UNIVERSIDAD
REY JUAN CARLOS

MÁSTER OFICIAL EN SISTEMAS TELEMÁTICOS E INFORMÁTICOS

Curso Académico 2011/2012

Trabajo Fin de Máster

SVNExplorer: HERRAMIENTA PARA EL ANÁLISIS DE CAMBIOS EN FUNCIONES ENTRE VERSIONES DE CÓDIGO

Autor : Daniel Domínguez Cuadrado

Tutor : Gregorio Robles

Índice

1. Resumen

Desde sus inicios, el software libre se ha caracterizado por tener un gran número de desarrolladores en cada uno de sus proyectos. Estos desarrolladores no se suelen conocer entre ellos, por lo que en los últimos años se han desarrollado muchos proyectos para ofrecer la integración de todo el equipo de desarrollo, tales como *Subversion*, *Git*, ...

En proyectos muy voluminosos, requiere una pérdida de tiempo enorme saber los cambios realizados por otro compañero. Por ello, disponer de una herramienta que ofrece información sobre los cambios realizados en el continuo desarrollo de un proyecto puede resultar muy interesante.

En este TFM se hace uso de *Subversion* para ofrecer una herramienta de análisis en el desarrollo de un proyecto específico. En él, se intenta ayudar a los desarrolladores indicando el nombre de las distintas funciones que conforman un programa en cada una de sus distintas versiones, cuando han sido añadidas o eliminadas, las modificaciones que han sufrido a lo largo de sus versiones, ...

Para ello, se han elegido distintas herramientas basadas en software libre, tales como *Python*, *Ctags*, *SQLite* o el mismo *Subversion*.

2. Introducción

En esta sección, nos centraremos en realizar una breve introducción al software libre, ofreciendo una descripción de sus principales paradigmas. También ofreceremos una descripción detallada de cada una de las herramientas que hemos utilizado para desarrollar este proyecto.

2.1. Software libre

Para que un programa pueda ser considerado software libre debe ofrecer una serie de características. Principalmente, se basa en estas cuatro libertades:

- Libertad de uso: un programa software libre puede ser usado como uno quiera, donde quiera, sin necesidad de pedir permiso a los propietarios del programa.
- Libertad de redistribución: un programa software libre puede ser redistribuido sin violar ninguna ley restrictiva de propiedad intelectual.
- Libertad de modificación: el código fuente debe estar disponible para poder realizar mejoras o adaptar la funcionalidad del programa a nuestras necesidades.
- Libertad de redistribución de las modificaciones: podemos facilitar a terceros las adaptaciones realizadas en un programa ajeno, sin incurrir en delito alguno.

Esta introducción la indicamos en esta memoria porque nuestro proyecto va a ser software libre, y siempre es importante indicarle a los futuros usuarios las libertades de las que disponen.

2.1.1. Desarrollo de software libre

El desarrollo software libre se nutre generalmente de un grupo heterogéneo de programadores de procedencia mundial. Por ello, las herramientas que facilitan la rápida integración de todos ellos están en auge.

Las comunidades de desarrolladores han experimentado un fuerte crecimiento en los últimos años siendo altamente necesario el uso de internet para poder establecer una comunicación efectiva entre los integrantes de la comunidad.

En esta linea, han surgido proyectos como *Subversion* o *Git* para facilitar el desarrollo de los proyectos cooperativos.

Nuestro TFM quiere facilitar el análisis de un proyecto cooperativo, ofreciendo una herramienta que obtenga información de los cambios producidos en cada una de las diferentes versiones y pasos seguidos.

2.2. Herramientas utilizadas en la realización del TFM

A la hora de realizar este TFM, nos hemos decantado por utilizar herramientas basadas en software libre. En este apartado ofreceremos una breve descripción de todas y cada una de las aplicaciones utilizadas.

2.2.1. Python

Python es un lenguaje de programación interpretado orientado a objetos. Una de sus mayores características es que es un lenguaje de programación multiparadigma, esto permite programar en *Python* en varios estilos: programación orientada a objetos, programación estructurada y programación funcional. Aparte, mediante uso de extensiones existen más paradigmas soportados por este lenguaje.

Python destaca por la facilidad para realizar extensiones, ya que se pueden escribir nuevos módulos en *C* o *C++*. Las características técnicas de *Python* son:

- Lenguaje de propósito general: se pueden escribir en *Python* todo tipo de aplicaciones, desde aplicaciones para Windows hasta páginas web.
- Multiplataforma: *Python* incluye un intérprete propio para permitir la ejecución de programas en distintos sistemas operativos.
- Lenguaje interpretado: Evita compilar el código antes de su ejecución.
- Funciones y librerías: Dispone de muchas funciones incorporadas en el propio lenguaje para tratar con estructuras de datos tales como strings, números reales, ficheros, ... Además, existen numerosas librerías para aumentar la funcionalidad y diversificación de sus programas.
- Sintaxis clara: Indentación obligatoria para separar bloques de código, ayudando a la claridad y consistencia del código escrito.

- Tipado dinámico y lenguaje fuertemente tipado: No es necesario indicarle a *Python* el tipo de la variable que vamos a utilizar, pues el mismo lo reconoce la primera vez que se le otorga un valor.

- Lenguaje de muy alto nivel: Oculta al programador los detalles de bajo nivel, como manejar la memoria empleada por nuestro programa. Incluye un recolector de basura para eliminar de memoria los objetos que ya no van a ser utilizados.

Una de las razones que nos ha impulsado a utilizar *Python* para implementar nuestro TFM ha sido la facilidad que tiene para comunicarse con bases de datos (incluye gestores para *MySQL*, *Oracle*, *SQLite*, ...).

2.2.2. SQLite

SQLite es un sistema de gestión de bases de datos relacionales contenido en una pequeña biblioteca escrita en *C*.

La biblioteca de *SQLite* se enlaza con el programa que lo utiliza pasando a ser una parte integral del mismo. Esto reduce muchísimo la latencia en el acceso a la base de datos debido a que las llamadas a funciones son más eficientes que la comunicación entre procesos.

El conjunto de la base de datos se guarda en un sólo fichero estándar. Para evitar la corrupción de los datos, el fichero que contiene la base de datos en cuestión se bloquea al principio de cada transacción. *SQLite* permite bases de datos de hasta 2TB de tamaño.

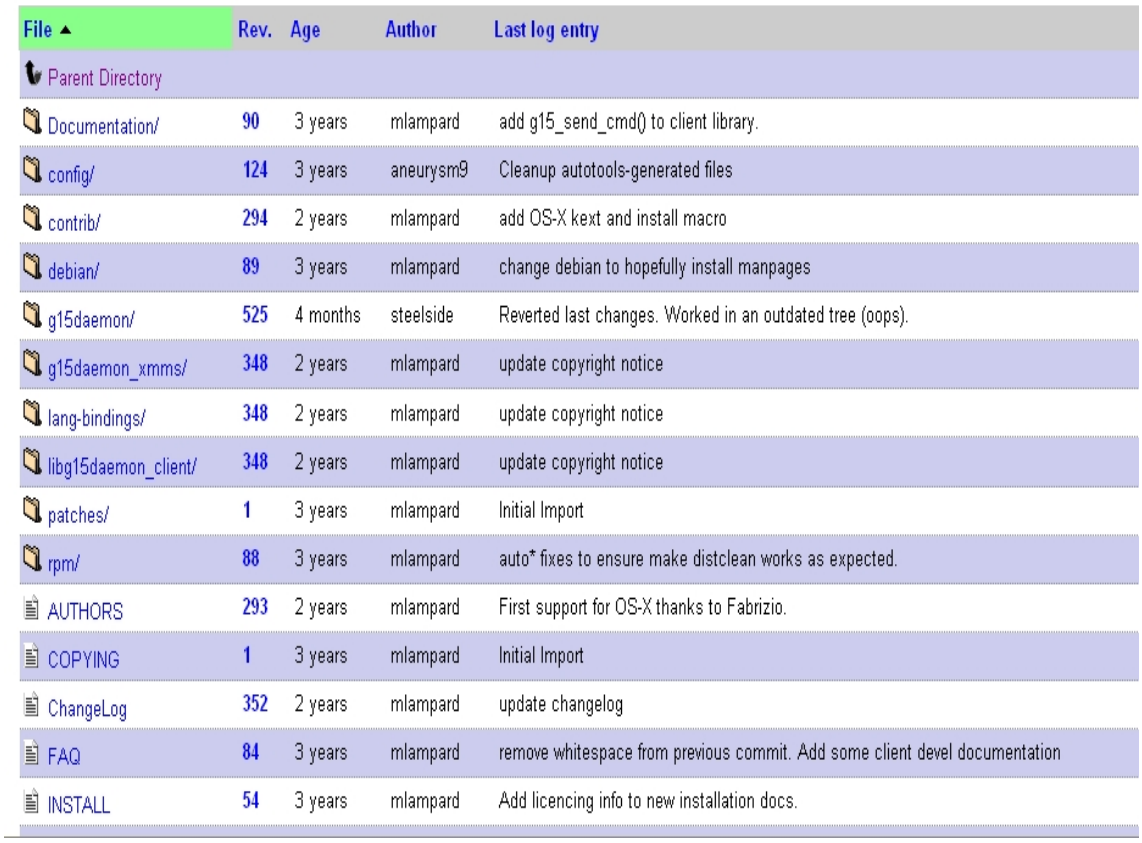
Una de las principales ventajas de *SQLite* es que puede ser usado desde multitud de diferentes lenguajes de programación, tales como *C/C++*, *Perl*, *Python*, *Visual Basic*, *Delphi*, *PHP*, ...

SQLite es un sistema gestor de bases de datos altamente fiable, ya que aplicaciones tan renombradas como Firefox, Amarok, Android o Yum (gestor de paquetes de Fedora Core) hacen uso de esta herramienta.

2.2.3. Subversion

Subversion es un software de sistema de control de versiones, diseñado para reemplazar a CVS. Ofrece ventajas importantes al desarrollo de un proyecto y sus principales características son las siguientes:

- Permite trabajar sobre múltiples plataformas: Linux, UNIX, Windows, MacOS, ...
- Las transacciones de las modificaciones en los archivos son atómicas, previniendo una posible corrupción en los datos del proyecto.
- Mantiene el repositorio más ordenado al contar con tres vías principales de desarrollo: Trunk (desarrollo principal), tags(ubicación de las versiones congeladas) y branches(versiones de desarrollo paralelas a trunk)
- Al enviarse sólo las diferencias al realizar una modificación en los archivos del repositorio, es más rápido y eficiente que su antecesor CVS, que siempre envía al servidor archivos completos.
- El repositorio *Subversion* puede ser accedido por múltiples protocolos: http, https, svn, svn+ssh, ...
- Existen numerosos interfaces para trabajar con *Subversion*, tales como TortoiseSVN, RabbitVCS, KDESvn, ... También incorpora múltiples *plugins* para poder integrarse con Eclipse.



File ▲	Rev.	Age	Author	Last log entry
Parent Directory				
Documentation/	90	3 years	mlampard	add g15_send_cmd() to client library.
config/	124	3 years	aneurysm9	Cleanup autotools-generated files
contrib/	294	2 years	mlampard	add OS-X kext and install macro
debian/	89	3 years	mlampard	change debian to hopefully install manpages
g15daemon/	525	4 months	steelside	Reverted last changes. Worked in an outdated tree (oops).
g15daemon_xmms/	348	2 years	mlampard	update copyright notice
lang-bindings/	348	2 years	mlampard	update copyright notice
libg15daemon_client/	348	2 years	mlampard	update copyright notice
patches/	1	3 years	mlampard	Initial Import
rpm/	88	3 years	mlampard	auto* fixes to ensure make distclean works as expected.
AUTHORS	293	2 years	mlampard	First support for OS-X thanks to Fabrizio.
COPYING	1	3 years	mlampard	Initial Import
ChangeLog	352	2 years	mlampard	update changelog
FAQ	84	3 years	mlampard	remove whitespace from previous commit. Add some client devel documentation
INSTALL	54	3 years	mlampard	Add licencing info to new installation docs.

Figura 1: Imagen de *Subversion* accediendo vía http

2.2.4. Ctags

Ctags es una aplicación que genera un archivo donde ofrece un resumen de las características de un programa. Dependiendo del lenguaje de programación en el que está escrito nuestro programa, ofrece múltiples opciones para realizar un resumen más o menos completo.

Podemos indicarle a *Ctags* que nos gustaría disponer de un resumen donde se especifique el nombre de las funciones, macros declaradas en nuestro programa, variables de las que hacemos uso... todo eso unido con la posibilidad de localizar donde han sido declaradas.

3. Objetivos

3.1. Descripción del problema

A la hora de realizar un proyecto de grandes dimensiones utilizando un software de control de versiones en el que mucha gente colabora, existe el riesgo de perder una ingente cantidad de tiempo y esfuerzo en determinar qué se ha hecho en el paso de una versión a otra.

Para determinar la calidad y reducir la complejidad en el proceso de desarrollo, hemos pensado que una herramienta de análisis de diferencias entre versiones puede ser de una gran ayuda. En el desarrollo de un proyecto es fundamental disponer de información en los cambios producidos cuando se practican mejoras a un software, ya que si no se corre el riesgo de no poder resolver ciertos problemas relacionados con el desarrollo. Nuestro objetivo fundamental es ayudar a recabar esta información más fácilmente.

El análisis del software puede resultar fundamental para ayudar a tomar decisiones complejas, cuanta más información tengamos, más eficiente será la decisión tomada. En nuestro caso, notamos que existía un vacío en la comparación entre versiones, porque aunque si es verdad que *Subversion* ofrece la herramienta *svn diff version1 version2* para comparar las versiones de un programa, nos pareció una herramienta pobre ya que sólo indica el cambio y la linea donde se produjo dicho cambio, sin dar ningún otro tipo de información, importante bajo nuestro punto de vista.

Pensamos que una herramienta que indique qué funciones han sido añadidas o eliminadas, o que indique el nombre de las funciones donde se han producido los cambios, podría ayudar a delimitar el alcance de un problema, revisando sólo un número determinado de líneas de código sin tener que revisar el proyecto entero, evitando así un esfuerzo impresionante de los programadores, permitiéndoles centrarse en el desarrollo para que puedan ofrecer un código de mayor calidad.

De esta misma forma, hemos intentado sacar los datos de nuestro análisis de la forma más sencilla posible, haciendo posible un análisis rápido de los cambios producidos en el avance del proyecto. Siendo posible también realizar un análisis pormenorizado en el caso de que nos interese recabar más información.

3.2. Requisitos

Hemos basado el desarrollo de nuestro proyecto en unos requisitos básicos:

- Facilitar una herramienta de análisis del software en un campo que nos parecía insuficientemente cubierto.
- Utilizar herramientas basadas en software libre, para que nuestra herramienta de análisis pueda ser accesible a la mayor cantidad de gente posible.
- Ofrecer un fácil manejo de la herramienta.
- Permitir una instalación fácil y sencilla.

3.3. Modelo de desarrollo

A la hora de desarrollar software, es importante llevar cada una de las fases de forma estructurada, ya que una mala planificación puede llevar al fracaso absoluto del proyecto.

Para llevar nuestro proyecto de forma metódica y estructurada hemos elegido el modelo de desarrollo de software por etapas, ya que nos permitía estructurar el proyecto de tal forma que:

- En cada una de las etapas estudiar las especificaciones técnicas, diseñar e implementar una nueva funcionalidad, sin tener que retocar el resto del proyecto, ya que esta forma de proceder ofrece una facilidad extrema a la hora de añadir funcionalidades o desarrollos posteriores.
- Además de permitir centrarse únicamente en una funcionalidad en cada una de las etapas, este modelo es capaz de ofrecer una gran batería de pruebas a nuestro software, ya que por cada una de las etapas:
    - Se realizan pruebas específicas para cada nueva funcionalidad.
    - Se realizan pruebas para comprobar la correcta integración de la funcionalidad desarrollada con el resto del proyecto.
    - Se realizan pruebas generales para el proyecto entero, permitiendo así la creación de un software altamente fiable.

Por cada una de las etapas, se han desarrollado las siguientes actividades:

- **Estudio de las especificaciones técnicas:** El tutor especifica hacia donde se debe encaminar el TFM. El autor del mismo debe sacar la síntesis de lo comunicado y hallar los requisitos de la nueva funcionalidad.
- **Planificación:** Se define el tiempo que nos costará desarrollar esta nueva funcionalidad, especificando claramente los resultados que deseamos generar.
- **Diseño e implementación:** Desarrollamos la nueva funcionalidad, primero estructurando claramente todas las posibles variantes, casos de uso, posibles problemas, ... y, posteriormente, implementándolo.
- **Batería de pruebas:** Se realizan las pruebas necesarias para comprobar que la nueva funcionalidad se comporta como nosotros habíamos deseado, y corroboramos que esta nueva etapa se integra perfectamente en el marco del proyecto.

4. Diseño e implementación

4.1. Arquitectura general

El primer objetivo que hemos intentado satisfacer, es la creación de una herramienta que permita acotar las funciones donde se han producido cambios en el paso de una versión a otra. Posteriormente, como tratábamos el tema de las versiones dentro de un mismo proyecto, decidimos integrar nuestra herramienta con *Subversion*, ya que es uno de los software de control de versiones más famosos. Por último, hemos añadido la forma de presentar los datos al usuario, eligiendo dos: formato texto o formato html. Lo podemos ver de forma más clara en la siguiente figura:

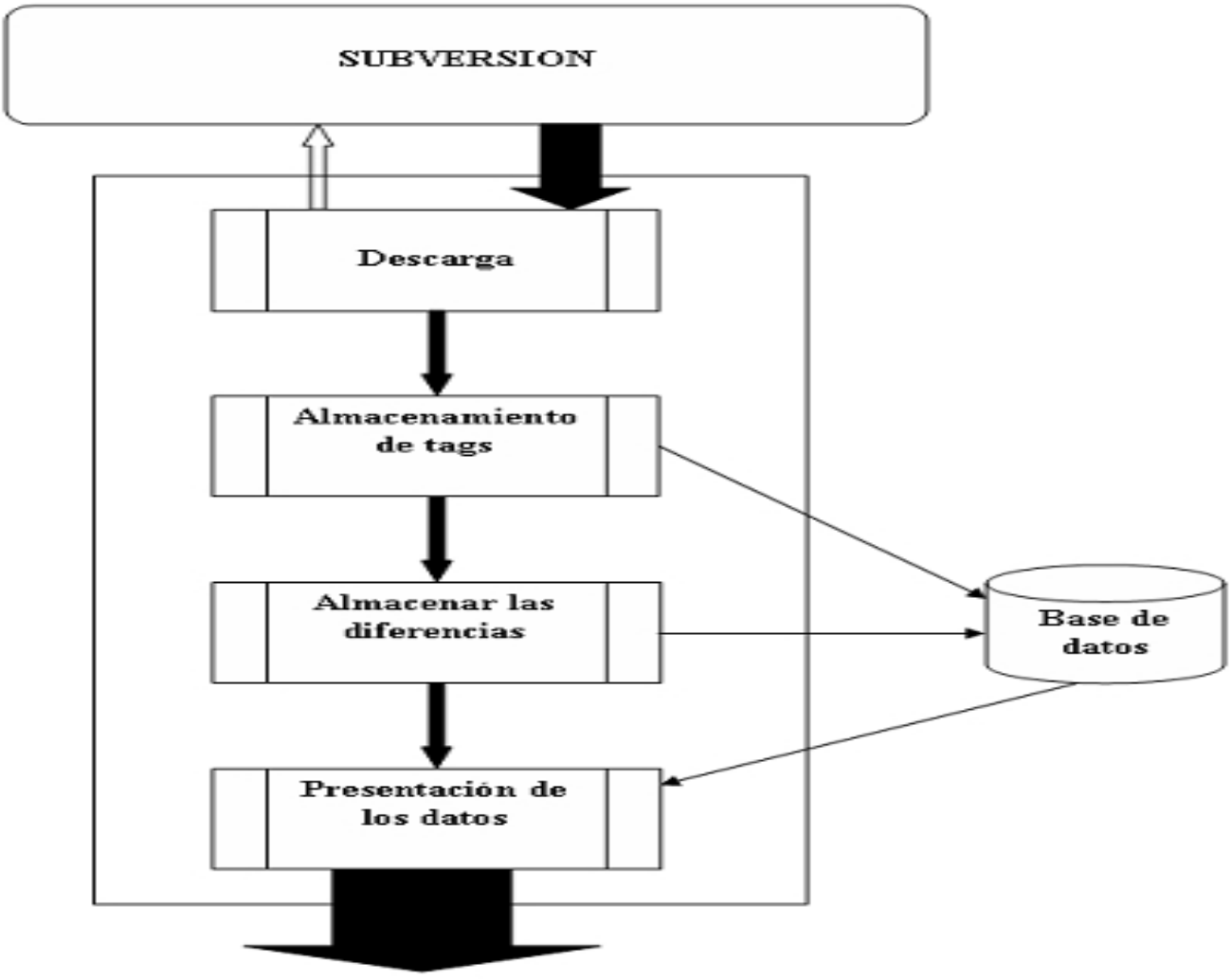


Figura 2: Diagrama de flujo

El funcionamiento de la herramienta consiste en pedir los datos al repositorio *Subversion* donde se encuentra el proyecto que queremos analizar, este proceso puede resultar largo, ya que depende del estado de la red, la capacidad del servidor donde se encuentra el repositorio, ...

Una vez conseguidos los datos, se analiza cada una de las diferentes versiones y se saca un resumen estructurado mediante la herramienta *Ctags*, donde se indicarán los nombres de cada una de las funciones, la línea donde están definidas, ...

Una vez hallada esta información, metemos los datos en una base de datos. Esta base de datos es temporal al uso del programa, una vez finalizada la herramienta, se borrará.

Cuando ya hemos almacenado los datos del análisis de cada una de las funciones, pasamos a comparar cada una de las versiones con la versión siguiente, para hallar, principalmente, las diferencias entre ellas. Estas diferencias serán:

- Funciones declaradas en la *versión n* y que no se encuentren en la *versión n+1*. De esta forma, hallamos las funciones eliminadas en el paso de una versión a otra.
- Funciones no declaradas en la *versión n* y que si se encuentran en la declaración de la *versión n+1*. Así podemos hallar las funciones añadidas en el paso de versiones.
- El nombre de la función donde se han añadido o eliminado líneas con respecto a la versión posterior.
- Por último, hallamos las diferencias entre los bloques de código que no se encuentran en ninguna función, tal como la declaración de estructuras de datos, importación de librerías, ...

Una vez sacada la información de las diferencias, se vuelve a hacer uso de la base de datos, guardando la información. En este momento, dependiendo de la opción que hallamos elegido para presentar los datos, se hará uso de un módulo u otro. Cada uno de estos módulos se encargará de revisar la base de datos y presentar la información de una forma estructurada.

Dado el modelo de desarrollo, podemos indicar el número de etapas que nos ha llevado construir este proyecto, y, nos permite especificar que se ha realizado en cada una de ellas. De esta forma, podemos deducir el siguiente esquema:

- **Etapa 0: Estudio previo.** Esta primera etapa se trata de un proceso de estudio vital para el buen desarrollo del proyecto. Se presentará como un acercamiento a las estructuras que

vamos a utilizar en el posterior desarrollo, profundizando en cada una de sus características.

- **Etapa 1: Desarrollo del módulo *Python* que obtiene las características principales de cada versión.** En esta etapa nos planteamos de que forma podemos sacar los datos que nos interesan a partir de la información que nos propone el software *Ctags*.

- **Etapa 2: Desarrollo del módulo *Python* que nos permite obtener las diferencias entre dos versiones consecutivas.** A partir de la información hallada en la etapa 1, desarrollamos un módulo de *Python* que, hallando las diferencias entre dos versiones consecutivas, nos permita indicar en que funciones del programa analizado se han producido dichas diferencias, almacenando los resultados en nuestra base de datos.

- **Etapa 3: Presentación de los datos.** En esta etapa, ya hemos podido almacenar en nuestra base de datos todos los datos dignos de mención. Por lo tanto, pasamos al desarrollo de módulos independientes que nos permitirán presentar los datos hallados previamente.

  - **Etapa 3.1: Salida en formato texto.** Desarrollo de un módulo de *Python* que nos permite sacar los datos en un fichero de texto plano.

  - **Etapa 3.2: Salida en formato HTML.** Desarrollo de un módulo de *Python* que nos permite sacar los datos a un fichero HTML.

- **Etapa 4: Añadiendo concurrencia a nuestra aplicación.** Examinando los tiempos de ejecución de nuestra aplicación, nos dimos cuenta que cuando el fichero que queremos evaluar tiene muchas versiones, los tiempos se disparan. Por eso llegamos a la conclusión que como mejora del proyecto deberíamos incluir concurrencia en las partes de nuestro código en las que ello sea posible.

- **Etapa 5: Creación de una interfaz gráfica.** Llegados a este punto, nos dimos cuenta que la gran mayoría de los usuarios están habituados a trabajar con las aplicaciones de una forma intuitiva, por eso nos decidimos a crear una interfaz gráfica que nos permita ejecutar el programa de una forma sencilla.

- **Etapa 6: Instalación/desinstalación de nuestro software.** En esta etapa final dentro del desarrollo, pensamos que la forma más fácil de instalar y desinstalar nuestro software es

a través de alguno de los gestores de paquetes que se encuentran habitualmente en las distribuciones Linux (aptitude, apt-get, yum, ...). Finalmente nos decidimos por crear un paquete RPM para que el proceso de instalación de nuestro proyecto sea posible realizarlo de una forma sencilla, eficiente y transparente.

- **Etapa 7: Batería de pruebas del software completo.** Para que una aplicación pueda dar el salto a producción, debe haber sido probado repetidamente, desde el proceso de instalación hasta el de presentación de datos, mostrando todas las pruebas realizadas y los resultados obtenidos.

- **Etapa 8: Caso de estudio de la aplicación comercial *Pidgin*.** En esta última etapa, se quiere mostrar como se utiliza la aplicación realizada para este TFM analiza un repositorio completo de *subversion*, mostrando distintas estadísticas y ofreciéndo una idea de la salud del proyecto que ha sido estudiado.

4.2. Etapa 0: Estudio previo.

4.2.1. Estructura de un programa escrito en lenguaje *C*

En esta sección, vamos a fijarnos en el formato de un programa escrito en *C*, ya que va a ser el lenguaje de programación soportado por nuestro proyecto. Vamos a llevar a cabo un exhaustivo análisis para facilitar el desarrollo posterior, ya que es fundamental diferenciar las distintas partes de las que se compone un programa.

Un programa escrito en *C*, tiene una estructura muy clara y delimitada, que aparece de la siguiente forma:

```
/*PARTE 1: Declaración de los módulos externos a importar.*/

#include <stdio.h>
#include <string.h>
...

/*PARTE 2: Definición de macros y constantes.*/

#define CIERTO 1
#define FALSO 0
#define NUM_PARAM 3
...
```

```
/*PARTE 3: Definición de estructuras de datos.*/

typedef struct{
    int dia;
    int mes;
    int anno;
}FECHA;
...

/*PARTE 4: Declaración e implementación de funciones.*/

int suma(int x,int y){
    return x+y;
}
...

/*Declaración de la función main.*/

int main(int argc,char *argv[]){
    int num_a,num_b,resultado;
    printf("Introduzca numero 1:\n");
    scanf("%d",&num_a);
    printf("Introduzca numero 2:\n");
    scanf("%d",&num_b);
    resultado=suma(num_a,num_b);
    printf("Resultado: %d\n",resultado);
    return EXIT_SUCCESS;
}

/*Fin del programa.*/
```

Como podemos observar, los programas escritos en *C*, tienen una estructura muy delimitada, completamente estructurada. De esta forma, hemos podido delimitar de forma clara y concisa, donde se han producido las modificaciones en el paso de una versión a la siguiente.

4.2.2. Estructura de la base de datos

En esta sección, vamos a indicar la estructura de la base de datos que utilizamos para almacenar la información.

Antes de todo, indicar que esta base de datos es temporal, la crearemos en cada ejecución que se haga de nuestro proyecto, y la eliminaremos cuando ya no vayamos a hacer uso de ella. De esta forma, sólo consumimos memoria un tiempo muy limitado, siendo totalmente

respetuosos con los recursos del sistema.

Al analizar la estructura de un programa en *C*, nos damos cuenta de que sólo necesitamos un par de tablas para almacenar nuestra información. Vamos a pasar a definir cada una de ellas.

Tabla 1: verfuncion. En esta tabla, sólo vamos a almacenar el nombre de la versión que estamos analizando, las funciones que componen la estructura de esta versión y la línea donde comienzan cada una de ellas. De esta forma, podemos indicar que la estructura de la tabla es la siguiente:

```
CREATE TABLE verfuncion(
      ver varchar2(50),
      funcion varchar2(50),
      linea int,
      PRIMARY KEY(ver,funcion));
```

```
NOMBRE_VERSION        NOMBRE_FUNCION        LINEA_DONDE_COMIENZA
r418.c                daemonise             57
r418.c                filesize              88
r418.c                tailf                 96
r418.c                helptext              143
r418.c                main                  151
...
...
```

Tabla 2: cambios. Esta otra tabla será la que almacene los cambios hallados en el paso de una versión a otra, por lo tanto será mucho más voluminosa. Los campos de esta tabla son el nombre de la versión, el nombre de la función donde se han producido los cambios, la línea donde se produjo el cambio, la iteración en la que se ha producido dicho cambio (pasaremos a explicar esto posteriormente) y la descripción del cambio realizado. La estructura de la tabla es la siguiente:

```
CREATE TABLE cambios(
      ver varchar2(50),
      funcion varchar2(50),
      linea varchar2(5),
      comp int,
      desc_cambio varchar2(500),
      PRIMARY KEY(ver,funcion,linea,comp),
      FOREIGN KEY(ver,funcion) REFERENCES verfuncion(ver,funcion));
```

```
NOMBRE_VERSION    NOMBRE_FUNCION    LINEA    COMP    DESC_CAMBIO
r415.c            helptext          d145     2       printf("Usage: g15tailf /path/to/file\n");
```

```
r416.c          helptext        a145    2       printf("Usage: g15tailf [options] /path/to/file\n");
...
...
```

Ahora vamos a pasar a explicar los campos de esta tabla:

- Los campos *ver* y *funcion* indican, respectivamente el nombre de la versión y el nombre de la función.
- El campo *linea*, en vez de estar definido como *int*, está definido como *varchar2* para poder indicar en la base de datos si se trata de una línea añadida (*a145*) o si se trata de una línea eliminada (*d145*).
- El campo *comp*, representa la iteración a la que pertenece este cambio. Este campo se incluyó para que no hubiese problemas de integridad en la base de datos, ya que si se producían dos cambios consecutivos en dos versiones sucesivas, se producía un error por repetición de claves. De tal forma, si tenemos tres versiones r415.c, r416.c y r417.c, la primera comparación entre las versiones r415.c y r416.c, será la iteración 0, y la comparación entre las versiones r416.c y r417.c será la iteración 1, y así sucesivamente. De esta forma, sabemos que se va a respetar en todo momento la integridad de los datos de esta tabla.
- El ultimo campo, indica la línea de código que ha sido añadida o eliminada.

4.3. Etapa 1: Desarrollo del módulo *Python* que obtiene las características principales de cada versión.

4.3.1. Diseño

En esta etapa, nos valemos de la información que nos ofrece el software *Ctags*, para sacar las principales características de la versión a tratar.

Esta es la información que nos presenta *Ctags* al analizar un archivo escrito en *C*:

```
!_TAG_FILE_FORMAT 2 /extended format; --format=1 will not append ;" to lines/
!_TAG_FILE_SORTED 1 /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_PROGRAM_AUTHOR Darren Hiebert /dhiebert@users.sourceforge.net/
!_TAG_PROGRAM_NAME Exuberant Ctags //
!_TAG_PROGRAM_URL http://ctags.sourceforge.net /official site/
```

```
!_TAG_PROGRAM_VERSION 5.8 //
daemonise r418.c /^int daemonise(int nochdir, int noclose) {$/;" line:57
filesize r418.c /^static size_t filesize(const char *filename)$/;" line:88
helptext r418.c /^void helptext() {$/;" line:143
main r418.c /^int main(int argc, char *argv[]){$/;" line:151
tailf r418.c /^static void tailf(g15canvas *canvas, const char *filename, int lines,int hoffset)$/;"
line:96
```

De este modo, podemos ver claramente como *Ctags* saca los datos siguiendo la siguiente estructura:

Columna 1: nombre de la función.

```
daemonise
```

Columna 2: nombre del programa o versión.

```
r418.c
```

Columna 3: declaración completa de la función.

```
int daemonise(int nochdir, int noclose);
```

Columna 4: número de línea donde comienza la función.

```
line:57
```

Ctags ofrece un gran número de opciones a configurar, permitiendo especificar al usuario el análisis que queremos hacer del programa. Es decir, dependiendo de las opciones que le pasemos al programa, nos dará una información u otra:

```
--fields=[+|-]flags

a   Access (or export) of class members
f   File-restricted scoping [enabled]
i   Inheritance information
k   Kind of tag as a single letter [enabled]
K   Kind of tag as full name
l   Language of source file containing tag
m   Implementation information
n   Line number of tag definition
s   Scope of tag definition [enabled]
S   Signature of routine (e.g. prototype or parameter list)
z   Include the "kind:" key in kind field
t   Type and name of a variable or typedef as "typeref:"
```

Ctags también ofrece opciones dependiendo del lenguaje de programación que queremos analizar. En nuestro caso, como queremos analizar un programa escrito en *C*, nos ofrece las siguientes opciones:

```
--c-kinds=[+|-]flags

C
    c  classes
    d  macro definitions
    e  enumerators (values inside an enumeration)
    f  function definitions
    g  enumeration names
    l  local variables [off]
    m  class, struct, and union members
    n  namespaces
    p  function prototypes [off]
    s  structure names
    t  typedefs
    u  union names
    v  variable definitions
    x  external and forward variable declarations [off]
```

En nuestro proyecto, ejecutamos *Ctags* indicándole que sólo nos interesa la opción *–c-kinds=+p*, para que nos indique los prototipos de las funciones definidas y la opción *–fields=+n* para que incluya el número de linea donde está definida la función. De esta forma, la instrucción completa que ejecutamos es la siguiente:

```
ctags --c-kinds=-c-d-e-g-m-n-s-t-u-v+p --fields=+n
```

4.3.2. Implementación

De esta forma, el módulo desarrollado en esta etapa se centra en sacar del archivo *tags* las características que queremos analizar.

En primer lugar, creamos una función de nuestro módulo de *Python* que se centre en sacar, por cada una de las funciones que tenemos en el programa analizado, el nombre de la función y la línea en la que comienza. Estos serán los datos que almacenaremos en la base de datos:

```
def sacar_datos_tags(version):
    """Función para sacar los datos del archivo tags"""
    #'version': version del programa tratado
    #'cadena': variable que nos permitirá saber si estamos analizando el archivo indicado
```

```
    #'dicc': diccionario que devolveremos para almacenar en la base de datos
    cadena=""
    dicc={}
    try:
        f_tags=file("tags")
    except:
        sys.stderr.write("Fallo al abrir el archivo tags\n")
        raise SystemExit

     while True:
        linea=f_tags.readline()
        if not linea: break
        if not linea.startswith("!"):
            cadena=linea.split('\t')
            if cadena[1]==version:
                aux=cadena[4].partition(":")
                dicc[int(aux[2])]=cadena[0]
    f_tags.close()
    return dicc
```

En esta segunda función, metemos los datos hallados anteriormente en nuestra base de datos. Más concretamente, esta función inserta los datos hallados en la tabla *verfuncion*, permitiendo delimitar claramente donde comienza y termina el bloque de código de cada función dentro de la versión analizada.

```
def meter_datos_verfuncion(version,dicc):
    """Función para meter los datos hallados previamente en la base de datos"""
    #'cursor_bbdd': variable que nos permite realizar las operaciones de manejo de la BBDD
    try:
        bbdd=dbapi.connect("test.db")
    except:
        sys.stderr.write("Fallo al abrir la base de datos\n")
        exit()
    cursor_bbdd=bbdd.cursor()
    func=dicc.keys()
    for i in func:
        cursor_bbdd.execute("""insert into verfuncion values(?,?,?)""",(version,dicc[i],i,))
        bbdd.commit()
    cursor_bbdd.close()
    bbdd.close()
```

4.4. Etapa 2: Desarrollo del módulo *Python* que nos permite obtener las diferencias entre dos versiones consecutivas.

4.4.1. Diseño.

Para el diseño de este módulo, decidimos hacer uso del mandato de UNIX *diff*, ya que ofrece una herramienta muy potente a la hora de comparar ficheros.

La estructura del fichero formado por el mandato *diff*, es el siguiente:

```
31c31
< $Revision: 414 $ -  $Date: 2008-01-15 03:52:06 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
---
> $Revision: 415 $ -  $Date: 2008-01-15 04:13:35 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
54a55
> #include <config.h>
141a143,150
> void helptext() {
>   printf("%s (c)2008 Mike Lampard, (c) 1996, 2003 Rickard E. Faith\n\n",PACKAGE_STRING);
>   printf("Usage: g15tailf /path/to/file\n");
>   printf("Options:\n--title (-t) \"Screen Title\" displays \"Screen Title\" at the top of each screen.\n");
>   printf("--daemon (-d) runs g15tailf in the background as a daemon\n");
>   printf("--offset 20 (-o) starts display at horizontal offset 20\n");
> }
>
153c162
<     int opts=1,title=0,name=0;
---
>     int opts=1,title=0;
169c174
<       if(0==strncmp(argv[i],"-d",2)) {
---
>       if(0==strncmp(argv[i],"-d",2)||0==strncmp(argv[i],"--daemon",8)) {
```

Vamos a explicar el formato en el que saca los datos el mandato *diff*:

- Cuando las líneas empiezan por un número de línea, dependiendo del formato que tengan, pueden indicar distintas cosas:
    - Si la línea es del tipo *54a55*, significa que se ha incluido una línea en el fichero que se le pasa como segundo parámetro.
    - Si la línea es del tipo *31c31*, significa que se ha producido un cambio similar sin alterar la composición de los ficheros.

- Si la línea es del tipo *55d32*, significa que se ha eliminado una línea en el fichero que se le pasa como segundo parámetro.

- Cuando las líneas no empiezan por números, lo que muestra es el cambio que se ha producido en el código de los fichero pasados como parámetro.

4.4.2. Implementación

Este módulo de *Python* no se va a incluir debido a su gran extensión, ya que nos quedaríamos sin espacio para el resto de la memoria.

Por ello, vamos a pasar a explicar lo que se ha hecho en dicho módulo:

- Se ha creado una función que se encarga de revisar el fichero creado por el comando *diff*, sacando los datos relevantes, tales como los cambios realizados, número de línea donde se ha producido el cambio...

- Otra función se encarga de, haciendo uso de la tabla *verfuncion* explicada anteriormente, dilucidar en que bloque de código se ha producido el cambio, ya sea en una función, en la declaración de las estructuras de datos, en la importación de módulos, ...

- La última función de este módulo se encarga de ingresar los datos obtenidos en la tabla *cambios* de nuestra base de datos, rellenando correctamente cada uno de los campos para que los módulos que se encargan de representar la salida de nuestro proyecto tengan a mano la mayor información posible.

4.5. Etapa 3: Presentación de los datos.

En esta nueva etapa, una vez que conseguimos tener toda la información guardada en nuestra base de datos, implementamos distintos módulos que se encargan de presentar la información al usuario del programa.

El usuario indica, por el uso de opciones, cual es el formato en el que le gustaría ver la salida de nuestro proyecto. Por ello, dependiendo de la opción elegida, se encargará de presentar la información un módulo u otro.

4.5.1. Etapa 3.1: Salida en formato texto.

En esta nueva subetapa, hemos implementado un módulo que sacará la información a un fichero de texto plano donde guardará la información.

La salida en este formato no será tan completa como la salida en formato HTML, pero si sólo ofreciéramos la salida de la información de nuestro proyecto en HTML, estaríamos limitando a muchos posibles usuarios, ya que deberían disponer de un interfaz gráfico para observar los resultados.

A la hora de presentar los datos, después de valorar distintas formas, nos decidimos por sacar por cada nueva versión del programa analizado los siguientes datos:

- **Funciones añadidas.** En este apartado facilitamos el código de las funciones que han sido añadidas, así como las líneas que ocupan en el nuevo fichero.
- **Funciones eliminadas.** En este otro apartado, ofrecemos el código de las funciones eliminadas, así como las líneas que ocupaban en el fichero antiguo.
- **Líneas añadidas.** También ofrecemos por cada línea que ha sido añadida, además del número de línea que ocupaba, a que bloque de código pertenecía, especificando el nombre de la función o, en su defecto, indicando que esa línea pertenecía al bloque de declaración de estructuras de datos, al de importación de módulos, ...
- **Líneas eliminadas.** Esta información es similar al apartado anterior, pero en vez de indicar qué líneas han sido añadidas, especificamos que líneas no se han considerado necesarias incluir en la nueva versión.

Para guardar la información que nos proporciona el software en un fichero de texto plano, deberemos invocarlo de la siguiente forma:

```
actualiza_funciones.py -t fichero.txt http://url_repositorio_subversion
```

Ahora, vamos a mostrar la salida de una ejecución de nuestro programa, en la que elegimos que nos guarde en un fichero de texto plano la información:

```
PFC de Daniel Dominguez
Tutor: Gregorio Robles
```

Paso de la version r413.c a la version r414.c

- No hay funciones eliminadas
- No hay funciones annadidas
- Lineas eliminadas en la declaración:

  31: $Revision:413 $- $Date: 2008-01-15 03:06:43 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $

- Lineas eliminadas en la funcion: main

  164: printf("ie. g15tailf /var/log/messages -t [\"Message Log\"]\n");

- Lineas anadidas en la declaracion:

  31: $Revision:414 $- $Date: 2008-01-15 03:52:06 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $

- Lineas anadidas en la funcion: main

  164: printf("ie. g15tailf -t [\"Message Log\"] /var/log/messages \n");

Paso de la version r414.c a la version r415.c

- No hay funciones eliminadas
- Funciones annadidas:

  143: void helptext() {
  144: printf("%s (c)2008 Mike Lampard, (c) 1996, 2003 Rickard E. Faith\n\n",PACKAGE_STRING);
  145: printf("Usage: g15tailf /path/to/file\n");
  146: printf("Options:\n--title(-t)\"Screen Title\"displays\"Screen Title\"at the top of each screen.");
  147: printf("--daemon (-d) runs g15tailf in the background as a daemon\n");
  148: printf("--offset 20 (-o) starts display at horizontal offset 20\n");
  149: }
  150:

- Lineas eliminadas en la declaracion:

  31: $Revision:414 $- $Date: 2008-01-15 03:52:06 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $

- Lineas eliminadas en la funcion: main

  153: int opts=1,title=0,name=0;
  161: printf("G15tailf (c) 2008 Mike Lampard\n");
  162: printf("Please put the /full/file/name to read on the cmdline and(optionally)title of the screen");
  163: printf("and try again.\n");
  164: printf("ie. g15tailf -t [\"Message Log\"] /var/log/messages \n");
  165: printf("if run with -d as the first option, g15tailf will run in the background\n");
  169: if(0==strncmp(argv[i],"-d",2)) {

```
    173: else if(0==strncmp(argv[i],"-t",2)) {
```

- Lineas anadidas en la declaracion:

```
    31: $Revision:415 $- $Date: 2008-01-15 04:13:35 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
    55: #include <config.h>
```

- Lineas anadidas en la funcion: main

```
    162: int opts=1,title=0;
    170: helptext();
    174: if(0==strncmp(argv[i],"-d",2)||0==strncmp(argv[i],"--daemon",8)) {
    178: else if(0==strncmp(argv[i],"-t",2)||0==strncmp(argv[i],"--title",7)) {
    183: else if(0==strncmp(argv[i],"-o",2)||0==strncmp(argv[i],"--offset",8)) {
    184: sscanf(argv[++i],"%i",&hoffset);
    185: opts+=2;
    186: }
    187: else if(0==strncmp(argv[i],"-h",2)||0==strncmp(argv[i],"--help",6)) {
    188: opts++;
    189: helptext();
    190: return 0;
    191: }
    230: if(hoffset<0)
    231: hoffset=0;
```

Paso de la version r415.c a la version r416.c

- No hay funciones eliminadas
- No hay funciones annadidas
- Lineas eliminadas en la declaracion:

```
    31: $Revision:415 $- $Date: 2008-01-15 04:13:35 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
```

- Lineas eliminadas en la funcion: helptext

```
    145: printf("Usage: g15tailf /path/to/file\n");
```

- Lineas anadidas en la declaracion:

```
    31: $Revision:416 $- $Date: 2008-01-15 04:18:07 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
```

- Lineas anadidas en la funcion: helptext

```
    145: printf("Usage: g15tailf [options] /path/to/file\n");
```

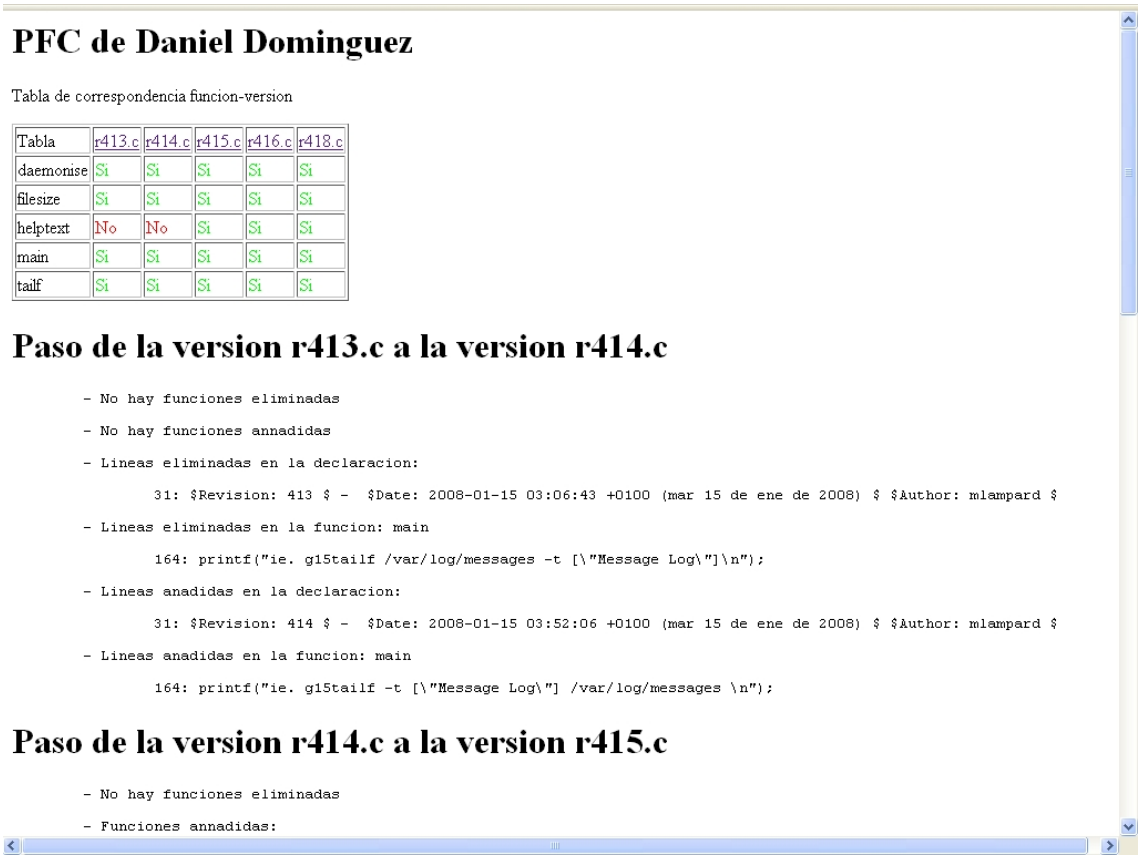
Paso de la version r416.c a la version r418.c

```
- No hay funciones eliminadas
- No hay funciones annadidas
- No hay lineas eliminadas
- No hay lineas anadidas
```

4.5.2. Etapa 3.3: Salida en formato HTML.

Aparte de ofrecer los datos en formato texto, decidimos ofrecer la información en formato HTML, ya que es un lenguaje muy versátil que nos permitía ofrecer una información más completa.

No podemos mostrar el código tal como lo hemos mostrado en el apartado anterior, debido a las limitaciones de LaTeX. Por lo tanto se ofrece una captura de la información de salida en formato HTML:



PFC de Daniel Dominguez

Tabla de correspondencia funcion-version

Tabla	r413.c	r414.c	r415.c	r416.c	r418.c
daemonise	Si	Si	Si	Si	Si
filesize	Si	Si	Si	Si	Si
helptext	No	No	Si	Si	Si
main	Si	Si	Si	Si	Si
tailf	Si	Si	Si	Si	Si

Paso de la version r413.c a la version r414.c

```
- No hay funciones eliminadas
- No hay funciones annadidas
- Lineas eliminadas en la declaracion:
        31: $Revision: 413 $ -  $Date: 2008-01-15 03:06:43 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
- Lineas eliminadas en la funcion: main
        164: printf("ie. g15tailf /var/log/messages -t [\"Message Log\"]\n");
- Lineas anadidas en la declaracion:
        31: $Revision: 414 $ -  $Date: 2008-01-15 03:52:06 +0100 (mar 15 de ene de 2008) $ $Author: mlampard $
- Lineas anadidas en la funcion: main
        164: printf("ie. g15tailf -t [\"Message Log\"] /var/log/messages \n");
```

Paso de la version r414.c a la version r415.c

```
- No hay funciones eliminadas
- Funciones annadidas:
```

Figura 3: Imagen de la información en formato html

Como podemos ver, en la salida con formato HTML, facilitamos una tabla donde se puede ver, de forma simple y explícita los cambios que ha sufrido el proyecto analizado con el paso de sus versiones.

En esta tabla, se explica una relación entre todas las versiones que componen el programa y todas las funciones que forman o han formado parte de este proyecto. Por cada versión se indica si una determinada función forma parte de ella o no, si se han sufrido cambios en una función en el paso de una versión a otra, ...

Lo que se quiere mostrar con esta tabla es una visión breve y completa de los cambios sufridos por un proyecto en el continuo paso de las versiones. Si se quiere ampliar la información, sólo es necesario hacer uso del vínculo que indica el número de versión para ver las líneas que han sufrido modificaciones, cuales han sido, ...

4.6. Etapa 4: Añadiendo concurrencia a nuestra aplicación.

En esta nueva etapa, una vez que hemos conseguido los requisitos básicos de la aplicación, nos debemos enfrascar en conseguir que el rendimiento sea el más óptimo posible.

4.6.1. Diseño.

Para conseguir mejorar el rendimiento, examinamos el código de nuestro programa y nos dimos cuenta que perdíamos mucho tiempo al utilizar un enfoque secuencial. La forma en la que estaba diseñado era muy lenta, ya que al pedir cada una de las versiones del fichero a examinar, sacar las características de la versión o examinar las diferencias entre versión y versión lo hacíamos secuencialmente.

Tal como tenemos diseñada nuestra aplicación, hemos considerado conveniente paralelizar tres partes fundamentales de ella:

4.6.2. Petición de las versiones a *Subversion*.

Uno de los principales problemas de rendimiento que teníamos en nuestra versión anterior de la aplicación era que se realizaban las peticiones de cada una de las versiones a *Subversion* de forma secuencial. Es decir, que sólo hacíamos una petición una vez que teníamos una copia guardada de la petición anterior.

En la nueva versión de nuestra aplicación, al introducir concurrencia y el uso de *threads* lo que conseguimos es que todas las peticiones se hagan de forma concurrente. Es decir, en un primer momento pedimos tantas versiones como existan del fichero que estamos examinando.

Una vez realizadas las peticiones, de las cuales se encargan cada uno de los procesos hijo que hemos lanzado, cada uno de estos *threads* se encarga de escribir y guardar en un fichero una copia de la versión almacenada en *Subversion*.

Realizándolo de esta forma nos hemos encontrado con un aumento en el rendimiento brutal, ya que el tiempo en el que ahora se descargan todas las versiones es el tiempo de la descarga del *thread* más lento, mientras que antes el tiempo de la descarga era la suma absoluta de cada una de nuestras descargas.

Con esta mejora y realizando diferentes pruebas, nos hemos dado cuenta que los tiempos de ejecución de nuestro programa se reducían entre un 20 % y un 40 %, dependiendo de elementos externos como la latencia de la red, el estado de carga del servidor *Subversion* al que le estamos realizando las peticiones o el estado de sobrecarga de nuestro propio sistema.

4.6.3. Hallar las características de la versión con *Ctags*.

Otro problema de rendimiento que teníamos en nuestro diseño anterior es que sacábamos las características de cada una de las versiones del programa a examinar de forma secuencial. Es decir, procesábamos una versión y hasta que no acabábamos no empezabamos a procesar la siguiente.

En la nueva versión de nuestra aplicación, al introducir concurrencia y el uso de *threads* lo que conseguimos es sacar las características de las versiones de forma concurrente. Es decir, en un primer momento lanzamos procesos hijos que se encargarán cada uno de ellos en coger una versión y mediante el uso de *Ctags* sacar sus principales características. Una vez realizado este proceso, cada uno de los procesos hijo lanzados se encarga de escribir en la base de datos que tenemos en nuestra aplicación los aspectos que nos interesan a nosotros para poder evaluar correctamente y de forma eficiente el programa elegido.

Con la introducción de la concurrencia en esta parte del programa, nos encontramos con un aumento espectacular del rendimiento, ya que el tiempo en el que ahora se evalúan y almacenan los datos de las versiones es el tiempo del *thread* más lento, mientras que en la versión anterior era la suma absoluta de cada una de nuestras evaluaciones con *Ctags*.

Con esta mejora y despues de distintas pruebas, nos encontramos con que hemos vuelto a reducir el tiempo de ejecución de nuestro programa entre un 20 % y un 30 %. La mejora en el rendimiento se nota más cuantas más versiones debemos evaluar, ya que es cuando la

paralelización que hemos realizado en nuestro código muestra todo su potencial.

4.6.4. Sacar las diferencias entre las versiones del programa examinado.

Otro aspecto en el que vimos que utilizando concurrencia podíamos aumentar el rendimiento de nuestro programa es el de evaluar cada una de las versiones con su versión consecutiva, ya que lo estábamos realizando de forma secuencial.

El proceso que estábamos siguiendo era el de examinar cada una de las versiones con la versión inmediatamente posterior de forma secuencial, siendo el tiempo de ejecución total de esta sección el sumatorio de cada una de nuestras evaluaciones de las diferencias entre versiones. Sin embargo con el nuevo enfoque, lo que hacemos es lanzar un proceso hijo por cada par de versiones que queremos evaluar, siendo el tiempo de ejecución total el tiempo de ejecución del *thread* más lento.

Una vez más, cuando más reducimos el tiempo de ejecución del programa es cuando tenemos que evaluar las diferencias de muchas versiones, ya que en un enfoque secuencial sería cuando más penalización tendríamos mientras que un enfoque concurrente la penalización no se nota apenas.

Con la paralelización de código en esta sección de nuestro programa, hemos advertido mediante múltiples pruebas que reducíamos el tiempo de ejecución del programa en un 25 %, aunque sigue siendo una de las partes del programa más pesadas.

4.6.5. Métodos utilizados para evitar problemas típicos de concurrencia.

Uno de los problemas que tenemos al utilizar la concurrencia es que con este modelo, la programación se hace más complicada, ya que no tienes un solo hilo de ejecución, si no que tienes muchos de ellos.

Principalmente, los problemas que tenemos con la concurrencia son los siguientes:

- **Sincronización de los *threads*.**

  Al emplear el uso de *threads* en nuestro programa y tener múltiples hilos de ejecución se complica mucho el cuándo y el cómo debemos paralelizar nuestra aplicación. Al no estar paralelizado todo el código, si no sólo las partes en las que la utilización de este estilo de programación nos proporciona unas ventajas indiscutibles, debemos investigar

adecuadamente cuales son los puntos en los que nuestro programa se separa en hilos y los puntos en los que debemos esperar a que cada uno de estos hilos finalice su ejecución.

```
for i in versiones:
        gtk.threads_enter()
        try:
                buffer.insert_at_cursor('Hallando los tags de '+str(i)+'...\n')
        finally:
                gtk.threads_leave()
                iteraciones=iteraciones+1
                hilo=threading.Thread(target=halla_tags,args=(i,iteraciones,))
                hilo.start()
                threads.append(hilo)

for t in threads:
t.join()
```

En este fragmento de código que hemos mostrado, tenemos un ejemplo representativo del problema de sincronización que obtenemos al utilizar el modelo de concurrencia. Como es posible ver, por cada una de las versiones que nos hemos descargado del repositorio *Subversion* lo que hacemos es lanzar un *thread* que se encarga de examinar cada uno de estos ficheros independientemente.

El problema viene cuando tenemos que volver a juntar todos los *threads* para proseguir con la ejecución normal de nuestro programa, ya que si alguno de estos hilos no termina adecuadamente o se nos olvida esperarle, nuestra aplicación tendrá problema de robustez y será una aplicación poco estable y propensa a fallos.

- **Acceso a variables compartidas.**

  Cuando programamos de forma concurrente, debemos tener mucho cuidado al acceder a variables compartidas por todos los hilos de ejecución, ya que la coherencia y la consistencia de nuestros datos se ve amenazada, por problemas tales como condiciones de carrera o inanición.

  El problema es cuando dos o más hilos de ejecución quieren acceder simultaneamente a una misma variable, ya que si uno escribe y el otro lee o, los dos escriben, ¿quién nos puede asegurar que el orden en el que se han ejecutado las instrucciones es el que nosotros habíamos proyectado?

Por este razón, todos los lenguajes de programación que permiten el modelo de concurrencia tienen herramientas a disposición de los programadores que nos ayudan a evitar estos problemas comentados.

En nuestro caso, el principal punto de peligro es cuando los *threads* tienen que acceder a la base de datos, ya que debemos establecer una política que les obligue a acceder de forma ordenada para mantener la consistencia de las tablas que estamos utilizando.

```
mutex=threading.Lock()

def meter_datos_verfuncion(version,dicc):
        global mutex
        mutex.acquire()
        try:
                bbdd=dbapi.connect("test.db")
        except:
                sys.stderr.write("Fallo al abrir la base de datos\n")
                raise SystemExit
        cursor_bbdd=bbdd.cursor()
        func=dicc.keys()
        for i in func:
                cursor_bbdd.execute("""insert into verfuncion values(?,?,?)""",(version,dicc[i],i,))
                bbdd.commit()
        cursor_bbdd.close()
        bbdd.close()
        mutex.release()
```

En las líneas de código mostradas anteriormente, tenemos un ejemplo de como establecer una política para que sólo un *thread* pueda acceder a la base de datos a la vez. De esta forma, podemos asegurar que la coherencia de los datos utilizados que debemos exigir está asegurada.

4.6.6. Uso de concurrencia: conclusiones.

Como conclusión, al hacer uso de un modelo concurrente en el desarrollo de nuestro software, podemos indicar que hemos obtenido una disminución en los tiempos de ejecución de nuestro software gratamente sorprendentes.

Hemos podido observar que dependiendo del volumen del proyecto a examinar conseguimos una reducción entre un 75 % y un 90 % del tiempo de ejecución entre la versión concurrente y la versión secuencial.

Al solventar adecuadamente los problemas inherentes al uso de la concurrencia, también podemos asegurar que nuestro programa será siendo igual de robusto que antes de aplicar esta mejora pero con una mejora en el rendimiento muy llamativa.

4.7. Etapa 5: Creación de una interfaz gráfica.

Una vez llegados a este estado en el desarrollo de nuestro proyecto, nos dimos cuenta que para que nuestro software tuviese mayor aceptación iba a ser indispensable desarrollar una interfaz gráfica que nos permitiese utilizar de forma sencilla e intuitiva todos los aspectos de nuestro programa descritos anteriormente.

Para desarrollar interfaces gráficas con *Python* tenemos multitud de módulos que podemos utilizar, ya que al ser un lenguaje de programación relativamente nuevo tiene una *API* muy completa para desarrollo gráfico. Al final, y después de una investigación profunda y ecuánime nos decidimos a utilizar el módulo *pygtk*, ya que es un módulo con muy buenas referencias y ha sido utilizado para desarrollar entornos tan reconocidos como *GNOME*.

4.7.1. Diseño.

A la hora de realizar el diseño, tuvimos claro que debíamos permitir al usuario utilizar nuestra aplicación de la forma más flexible posible pero también garantizando que su uso fuese sencillo. De esta forma, los usuarios noveles de nuestro software no echarán en falta un manejo más simple de la herramienta ni los usuarios más experimentados echarán de menos un análisis exhaustivo y eficiente de un repositorio de *Subversion*.

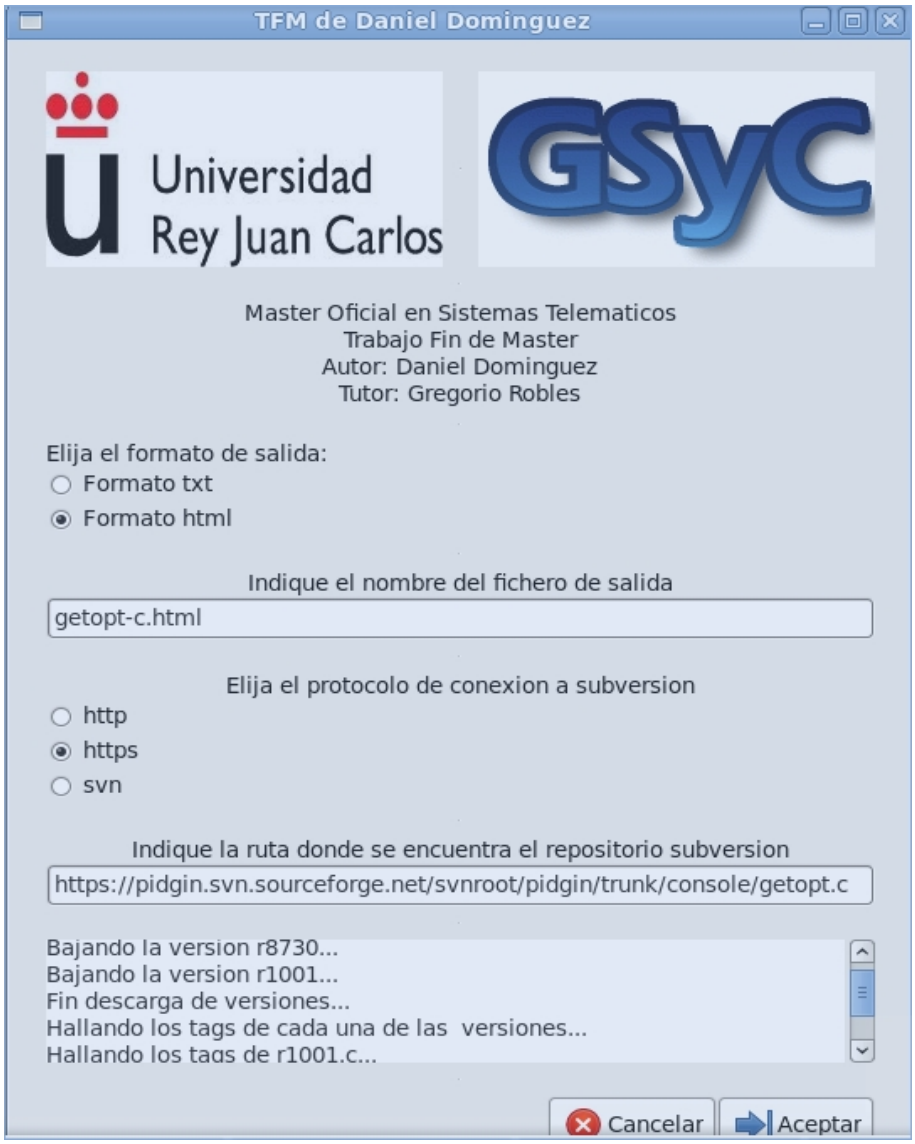


Figura 4: Imagen de de la interfaz gráfica diseñada para nuestro software

Como podemos ver en la imagen, hemos desarrollado una interfaz simple e intuitiva con todas las opciones que podemos utilizar para permitir una flexibilidad total en el uso de nuestra aplicación.

Una de las opciones que podemos elegir es el formato de salida en el que queremos visualizar los datos. Por un lado, en formato texto, que nos permite un análisis menos exigente en los requisitos de la máquina cliente y, por otro lado, en formato HTML, que nos permitirá un análisis más exhaustivo y a mayor profundidad del proyecto elegido.

Otra de las opciones que podemos configurar es la forma en la que el repositorio *Subversion* donde se encuentra alojado el proyecto que queremos examinar acepta conexiones. Por defecto,

nuestro software permite tres tipos de protocolo: HTTP, HTTPS y SVN.

A continuación, habilitamos una caja de texto donde el usuario debe poner la URL donde se encuentra el fichero del que quiere realizar el análisis.

También hemos habilitado un espacio donde nuestra aplicación proporciona *feedback* al usuario indicándole en que paso de la ejecución se halla en cada momento. De esta forma, en caso de que el proyecto a examinar sea muy voluminoso y nos lleve tiempo procesar toda la información, el usuario podrá comprobar que la aplicación cliente sigue respondiendo y no se ha quedado congelada o ha tenido problemas en su ejecución.

4.8. Etapa 6: Instalación/desinstalación de nuestro software.

Una de las cosas fundamentales que marca el éxito o el frácaso de una aplicación puede ser la forma en la que se instala y configura el software proporcionado. En una primera vuelta de reconocimiento, programamos un *script* para llevar a cabo la instalación.

Esta solución nos funcionó bastante bien, pero después de darle un uso más intensivo a nuestro software y ver como las dependencias y el uso de otros programas iba aumentando nos decidimos a integrarlo en un paquete que un gestor de paquetes (tales como *aptitude*, *apt-get*, *yum*, ...) pudiese instalar de forma totalmente automática.

Al final, optamos por desarrollar un paquete RPM para instalar y desinstalar nuestro software. La razón de esta opción elegida es que la distribución Red Hat y sus derivados (*Fedora*, *CentOS*, *Scientific Linux*, ...) son ampliamente utilizados por toda la comunidad Linux alrededor del mundo.

4.8.1. Creación de un paquete RPM.

Para crear un paquete RPM debemos seguir una serie de reglas para que nuestro software pueda ser empaquetado correctamente. Estos son los pasos que seguimos para conseguir un paquete RPM que nos permite que un gestor de paquetes como *yum* pueda realizar las operaciones de instalación y desinstalación de nuestro software:

- **Instalación de los paquetes necesarios para crear un paquete RPM.**

```
root@dani-laptop:~# yum install -y rpm-build make
```

- **Creación del árbol de directorios necesario para crear un paquete RPM.**

  Para crear un paquete RPM es necesario tener un árbol de directorios definido y con unos nombres específicos, ya que si no cuando el sistema intente construir nuestro paquete, fallará. En total, debemos tener los siguientes directorios: *BUILD*, *BUILDROOT*, *RPMS*, *SOURCES*, *SPECS* y *SRPMS*.

```
root@dani-laptop:~# mkdir -p /tmp/redhat/{BUILD,BUILDROOT,RPMS,SOURCES,SPECS,SRPMS}
```

- **Empaquetado del código fuente de nuestro programa.**

  Para que la herramienta que construye los paquetes RPM pueda examinar correctamente nuestro código fuente, exige que éste debe ser pasado comprimido con gzip. Por lo tanto, debemos comprimirlo y ponerlo en la carpeta *SOURCES*.

```
root@dani-laptop:~# tar -czvf svnexplorer.tar.gz svnexplorer/
root@dani-laptop:~# mv svnexplorer.tar.gz /tmp/redhat/SOURCES/
```

- **Creación del fichero *SPEC*.**

  En uno de los últimos pasos en la creación de un paquete RPM, debemos crear un fichero llamado *svnexplorer.spec* que contendrá los pasos a seguir en la instalación y configuración de nuestro software.

```
root@dani-laptop:~# cd /tmp/redhat/SPECS/
root@dani-laptop:~# vim svnexplorer.spec

Name: svnexplorer
Version: 1.0
Release: 11%{dist}
Summary: A program that explores svn spaces

Group: System Environment/Base
License: GPL
Source: %{name}.tar.gz
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root

Requires: python, sqlite, ctags, subversion, libnotify

%description
Este programa permite explorar un repositorio subversion para ver los cambios
habidos entre las distintas versiones de un mismo programa
```

```
%prep
%setup -n svnexplorer

%install
rm -rf %{buildroot}
mkdir -p "$RPM_BUILD_ROOT/usr/share/applications"
cp svnexplorer.desktop "$RPM_BUILD_ROOT/usr/share/applications"
mkdir -p "$RPM_BUILD_ROOT/usr/share/icons"
cp svn.png "$RPM_BUILD_ROOT/usr/share/icons"

mkdir -p "$RPM_BUILD_ROOT/usr/local/lib/svnexplorer"
mkdir -p "$RPM_BUILD_ROOT/usr/local/bin"
cp aceptar.png cancelar.png gsyc.gif README svnexplorerlib.py urjc.jpg "$RPM_BUILD_ROOT/usr/local/lib/sv
cp svnexplorer "$RPM_BUILD_ROOT/usr/local/bin"


%files
/usr/local/bin/svnexplorer
/usr/local/lib/svnexplorer
/usr/share/applications/svnexplorer.desktop
/usr/share/icons/svn.png


%clean
rm -rf $RPM_BUILD_ROOT


%post
chown root:root -R /usr/local/lib/svnexplorer
chmod 755 /usr/local/bin/svnexplorer
echo "PYTHONPATH=/usr/local/lib/svnexplorer" >> /etc/profile
notify-send -i /usr/share/icons/svn.png -t 10000 "SVNExplorer: Instalacion completada. Es necesario cer
```

Vamos a parar un poco en los detalles de este fichero ya que es el principal responsable de que un paquete RPM pueda ser creado correctamente o no.

En el apartado *Requires*, indicamos que para que este software pueda ser instalado en un sistema, deben estar instalados los paquetes *Python*, *SQLite*, *Ctags*, *Subversion* y *libnotify*

En el apartado *%install* lo que hacemos es crear los directorios donde nuestro software necesita crear ficheros. Como podemos ver creamos los siguientes directorios:

- **/usr/share/applications:** Este directorio es necesario para crear un enlace directo a

nuestra aplicación desde el menu de GNOME.

- **/usr/share/icons:** Este directorio es necesario para darle una imagen al enlace directo que queremos crear en GNOME.
- **/usr/local/lib/svnexplorer:** Creamos este directorio donde vamos a almacenar la librería que utiliza nuestro software y las imágenes de las que hacemos uso.
- **/usr/local/bin:** En este directorio vamos a colocar nuestro ejecutable.

Por último en el apartado *%post* le estamos indicando cuáles son las instrucciones que queremos que ejecute antes de cerrar la instalación de nuestro software. Nosotros hemos definido algunas como cambiar el propietario, los permisos, definir y exportar una variable de entorno o notificar que el software ha sido instalado.

- **Creación del paquete RPM.**

  Después de todos estos pasos, sólo nos queda el paso final. Lo que debemos hacer es ejecutar *rpmbuild* para que nos fabrique nuestro nuevo paquete RPM:

```
root@dani-laptop:~# rpmbuild -v -bb /tmp/redhat/SPECS/svnexplorer.spec
```

4.9. Etapa 7: Batería de pruebas del software completo.

Para esta etapa, vamos a mostrar una de las pruebas realizadas en nuestra batería de pruebas. Se mostrará todo el proceso, desde la instalación de nuestro software, hasta los datos mostrados en formato HTML y en formato texto.

4.9.1. Prueba de un proyecto.

Para nuestra prueba, elegimos un proyecto que contase con un par de revisiones, para ver si la salida era correcta y nuestro software no presentaba fallos en su fase final. Al final nos decidimos por un proyecto que desarrolla un *plugin* para la aplicación de mensajería instantánea *Gaim*.

Ejecutamos el proyecto indicándole que queremos la información mostrada por salida estándar:

Figura 5: Imagen de la interfaz gráfica configurada para la prueba

Y aquí tenemos los datos en formato texto:

```
    PFC de Daniel Dominguez


Paso de la version r1.c a la version r4.c

  - No hay funciones eliminadas
  - No hay funciones annadidas
  - Lineas eliminadas en la funcion: plugin_load

    579: /*
    580: static GList *
    581: multiblock_actions (GaimPlugin *plugin, gpointer context)
    582: {
```

```
583: GaimPluginAction *act = NULL;
584:
585: act = gaim_plugin_action_new (_("Manage blocking groups"), dialog_manage_groups);
586: actions = g_list_append (actions, act);
587:
588: return actions;
589: }
590: */
591:

- No hay lineas anadidas
```

En la salida de nuestro programa podemos observar los siguientes apartados:

- **Funciones eliminadas.** En este primer apartado, el programa nos muestra si se ha eliminado alguna función en el paso de versiones. En este caso, no se eliminó ninguna función, por lo que el programa nos lo indica mostrando la cadena:

  ```
  - No hay funciones eliminadas
  ```

- **Funciones añadidas.** En este segundo apartado, el programa nos muestra si se ha añadido alguna función en el paso de versiones. En este caso, no se añadió ninguna función, por lo que el programa nos lo indica mostrando la cadena:

  ```
  - No hay funciones annadidas
  ```

- **Líneas eliminadas.** En este bloque, la salida del programa nos muestra las líneas que han sido eliminadas en una función. Estos cambios se muestran indicando que se han eliminado líneas y a continuación muestra el nombre de la función que ha sufrido dichos cambios. Posteriormente, muestra cada uno de los cambios con el siguiente formato:

  ```
  número de linea: línea de código eliminada
  Ejemplo:
  - Lineas eliminadas en la funcion: plugin_load

      579: /*
      580: static GList *
      581: multiblock_actions (GaimPlugin *plugin, gpointer context)
      ...
  ```

- **Líneas añadidas.** En este bloque, la salida del programa nos muestra las líneas que han sido añadidas en una función. Estos cambios se muestran indicando que se han añadido líneas y a continuación muestra el nombre de la función que ha sufrido dichos cambios. El formato con el que saca las líneas de código añadidas es similar al formato de las líneas eliminadas.

Ahora, volvemos a ejecutarlo indicándole que nos gustaría ver los datos en formato HTML:

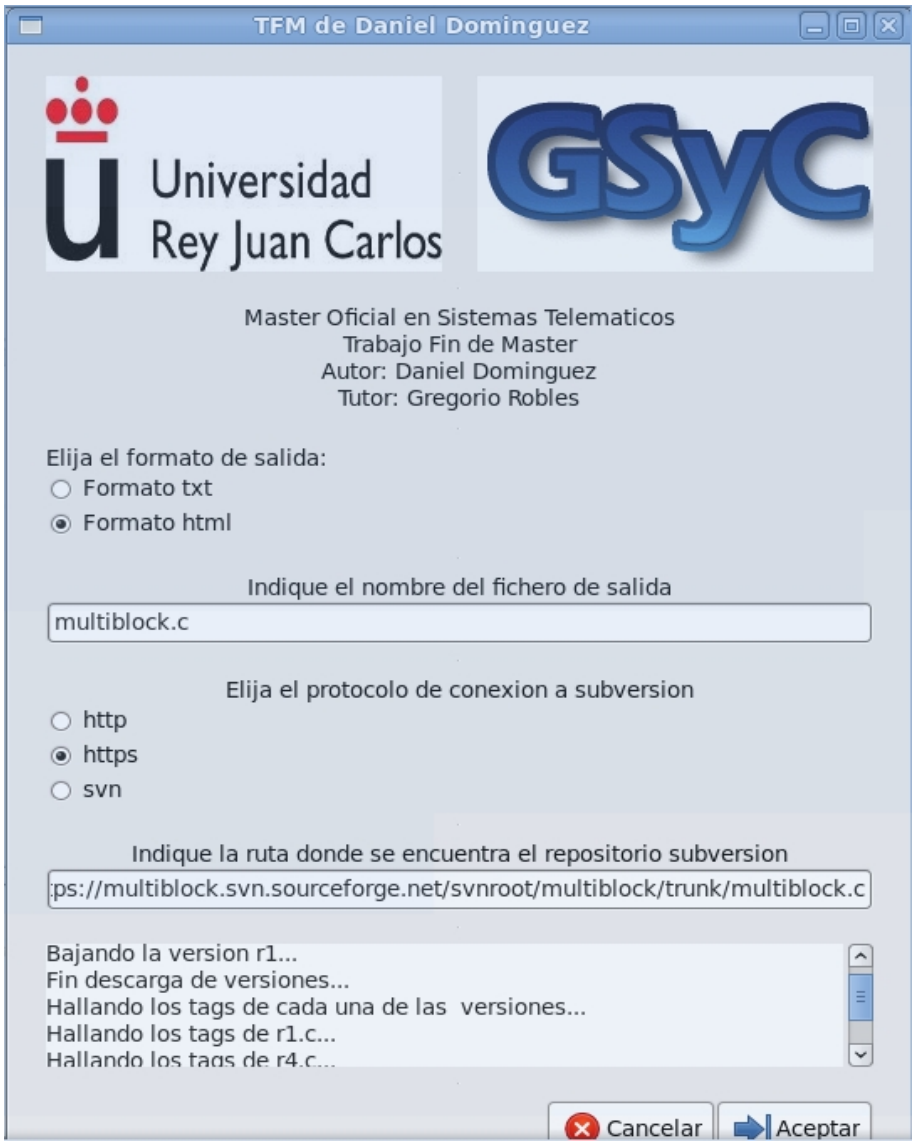


Figura 6: Imagen de la interfaz gráfica configurada para la prueba

Y estos son los datos ofrecidos por el software:

PFC de Daniel Dominguez

Tabla de correspondencia funcion-version

Tabla	r1.c	r4.c
blocked_toggled_cb	Si	Si
buddies_list_store_set_cb	Si	Si
change_new_type_cb	Si	Si
create_group	Si	Si
delete_group	Si	Si
destroy_manage_window_cb	Si	Si
dialog_manage_groups	Si	Si
dialog_new_group	Si	Si
get_plugin_pref_frame	Si	Si
group_block_cb	Si	Si
init_plugin	Si	Si
multiblock_extended_menu_cb	Si	Si
plugin_load	Si	Si
prefs_load_cb	Si	Si
prefs_write_cb	Si	Si

Paso de la version r1.c a la version r4.c

- No hay funciones eliminadas

- No hay funciones annadidas

- Lineas eliminadas en la funcion: plugin_load

```
579: /*
580: static GList *
581: multiblock_actions (GaimPlugin *plugin, gpointer context)
582: {
```

Figura 7: Imagen de la salida en formato html

Como podemos ver en la figura anterior, vemos una tabla donde se especifica el número de versiones que componen el proyecto analizado, indicando por cada versión si una determinada función está implementada. Además, por cada función se especifica si se han añadido o eliminado líneas en el paso de una versión a la siguiente.

Aparte de la tabla, la página en formato HTML que se muestra, dispone de una información más extensa en la parte inferior (accesible mediante vínculos desde la tabla) donde se muestran los cambios exactos que se han producido en una determinada función.

4.10. Etapa 8: Caso de estudio de la aplicación comercial *Pidgin*.

Para comprobar que nuestro software puede ser realmente útil y terminar con el periodo de pruebas del software, hemos decidido realizar un caso de estudio. En este caso la aplicación elegida es *Pidgin*, ya que la gran mayoría de su código está escrito en *C*, está alojado en un repositorio *Subversion* y se trata de un proyecto de gran envergadura.

Los datos que queremos obtener son unas gráficas y unas estadísticas que nos permitan observar de forma sencilla y rápida si el proyecto ha tenido un desarrollo continuado, si con cada cambio de versión han sido muchas las funciones añadidas o eliminadas o el número de líneas que han sido añadidas o eliminadas en el paso de una versión a la siguiente.

4.10.1. Versiones.

En primer lugar, lo que queremos saber son las versiones existentes de cada uno de los ficheros que componen *Pidgin*:

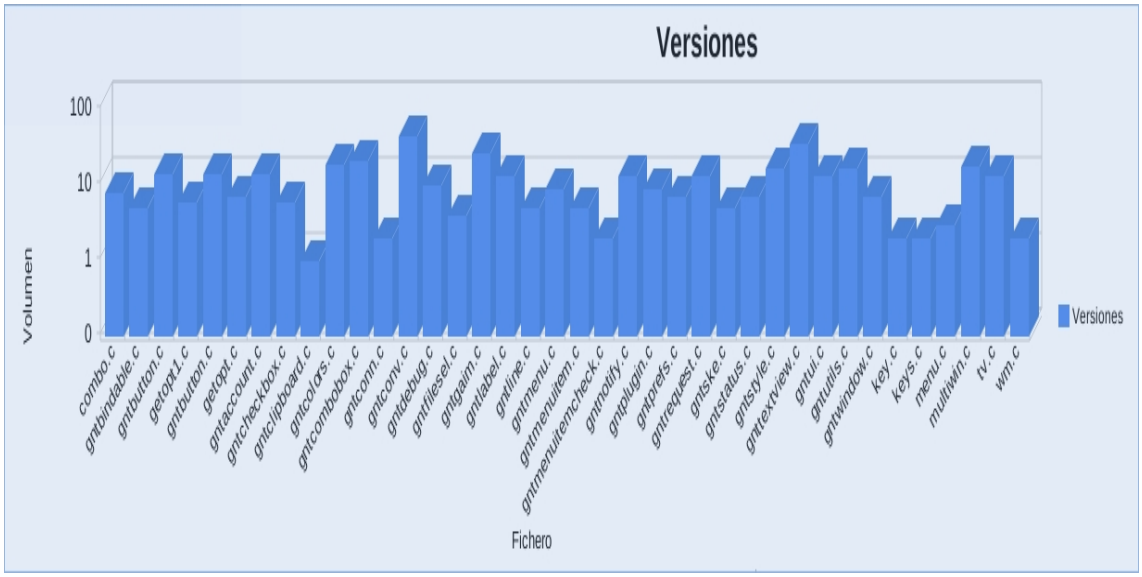


Figura 8: Gráfico con el número de versiones por cada uno de los ficheros que componen *Pidgin*

A continuación mostramos una tabla con el número exacto de versiones por fichero:

Nombre fichero	Número de versiones
combo.c	8
gntbindable.c	5
gntbutton.c	14
getopt1.c	6
gntbutton.c	14
getopt.c	7
gntaccount.c	14
gntcheckbox.c	6
gntclipboard.c	1
gntcolors.c	19
gntcombobox.c	21
gntconn.c	2
gntconv.c	45
gntdebug.c	10
gntfilesel.c	4
gntgaim.c	26
gntlabel.c	13
gntline.c	5
gntmenu.c	9
gntmenuitem.c	5
gntmenuitemcheck.c	2
gntnotify.c	13
gntplugin.c	9
gntprefs.c	7
gntrequest.c	13
gntske.c	5
gntstatus.c	7
gntstyle.c	17
gnttextview.c	35

Nombre fichero	Número de versiones
gntui.c	13
gntutils.c	17
gntwindow.c	7
key.c	2
keys.c	2
menu.c	3
multiwin.c	18
tv.c	13
wm.c	2

Una vez obtenidos estos datos, nos interesa saber cuál es el número medio de versiones por cada uno de los ficheros analizados. Realizando una división entre el número total de versiones obtenidas entre el número total de ficheros analizados, nos queda que el número medio de versiones por fichero es **11**

4.10.2. Funciones añadidas.

A continuación, queremos saber el número de funciones añadidas en cada uno de los ficheros que componen *Pidgin*:

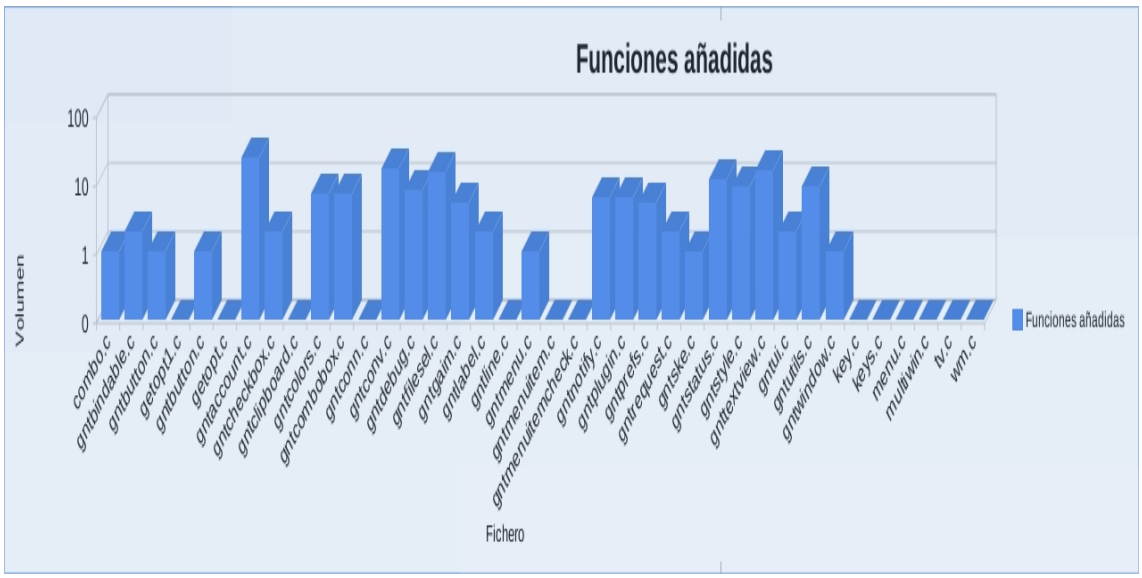


Figura 9: Gráfico con el número de funciones añadidas por cada uno de los ficheros de *Pidgin*

A continuación mostramos una tabla con el número exacto de funciones añadidas por fichero:

Nombre fichero	Número de funciones añadidas
combo.c	1
gntbindable.c	2
gntbutton.c	1
getopt1.c	0
gntbutton.c	0
getopt.c	0
gntaccount.c	23
gntcheckbox.c	2
gntclipboard.c	0
gntcolors.c	7
gntcombobox.c	7
gntconn.c	0
gntconv.c	16
gntdebug.c	8
gntfilesel.c	14
gntgaim.c	5
gntlabel.c	2
gntline.c	0
gntmenu.c	1
gntmenuitem.c	0
gntmenuitemcheck.c	0
gntnotify.c	6
gntplugin.c	6
gntprefs.c	5
gntrequest.c	2
gntske.c	1
gntstatus.c	11
gntstyle.c	9
gnttextview.c	15

Nombre fichero	Número de funciones añadidas
gntui.c	2
gntutils.c	9
gntwindow.c	1
key.c	0
keys.c	0
menu.c	0
multiwin.c	0
tv.c	0
wm.c	0

Una vez obtenidos estos datos, nos interesa saber cuál es el número medio de funciones añadidas por cada uno de los ficheros analizados. Realizando una división entre el número total de funciones añadidas entre el número total de ficheros analizados, nos queda que el número medio de funciones añadidas por fichero es **4**

4.10.3. Funciones eliminadas.

A continuación, queremos saber el número de funciones eliminadas en cada uno de los ficheros que componen *Pidgin*:

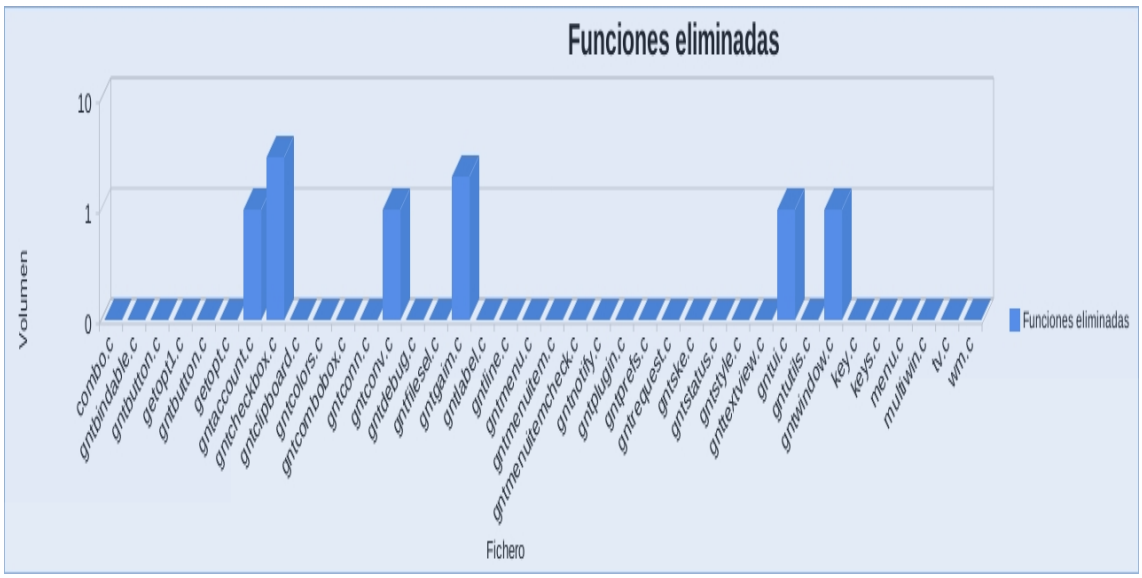


Figura 10: Gráfico con el número de funciones eliminadas por cada uno de los ficheros de *Pidgin*

A continuación mostramos una tabla con el número exacto de funciones eliminadas por fichero:

Nombre fichero	Número de funciones eliminadas
combo.c	0
gntbindable.c	0
gntbutton.c	0
getopt1.c	0
gntbutton.c	0
getopt.c	0
gntaccount.c	1
gntcheckbox.c	3
gntclipboard.c	0
gntcolors.c	0
gntcombobox.c	0
gntconn.c	0
gntconv.c	1
gntdebug.c	0
gntfilesel.c	0
gntgaim.c	2
gntlabel.c	0
gntline.c	0
gntmenu.c	0
gntmenuitem.c	0
gntmenuitemcheck.c	0
gntnotify.c	0
gntplugin.c	0
gntprefs.c	0
gntrequest.c	0
gntske.c	0
gntstatus.c	0
gntstyle.c	0
gnttextview.c	0

Nombre fichero	Número de funciones eliminadas
gntui.c	1
gntutils.c	0
gntwindow.c	1
key.c	0
keys.c	0
menu.c	0
multiwin.c	0
tv.c	0
wm.c	0

Una vez obtenidos estos datos, nos interesa saber cuál es el número medio de funciones eliminadas por cada uno de los ficheros analizados. Realizando una división entre el número total de funciones eliminadas entre el número total de ficheros analizados, nos queda que el número medio de funciones eliminadas por fichero es **0,236**

4.10.4. Líneas añadidas.

A continuación, queremos saber el número de líneas añadidas en cada uno de los ficheros que componen *Pidgin*:

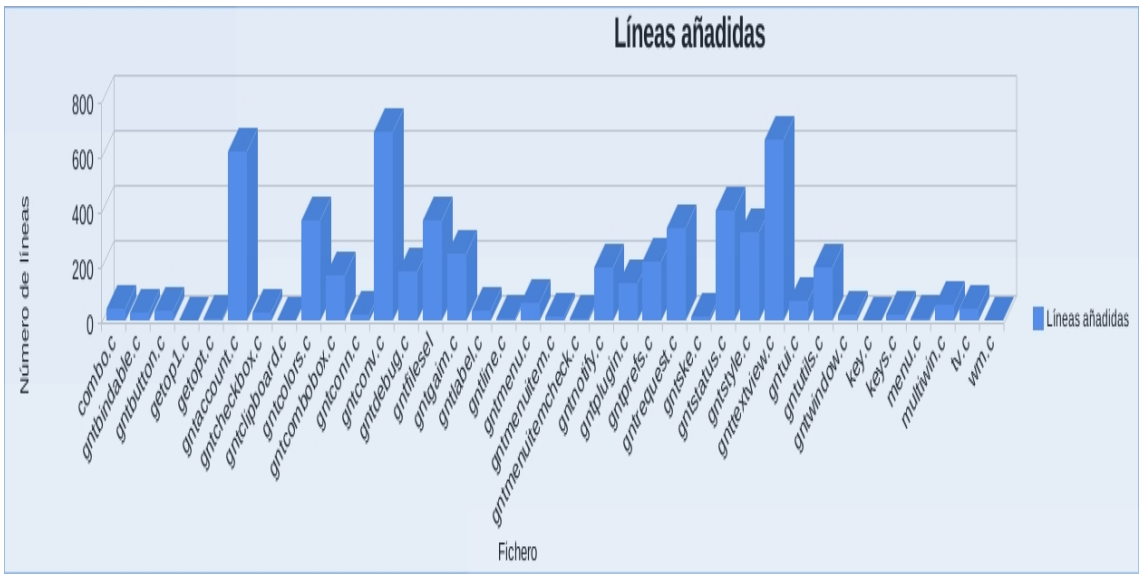


Figura 11: Gráfico con el número de líneas añadidas por cada uno de los ficheros de *Pidgin*

A continuación mostramos una tabla con el número exacto de líneas añadidas por fichero:

Nombre fichero	Número de líneas añadidas
combo.c	42
gntbindable.c	25
gntbutton.c	36
getopt1.c	3
gntbutton.c	36
getopt.c	7
gntaccount.c	616
gntcheckbox.c	29
gntclipboard.c	0
gntcolors.c	364
gntcombobox.c	163
gntconn.c	18
gntconv.c	683
gntdebug.c	175
gntfilesel.c	365
gntgaim.c	240
gntlabel.c	34
gntline.c	6
gntmenu.c	61
gntmenuitem.c	13
gntmenuitemcheck.c	5
gntnotify.c	196
gntplugin.c	135
gntprefs.c	215
gntrequest.c	337
gntske.c	16
gntstatus.c	402
gntstyle.c	319
gnttextview.c	657

Nombre fichero	Número de líneas añadidas
gntui.c	74
gntutils.c	192
gntwindow.c	19
key.c	1
keys.c	19
menu.c	9
multiwin.c	57
tv.c	46
wm.c	3

Una vez obtenidos estos datos, nos interesa saber cuál es el número medio de líneas añadidas por cada uno de los ficheros analizados. Realizando una división entre el número total de líneas añadidas entre el número total de ficheros analizados, nos queda que el número medio de líneas añadidas por fichero es **147,84**

4.10.5. Líneas eliminadas.

A continuación, queremos saber el número de líneas eliminadas en cada uno de los ficheros que componen *Pidgin*:

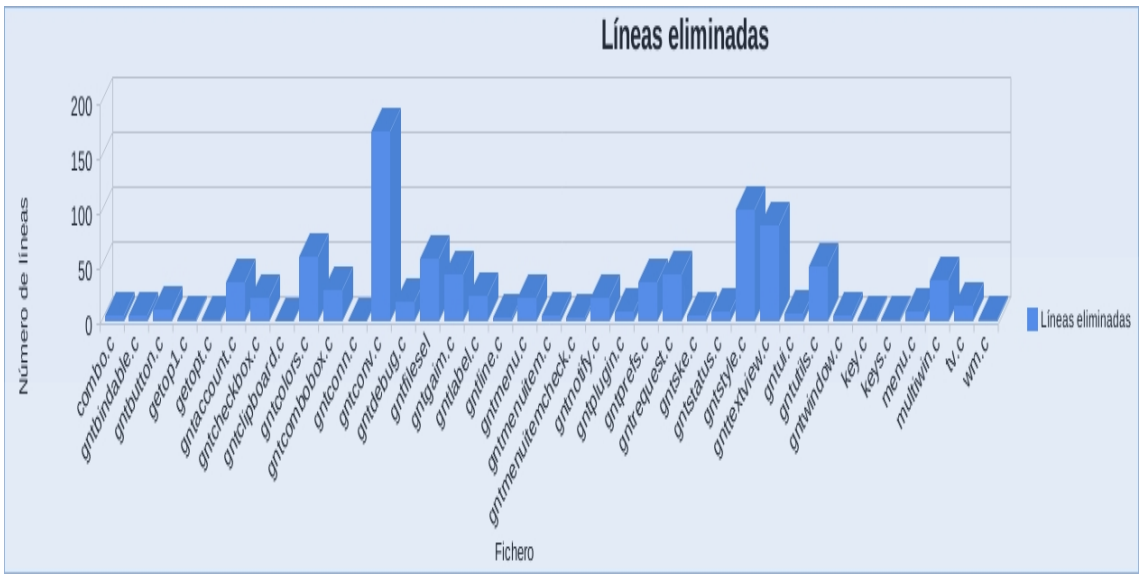


Figura 12: Gráfico con el número de líneas eliminadas por cada uno de los ficheros de *Pidgin*

A continuación mostramos una tabla con el número exacto de líneas eliminadas por fichero:

Nombre fichero	Número de líneas eliminadas
combo.c	6
gntbindable.c	5
gntbutton.c	10
getopt1.c	2
gntbutton.c	10
getopt.c	2
gntaccount.c	36
gntcheckbox.c	21
gntclipboard.c	0
gntcolors.c	59
gntcombobox.c	28
gntconn.c	0
gntconv.c	173
gntdebug.c	17
gntfilesel.c	58
gntgaim.c	43
gntlabel.c	23
gntline.c	4
gntmenu.c	22
gntmenuitem.c	6
gntmenuitemcheck.c	4
gntnotify.c	21
gntplugin.c	9
gntprefs.c	36
gntrequest.c	42
gntske.c	6
gntstatus.c	9
gntstyle.c	102
gnttextview.c	87

Nombre fichero	Número de líneas eliminadas
gntui.c	8
gntutils.c	50
gntwindow.c	6
key.c	1
keys.c	1
menu.c	9
multiwin.c	37
tv.c	15
wm.c	1

Una vez obtenidos estos datos, nos interesa saber cuál es el número medio de líneas eliminadas por cada uno de los ficheros analizados. Realizando una división entre el número total de líneas eliminadas entre el número total de ficheros analizados, nos queda que el número medio de líneas eliminadas por fichero es **25,5**

5. Conclusiones

El proyecto supone un instrumento de colaboración a los desarrolladores de software, ofreciéndoles el manejo de una herramienta de análisis, permitiéndoles un ahorro de tiempo en el desarrollo de aplicaciones distribuidas.

Podemos concluir que el desarrollo del proyecto terminó con éxito, pues se han alcanzado todos los objetivos propuestos al inicio del mismo.

Se ha logrado crear una herramienta de análisis que permite observar las características de desarrollo de un proyecto específico, ofreciendo así el número de versiones que lo componen, los cambios sufridos en el paso de una versión a la siguiente, las funciones que componen cada una de las versiones, ...

Además, se permitió un alto grado de automatización de la herramienta, ya que dispone de un interfaz muy sencillo, así como un alto grado de sencillez en la instalación/desinstalación del software.

Por último, comentar que todas las herramientas de las que se ha hecho uso en este proyecto han sido software libre, pudiendo comprobar su gran eficiencia y eficacia.

5.1. Lecciones aprendidas

El desarrollo de este proyecto ha sido altamente enriquecedor, ya que nos ha permitido aprender muchísimas herramientas nuevas, además del aprendizaje de como debe ser el desarrollo de un proyecto, puesto que jamás habíamos puesto en práctica algunos de los conocimientos adquiridos en el transcurso de la carrera y del máster. Podemos enumerar los principales aspectos que nos han resultado especialmente llamativos:

- La comprensión de la variedad de medios de colaboración existentes en el desarrollo del software libre: hemos conocido herramientas como *Subversion* o *Git* que no se conocían previamente.
- La iniciación en el lenguaje *Python*, pudiendo comprobar por qué tanta gente hace uso de él. Nos hemos encontrado un entorno agradable, con una sintaxis muy clara y una gran cantidad de bibliotecas de las que poder hacer uso, cosa que nos ayudó mucho.
- La creación de un proyecto de software de ciertas dimensiones siguiendo una metodología de desarrollo, en este caso el modelo de desarrollo en etapas.
- El aprendizaje de *SQLite*, así como su manejo desde un lenguaje de programación, en este caso *Python*.
- Hemos aprendido a desarrollar interfaces gráficas con el módulo de *Python pygtk*.
- Se ha aprendido a crear un paquete RPM que facilita la instalación/desinstalación de software.
- Se ha aprendido también a manejar LaTeX, el sistema de procesamiento de documentos empleado en la realización de esta memoria.

5.2. Trabajos futuros

Una vez finalizado el proyecto, se plantean varios campos de trabajo que supondrían una extensión de su funcionalidad.

Una posible mejora es ampliar los proyectos programados en otros lenguajes para permitir su análisis. Si este proyecto pudiese analizar proyectos programados en lenguajes tan amplia-

mente utilizados como *C++*, Java o .NET, estamos convencidos que podría ser de gran ayuda para multitud de desarrolladores.

Resultaría también interesante exportar la información recopilada a otros formatos, tales como PDF, XML, ...

Todos estos nuevos objetivos (y los que pueda plantear una tercera persona), podrán implementarse para potenciar la herramienta, que quedará amparada por una licencia de software libre.

A. Instalación y uso

La instalación del software es tan sencilla como, una vez obtenido el rpm de instalación, instalarlo con el gestor de paquetes *yum*:

```
root@dani-laptop:~# yum install -y svnexplorer-1.0-11.fc14.i686.rpm
```

El modo de empleo del programa es muy sencillo, puesto que sólo debemos elegir el nombre del fichero que se va a crear, si queremos los resultados en formato texto o en formato HTML, el protocolo de conexión a *Subversion* y la URL donde se encuentra el fichero que queremos analizar.

En concreto:

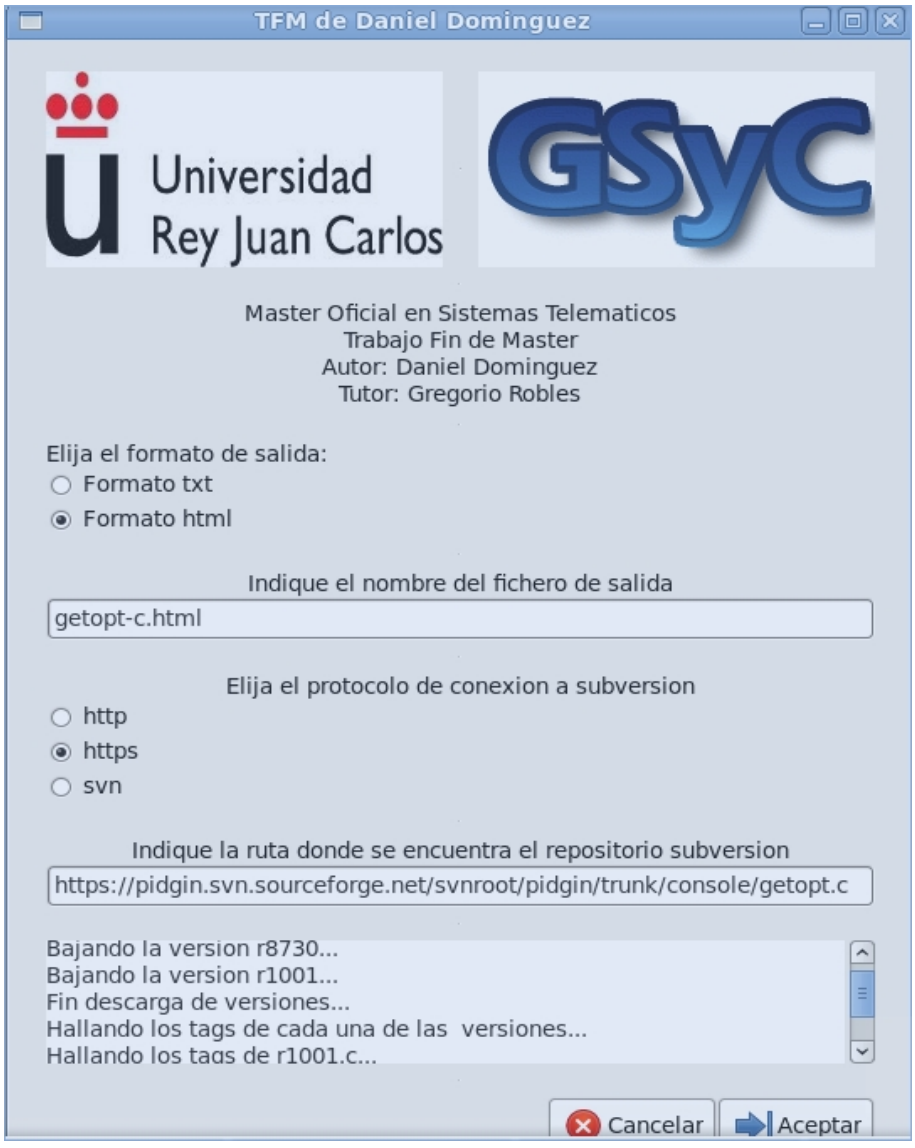


Figura 13: Imagen de de la interfaz gráfica diseñada para nuestro software

A.1. Requisitos

Los requisitos previos a la puesta en marcha de las herramientas son:

- *Intérprete de Python*.

  Debería funcionar con la versión 2.5 o posterior. Durante el desarrollo se usó la versión 2.6

- *Sistema gestor de base de datos SQLite3*.

- *Herramienta Ctags*.

- *Herramienta Subversion*.

Bibliografía

[1] Mark Lutz. *Programming Python*. O'Reilly, 2001.

[2] Fredrik Lundh. *Python standard library*. O'Reilly, 2001.

[3] Grupo de Sistemas y Comunicaciones - Universidad Rey Juan Carlos.

`http://gsyc.escet.urjc.es`

[4] Web del proyecto Libre Software Engineering - GSYC.

`http://libresoft.dat.escet.urjc.es`