo le gusta colaborar en una Wiki; por lo que es una buena idea enviar de todas maneras nuestros casos de uso a la lista de correo.

Podemos ver un ejemplo de la especificación de una funcionalidad en la figura 4

**Automatic scrolling**

If mouse is positioned at the very end of the navigation windows; it should scroll down automatically.

**Usecase**

1. User arrives at the very end of bottom navigation window
2. Window automatically scrolls down. Now the bottom row is the top one ; in the current user navigation window.
3. End of usecase.

**Possible alternatives**

1. Decide how far from bottom mouse is before it scrolls down ( I would vote: if user positionates mouse at 3/4 of height* of the last row.)
2. Maybe decide transition speed , and if it will be animated or instant view change.

3. Make automatic scrolling only work full screen view mode when using pan view.

Figura 4: Propuesta de nueva funcionalidad como caso de uso.

## 4.3. Negociaciones y toma de decisiones

A medida que vayamos desarrollando nuestra idea e implementando nuestras funcionalidades, es posible que vayan surgiendo problemas y que aparezcan nuevas oportunidades. Vamos a intentar enumerar algunos de los casos que pueden aparecer a lo largo del desarrollo del proyecto.

- **Todo o parte de lo que queremos implementar está siendo desarrollado por otra u otras personas.** Es importante que nos pongamos en contacto con ella/as y averiguamos el estado su estado de desarrollo. ¿Podemos colaborar con ella? ¿Podemos diseñar e implementar parte de la funcionalidad que se está buscando? ¿Tenemos que esperar a que dicha persona acabe lo que está haciendo para realizar lo que queremos hacer nosotros? ¿Cuál es el plazo previsto para que sea terminado?

- **Alguna de las implementaciones que se están realizando pueden afectar a la mejora que hemos propuesto.** Es una variante del caso anterior. La implementación de nuestra funcionalidad usa partes del programa que están siendo actualmente cambiadas. ¿De que manera tenemos que adaptar nuestro código para poder aprovechar las futuras mejoras? ¿Hay alguna estructura de datos nueva que debiéramos tener en cuenta?

> Imaginar que queremos ampliar u explorador de archivos para muestre una previsualización de los archivos fotográficos. Actualmente, la información que se guarda de los archivos se hace un fichero de texto plano; pero se esta implementando una mejora para que se haga en una base de datos. ¿Debemos indicar a los desarrolladores de dicha que modifiquen la implementación de alguna manera para que contemple lo que queremos hacer? ¿Valdrá más la pena que nos dediquemos primero a ayudar a completar la funcionalidad de base de datos? Sobre funciones que se usan para guardar la información de los archivos en la base de datos: ¿están lo suficientemente encapsuladas como para que no tengamos que cambiar demasiado nuestro código cuando se implemente el guardado de información en base de datos?

- **Nuestra propuesta no se ajusta nada a las lineas generales de desarrollo.** Es posible que lo que queremos implementar no esté pensado a incorporarse en las lineas generales de trabajo del proyecto. Normalmente, existe cerca flexibilidad en incorporar nuevas funcionalidades al proyecto si están resultan interesantes; y tarde o temprano pueden ser incorporadas a la versión general y estable del proyecto. Sin embargo, si nuestra propuesta se aleja tanto de lo que es el programa original en si, podemos siempre crear una nuevo. Otra solución es pedir a los responsables del proyecto que nos abran una *rama de desarrollo alternativa*. Esto es, una carpeta dentro del repositorio de código del proyecto donde podemos ir enviando nuestras contribuciones; para no entorpecer las lineas generales de desarrollo del proyecto. Eventualmente, nuestras funcionalidades serán integrada en la versión oficial y estable del programa si así lo decide el equipo de desarrollo. Si no, siempre podemos mantener esa rama de desarrollo alternativa; e invitar a todas las personas interesadas a colaborar en ella.

| File ▲ | Rev. | Age | Author | Last log entry |
|---|---|---|---|---|
| branches/ | 1886 | 3 months | DarkEndymion | |
| tags/ | 1909 | 5 weeks | nadvornik | tagging 1.0 release |
| trunk/ | 1915 | 4 weeks | mow | Add unknown file class to grouping |

Figura 5: Aspecto usual de los directorios de un proyecto.

En *trunk* tenemos el código de la versión estable oficial; mientras que en *branches* todas las ramas de desarrollo alternativas de este mismo proyecto.

# 5. Diseño e implementación

## 5.1. Diseñando nuevos módulos y funciones

En este apartado daremos algunos consejos sobre el diseño e implementación de nuevas funciones. Corresponde al lector usar su pericia para el diseño e implementación de nuevas funciones; así como crear nuevas módulos cuando sea necesario. Nos centraremos pues en otros aspectos que bien se podrían aplicar en la programación de nuevas funcionalidades para cualquier tipo de software; pero que cobran especial importancia cuando hablamos de desarrollo colaborativo de software libre; pues es un producto que seguramente va a perdurar en el tiempo, y en él van a trabajar un número más o menos grande de colaboradores; y donde puede ocurrir una alta rotación de ellos.

- **Aprovechar el trabajo realizado.** Es una máxima en el desarrollo de software libre; y lo es también a la hora de programar nuevas funcionalidades. Es posible que muchas de las cosas que necesitamos ya estén realizadas. Es muy recomendable pues buscar en el código antes de implementar nada; e incluso preguntar al equipo si creemos que cierta funcionalidad ya debe estar implementada; pero no la encontramos.

> Una buena forma de buscar funciones útiles es revisar los archivos descriptores de funciones (.h o .hpp, en C y C++ respectivamente). Normalmente, ahí encontraremos todas las funciones que están bien encapsuladas y preparadas para ser usadas.

- **Seguir el estilo de programación.** Es importante que todos los programadores sigan un estilo común de programación; para que se pueda enteder el código con mayor facilidad. Cuando hablamos de estilo de programación, nos referimos a varias partes de el: estructura, nombre y orden declaración de variables, creación de bucles, etc.

  El estilo de programación no solo atañe a la manera de introducir el código, si no también a la forma de estructurar nuestros ficheros; sobretodo si introducimos funcionalidades parecidas a las ya implementadas.

> Tenemos un explorador de ficheros, que permite realizar varias operaciones sobre los ficheros. Tenemos implementada la funcionalidad Borrar fichero en la clase borrar.h y borrar.c; cuyas funciones públicas son: *borrar_fichero(parametro 1,parametro 2) y borrar_lista(parametro 1, parametro 2).* Si posteriormente implementamos la funcionalidad Renombrar; es de esperar que nuestros ficheros se llamen renombrar.h y renombrar.c; y las funciones Borrar fichero en la clase borrar.h y borrar.c; cuyas funciones públicas son: *renombrar_fichero(parametro 1,parametro 2) y renombrar_lista(parametro 1, parametro 2).*

- **Controla el impacto de los cambios en el código original.** Siguiendo el primer consejo de este apartado, es conveniente siempre que sea posible minimizar el impacto

de los cambios en el código del programa original. De hecho, deberíamos crear ficheros nuevos para funcionalidades nuevas, y no intentar modificar el código original del programa para que se *adapte* a lo que queremos hacer; debido a que podemos generar una serie de efectos colaterales que pueden desestabilizar el programa. En ese sentido, he aquí algunas recomendaciones:

- Si creamos ficheros nuevos en el proyecto, deberemos indicar de algún modo que estos han de ser compilados (ficheros Makefile [6] y herramientas Autotools [7]; generalmente si se desarrolla en C o C++). Intentaremos minimizar el numero de dependencias de los módulos originales, y usaremos las funciones de los descriptores de ficheros que nos ofrecen las funciones del código original siempre que sea posible.
- Si nos vemos en la necesidad de modificar código de funciones existentes, intentaremos que el impacto sea mínimo. Una posible escala de preferencias seria la siguiente (de más recomendable a menos recomendable):
    1. Crear una nueva función pública, e implementarla sin modificar el resto de ese fichero.
    2. Crear una nueva función privada, que será llamada por alguna pública del fichero cuando sea necesario.
    3. Modificar una función pública o privada ya creada para que haga lo que nosotros deseamos. Este es el peor escenario[4], y una buena estrategia, es condicionar la ejecución de cierto código mediante algún parámetro que indique que efectivamente lo que se debe ejecutar es parte de nuestro código.

Mostramos los 3 supuestos anteriores con ejemplos. Son ejemplos independientes, aunque traten de funcionalidades parecidas. Comentar también que son ejemplos muy básicos, solo para dar una idea al lector, pero la cosa se podría complicar mucho a medida que crecen la complejidad de los cambios. Por ello, cuanto mejor encapsulemos nuestras nuevas funcionalidades, mucho mejor.

```
/** CASO 1:Nueva función pública 'renombrar_fichero' en fichero.h */

copiar_fichero(FileData *file);
borrar_fichero(FileData *file);
renombrar_fichero(FileData *file); /** nueva función*/

/** CASO 2: La función pública 'elegir_funcionalidad' llama a una u otra
    dependiendo un parametro */

void elegir_funcionalidad(FileData *file, int option)
```

[4] A no ser que el la función haya sido creada con ese propósito, es decir, considerar una serie de acciones en función de un parámetro de entrada.

```
{
   if (option==0) copiar_fichero(file);
   else if (option==1) borrar_fichero(file);
   else if (option==2) renombrar_fichero(file); /** nuevo */
}

/** CASO 3: Necesitamos ejecutar cierto código cuando nos encotremos en nuestro
    modo de funcionamiento recién implementado */

int n,m;

funcion1(...) /** Esto se ejecuta siempre */

if (modo->funcionamiento==RENOMBRAR_FICHERO) funcion2(...)
/** NUEVO:Esto ahora es necesario que se ejecute cuando estemos el
 modo de funcionamiento 'renombrar fichero' */

else funcion 3(..) /** NUEVO:El hecho de ejecutar nuestra funcion2 implica que
no debemos ejecutar la funcion 3 que ya estaba implementada; introduciendo así
otro cambio considerable en el código existente */
```

## 5.2. Documentando el código

Todas las funciones que creamos deben estar debidamente documentadas; para entender en el futuro que hacen dichas funciones; facilitando la comprensión de su funcionamiento y su reusabilidad. Es más. si durante el estudio del código a modificar encontramos funciones no documentadas, es del todo recomendable documentarlas también. No importa si en un primer momento no sabemos al cien por cien que hacen exactamente dichas funciones; podemos ir aproximando la explicación a medida que conozcamos el funcionamiento real y más general de todo el programa.

A continuación, la documentación del cuerpo de una función; cuando no disponemos de demasiada información sobre el funcionamiento general del programa:

```
/** Coordenadas x e y de la ventana */
int x,y;

/** Explora el directorio 'dir_fd' y obtiene los archivos en una lista */
lista = explorar_directorio(dir_fd,TRUE);

while(i!=lista.tamaño())
{
   /** Aplica la funcion_generia a todos los archivos de la lista */
   funcion_generica(lista[i],OPTION_D);
```

```
   i++;
}
```

Una segunda versión de esa misma documentación, una vez conocemos mejor el software y el resto de funciones:

```
/** Explora el directorio 'dir_fd' y obtiene los archivos en una lista,
y los ordena alfabeticamente de manera ascendente */
lista = explorar_directorio(dir_fd,TRUE);

while(i!=lista.tamaño())
{
   /**  Borra todos los archivos de la lista (OPTION_D de funcion generica) */
   funcion_generica(lista[i],OPTION_D);
   i++;
}
```

Puede resultar un poco tedioso documentar cada linea del código no explicado que nos encontremos, pero los beneficios a medio plazo son enormes en contra del relativo tiempo que vamos a perder documentando. Además, explicar como funciona algo, aunque sea a nosotros mismo, es una buena manera para comprobar si lo hemos entendido. Finalmente, el equipo de desarrollo nos agradecerá enormemente esta labor.

### 5.2.1. Maquetando la documentación

Existen algunas herramientas que nos permiten maquetar la documentación que introducimos en el código. Esto nos da dos ventajas: una es que todos los programadores van a documentar siguiendo el mismo patrón. Además, mediante ciertas aplicaciones podemos generar documentación en HTML u otros formatos; y realizar búsquedas en el código, entre otras ventajas.

A continuación, un ejemplo de código maquetado con comentarios del estilo del maquetado Doxygen:

```
/*! A test class */

class Test
{
  public:
    /** An enum type.
     *  The documentation block cannot be put after the enum!
     */
    enum EnumType
```

```
  {
    int EVal1,     /**< enum value 1 */
    int EVal2      /**< enum value 2 */
  };
  void member();   //!< a member function.

 protected:
  int value;       /*!< an integer value */
};
```

Una vez compilado con el programa, podemos obtener un HTML con toda la documentación, tal como vemos en la figura 5.2.1

Main Page | Classes | Files

Class List | Class Members

#include <afterdoc.h>

List of all members.

**Public Types**

enum **EnumType** { **EVal1**, **EVal2** }
An enum type. More...

**Public Member Functions**

void **member** ()
a member function.

**Protected Attributes**

int **value**

**Detailed Description**

A test class

**Member Enumeration Documentation**

**enum Test::EnumType**

An enum type.

The documentation block cannot be put after the enum!

**Enumerator:**

*EVal1* enum value 1
*EVal2* enum value 2

Hay más estilos y maquetadores de código. Podemos explorar un poco en la red para encontrar el que mas nos convenza; aunque si el equipo de desarrollo ya ha propuesto uno o lo está usando; es conveniente considerar este antes de proponer ningún otro; ya que la mayoría suelen dar resultados similares.

# Referencias

[1] Estructura GDKPixbuff. Biblioteca Gnome. http://library.gnome.org/devel/gdk-pixbuf/stable

[2] Portal Sourceforge. Página oficial. http://www.sourceforge.net

[3] The Linux Documentation Project. Página oficial. http://www.tldp.org/HOWTO/Software-Proj-Mgmt-HOWTO/users.html#ALPHABETA

[4] Control de versiones con Subversion. Wikilearning. http://www.wikilearning.com/tutorial/control_de_versiones_con_subversion-el_control_

[5] Listas de correo electrónico. Wikipedia. http://es.wikipedia.org/wiki/Lista_de_correo_electr\

[6] Tutorial fichero Makefile. Opus Software. http://www.opussoftware.com/tutorial/TutMakefile.ht

[7] Autotools tutorial. EPITA Research and Development Laboratory. http://www.lrde.epita.fr/~adl/autotools.html

[8] Doxygen: Source code documentation tools. Página oficial. http://www.stack.nl/~dimitri/doxygen/