UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
PROGRAMA DE MAGISTER EN CIENCIAS DE LA COMPUTACIÓN

Estudio empírico de prácticas de desarrollo de software en proyectos de código abierto

Profesor Guía: Ricardo Contreras Arriagada
Dpto. de Sistema
Facultad de Ingeniería
Universidad de Concepción

Tesis para ser presentada a la Dirección de Postgrado de la Universidad de Concepción

GERMÁN EMILIO POO CAAMAÑO
CONCEPCIÓN-CHILE
2010

Índice general

Índice de figuras

Índice de tablas

Resumen

El uso de Internet ha permitido la proliferación del Software Libre o código abierto. Internet ha facilitado la existencia de proyectos en donde los desarrolladores se encuentran geográficamente distribuidos. Dada su naturaleza distribuida, las contribuciones a estos proyectos pueden provenir de diversas fuentes, las cuales no siempre ocurren desde el núcleo de desarrolladores, sino que también mediante contribuciones externas.

En los proyectos de Software Libre, usualmente no hay estructuras jerárquicas rígidas de la forma en que tradicionalmente existen en las empresas. La necesidad de coordinación ha visto nacer subproductos que buscan facilitar el desarrollo y la comunicación en un proyecto, tales como herramientas de seguimiento de errores, sistemas de control de revisiones del software y registro de las comunicaciones y decisiones tomadas para alcanzar los objetivos de cada proyecto. Tanto los proyectos, como los productos derivados para lograr su desarrollo, se encuentran disponibles públicamente, con los cuales es posible realizar análisis que de otra forma serían muy difícil lograr.

El objetivo principal de esta tesis es estudiar la evolución y propiedades de un proyecto de software a través del estudio empírico y aplicación de análisis estadístico de su código fuente que permitan determinarl el mayor esfuerzo de desarrollo que recibe un proyecto según se liberan nuevas versiones.

Capítulo 1

Introducción

1.1. Motivación

Desde sus inicios la Ingeniería de Software ha intentado obtener conocimiento del desarrollo de software con el objetivo de cuantificar y predecir el tiempo de desarrollo, el costo en horas hombres y recursos técnicos. Sin embargo, los estudios en muchos casos se han podido realizar con datos que sólo han estado disponibles para el investigador (debido a restricciones de propiedad intelectual)[30]. Además, otros datos como la evolución del código no siempre ha estado disponible para su estudio o éste lo ha estado en un ámbito reducido. Por lo que la caracterización del proceso de desarrollo de software ha sido incompleta[15].

En los proyectos de código abierto o Software Libre (FOSS[1]) no existe una clara separación entre el proceso de desarrollo y el de mantención, ni liberación de versiones; y, en donde prima el lema "liberar pronto, liberar temprano"[2][24]. Dichas interacciones ocurren abiertamente y resulta interesante estudiar y explicar su funcionamiento.

Las razones que motivan esta tesis son:

- Estudiar el comportamiento de los proyectos FOSS, en particular del proyecto GNOME, cuyo objetivo es proveer de un sistema que facilite el uso del computador a un espectro amplio de usuarios y para ello utiliza la metáfora de escritorio. A su vez, el proyecto GNOME provee una plataforma de desarrollo para que terceros puedan construir aplicaciones.

[1] Free/Open Source Software

[2] Del inglés "release early, relase often"

- Los proyectos FOSS se diferencian de proyectos tradicionales en diferentes aspectos, en particular, éstos tienen una naturaleza descentralizada, cuyo desarrollo se base a intereses comunes en donde no hay necesariamente un interés pecuniario detrás. Además, no tienen fronteras, y por lo tanto, requieren el desarrollo de una comunidad de usuarios y desarrolladores que se comunican a través de distintas herramientas y cuyo factor común es Internet, lo que gatilla un particular interés por estudiar su funcionamiento.
- Al utilizar Internet y guardar registro de varias actividades que se realizan, hace interesante la búsqueda de más información con los datos disponibles actualmente.

1.2. Definición del problema

El Software Libre permite a los desarrolladores realizar un completo análisis cuantitativo del código y todos los otros parámetros involucrados en su producción, debido a que están disponibles públicamente[10]. Al contar con datos de acceso público del desarrollo de software permite llevar a cabo los estudios estadísticos propuestos por Lehman[18] para estudiar la evolución del software.

A pesar que la Ingeniería de Software es una disciplina que se ha consolidado a través de los años y en donde existe una literatura bastante difundida en el área[31] sobre como desarrollar software, en el último tiempo el desarrollo seguido por proyectos FOSS abre nuevas perspectivas, en donde muchas indicaciones propuestas para un mejor desarrollo se utilizan ampliamente, hay otros aspectos de la disciplina que no se siguen precisamente (por ejemplo, un levantamiento formal de requerimientos). Aún así, el software sigue creciendo y mejorando[12].

La Ingeniería de Software Libre es un campo nuevo donde hay mucho que explorar para ayudar a entender desde una perspectiva de ingeniería el proceso de desarrollo de software[10].

Luego, por las razones expuestas, el atractivo del desarrollo comunitario del Software Libre, con datos disponibles públicamente esperando ser escrutados y la importancia para la Ingeniería de Software de modelar con datos empíricos surge el interés por estudiar empíricamente las prácticas de desarrollo de un proyecto de software como GNOME.

1.3. Hipótesis

Es posible realizar un estudio empírico de prácticas de desarrollo de software, aplicar minería de datos en el proyecto, estudiar su evolución a través del tiempo y correlacionar datos, determinar y aplicar métricas. A partir de lo cual es posible comprobar que la longevidad y mantención del código en el proyecto GNOME es heterogénea y difiere entre componentes de un mismo proyecto.

1.4. Objetivo

Estudiar la evolución y propiedades de un proyecto de software a través del estudio empírico y aplicación de análisis estadístico de su código fuente.

Los objetivos específicos son:

- Estudiar el comportamiento de los proyectos FOSS, en particular del proyecto GNOME, cuyo objetivo es proveer de un sistema que facilite el uso del computador a un espectro amplio de usuarios y para ello utiliza la metáfora de escritorio. A su vez, el proyecto GNOME provee una plataforma de desarrollo para que terceros puedan construir aplicaciones.
- Identificar las fuentes de datos que ofrecen los proyectos FOSS de manera pública, presentar una metodogía para el análisis de las fuentes y de los datos que se puedan extraer de las mismas.
- Identificar en donde ocurre el mayor esfuerzo en programación respecto a otras tareas como documentación y traducción, en un proyecto como GNOME.
- Conocer mejor el fenómeno del Software Libre, el proceso de creación de software y cómo se puede aplicar en cualquier otro entorno de desarrollo.
- Describir la mecánica de desarrollo de los movimientos de Software Libre.

Es así que, por una parte se encuentra la comprensión de un proyecto de software y por otro se encuentra la posibilidad de retroalimentar a dichos proyectos a partir del análisis de los datos que ya se encuentran disponibles y que no se han explotado completamente.

1.5. Organización

Es tesis está estructurada de la siguiente forma:

Capítulo 1. Consta de la presente introducción.

Capítulo 2. Se describe el proyecto GNOME, los aspectos de estudio, los sistemas de control de versiones con su terminología y su relación con GNOME.

Capítulo 3. Se detalla la metodología a utilizar, la caracterización de los datos en el contexto del proyecto GNOME, métodos de obtención, herramientas y el proceso de refinamiento de los datos.

Capítulo 4. Detalle de los resultados obtenidos, con los parámetros que fueron utilizados y la interpretación de éstos.

Conclusiones. Se presentan las conclusiones obtenidas durante este estudio en base a la evidencia obtenida en el Capítulo 4, para finalizar con el cumplimiento de la hipótesis y objetivos, así como propuestas para futuros estudios.

Capítulo 2

Antecedentes

2.1. El proyecto GNOME

El proyecto GNOME se puede resumir como un proyecto de Software Libre compuesto por una colección de bibliotecas y aplicaciones destinadas al usuario final que proveen un "escritorio" gráfico a sistemas Unix[9].

Es un esfuerzo comunitario para crear un software de ambiente de escritorio libre, en conjunto con una plataforma de desarrollo de aplicaciones. Así, el proyecto provee: el ambiente de escritorio GNOME, el cual es un escritorio intuitivo y atractivo para los usuarios, y la plataforma de desarrollo GNOME, la cual es una estructura que permite construir aplicaciones que se integren con el resto del escritorio[1][2][6].

Cabe notar que, el concepto de ambiente de escritorio no es familiar para personas que no han tenido acceso a sistemas Unix, por cuanto sistemas como MacOSX y Windows integran la interfaz gráfica con el sistema operativo sin posibilidades de cambio sustanciales. Para entenderlo mejor, Woods y Guliani[32] separan un escritorio en tres capas, como se muestra en la figura 2.1. De acuerdo a dicha figura, GNOME se encuentra en las dos capas superiores, mientras que la capa inferior corresponde al sistema operativo más XWindow.

La existencia de múltiples ambientes de escritorio tienen como raíz el diseño del sistema XWindow, el cual deliberadamente no impone restricciones ni políticas para usar algún tipo de interfaz de usuario en particular. Los ambientes de escritorio,

[1] `http://live.gnome.org/AboutGnome`.
[2] `http://www.gnome.org/about/`.

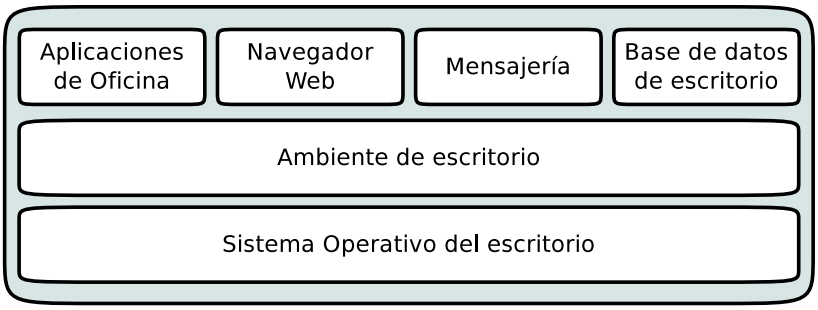

Figura 2.1: Capas de funcionalidades en un escritorio

como GNOME y KDE[3], agregan una capa superior, en donde se establecen las políticas de un diseño consistente.

2.1.1. Descripción del proyecto e historia

El desarrollo del proyecto de software GNOME se inició en 1997[4] por Miguel de Icaza y Federico Mena, ambos de la Universidad Autónoma de México, en conjunto con un pequeño equipo de desarrolladores de distintos lugares del mundo. El proyecto fue inspirado por tres factores:

1. KDE[5], el cual no era completamente libre:
2. La popularidad del sistema operativo Linux; y,
3. Las potencialidades del toolkit gráfico GTK+.

GNOME es el acrónimo de GNU Network Object Model Environment, debido a que en sus inicios la intención fue crear un sistema de componentes similar, en espíritu, a las tecnologías OLE2, ActiveX y OpenDOC de Microsoft[6]. Sin embargo, el ámbito del proyecto se amplió al determinar que se requería mucho más que tener objetos en red. Se creó una arquitectura para incrustar objetos en las aplicaciones llamada Bonobo, basado en CORBA 2.2 y para lo cual se construyó ORBit[21].

[3]KDE es el acrónimo de K Desktop Environment (Ambiente de Escritorio K).

[4]`http://mail.gnome.org/archives/gtk-list/1997-August/msg00123.html`.

[5]El proyecto KDE se inició el 14 de octubre de 1996 y fue anunciado en un grupo de noticias de Alemania: `http://groups.google.com/group/de.comp.os.linux.misc/msg/cb4b2d67ffc3ffce`.

Desde la perspectiva de un usuario es un ambiente de escritorio integrado y un conjunto de aplicaciones. Desde la perspectiva de un programador, es un infraestructura de desarrollo de aplicaciones, la cual está constituida por varias bibliotecas. Las aplicaciones escritas con las bibliotecas de GNOME se pueden ejecutar sin el ambiente de escritorio, pero no se puede aprovechar la integración del entorno como un todo. El ambiente de escritorio, en su forma más básica[6], incluye un administrador de archivos, un panel de tareas, acceso rápido a aplicaciones, mini–aplicaciones[7], un gestor de ventanas, un gestor de sesión con la configuración asociada. A su vez, la infraestructura de desarrollo de GNOME permite construir aplicaciones consistentes, fáciles de utilizar e interoperables entre sí.

2.1.2. Composición del proyecto GNOME

El proyecto GNOME está compuesto por diferentes equipos o proyectos, algunos ligados directamente con el desarrollo de distintas partes de la aplicación y otros encargados de mantener la infraestructura que permite a los equipos funcionar. Aunque existen diversos equipos, es común encontrar colaboradores que trabajan en múltiples equipos o que estos cambien de equipo en el transcurso del tiempo.

2.1.2.1. Estructura legal

La Fundación GNOME es la entidad legal que se formó para servir como interlocutor con las empresas interesadas en el proyecto, así como para poder recibir y administrar donaciones. La Fundación GNOME agrupa a los colaboradores del proyecto, quienes llegan a ser miembros mediante una postulación en donde muestren su contribución[8] y por recomendación de otros miembros.

Como organización, la Fundación GNOME[9] consta de un consejo de directores y el consejo consultivo. El consejo directivo está compuesto por miembros de la fundación, quienes son elegidos por los miembros mediante votación. Son el nexo administrativo y técnico entre la comunidad de desarrolladores y las empresas interesadas en el éxito de GNOME. El consejo consultivo está compuesto por representantes de las empresas que patrocinan a la fundación, quienes pueden plantear sus puntos de vistas, preocupaciones, así como formular recomendaciones. Sin

[6] Lo que correspondería a la capa intermedia de la figura 2.1.

[7] Se les conocen como *applets*, En MacOSX se les conoce como *widgets*. Son aplicaciones que se incrustan en una aplicación mayor, como un panel.

[8] Dicha contribución no debe ser trivial.

[9] `http://foundation.gnome.org/about/charter/`.

embargo, el consejo consultivo sólo tiene derecho a voz; no tiene derecho a voto ni tampoco puede vetar las decisiones del consejo directivo.

2.1.2.2. Equipos de desarrollo

- *Accesibilidad*. Este equipo se encarga de garantizar que las personas con discapacidades puedan utilizar todas las funcionalidades de GNOME, habilitando el soporte para distintas discapacidades en el software que se construye. Además, permite que las personas con discapacidades puedan participar activamente en el proyecto.

- *Arte*. Es el encargado del aspecto gráfico y visual de GNOME, esto incluye fondos de pantalla, iconografía, protectores de pantalla, diseño visual, los temas que se muestran en el escritorio, etc. Aunque también cumplen funciones de apoyo, se considera parte del desarrollo dado que en los programas se incluye el trabajo realizado por este equipo.

- *Aseguramiento de calidad*. El equipo de aseguramiento de la calidad está compuesto por dos partes: el escuadrón de fallos[10] se encarga del sistema de seguimiento de eventos[11], el cual es una base de datos que almacena todos los problemas y fallos que ha tenido el software para que los desarrolladores puedan repararlos, así como las peticiones de nuevas características. Además, este equipo se encarga de clasificar[12] cada uno de los reportes de fallos, para facilitar el trabajo de los desarrolladores. La otra parte es la brigada de construcción de software[13], que se encarga de mantener el proyecto en construcción en forma continua y automática, para detectar los fallos en la construcción lo más pronto posible, incluso antes que otros desarrolladores lo noten[14]; así, se garantiza que el sistema siempre esté funcionando.

- *Documentación*. El equipo de documentación es el responsable de proveer a los usuarios con documentación de alta calidad[15], esto significa manuales, ayuda en línea, tutoriales, guía de referencia de la interfaz de programación, lineamientos de interfaces de usuario, etc.

[10] Bugsquad

[11] La herramienta utilizada en GNOME es Bugzilla.

[12] El término empleado es *triage*, que es el mismo que se emplea en las salas de emergencia (ER) para clasificar los pacientes que requieren mayor (o menor) prioridad de atención de acuerdo a la gravedad y antecedentes del caso.

[13] Build brigade

[14] Un software que no se puede construir, no se puede probar.

[15] No sólo debe existir documentación, sino que también ésta debe ser útil y usable.

- *Administración de entrega de productos o versiones*. Coordina el proceso de los distintos equipos involucrados en el desarrollo de una nueva versión, sea de desarrollo o estable, de GNOME. Define el calendario (el cual comprende un período de 6 meses) y mantiene informado a los desarrolladores sobre el estados y las etapas en que se encuentran dentro de dicho calendario. Este equipo decide los módulos que se incluirán en una nueva versión del escritorio.

- *Rendimiento*. Está constituido por desarrolladores dedicados especialmente a acelerar el tiempo de arranque de las aplicaciones, disminuir el consumo de memoria de éstas, encontrar y solucionar los cuellos de botella que pueden afectar al rendimiento del escritorio como un todo.

- *Traducción*. Se encarga de coordinar el esfuerzo de traducción, a distintos idiomas, de la interfaz de usuario de las aplicaciones y su correspondiente documentación. Además, se preocupa de mantener la consistencia de la traducción entre distintas las aplicaciones que conforman el escritorio.

- *Usabilidad*. Se preocupa de mejorar la experiencia del usuario de GNOME. Ayuda en el esfuerzo de programación para crear apelaciones intuitivas, lidera la creación de diseños y propuestas detalladas de interfaces de usuario que puedan mejorar el escritorio. También realizan pruebas con usuarios, definen guías de estilo, visualización de información e interacción humano–computador.

2.1.2.3. Equipos de apoyo

- *Administración de sistemas*. Se encarga de mantener toda la infraestructura que mantiene los servicios que necesita el proyecto para su funcionamiento, entre los que se encuentran las listas de correo, sistema de control de versiones, servidores web con su productos asociados, directorio de colaboradores, seguridad, respaldos, sistema de seguimiento de fallos, etc.

- *Web*. Construye y mantiene el contenido de los distintos sitios de GNOME. Algunos sitios son generados de manera automática, mientras que otros requieren intervención manual para mantenerse actualizados. Trabajan muy relacionados con el equipo de arte y marketing.

- *Marketing*. Este equipo trabaja para mejorar la comunicación e imagen del proyecto. Se encargan de los comunicados de prensa, establecer contacto con los voluntarios alrededor del mundo para realizar seguimiento de los

eventos relacionados con GNOME, de tal forma de garantizar la presencia del proyecto y hacerla más visible a las empresas que patrocinan el proyecto.

- *Moderadores*. Se encarga de mantener sanas las listas de correo. Son los encargados de filtrar la comunicación que proviene de entes externos que no están suscritos a las listas de correos, de tal forma, no permitir el ingreso de SPAM, correos no deseados o mensajes demasiado grandes.

- *Viajes*. Este equipo está a cargo de canalizar las solicitudes de patrocinio que realizan los desarrolladores para asistir a congresos y asignarles fondos de acuerdo al presupuesto definido por la Fundación GNOME.

- *Legal*. Es un equipo en donde se discuten los temas de licencias, protección de marcas y todo tema que pueda tener alguna repercusión legal con respecto a los intereses del proyecto.

2.2. Canales de comunicación y fuentes de información

El modelo de desarrollo de GNOME es muy cercano a otros proyectos de FOSS; las discusiones tienen lugar principalmente en listas de correo de carácter público, habiendo también contacto en listas de correo de carácter privado, en donde los desarrolladores toman decisiones importantes de un proyecto disminuyendo el ruido. El acceso a los repositorios del código fuente del proyecto es público para lectura y los desarrolladores tienen también acceso de escritura[6]. Esto último, ha perdido relevancia luego que el proyecto GNOME migró el sistema de control de versiones, desde Subversion a *git*, lo cual se explica en la sección 2.3.

Existen diversos canales y fuentes de información del proyecto, algunas se describen detalladamente en los trabajos de tesis de doctorado de Gregorio Robles[25] e Israel Herraiz[12].

- Entrega de nuevas versiones de software. Los proyectos FOSS ofrecen sus productos con cada nueva versión, las cuales se realizan en forma de código fuente y el aparato necesario para construir el sistema final a partir de dicho código. Además, se suele incluir documentación para el usuario, el desarrollador y el administrador de sistemas encargado de instalarlo.

- Sistemas de control de versiones. Los sistemas de control de versiones se utilizan para controlar y coordinar el desarrollo concurrente de muchos actores de un proyecto. Los sistemas de control de versiones permiten almacenar y

realizar seguimiento a todos los cambios que se realizan en el repositorio de un proyecto. A su vez, guarda meta–información, como el autor del cambio, fecha y hora, así como comentarios del autor sobre los cambios.

- Listas de correo. Usualmente el medio de comunicación más activo en el proyecto y es un método que permite guardar todas las discusiones que se llevan a cabo. Existen múltiples listas de correo, algunas de carácter más general y otras de carácter más detallado, que permiten filtrar las discusiones que tienen los colaboradores, a medida que van ocurriendo. En general, cada subproyecto tiene una o más listas de correo, según el tráfico que generan. También existen listas de correo para mantener informado a los entes externos y que son completamente moderadas, tales como anuncios de actividades o entrega de nuevas versiones y un resumen de su contenido.

- Seguimiento de fallos y aseguramiento de calidad[16]. Permite a los desarrolladores mantener un registro de todos los fallos que se han reportado, su estado, así como nuevas características que desean o requiere implementar. Habilita que todos puedan consultar el estado de los fallos, sugerir mejoras y enviar los programas corregidos. Sirve como canal de comunicación con los proyectos que utilizan GNOME como subproducto, por ejemplo, proveedor de Linux o Unix que incluyen GNOME dentro de la distribución de su software.

- Internet Relay Chat (IRC)[17]. Es un lugar para conversaciones interactivas a través de distintos canales separados por temas. Incluye discusiones ligeras del proyecto, intercambio de información mediante pregunta–respuesta, así como conversaciones de temas no técnicos. También es punto de encuentro para realizar reuniones de equipos de trabajo, realizar tutoriales y/o conferencias en línea, las cuales posteriormente se publican en listas de correo o sitios web, como una forma de permitir que toda la comunidad se entere. Lo mismo ocurre con las decisiones que se hayan tomado por este medio, de manera de evitar la exclusión de colaboradores que por su zona horaria u otros compromisos no puede utilizar este medio.

- Agregador de noticias[18]. Permite seguir el rastro de los principales desarrolladores del proyecto GNOME. Se reúne lo que escribe cada uno de sus sitios personales mediante *weblogs*. Aunque usualmente los desarrolladores

[16] `http://bugzilla.gnome.org/`.
[17] irc.gnome.org
[18] `http://planet.gnome.org/`.

escriben sobre temas técnicos relacionados con los productos que ellos están trabajando, también abarcan sobre distintos intereses de dichas personas. Así, los agregadores de noticias reúnen toda la información en un lugar centralizado.

- Wiki[19]. Se guarda los registros de todas las propuestas, lo cual no significa que sean adoptadas oficialmente, sino que funciona como canal para fomentar una discusión más elaborada y asíncrona, en donde se mantiene la historia de los cambios acorde se complementan los contenidos que allí se editan. También sirve como punto de partida y para entregar más información de contexto a las discusiones que se llevan a cabo mediante listas de correos, de esta forma se combinan ambos medios, uno para mantener accesible la información de una manera más cómoda y otra para discutir e informar a la comunidad de los trabajo que se realiza y que se desea discutir.

- Library[20]. El sitio de la biblioteca de GNOME, donde hay información para usuarios, administradores de sistemas y desarrolladores de software. Para los usuarios finales se encuentra ayuda en línea e información general de cada proyecto. Para los administradores, se encuentra documentación para instalar y administrar GNOME en sitios con muchas instalaciones, como suele sucede en empresas e instituciones gubernamentales. Para los desarrolladores de software, se encuentra información para construir aplicaciones para GNOME, guías para iniciados, tutoriales, normas, información detallada de todos los equipos de trabajo, y, en general, el modelo de desarrollo de GNOME.

- Sitios estáticos.

  - `http://www.gnome.org/` es el punto de entrada del proyecto GNOME.
  - El sitio `http://foundation.gnome.org/` tiene toda la información de la Fundación GNOME, de interés para los miembros y para las instituciones externas que necesitan comunicarse con el proyecto.
  - En `http://projects.gnome.org/` se encuentran los subproyectos relacionados con GNOME, donde cada uno tiene su espacio para colocar su sitio *web*.

[19] `http://live.gnome.org/`.
[20] `http://library.gnome.org/`.

2.3. Sistemas de Control de Versiones

No hay un modelo único de desarrollo, ni en el software propietario ni en FOSS. Raymond plantea dos visiones del desarrollo de un proyecto (con especial énfasis en el FOSS), los cuales denomina Catedral y Bazaar[24]:

- En la Catedral, hay una distribución clara de los roles, con el diseñador a la cabeza, controlando el proceso de inicio a fin. La planificación es controlada en forma estricta, donde el período de tiempo entre nuevas versiones es amplio. No sólo corresponde a proyectos grandes en la industria, sino que también a proyectos FOSS, tal como GNU o NetBSD, los cuales son proyectos centralizados.
- En el Bazaar, no existe autoridad que controle el proceso cuidadosamente conforme a un plan, en donde se dispone el código públicamente y aprovecha la interactividad de la comunicación y herramientas en red, considera a los usuarios como co-desarrolladores, y por la amplitud que alcanza, Raymond plantea que disminuye los costos de prueba y los aplica a múltiples escenarios de hardware, sistemas operativos, etc.

Sin importar el modelo de desarrollo, una buena administración exige controlar los cambios que ocurren, manteniendo registro de las actividades que se realizan sobre el código fuente. No obstante, dada la naturaleza distribuida del desarrollo siguiendo la visión del Bazaar, dicho registro es obligatorio para mantener el orden entre desarrolladores dispersos geográficamente y con diferentes zonas horarias.

2.3.1. Control de revisiones

El control de versiones (o revisiones) es el proceso de administrar múltiples versiones de una pieza de información. En su forma más simple, es algo que muchas personas realizan manualmente: cada vez que modifican un archivo, lo guardan con un nuevo nombre que contiene un número, el cual es mayor al número (versión) precedente.

La tarea de manejar manualmente múltiples versiones es propensa a errores, ya que confía en la capacidad de memorización del autor respecto a las diferencias que hay en cada versión. Desde hace mucho tiempo han existido herramientas que asisten a una persona a manejar distintas versiones de un archivo de forma automática. En las últimas décadas, el ámbito de las herramientas de control de versiones se ha expandido al punto de contar con herramientas que no tienen dificultades en

controlar versiones para múltiples archivos o proyectos en el cual hay miles de archivos[19].

El control de revisiones es un campo diverso, en donde se puede encontrar con diferentes nombres y acrónimos. Algunas variantes que es posible encontrar son:

- Sistema de Control de Revisiones (RCS[21]);
- Administración de Configuración de Software (SCM[22]) o Administración de Configuración (CM[23]);
- Administración del Código Fuente (SCM[24]);
- Sistema de Control de Versiones (VCS[25])

Dependiendo del autor, es posible encontrar diversos significados para estos términos, pero en la práctica se traslapan y no hay un consenso que permita separarlas claramente[1][26].

Hay diversas motivaciones para utilizar un sistema automatizado de control de versiones:

- Registro de la historia y evolución de un proyecto. Por cada cambio se registra quién realizó un cambio, por qué se hizo, cuándo se hizo y cuál ha sido el cambio.
- Cuando se trabaja con otras personas, un sistema facilita la colaboración. Por ejemplo, cuando una o más personas realizan un cambio simultáneo que son incompatibles entre sí. Un sistema automatizado ayuda a identificar y resolver dichos conflictos.
- Ayuda a la recuperación frente a errores. Si se realiza un cambio, el cual resultó ser un error, es posible revertir los cambios a una versión anterior en uno o más archivos. Un buen sistema de control de versiones debiera ayudar eficientemente a averiguar en dónde se produjo el problema en forma exacta[27].

[21] Del inglés Revision Control System
[22] Del inglés Software Configuration Management
[23] Del inglés Configuration Management
[24] Del inglés Source Code Management
[25] Del inglés Version Control System
[26] `http://www.cmcrossroads.com/cgi-bin/cmwiki/view/CM/SoftwareConfigurationManagement`.
[27] El comando *bisect* disponible en *Mercurial* y *git* facilitan la búsqueda de la versión en donde se introdujo el error.

- Trabajar y administrar simultáneamente varias versiones de un proyecto.

Como existen diferentes sistemas de control de versiones, la herramienta más adecuada dependerá de los beneficios que ofrece frente a los costos o imposiciones que suponga el uso de dicho sistema. Un proyecto que involucra decenas de personas es más propenso a colapsar si no se emplea un herramienta para control de versiones; una persona que trabaja sola podría estar menos dispuesta a utilizar una herramienta de control de versiones dado el costo que significa (tiempo en aprender, adaptar costumbres, etc.) frente al manejo que podría requerir de sus proyectos. Como toda herramienta, funcionará apropiadamente si el proyecto es bien administrado.

2.3.2. Evolución en los sistemas de control de versiones

La primera generación de sistemas de control de versiones consistían en controlar archivos de manera independiente, y aunque es un avance respecto del método manual, tienen como problema el sistema de bloqueo de archivos y la limitación de trabajar en un solo computador, dificultando el trabajo en equipo.

La segunda generación resolvió las limitaciones del primero, permitiendo una arquitectura centrada en la red y en donde se pueden controlar proyectos completos de una sola vez. Para proyectos grandes, el servidor del repositorio central podría tener problemas de escala. Y en caso que un cliente tuviera problemas con la red, no podría acceder al servidor. Además, en proyectos de código abierto, el código está disponible para todos en modo lectura (anónimo) pero sólo algunos desarrolladores tienen acceso de escritura al repositorio. De esta forma, muchos colaboradores no pueden registrar sus cambios.

Una tercera generación se acerca al modelo de pares[28], en donde se elimina la dependencia de un servidor central y permite distribuir los datos de la historia de un proyecto a todos las partes en donde se necesite. Los sistemas más nuevos permiten trabajar en forma autónoma, sin depender de la conexión de red.

2.3.3. Sistemas de control de versiones centralizados

Los sistemas de versiones centralizados se caracterizan por almacenar los metadatos e historia del proyecto en un solo lugar —comúnmente un servidor central—, desde el cual los desarrolladores obtienen una copia del repositorio en un determinado estado o revisión, y a partir del cual se sincronizan.

[28] peer–to–peer

En la copia local hay suficiente información para sincronizarse contra el repositorio central, y todas las operaciones de manejo se deben aplican al repositorio principal. Por consiguiente, el desarrollador debe tener permisos de escritura en el repositorio remoto.

2.3.4. Sistemas de control de versiones distribuidos

Los sistema de versiones distribuidos se caracterizan por permitir que cada desarrollador del proyecto, al momento de obtener una copia del repositorio, obtenga todos los meta-datos e historia del proyecto. A partir de allí, dicho desarrollador ya no depende de la red para realizar las operaciones con el repositorio dado que se realizan en forma local, acelerando el trabajo del desarrollador.

Además, al disponer del código y todos los meta-datos, cada copia es igual al origen y por lo tanto, cada desarrollador mantiene un respaldo completo de un proyecto. Del mismo modo, es posible realizar operaciones entre múltiples repositorios sin que ello se requiera acceso al repositorio "*principal*".

2.3.5. Terminología

Para poder considerar un estudio de un sistema de control de versiones o control de cambios es necesario conocer algunos términos básicos, sin necesidad de entrar en detalle de uso de tales sistemas. Existe una terminología que es básica, y es independiente del sistema de control de versiones. Como lo indica Fogel[7], el problema es la administración de cambios.

- *repositorio*. Es una base de datos donde se almacenan todos los *cambios* que se han producido en un proyecto. En los sistemas de control de versiones centralizados hay sólo un repositorio *maestro*, en donde se almacenan todos los cambios del proyecto y los desarrolladores sólo tienen copias parciales de su historia. En los sistemas descentralizados cada desarrollador puede tener la historia completa del repositorio, incluso pueden intercambiar *cambios* con otros repositorios en forma arbitraria e independiente.

- *commit*. Implica hacer efectivo un cambio en el proyecto, almacenando dicho cambio en la base de datos del sistema de control de versiones utilizado. En el lenguaje coloquial de los desarrolladores, el término *commit* puede ser utilizado como verbo o como sustantivo, en este último caso es sinónimo de *cambios*. El término se puede asociar al concepto utilizado en los sistemas de base de datos cuando se finaliza una transacción.

- *log* (*mensaje de registro*). Es un comentario breve que se adjunta a cada *commit*; describe la naturaleza y el propósito del *commit*. Estos mensajes se consideran parte importante de la documentación e historia en cualquier proyecto, ya que ellos son el puente entre un lenguaje altamente técnico (de los cambios en el código fuente que realizan los individuos) y un lenguaje orientado a comunicar los cambios a otras personas. Un mensaje efectivo debe contener información concisa de las modificaciones realizadas, sus motivos y su incidencia.

- *update* (*actualización*) –*fetch* o *pull*–. Es el proceso en donde se consulta por los *cambios* o *commits* que otros han incorporado en un proyecto y que el desarrollador lleva a su copia local. Corresponde a actualizar la copia local con los cambios hechos por terceros en el exterior. El concepto es ligeramente distinto entre un sistema de control de versiones centralizado a uno distribuido. En uno centralizado significa sincronizar el repositorio local contra el servidor remoto, es decir, dejar la copia local igual al repositorio central. En un sistema distribuido significa sincronizar el repositorio con otros repositorios que puedan existir en otros lados (no necesariamente uno central). Dentro de las prácticas de desarrollo, algunos desarrolladores pueden realizar varias actualizaciones en un mismo día o una vez al día, a la semana o en demanda; dependiendo de su nivel de actividad en un proyecto.

- *checkout*. Es el proceso de obtener una copia del proyecto de un repositorio. Usualmente un *checkout* produce un árbol de directorios que corresponde a la *copia de trabajo*, lugar en donde el desarrollador efectúa los cambios para luego realizar una o varias operaciones de *commit*, los cuales, en un proceso posterior, pueden ir al repositorio desde donde provienen. Dado que en los sistemas de control de versiones distribuidos cada copia de trabajo es un repositorio en sí, las modificaciones pueden ser intercambiadas con cualquier repositorio (cuyo origen sea común) que esté dispuesto a aceptarlo.

- *clone*. Corresponde a la obtención de una copia de un repositorio completo.

- *copia de trabajo*. Es el directorio privado de un desarrollador y es el árbol que contiene todo el código fuente del proyecto[29]. La copia de trabajo contiene, además, los meta-datos necesarios que le permiten trabajar con el sistema de control de versiones, por ejemplo, el origen del repositorio de donde se obtuvo la copia de trabajo, la última versión extraída, etc. En un sistema centralizado, los cambios deben ir al repositorio *maestro* y por consiguiente,

[29]En estricto rigor, puede ser cualquier documento o archivo que esté siendo registrado en el repositorio (imágenes, código fuente, páginas web, etc.).

requiere conexión a dicho repositorio. En un sistema distribuido, el desarrollador tiene la libertad de realizar todos los cambios y *commits* que estime pertinente; y en cualquier momento posterior podría enviar sus cambios o bien recibir otros (*update*, *fetch* o *pull*).

- *revisión* (*cambio, conjuntos de cambios, changeset*). Una revisión usualmente es una identificación de uno o más archivos o directorios en un instante del tiempo. Algunos sistemas pueden utilizar los términos revisión, *change* (cambio), o bien *changeset* para referirse precisamente al conjunto de cambios comprometidos (*commit*) como una sola unidad conceptual en el repositorio. Dependiendo de los sistemas de control de versiones pueden tener significados técnicos distintos, pero la idea general es la misma. Es una forma de poder identificar en forma precisa ciertos puntos en el tiempo de la historia de un archivo o un conjunto de éstos.

- *diff* (*diferencias*). Es una forma de comparación que busca, por una parte, minimizar el espacio necesario para describir diferencias y, por otra, poder mostrar esas diferencias haciendo más comprensibles los cambios entre instancias o versiones de archivo de origen común. Se podría entender, además de lo descrito, como una secuencia lo más corta posible de operaciones (generalmente sobre texto, pero en algunos casos sobre contenido binario) que si se aplicaran sobre una instancia/versión de un archivo o fragmento generarían la otra instancia/versión con la que se está comparando.

- *tag* (*etiqueta*). Es un identificador arbitrario del proyecto en una revisión especifica. Se utilizan normalmente para marcar hitos en un proyecto, y que a su vez, sea fácil de recordar y/o asociar por las personas. Un práctica común de etiquetado se produce cuando un proyecto entrega una nueva versión al público o cuando éste ha alcanzado un hito importante, etiquetando dicha revisión con un nombre simbólico, por ejemplo, GNOME_1_0. En sistemas modernos, como *git,* Mercurial o Subversion[30], una etiqueta se asocia a una revisión (*changeset*); en sistemas más antiguos, como CVS, una etiqueta se aplica a nivel de archivos, lo cual es mucho más flexible pero también más complejo y en su forma más simple, se puede obtener el mismo efecto que en *git* o Mercurial si se etiquetan todos los archivos simultáneamente.

- *branch* (*rama*). Las ramas también son conocidas como líneas de desarrollo. Una rama corresponde a una copia del proyecto que se mantiene bajo el sistema de control de versiones, pero cuyos cambios no afectan la línea de desarrollo principal del proyecto (u otras ramas) y viceversa; exceptuando

[30]Estos difieren en los detalles de implementación.

el caso en que explícitamente se decida aplicar cambios de un lado hacia otro. Aunque un proyecto no defina ramas en su desarrollo, se puede considerar que siempre hay una línea principal de desarrollo y que suele ser la más activa[31]. Un uso básico de las ramas consiste en mantener una línea de desarrollo estable y otra experimental, cuando se terminan de desarrollar las características experimentales, es posible incorporarlas a la línea estable (ver el término *merge* o *mezcla*).

- *merge* (*mezcla*). Consiste en incorporar los cambios desde una rama a otra. Hay situaciones en los cuales los cambios realizados en dos ramas no se intersectan, en cuyo caso, se puede mezclar automáticamente sin inconvenientes. En aquellos casos en donde hay intersección de cambios, se produce un conflicto. Todos los sistemas de control de versiones son capaces de detectar conflictos y notificar los cambios que producen el conflicto, de tal manera que una persona pueda resolver dichos conflictos y luego comunicar el resultado al sistema de control de versiones. A medida que los SCM han evolucionado, la operación de mezcla se ha ido simplificando[32]y no tiene relación con que se trate de un sistema centralizado o distribuido, está relacionado con el instante del tiempo en que se gestaron dichas aplicaciones. En las aplicaciones más recientes ha habido mejor entendimiento de las necesidades y los problemas; así como más experiencia.

- *lock* (*bloqueo*). El bloqueo es una forma antigua y limitada de controlar el acceso a un archivo que será modificado; y, se explica solamente como antecedente histórico dado que hay sistemas comerciales que aún se basan en RCS, predecesor de CVS. El bloqueo es una forma de declarar exclusivo el acceso de modificación de un archivo, y como concepto, es similar al empleado en base de datos. Los SCM modernos, aún CVS, no utilizan un modelo de bloquear–modificar–desbloquear, sino uno de copiar–modificar–mezclar, que permite a más colaboradores trabajar en un mismo archivo de manera concurrente.

2.3.6. Minería de datos y Sistemas de Control de Versiones

El uso de herramientas SCM permite mantener registro histórico de las operaciones que se realizan sobre el código durante su desarrollo. Como efecto colateral permite realizar minería de datos de los proyectos y, por consiguiente, estudiar su

[31] En *git* se denomina *master*, en CVS se denomina *head* y en Subversion es *trunk*

[32] Es así que en CVS el proceso es más complicado que en Subversion; a su vez en Subversion es más complicado que en *git* o Mercurial.

evolución, aprender de ella y extraer lecciones para ayudar a tomar decisiones más informadas en proyectos nuevos o en proyectos en curso.

2.4. Uso de Sistemas de Control de Versiones en GNOME

2.4.1. CVS

El proyecto GNOME utilizó CVS desde el inicio del proyecto. CVS es el sistema utilizado en los proyectos de GNU, muy popular en la época porque ofrecía características superiores para el trabajo colaborativo por sobre RCS y SCCS. Además, el hecho de utilizar las herramientas en uso por otros proyectos facilita la incorporación de nuevos colaboradores a un proyecto que se inicia, dado que mantiene bajas las barreras de entrada[6].

2.4.2. Subversion

Subversion[33] es un SCM que fue concebido inicialmente como un reemplazo de CVS, conservando sus ideas básicas, corrigiendo sus fallos y supliendo sus carencias. En consecuencia, utiliza el mismo modelo de trabajo que CVS[22][7].

En GNOME, después de 9 años utilizando CVS, se decidió migrar a Subversion, dado que las limitaciones de CVS empezaron a ser más evidentes y era necesario realizar algún cambio, pero sin modificar considerablemente la forma de pensar de los desarrolladores, de tal manera que uno de los criterios empleados fue que la migración mejorara el desempeño o trabajo de los desarrolladores con el menor impacto.

2.4.2.1. Migración de CVS a Subversion

Cuando se migró de CVS a Subversion en el proyecto GNOME, se esgrimieron diferentes argumentos, de los que técnicamente aceptables se pueden rescatar:

- Similaridad a CVS. Dado que Subversion nació como una mejor versión de CVS y hereda sus conceptos básicos. Además, la interfaz del cliente de línea de comandos de Subversion es muy similar o igual a CVS, variando cuando hay una razón justificada para hacerlo de otra forma, como es el caso de revisiones por conjunto de cambios y no por archivos.

[33] También se le conoce como *svn*, por el nombre del comando que se ejecuta.

- Subversion guarda registro de la estructura de directorios, es decir, son parte del control de versiones. CVS solamente registra cambios en el contenido de los archivos y su existencia; en cambio en Subversion es posible controlar las versiones de los directorios, registrar los cambios de nombre, el copiado de archivos o cualquier operación de archivos que permita guardar meta-datos, entre las que también se incluye el cambio de permisos y atributos. Esta es una de las carencias principales de CVS y uno de los principales motivos esgrimidos para construir Subversion[22].

- La transacción que identifica un cambio se llama *commit*. En Subversion (y los nuevos SCM) la operación de *commit* es una transacción atómica, es decir, ninguna parte del *commit* toma efecto hasta que toda la transacción sea exitosa. En consecuencia, y a diferencia de CVS, los números de revisión son asignados a un *commit* y no a un archivo, lo cual permite tratar los cambios como una sola unidad. Como efecto secundario, los mensajes de registro de cambios se almacenan una sola vez y no por cada archivo modificado.

- Crear ramas y etiquetar revisiones[34] no tiene costo en Subversion y, por lo tanto, es una operación que tarda tiempo constante. Toda vez que se crea una rama o una etiqueta, lo que ocurre es una operación de *copy–on–write*, es decir, cuando se lleva a cabo una operación de *commit* (escritura en el repositorio), entonces se convierte en una rama de desarrollo y se guardan solamente los cambios. Por lo tanto, la diferencia entre una rama y una etiqueta radica en la existencia de *commits* posteriores. A diferencia de CVS, que almacena información de ramas y etiquetas una por cada archivo, por su limitación en la operación de *commit*[2].

- El protocolo de red de Subversion es mucho más eficiente en el uso de recursos, en la comunicación cliente–servidor, se envían sólo las diferencias en ambos sentidos; a diferencia de CVS, que por limitaciones en su diseño, envía las diferencias desde el servidor al cliente, pero no así desde el cliente al servidor.

- El modelo del repositorio de Subversion permite que los costos de las operaciones sean proporcionales al tamaño de los cambio realizados y no respecto al tamaño absoluto del proyecto en el cual se efectúan los cambios.

- Como consecuencia del control de la estructura del directorio de un proyecto, por sobre el manejo disperso de archivos, también es posible registrar

[34] *Branch* y *tag*, respectivamente.

los cambios con los enlaces simbólicos en sistemas UNIX. No obstante, en sistemas de archivos que no manejan el concepto de enlace simbólico no es posible[35].

- Manejo eficiente de archivos binarios. Subversion utiliza un algoritmo de diferencias binario, tanto para transmitir como para almacenar las revisiones de forma sucesiva. De esta forma, es eficiente en el manejo de archivos binarios como en archivos de texto. CVS sólo maneja correctamente archivos de texto, para archivos binarios almacena cada revisión[36] con el tamaño completo del archivo, lo cual es particularmente ineficiente para las aplicaciones que disponen de muchos iconos, audio u otros archivos binarios.

Otras razones, técnicamente débiles, fueron:

- Todas las salidas del cliente de línea de comandos Subversion se diseñaron para que fueran legibles para el humano y a su vez pudieran ser procesadas automáticamente por programas externos. Es débil, dado que existían programas que procesan la salida de CVS y aunque no haya sido trivial su implementación era un problema resuelto.
- Subversion hace uso de la biblioteca *gettext*, lo que permite disponer de sus herramientas en diferentes idiomas, esto es, mostrar los mensajes traducidos ya sean de información o de ayuda de acuerdo a las configuraciones locales, lo que *potencialmente* permitiría la incorporación de nuevos colaboradores al proyecto (aquellos con limitaciones en el idioma inglés). Esta argumentación es débil porque posteriormente se migró de Subversion a *git*, y este último no usa *gettext* y sólo está disponible en inglés.

2.4.3. *git*

Cuando se evaluó la necesidad de utilizar un sistema de control de versiones descentralizado, se tuvo en consideración tres herramientas: Bazaar-NG, Mercurial y *git*. Se documentaron los requerimientos, ventajas y desventajas de cada herramienta[37]. El 19 de marzo de 2009, se anunció la migración a *git*[38], usando como referencia principal en análisis de la votación realizada en diciembre de 2008, en donde

[35] Es un detalle menor, porque la mayor parte del desarrollo se realiza en sistemas UNIX y Linux.

[36] *Revisión* corresponde a un cambio o conjunto de cambios. Para más detalles ver la sección 2.3.5.

[37] `http://live.gnome.org/DistributedSCM`.

[38] `http://mail.gnome.org/archives/gnome-infrastructure/2009-March/msg00064.html`.

votaron todos los colaboradores con acceso de escritura al repositorio Subversion de GNOME[39]. Se formó un equipo, conformado por los principales propulsores del uso de *git*, y se documentó el calendario y proceso de migración.

Cabe destacar que, para entender el cambio de GNOME desde Subversion a *git*, es necesario entender la relación que existe entre el proyecto GNOME y la iniciativa Free Desktop[40]. Free Desktop nació por la necesidad de compartir elementos comunes entre los distintos escritorios disponibles para el sistema XWindow, así como construir extensiones a XWindow[41]; de tal manera de establecer estándares con los cuales garantizar interoperabilidad. Dicha iniciativa fue impulsada inicialmente por colaboradores de GNOME[42], por lo que proyectos de GNOME que eran independientes de un escritorio se movieron a Free Desktop.

En Free Desktop no existe una organización establecida como GNOME o KDE, más bien agrupa proyectos independientes que buscan ser utilizados ampliamente. De esta manera, algunos proyectos comenzaron a utilizar *git* como SCM, aparentemente motivados por la influencia de Keith Packard[43] cuando se decidió migrar de SCM para mantener el código fuente de X.org desde CVS a *git*[44]. Sería interesante estudiar la influencia que pueda tener uno o más miembros de una comunidad de desarrolladores al momento de elegir una herramienta por influencia más que por asuntos meramente técnicos.

2.5. Ciclo de entrega de nuevas versiones de GNOME

El ciclo de desarrollo de GNOME se rige por plazos de 6 meses. En la tabla 2.1 se indican las fechas con los principales hitos en el desarrollo de GNOME relacionado con la entrega de versiones. En el período entre GNOME 2.0 y 2.2 se estableció un calendario que comenzó a regir una vez que se entregó esta última. Desde allí en adelante, se fue afinando hasta alcanzar reguralidar en la entrega cada 6 meses de versiones mayores, y en el intermedio se entregan versiones menores que corresponden a correcciones de fallos, actualización de documentación y/o traducciones.

[39] http://blogs.gnome.org/newren/2009/01/03/gnome-dvcs-survey-results/.

[40] http://www.freedesktop.org/.

[41] http://www.freedesktop.org/wiki/MissionStatement.

[42] Principalmente Havoc Pennington, http://mail.gnome.org/archives/gnome-list/2000-April/msg00476.html.

[43] Keith Packard es uno de los desarrolladores principales de X.org y ha permitido reunir las necesidades de los proyectos de escritorios con las funcionalidades que debe proveer XWindow, http://www.xfree86.org/pipermail/forum/2003-March/002165.html.

[44] http://keithp.com/blogs/Repository_Formats_Matter/.

Hitos	Fecha	Días	Observaciones
Inicio	15-08-1997		Anuncio en múltiples listas de correo
1.0	03-03-1999	565	Primera versión “estable”
1.0.53	12-10-1999	223	Primera versión realmente estable
1.2	25-05-2000	226	
1.4	02-04-2001	312	
2.0	**27-01-2002**	300	Reescritura de muchos componentes
2.2	06-02-2003	375	Se establece calendario de versiones
2.4	11-09-2003	217	
2.6	31-03-2004	202	
2.8	15-09-2004	168	Separación entre plataforma y escritorio
2.10	10-03-2005	176	
2.12	07-09-2005	181	
2.14	15-04-2006	220	Proyecto Ridley (limpieza de código)
2.16	06-09-2006	144	Cambio de SCM de CVS a Subversion
2.18	14-03-2007	189	
2.20	19-09-2007	189	
2.22	12-03-2008	175	
2.24	24-09-2008	196	
2.26	18-03-2009	175	Cambio de SCM de Subversion a Git
2.28	23-09-2009	189	
2.30	29-03-2009	187	

Tabla 2.1: Entrega de versiones de GNOME a través de los años

El promedio del ciclo de desarrollo de GNOME desde la versión 2.4[45] hasta la versión 2.28[46] es de 186 días.

En el desarrollo de 3.0 se considera la concreción del Proyecto Ridley[47], esfuerzo iniciado en la versión 2.14 (ver tabla 2.1) y cuyo objetivo es consolidar bibliotecas externas en GTK+, es decir, aquellas que partieron como bibliotecas experimentales pero posteriormente comenzaron a utilizarse ampliamente. El Proyecto Ridley también considera la eliminación de aquellas bibliotecas para las cuales no existe un propósito bien definido, y además, integrar en una biblioteca principal aquellas funcionalidades que van en directo beneficio de todos los subproyectos.

El cambio de numeración permite romper la compatibilidad a nivel de API y de ABI, lo que permitirá eliminar todas aquellas funciones que han sido declaradas como no recomendabas u obsoletas, pero que por el compromiso de compatibilidad autoimpuesto para la serie 2.*x* se siguen manteniendo.

2.5.1. Calendario de entrega de nuevas versiones

El proceso de desarrollo de GNOME es definido respecto de los plazos. Puede variar la cantidad de versiones estables e inestables dentro de un ciclo de desarrollo, pero siempre se mantiene en márgenes medibles en semanas.

Las versiones se entregan los días miércoles, lo cual permite definir el calendario del ciclo de desarrollo en términos de semanas más que días. Sólo en casos excepcionales una entrega se puede diferir.

En la figura 2.2 se puede apreciar el proceso de entrega para las versiones 2.26 y 2.28, la cual es muy similar a todos los procesos anteriores a partir de la versión 2.2, que como se aprecia en la columna *Días* de la tabla 2.1, es bastante regular.

Mediante la numeración de las versiones se puede distinguir una versión estable de una versión de desarrollo, siguiendo el formato *z.x.y*, en donde *z* es la línea de desarrollo principal en donde se garantiza compatibilidad a nivel de API y ABI, tal como se indicó anteriormente; si *x* es par indica una versión estable y en caso contrario es de desarrollo; finalmente *y* indica las versiones subsecuentes de *x*. En la figura se puede ver la separación que ocurre cuando al momento de entregar 2.26.0, en donde la versión de desarrollo se estabiliza y se crea una rama estable

[45] La versión 2.4 corresponde a la primera versión entregada con un calendario definido de desarrollo.

[46] 2.28 es la última versión entregada al momento de escribir esta tesis.

[47] `http://live.gnome.org/ProjectRidley`.

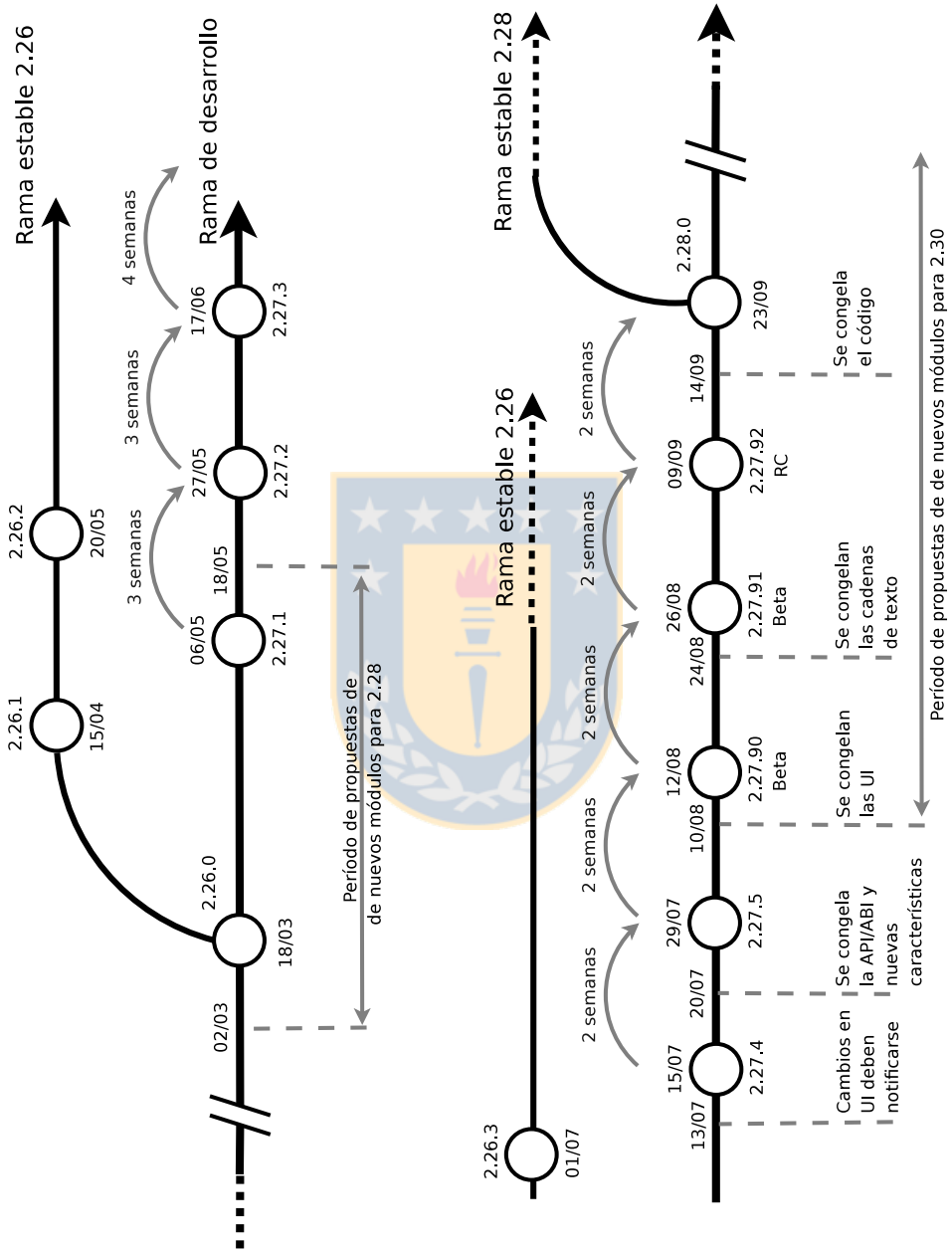

Figura 2.2: Diagrama con el ciclo de desarrollo de 6 meses de GNOME

en donde en el ciclo de desarrollo de 6 meses habrán 4 ó 5 nuevas entregas estables (en el ejemplo, continuando desde 2.26.1 a 2.26.3), y la cantidad dependerá de las necesidades. Por otro lado, en la rama de desarrollo también se producen entregas (2.27.1 a 2.27.92), las que pueden varíar entre 6 y 9 en un mismo ciclo, las primeras corresponden a versiones de desarrollo mientras que las 3 últimas son de estabillización y son consideradas *beta*[48]. Es necesario notar, que las versiones de desarrollo en un principio se relacionan a incorporación de nuevas características (2.27.1 a 2.27.5), luego a permitir el trabajo de documentación, traducción y pruebas (2.27.4 en adelante) y finalmente, a corrección de fallos y refinamiento (2.27.90 a 2.27.92).

Cuando se da curso al período de estabilización (dos días antes de entregar 2.27.90), en paralelo se inician las propuestas y discusiones de las características que se incorporarán en el siguiente ciclo. Algunas pueden ser completamente nuevas, otras pueden ser características que fueron retrasadas en las discusiones de ciclos anteriores.

[48] El concepto *beta* varía un poco del contexto del software comercial, dado que todas las versiones de desarrollo siempre están disponibles para el público. Sin embargo, el concepto es de utilidad para las distribuciones (consumidores directos).

Capítulo 3

Metodología

En este capítulo se presenta la metodología desarrollada, la cual se compone de cinco etapas. La primera es el proceso de caracterización de los datos que se recopilarán durante el proyecto. La segunda etapa corresponde a la preparación de los datos, lo cual involucra obtener copias del historial de desarrollo de los subproyectos relacionados con el escritorio. Posteriormente, en la tercera etapa, considerando que GNOME es un proyecto que reune varios subproyectos, es necesario definir los subproyectos que deben ser procesados para obtener los datos en un formato que permita realizar análisis sobre ellos. La cuarta etapa es la evaluación de uso de herramientas de extracción de datos de repositorios y determinar si resulta apropiado emplearlas, extenderlas o escribir herramientas nuevas. Finalmente, en la quinta etapa, se deben evaluar los datos utilizando análisis estadísticos.

3.1. Caracterización de los datos a recopilar

Esta etapa tiene como objetivo conocer e identificar las características que permitan clasificar el proyecto GNOME a través del tiempo y de los elementos de interés de este estudio. De esta manera, se puede enfocar el análisis en grupos más representativos, reduciendo el objeto de estudio.

Una razón para reducir el objeto de estudio es el tiempo necesario que se requiere para el procesamiento de los datos, aunque también contribuye a realizar un análisis más localizado.

Como se ha indicado anteriormente en la sección 2.1, GNOME no es un producto que resulte de la producción de una sola pieza de programa sino que es el resultado

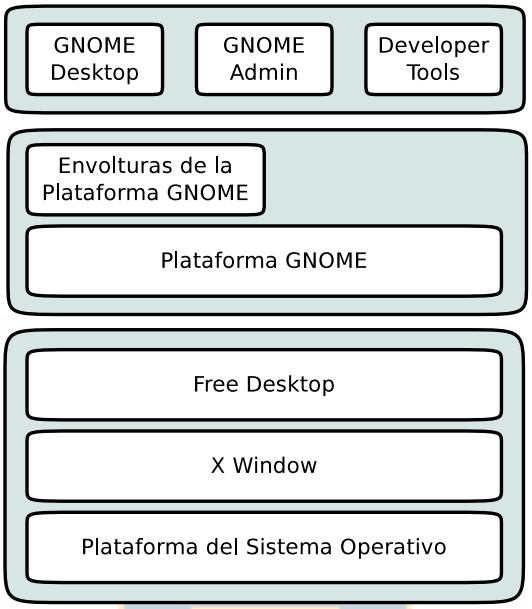

Figura 3.1: Arquitectura de las plataformas subyacentes en GNOME

de la unión de varios proyectos. Además, cada proyecto cuenta con su repositorio, y cuando se define una entrega como producto, se agrupa un conjunto de ellas que cuentan con características comunes que las describen.

En la figura 3.1 se aprecia como se relacionan las plataformas subyacentes en GNOME con los productos que se ofrecen. En la capa inferior, se encuentran apiladas la plataforma del sistema operativo, luego X Window y encima Free Desktop. La capa inferior no corresponde al proyecto, aunque es común que los desarrolladores de ambientes de escritorio como GNOME y KDE, colaboren o interactúen también con ellos. La capa intermedia, está compuesta por la plataforma GNOME, que es el conjunto de bibliotecas que provee las funcionalidades generales sobre las cuales construir aplicaciones. Sobre la plataforma GNOME, se encuentran las envolturas que permiten construir aplicaciones para GNOME en otros lenguajes distintos de C. Finalmente, en la capa superior, se encuentran los paquetes de aplicaciones construídas que utilizan la plataforma GNOME en cualquiera de los lenguajes disponibles. En esta última capa, se encuentra el escritorio, las herramientas de administración orientado a entornos de escritorio, las herramientas de desarrollo y, últimamente, la plataforma para dispositivos móviles.

3.1.1. La plataforma de desarrollo

Esta plataforma es parte esencial del funcionamiento de GNOME, y no ha habido variación desde la versión 2.16. Es la plataforma de más bajo nivel de GNOME, escrita en C y provee la base para los servicios que todas las aplicaciones de GNOME pudieran esperar, incluyendo las plataformas incrustadas.

Este producto existe desde la versión 2.8. Entre las versiones 2.0 y 2.6 estas bibliotecas eran parte del núcleo de GNOME (también conocido como el Escritorio GNOME[1]).

Sólo los módulos de este producto están afectos a las reglas de compatibilidad a nivel API y ABI. Para que un módulo sea incorporado a esta plataforma no es suficiente cumplir con la regla de compatbilidad, sino que también debe ser ampliamente utilizado por otros proyectos. No obstante, y como se indicó en la sección 2.5, a través del Proyecto Ridley se busca disminuir la cantidad de bibliotecas, por lo que este producto es poco susceptible a cambiar.

En la figura 3.2 se observa el grafo de dependencias entre las bibliotecas de la plataforma de desarrollo de GNOME en la versión 2.28. Las bibliotecas con tono distinto de blanco corresponden a aquellas que se mantienen por el compromiso que existe en el proyecto de compatibilidad[2] a nivel de API y ABI dentro de la serie 2.*x*, pero cuyo uso en proyectos nuevos es desaconsejado.

En el diagrama también se puede observar que las bibliotecas *at-spi* y *gconf* son las únicas que dependen de bibliotecas cuyo uso se desaconseja, las cuales son *libbonobo* y *ORBit2*, respectivamente. Con las bibliotecas *libbonobo*, *ORBit2* y *libIDL* se implementa el sistema de componentes reusables Bonobo –por medio de CORBA– y es este sistema de componentes el que se intenta desincentivar en las capas superiores.

Finalmente, bajo la línea punteada se encuentra *gnome-doc-utils*, que no es parte de la plataforma de desarrollo sino que es parte de las herramientas de escritorio de GNOME, lo cual parece una contradicción, dado que sobre la plataforma se construye el escritorio. *gnome-doc-utils* es una colección de utilidades de documentación y hojas de estilo XLST que se utilizan para construir la documentación de GNOME.

[1] *GNOME Desktop*, por su nombre en inglés.

[2] `http://live.gnome.org/ReleasePlanning/ModuleRequirements/Platform`.

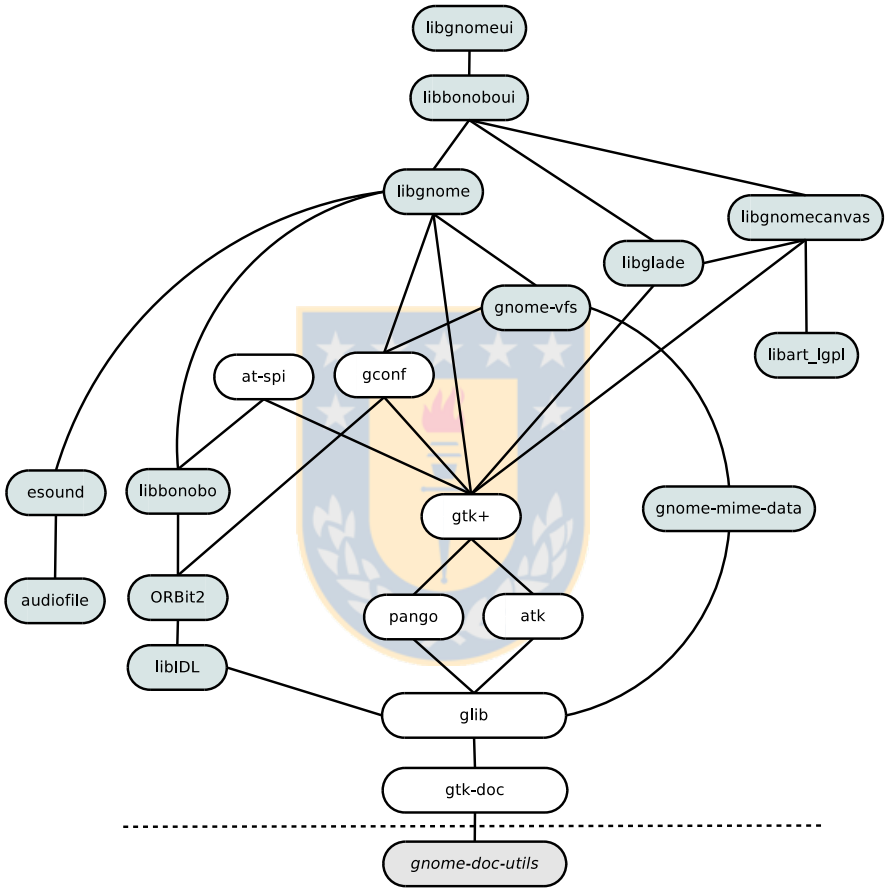

Figura 3.2: Dependencias de bibliotecas de la plataforma de desarrollo de GNOME a la versión 2.28

3.1.2. Envolturas de la plataforma de desarrollo

Aunque la plataforma de desarrollo de bajo nivel está escrita en C, se provee una plataforma de desarrollo que la envuelve y ofrece acceso mediante diversos lenguajes, tales como C++, Python, Java, Mono[3], Perl y Javascript. Esta plataforma permite que las aplicaciones puedar ser escritas en el lenguaje de preferencia de cada programador.

3.1.3. El núcleo o escritorio

El núcleo corresponde a las aplicaciones que representan la semántica del escritorio hacia el usuario. En este producto reside el panel, el administrador de archivos, herramientas y accesorios básicos, navegador web, herramientas de trabajo colaborativo (correo electrónico, contactos, calendario).

Este producto es el más grande, y en el que cada ciclo se añaden nuevas bibliotecas y/o aplicaciones, las cuales se discuten por un período de dos meses (ver diagrama 2.2).

3.1.4. Herramientas de desarrollo y administración

Estos productos son pequeños por la cantidad de proyectos que involucran.

Las herramientas de desarrollo están constituídas por proyectos para asistir a la programación, creación de documentación y verificador de accesibilidad de las aplicaciones (*accerciser*, *anjuta*, *devhelp*, *gdl*, *glade3*, *gnome-devel-docs*).

El producto de administración contiene software útil para la administración de instalaciones en gran escala, con los cuales es posible realizar administración remota y bloqueo/habilitación de funcionalidades. En la versión 2.26 sólo hay dos proyectos dentro de este producto: *pessulus* y *sabayon*.

3.2. Preparación de los datos

El código fuente de GNOME entre el 15 de agosto de 1997 y el 06 de septiembre de 2006 estuvo albergado en un gran repositorio CVS con distintos módulos, en donde cada módulo era un proyecto, tal como se muestra en la figura 3.3. Cada proyecto es libre de definir las ramas y etiquetas en el momento que las personas

[3]Mono es la implementación libre de .NET, por lo tanto, abarca más de un lenguaje.

a cargo lo estimen conveniente, los cuales usualmente siguen las convenciones recomendadas[4] para etiquetar el estado del repositorio luego de una entrega de una versión.

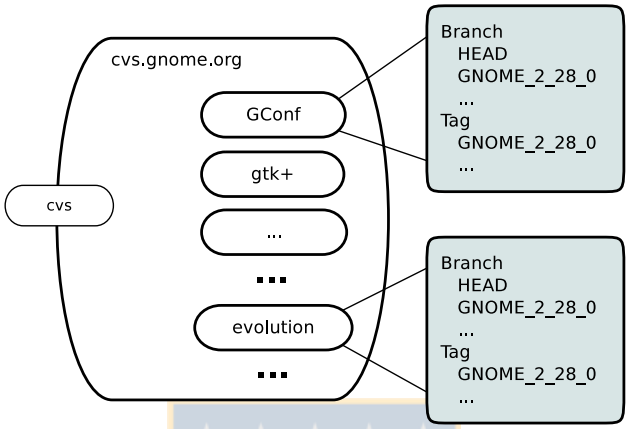

Figura 3.3: Repositorio *cvs* de GNOME

El repositorio, como se muestra en la figura 3.3, aún se encuentra almacenado, aunque sin acceso público. Los desarrolladores de GNOME que tienen acceso a *shell* al servidor *git.gnome.org*[5] pueden obtener una copia de dicho repositorio, que se encuentra localizado en /cvs/gnome. El tamaño del repositorio es de 12GB.

El hecho de contar con un repositorio con varios módulos puede ser fuente de problemas cuando se comete un error, puesto que éstos se pueden propagar en todo el repositorio, especialmente cuando existen casos especiales con módulos que son compartidos por un conjunto amplio de otros módulos. En el caso de GNOME, el módulo *gnome-common* contenía las macros de *autoconf* para construir cualquier módulo.

En el caso de *git*, se trata de un servidor[6] que alberga múltiples repositorios, como se muestra en la figura 3.4. Si en CVS cada proyecto se almacenaba en un módulo del repositorio, en *git* cada proyecto cuenta con su propio repositorio, los que se encuentran almacenados bajo el mismo directorio en el sistema de archivos del servidor.

[4] `http://live.gnome.org/MaintainersCorner`.

[5] Los mantenedores de proyectos albergados en los servidores de GNOME tienen habilitado el acceso, el cual es necesario para llevar a cabo la liberación o entrega de una nueva versión. En estricto rigor, basta el acceso a *shell* en cualquier máquina que tenga montado dichos directorios vía NFS.

[6] El cual se puede acceder vía http, *git* o *ssh* en *git.gnome.org*.

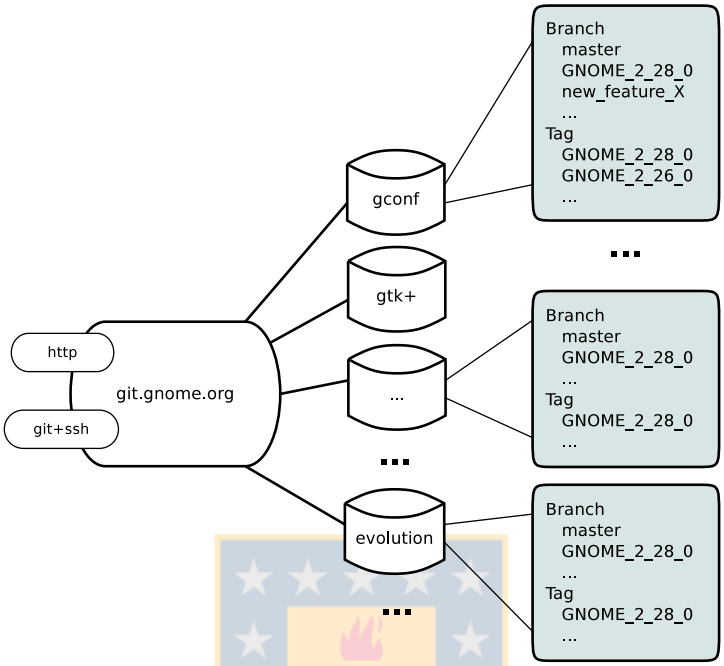

Figura 3.4: Repositorio *git* de GNOME

Cada repositorio es independiente y contiene su propia historia. Así por ejemplo, si realiza un clonado de *evolution* se obtiene una copia con toda la historia del proyecto desde sus inicios, incluyendo la historia de CVS y Subversion que la precedieron.

Es necesario aclarar que es una convención que *git.gnome.org* sea el repositorio central del proyecto, y se relaciona al flujo de trabajo de las personas involucradas en el proyecto, más que al carácter técnico en que cada copia es idéntica al original con toda su historia.

No obstante, en adelante, se utilizará el concepto repositorio de GNOME como el grupo de repositorios albergados en *git.gnome.org*.

El tamaño del repositorio de GNOME, a agosto de 2009, es de 4,7GB, la cual corresponde solamente a la historia. Si se añade la copia de trabajo, el tamaño aumenta a 8GB. Desde entonces hasta noviembre de 2009, el tamaño del repositorio se incrementó a 5,5GB; es decir, un 14,9 % y que esboza la salud del proyecto. Es conveniente recordar, que el tamaño involucra toda la historia del proyecto, y a pesar que tiene 7 años más de historia almacenada que CVS el espacio requerido es mucho menor, dado que la estructura con la cual *git* almacena los datos es más efi-

ciente que CVS, y además, se almacena comprimido[5]. El tamaño del repositorio influye en el intercambio de datos por Internet.

Para el caso de Subversion, el esquema de repositorios es similar al de *git* (cada módulo es un repositorio), pero desde el punto de vista de los clientes el comportamiento es similar a CVS (sólo se obtiene una copia de trabajo y no la historia del proyecto). El tamaño del repositorio, que incluye la historia de CVS y migración a Subversion hasta abril de 2009, es de 24,2GB.

3.3. Obtención de los datos

El conjunto de repositorios *git* de GNOME contienen toda la historia de los proyectos. Sin embargo, los otros repositorios (CVS y Subversion) pueden ser útiles si fuese necesario determinar y/o corregir errores en la migración de un sistema a otro, que no afectan a los desarrolladores del proyecto pero que sí pueden hacerlo a quienes lo estudian.

3.3.1. Obtención de repositorios CVS y Subversion de GNOME

Dado que los repositorios de CVS y Subversion no reciben escrituras desde fines de 2006 y principios de 2009, respectivamente (ver tabla 2.1); sólo es necesario realizar la copia una sola vez. Para preservar las fechas y permisos se emplea *rsync*, que además, permite retomar el proceso si éste fuese interrumpido.

La forma de obtener los datos de CVS y Subversion es la siguiente:

```
$ rsync -e ssh -aH gpoo@git.gnome.org:/cvs/gnome dst
$ rsync -e ssh -aH gpoo@git.gnome.org:/mnt/git-data/svn dst
```

Donde *dst* corresponde al directorio de destino para el repositorio en forma local y *gpoo* es la cuenta de acceso utilizada. Si no se dispone de una cuenta *shell*, se puede solicitar el acceso a los administradores de sistemas o con algún desarrollador de GNOME con acceso.

Lo mismo podría aplicarse para */cvs/archive* y */mnt/git-data/svn-archive*, ambos directorios contienen los repositorios de proyectos o subproyectos que han sido declarados obsoletos porque quedaron huérfanos[7], fueron reemplazados, o porque fueron experimentos en donde se utilizó un repositorio/módulo nuevo en vez de

[7] Sin mantención

crear una rama de desarrollo. Un posible estudio de dichos archivos podría ser determinar la naturaleza de dichos proyectos, estimar el esfuerzo de desarrollo que se ha descartado, el tamaño de éstos y cómo pudo afectar en el resto de los proyectos. No es objetivo de este trabajo realizarlo, pero sí vale la pena mencionar.

3.3.2. Obtención de repositorios *git* de GNOME

Dado que *git* es el SCM en uso, se utiliza esta misma herramienta para replicar cada repositorio; pues no hay que olvidar que al realizar la operación *clone* en un SCM distribuido se obtiene toda la historia del repositorio, convirtiéndole en un espejo o respaldo hasta ese instante del tiempo.

Es necesario conocer de antemano el nombre de los repositorios, para lo cual existen dos archivos disponibles públicamente dicha información:

- `http://git.gnome.org/repositories.txt`: El cual contiene un listado plano con los nombres de los repositorios disponibles.
- `http://git.gnome.org/repositories.doap`: El cual es un archivo XML que contiene la definición de cada proyecto[8], nombre y contacto de los mantenedores, entre otros datos.

La forma de obtener los datos de *git* –utilizando comandos propios de Linux/UNIX y el *shell*– es la siguiente:

```
$ wget http://git.gnome.org/repositories.txt
$ for i in `cat repositories.txt`
  do
    git clone --mirror git://git.gnome.org/$i
  done
```

Este acceso se realiza en forma anónima, por lo tanto, es replicable por cualquier persona interesada o en desarrollar software para GNOME o para estudiar dicho software. La opción '*–mirror*'[9] es para obtener una copia idéntica del servidor, sin necesidad de obtener una copia de trabajo.

Al tratarse de repositorios con desarrollo activo, en constante cambio, posteriormente se puede utilizar la misma herramienta para mantener actualizado cada uno

[8] DOAP es un esquema RDF utilizado para describir proyectos FOSS. La sigla significa *Description Of A Project*.

[9] Esta opción está disponible sólo a partir desde *git* 1.6.

de ellos. Suponiendo como directorio de trabajo aquel que es base para los repositorios *git*, dicho proceso se puede realizar de la siguiente forma:

```
$ for i in `ls -1`
  do
    (cd $i; git pull)
  done
```

El primer proceso requiere de mayor tiempo, puesto que es necesario transmitir 5,5GB a través de Internet y dependerá de la capacidad del enlace disponible. Sin embargo, las actualizaciones posteriores requerirán menos tiempo, dado que sólo se transmitirán los objetos nuevos de cada repositorio.

El proceso anterior corresponde a una simplificación, el cual se puede automatizar considerando validaciones, tanto para la actualización como la recuperación de nuevos repositorios, de la siguiente forma:

```
#!/bin/bash
BASE=gnome.git/mirror
REPO=$(dirname $BASE)/repositories.txt
# Get the minor version (Y) from 'git version x.Y.z'
GIT_VERSION=$(git version | cut -f2 -d.)
if [ ! -d $BASE ]; then
    mkdir -p $BASE
fi
wget -O $REPO http://git.gnome.org/repositories.txt
for i in $(cat $REPO)
do
    if [ -d $BASE/$i ]; then
        echo "Fetching $i"
        (cd $BASE/$i && git fetch)
    else
        echo "Getting $i"
        URL="git://git.gnome.org/$i"
        # git >= 1.6 supports the --mirror shorthand
        if [ $GIT_VERSION -gt 5 ]; then
            git clone --mirror $URL $BASE/$i
        else
            # Workaround for git < 1.6
            git clone --bare $URL $BASE/$i
```

```
            (cd $BASE/$i &&
             git remote add --mirror origin $URL)
        fi
    fi
done
```

Los repositorios obtenidos fueron 673, de los cuales es necesario elegir aquellos que serán objeto de estudio.

3.4. Consideraciones generales respecto a *git*

Inicialmente fue concebido como un *toolkit* para tener registro de versiones (SCM) y no como un SCM orientado al usuario final. Conceptualmente, se emplean conjuntos de *verbos*[10] que realizan operaciones de bajo nivel, los cuales están diseñados con la filosofía de UNIX: realizar una función simple, bien hecha y que se pueda encadenar mediante filtros[5].

Internamente, *git* es un sistema de archivos direccionables por su contenido con una interfaz de SCM encima. Lo que almacena son pares de datos *clave–valor*. Por lo tanto, es posible acceder y modificar los recursos de *git* utilizando comandos del sistema. Sin embargo, para la mayor parte de los casos, *git* ofrece una forma cómoda y segura de acceder a bajo nivel sobre los objetos que almacena. Esto puede ser una característica importante de tener en cuenta al momento de diseñar herramientas que intenten analizar información.

Bird y otros[3] han señalado un conjunto de promesas y peligros a los que se expone un investigador al analizar la historia del desarrollo de un software llevada por *git*. de Las promesas describen las potencialidades de estudiar un repositorio con git, mientras que los peligros son los riesgos que es necesario prevenir.

Las promesas son:

1. Cada desarrollador puede tener una copia del repositorio, por consiguiente, tiene toda la historia del proyecto. Es posible revisar la historia, incluso de aquellas partes que están en progreso o que desarrollo experimentales que no necesariamente se utilizarán.

2. *git* facilita revisar la historia de un proyecto de una manera mucho más rica, dado que es capaz de seguir el desarrollo luego de mezclar distintas ramas o de distintos repositorios.

[10] en forma de comandos o instrucciones

3. *git* almacena información de *fast-forward merges*, *rebase* y *pulls* en los *logs*. Cada información del flujo de trabajo de un desarrollo se puede almacenar, tal como, cuando realizó un *checkout*. Sin embargo, es necesario tener acceso al repositorio privado del desarrollador. Esta información se puede perder si el desarrollador utiliza *git gc*, o incluso *git gc –aggresive*.

4. Hay atributos que permiten registrar responsabilidades dentro de un proyecto (*signed-off-by*).

5. *git* registra explícitamente la autoría de aquellos colaboradores que no son parte del núcleo de desarrollo.

6. Toda la metadata es local.

7. *git* guarda registro del contenido, es posible seguirle el rastro a cada línea, incluso cuando se mueven o se copian.

8. *git* es más rápido y a menudo requiere menos espacio de almacenamiento que los sistemas centralizados.

9. Muchos SCM, tales como CVS, SVN, Perforce y Mercurial ; pueden ser convertidos a *git* con el historial de sus ramas, mezclas y tags en forma intacta.

De las promesas enumeradas, en este estudio se han considerado todas excepto la número 3 y la 9. De esta forma, las dos primeras corresponden a una de las motivaciones para llevar a cabo este estudio. La cuarta no es una promesa fuerte, dado que depende mucho del hábito de los desarrolladores el cual puede o no estar muy difundido; vale decir, aquellos que migran de otros SCM no necesariamente explotarán las potencialidades de uno nuevo, aunque sin duda es objeto de estudio el determinar si es así o no. La quinta promesa se puede evidenciar en el aumento de desarrolladores desde el momento que se produce el cambio de SCM, según muestran Bird y otros[3], es un hecho para el caso del lenguaje Ruby. La sexta está muy relacionada con las dos primeras. La séptima es muy útil, por cuanto permite contabilizar correctamente las modificaciones sin importar si los archivos fueron renombrados o copiados[11]. La octava promesa quedó en evidencia en la sección 3.2.

Las promesas 3 y 9 no aplican en este estudio. La tercera promesa guarda relación con la forma de trabajar de un desarrollador en particular mediante el análisis de

[11] Basta utilizar las opciones *-M* y *-C* para que el comando *git log* muestre el resultado de las copias y modificaciones.

su repositorio. En este estudio, se analizará un repositorio "*central*". La novena promesa no aplica, por cuanto los datos ya se encuentran migrados a *git*.

Con respecto a los peligros, se enumeran a continuación:

1. La nomenclatura de *git* difiere de aquella empleada por los SCM centralizados: acciones similares tienen distintos comandos y comandos que comparten el nombre pueden tener significados diferentes.
2. En *git* existen ramas implícitas, que en SCM centralizados no existen.
3. En *git* no existe una línea principal de desarrollo. La historia no es líneal, sino un grafo dirigido acíclico.
4. Es posible reescribir la historia.
5. No siempre es posible determinar el origen de una rama.
6. No siempre es posible rastrear el origen de una mezcla o incluso determinar si hubo mezcla.
7. Los datos asequibles pueden contener sólo a los commits realizados en forma exitosa.

En el primer caso se trata de una advertencia, dado que la semántica de algunos comandos en git son distintos a otros. No obstante, dichas acciones tienen un mayor impacto en el usuario de SCM, mas no en quien lo estudia. El punto de partida del análisis de un repositorio es su historia, para lo cual es necesario interiorizarse en su formato.

Los peligros 2, 5 y 6 no aplican en este estudio, puesto que no se abordan las ramas de desarrollo. Hay suficiente material de estudio siguiendo el desarrollo principal. No obstante, es importante tener en mente que el esfuerzo final, aunque pueda ser representativo, no será completamente preciso.

El tercer peligro indicado es un riesgo controlado, dado que depende más del flujo de desarrollo del proyecto que de la herramienta en sí. En el caso de GNOME, la línea principal de desarrollo está dada por la rama principal en el repositorio central; y en donde el repositorio central es una convención aceptada implícitamente (se puede revisar la figura 3.4).

Con respecto al cuarto peligro, es un riesgo que se debe considerar al momento de analizar los datos y esbozar las conclusiones. El hecho que la historia se escriba

localmente, indica que ésta tiene movilidad[12] dentro del repositorio hasta antes que sea publicado. Esto no afecta en el tamaño o complejidad final del código, sino más bien, en determinar en forma precisa cuando ocurrió. Una consideración similar corresponde al séptimo peligro indicado.

3.5. Evaluación de uso de herramientas

3.5.1. CVSAnalY

CVSSAnalY[29] es un proyecto dentro del grupo LibreSoft de la Universidad Rey Juan Carlos y es la herramienta más completa de las estudiadas, aunque su lógica está muy amarrada a CVS y Subversion, en donde el soporte para *git* se encuentra aún en la rama de desarrollo y en carácter de experimental. El rendimiento es pobre para procesar repositorios con harta historia. Como es necesario trabajar con muchos repositorios, entonces se vuelve complejo. Sin embargo, en comparación con *Git Mining Tools*, procesa los datos entre 3 y 6 veces más rápido, además que permite parcelarlos por tipo de archivos, acciones aplicadas en los tipos de archivos, autores.

Otra ventaja de esta herramienta, es la inclusión de extensiones para calcular las métricas más populares (SLOC, McCabe), así como la posibilidad de escribir nuevas extensiones.

Por otro lado, no tiene soporte para el manejo de identidades, múltiples proyectos ni la relación entre múltiples repositorios. Sin embargo, se puede extender la herramienta para lograr dicho objetivo. Es justo indicar, que ninguna de las herramientas analizadas puede hacerlo.

Ha sido reescrito, pero aún tiene trozos de código que es necesario mejorar, así como acelerar el procesamiento de los logs de *git*. En la primera evaluación parecía ser la herramienta más recomendable para trabajar, por cuanto se puede extender y se pueden añadir las funcionalidades que otros programas, como *Carnarvon*, entregan.

Por otro lado, CVSAnalY procesa los datos para ser incorporados a una base de datos y a partir de allí realizar análisis. En diferentes estudios de Robles y otros[29, 30, 28, 26] se ha utilizado para analizar repositorios independientes, pero no un conjunto de repositorios como es el caso de GNOME. Aunque el modelo de datos (ver figura 3.5) permite ser extendido hacia dicho objetivo, es necesario poner a prueba la herramienta para hacerlo.

[12] Por ejemplo, al utilizar el comando *rebase* de *git*.

Para este estudio, se deben añadir tablas nuevas en donde se puedan indicar las relaciones entre distintos repositorios (por ejemplo, *GNOME Development Platform*, *GNOME Desktop*, *GNOME Mobile*, etc.), realizar asociaciones de identidades, asociar identidades con organizaciones (empresas, fundaciones, otras) para luego realizar otro tipo de análisis.

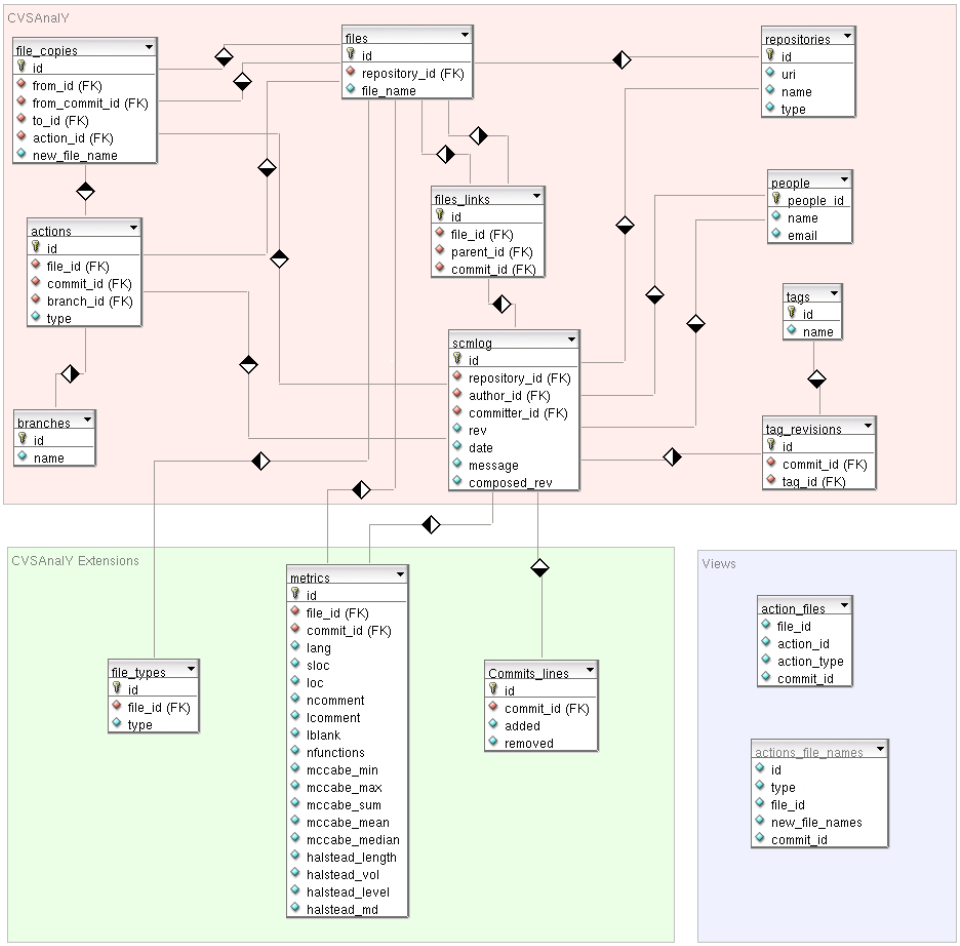

Figura 3.5: Modelo de datos de CVSAnalY

Se realizaron pruebas incrementales con un conjunto pequeño de repositorios. Se fue ajustando el código, añadiendo funcionalidades[13] y mejorando el código. Se realizaron pruebas con el motor de base de datos en la misma máquina y en un

[13]Por ejemplo, soporte para Postgresql y mediante el cual se encontraron algunas debilidades del soporte para otras bases de datos.

equipo dedicado, sin embargo, el rendimiento no resultó como se esperaba. Procesar 6 repositorios tardó más de 2 semanas, y en algunos hubo problemas de memoria. Ello se contradice con la rapidez con que *git* realiza las operaciones más comunes y que se describieron en la sección 2.4.3.

3.5.2. Carnarvon

Carnarvon fue un proyecto dentro del grupo LibreSoft –al igual que CVSAnalY– y cuyo propósito es analizar la edad de un programa. Para determinar la edad, extrae e indexa datos de modificaciones de cada línea de código. A partir de la historia de un proyecto, determina el nivel de mantención y el esfuerzo que podría suponer mantenerlo en el futuro[16].

La idea del proyecto es interesante. Sin embargo, y aunque el proyecto se consideraba estable, no ha sido mantenido desde mayo de 2007. La última versión está limitada a analizar repositorios de CVS y Subversion. Su construcción sigue una lógica similar a las primeras versiones de CVSAnalY, la cual no es suficientemente modular como para que permita incorporar nuevas características sin realizar cambios estructurales del proyecto.

Carnarvon procesa la salida que genera el comando *annotate* de CVS o Subversion. Dicho comando provee información de la operación *commit* responsable de la incorporación de cada línea en un archivo. El equivalente, de manera conceptual, en *git* es el commando *blame*, aunque la salida difiere de CVS y Subversion.

Conceptualmente es interesante para estudios futuros porque permite el análisis a través del tiempo de las líneas de código, así como estudiar su edad en la forma que lo describe Parnas[20].

3.5.3. Git Mining Tools

Es una herramienta utilizada para realizar un análisis preliminar de los datos que se pueden estudiar en repositorios *git*. y con el cual se obtienen algunos datos básicos mostrados en el trabajo de Bird y otros[3]. Con ella se analizaron los datos para determinar el impacto en el número de autores distintos en el proyecto Ruby a través de los meses, el cual se uso como prueba de concepto que *git* expone más datos que Subversion, y por consiguiente, es más conveniente para estudiar.

La herramienta analiza el resultado entregado por el comando *log* de *git*. La herramienta fue escrita originalmente en Perl y posteriormente se reescribió en Ruby. En ambos casos, el rendimiento es de 3 a 6 veces más lento que CVSAnalY. Aunque

su código es mucho más simple, también es mucho más limitado en la información recolectada. La simplicidad de su código radica en que no intenta abstraer la mecánica de funcionamiento para procesar varias herramientas de SCM, sino que se enfoca solamente en *git*.

Esta herramienta puede ser útil para procesar repositorios pequeños o una cantidad limitada de éstos. Para el tamaño de los repositorios de proyectos en GNOME, no resulta viable. En una misma máquina, un repositorio pequeño como el del programa *evince* en CVSAnalY tardó 2:33 minutos de procesamiento; mientras que con *Git Mining Tools* el tiempo de procesamiento fue de 8:49 minutos. Considerando que el repositorio de GNOME alberga más de 600 proyectos, y que en la misma máquina el procesamiento del proyecto *evolution* tardó más de 20 horas con CVSAnalY, es de sentido común que *Git Mining Tools* no es la herramienta adecuada para el trabajo de esta tesis.

3.5.4. Git Data Miner

Git Data Miner es una herramienta construída por Jonathan Corbet para estudiar el código de Linux, y que es la base del informe técnico elaboarada bajo el alero de Linux Foundation[11], para determinar las mayores contribuciones en el núcleo de Linux, así como el aporte que las empresas realizan mediante desarrolladores pagados por ellas.

Esta herramienta analiza sólo un repositorio, dado que está enfocada al núcleo de Linux. Como se muestra en la figura 3.4, la situación de GNOME es distinta debido a que son múltiples, uno por cada subproyecto. Además, esta herramienta toma como variable principal la cantidad de operaciones de *commit* que realiza cada desarrolador. No obstante, esta herramienta contiene ideas interesantes para resolver ciertos problemas como la asociación temporal entre un desarrollador y una empresa, así como la medición del proceso de revisión de contribuciones realizados por los desarrolladores con más jerarquía en el núcleo de Linux, lo cual se logra por el uso estricto que se realiza en dicho proyecto de la opción *–signof*, situación que primero es necesario corroborar cómo se cumple dentro de los subproyectos de GNOME.

Git Data Miner no está escrito de manera modular, contiene muchas variables globales y está escrito de manera ad-hoc al núcleo de Linux. Luego de varias pruebas resultó ser la herramienta más rápida en realizar el procesamiento de un repositorio. El procesamiento de *evolution* tardó 13 minutos, frente a las 20 horas iniciales de CVSAnalY; y luego de reescribir ciertas partes del código se logró reducir a 7 minutos. Esto permite que el proceso de refinamiento de los datos sea más fluído.

Es necesario notar que CVSAnalY realizó algunas tareas distintas de *Git Data Miner,* tales como, almacenar los datos en una base de datos, registrar el renombrado y copiado de archivos, establecer las ramas del repositorio.

3.6. Análisis de los datos y refinamiento

Aunque es posible automatizar la extracción de datos a partir del registro histórico en el repositorio, hay aspectos de los datos que necesitan ser interpretados para darles un correcto tratamiento. La interpretación de ciertos datos los puede dar el investigador que conoce el contexto de los datos que estudia y que se logra mediante el refinamiento de los datos considerando aspectos que incluyen la afiliación de un colaborador con una empresa y/o un período de dependencia laboral, agrupación de múltiples identidades de un mismo individuo y la selección de las etiquetas asociadas a las versiones que se desea estudiar.

3.6.1. Determinación de identidades

Cuando se almancena la información del autor, lo que se guarda es el nombre y el correo electrónico indicado por dicho colaborador. Existen varias razones por las cuales un colaborador podría identificarse de distintas formas a través del tiempo, tales como:

1. Cambio de correo electrónico.

    a) Cambio de institución. Puede ocurrir por cambio de trabajo, cambio de institución en donde estudia o por poseer varias cuentas de correo electrónico y que usa indistintamente una u otra.

    b) Cambio del nombre de la institución. Una empresa puede cambiar de nombre, y por consiguiente de dominio de la cuenta de correo. Por ejemplo, GNOME Support International (*gnome-support.com*) cambió a HelixCode (*helixcode.com*), y posteriormente a Ximian (*ximian.com*).

    c) Fusión de empresas. Puede ser fusión o absorción, en donde prevalece sólo una dirección. Por ejemplo, Ximian fue adquirida por Novell, y por consiguiente, todos sus empleados cambiaron de correo electrónico.

2. Distinto nombre. Dos nombres distintos podrían asociarse a un mismo correo electrónico.

a) Cambio de nombre, como suele suceder en los países anglosajones en donde la mujer, al contraer matrimonio, adopta el apellido del hombre.

b) Error en la conversión de caracteres. Por problemas en el origen o en el destino, la codificación de caracteres puede cambiar el nombre de una persona. Por ejemplo, "*Germán*" podría quedar almacenado como "*GermÃ¡n*" lo cual ocurre porque el receptor interpretó la codificación UTF-8 como ISO-8859-1. Para evitar dichos problemas, era común encontrar colaboradores que simplemente usan ASCII y con la adopción de UTF-8, se ha comenzado a utilizar los caracteres extendidos.

c) El colaborador se identifica de manera distinta. Un usuario puede configurar, conciente o inconcientemente, su identidad de manera distinta en cada computador. Por ejemplo, "*Anna Dirks*" es alfabéticamente distinto de "*Anna Marie Dirks*", aunque se trata de la misma persona.

3. Uso de pseudónimos en vez de nombres reales. Estos pueden cambiar en el tiempo, por ejemplo, cuando un desarrollador comienza a colaborar por parte de una empresa.

4. Combinaciones de las anteriores para una misma persona.

Robles y González-Barahona[27] propusieron un método para identificar a los desarrolladores mediante el cruce de datos a partir de múltiples fuentes. El repositorio del código fuente es sólo una fuente de datos, y en este estudio es la única disponible. Aunque es posible identificar múltiples identidades de un mismo colaborador mediante un proceso de refinamiento semi–automático.

Git Data Miner maneja el concepto de múltiples identidades en el procesamiento de un repositorio, el cual es alimentado *a priori*. No obstante, también es posible vaciar los resultados de la base de datos de *Git Data Miner*, el cual es una estructura de objetos guardados como un diccionario. El proceso de refinamiento de identidades consiste en ejecutar varias veces *Git Data Miner*, en cada iteración se debe revisar las identidades comunes y proceder a definir los alias de un mismo desarrollador y obtener una salida consolidada en las siguientes iteraciones.

Si se tiene conocimiento de la comunidad de desarrolladores de un proyecto, el proceso puede resultar más preciso y más rápido en la inspección visual, aunque no por ello, menos tedioso.

3.6.2. Identificación de empresas

Por otro lado, este mismo proceso ayuda también a establecer una relación entre la identidad de un desarrollador y la empresa en la cual trabaja. Al conocer la comu-

nidad, es posible darse cuenta que los desarrolladores no siempre se identifican con la empresa en la cual trabajan, por lo que una parte de esta asociación sólo puede realizarse en forma manual.

Git Data Miner, como herramienta, está muy ligada al desarrollo del núcleo de Linux. Las direcciones están asociadas con las empresas que desarrollan en torno al núcleo de Linux, el cual no necesariamente corresponde al mismo conjunto de empresas que pagan a sus desarrolladores para trabajar en GNOME. Es necesario identificar dichas direcciones y asociarlas con su correspondiente empresa.

Git Data Miner provee un archivo en donde se puede asociar dominio–empresa. Cuando se ejecuta por primera vez, aquellas direcciones cuyo dominio no tiene asociado ninguna empresa aparecen como "*desconocidos*". Antes, es conveniente añadir los dominios de las empresas que se conocen por colaborar con GNOME, partiendo por aquellas que patrocinan a la Fundación GNOME.

Posteriormente, se deben revisar las direcciones del grupo de "*desconocidos*" y asociarlas con una empresa, con especial énfasis en aquellos dominios que se repiten. El proceso de filtrado se puede automatizar, no así el proceso de asociación. Si el vaciado de la base de datos se puede obtener en un archivo separado por comas (CSV), se pueden usar los filtros de un sistema UNIX/LINUX:

```
$ awk -F, '{print $2}' bd.csv | cut -f2 -d@ | sort |
> uniq -c | sort -n
```

En este ejemplo, los datos están en *bd.csv* y la segunda columna o campo contiene los correos electrónicos de los desarrolladores. Por lo que se extrae el dominio, se agrupan y se ordenan de menor a mayor.

Es conveniente partir con este proceso con aquellos repositorios más grandes y que, presumiblemente, tienen una base más amplia de desarrolladores. Aunque es necesario aplicarlo con todos los repositorios que se quieren analizar, en el caso de GNOME, es natural esperar que los desarrolladores tiendan a repetirse entre repositorios.

También es necesario considerar que en el transcurso del tiempo algunas empresas han sido absorbidas por otras, como es el caso de Openedhand, que fue creada por algunos desarrolladores independientes –y colaboradores de GNOME– y que a su vez, contrató a otros desarrolladores del mismo proyecto. Posteriormente OpenedHand fue adquirida por Intel. Situación similar ocurre con Ximian, la cual fue posteriormente adquirida por Novell. Entonces, se pueden encontrar muchos dominios asociados a una misma empresa.

Cuando una empresa absorve a otra, automáticamente aquella contribución realizada al proyecto pasa a sumarse en una sola empresa. No obstante, es interesante estudiar la evolución en el aporte de una empresa tras su adquisición. Por ejemplo, determinar el aporte que hizo Ximian con respecto al aporte que continuó haciendo Novell después de ser adquirida por esta última. Es así que se puede analizar el impacto que tienen las relaciones comerciales en el proyecto GNOME. Cabe recordar que estas empresas, en un principio, contrataron un grupo importante de desarrolladores de GNOME, y al ser absorbidos por otra empresa más grande, con un nicho de mercado más amplio y diverso, es posible plantear como hipótesis que las empresas más grandes pueden reorientar a sus desarrolladores a otros productos; y de cierta forma "*envenenar*" al proyecto, dado que como proyecto se pierden dichos colaboradores.

3.6.3. Refinamiento de identidades y empresas

Si bien el proceso inicial de refinamiento es manual, es necesario aclarar que es también un proceso incremental. El conocimiento que se expresa en las relaciones no se pierde y sólo es necesario añadir los nuevos antecedentes que se conocen.

También hay que asociar el dominio de un desarrollador con un empleador en particular, cundo no se puede determinar a partir del dominio de su correo electrónico. Hay desarrolladores que realizan sus contribuciones utilizando siempre una misma dirección electrónica, la cual no necesariamente corresponde a una empresa.

Entre las situaciones que ameritan su estudio, se encuentran aquellos desarrolladores que adquieren una relación con una una empresa en un período de tiempo definido y bajo la modalidad de servicios externos. Otra situación que es necesario considerar corresponde cuando un desarrollador se mueve de un empleador a otro, por ejemplo, de Novell a Canonical, de Sun a Red Hat, etc.

3.6.4. Cruce de datos

Como se analiza cada repositorio en particular, es importante poder vaciar el contenido en una base de datos común, en el cual se pueda realizar un análisis cruzando la información de los distintos repositorios. Si bien, la base de alias y empresas es la misma, se puede obtener más información del comportamiento de los desarrolladores al estudiar los repositorios como un conjunto.

En el caso de *Git Data Miner*, fue construído con el fin de estudiar el núcleo de Linux, lo que corresponde a un solo repositorio. Por lo tanto, es necesario extenderlo,

o mendiante otro proceso, asociar las etiquetas que se utilizan cuando se libera una nueva versión. Algunos utilizan, por ejemplo, gnome-2-28, otros GNOME_2_18, etc. Si se desea estudiar la evolución de GNOME entre la versión 2.20 y 2.22, es necesario realizar este refinamiento y éste debe ser guiado manualmente, tanto para poder discriminar las etiquetas utilizadas en cada repositorio como para determinar qué paquetes se consideran parte de GNOME en un instante de tiempo (Desktop, Platform, Admin, etc.)

Para cruzar los datos es importante contar con las etiquetas que relacionan versiones e instantes del tiempo, además de la identidad de los desarrolladores y las empresas.

3.6.5. Categorización de las contribuciones

Es importante poder diferenciar el tipo de contribuciones en el tiempo, según la naturaleza del archivo. En el refinamiento de los datos hay que determinar el tipo de archivo, dado que hay muchas contribuciones que corresponden a traducciones, documentación y código; por consiguiente, el tipo de aporte de cada colaborador y/o empresa pueden estar o no enfocados a un determinado tipo. Luego, es necesario determinar quién contribuye qué.

Entre otras limitaciones, *Git Data Miner* no discrimina por aporte, principalmente porque fue concebido para procesar la historia de desarrollo del núcleo de Linux, en donde no hay traducciones y el tipo de archivo esperado es código en C, y en algunas partes en Assembler. A diferencia de los programas de escritorio, en donde la documentación es distinta (orientada al desarrollador o al usuario), se dispone de aplicaciones traducidas a múltiples idiomas y hay variedad en los lenguajes con que se escriben las aplicaciones, por ejemplo, En proyectos como GNOME, se emplean múltiples lenguajes, tales como C, C++, Python, Perl, C#, Scheme y Ruby, por nombrar algunos.

Además, al determinar el tipo de archivo se puede, posteriormente, realizar un análisis más detallado en el contenido de éstos, tales como sus comentarios, definición de licencias y/o complejidad de los programas.

3.6.6. Determinación de ramas de desarrollo

Posteriormente se debe determinar el estudio de cada rama de desarrollo y ajustar el rango de estudio de cada rama. Dependiendo de la complejidad y madurez del programa en estudio, las ramas pueden tener más o menos desarrollo; en algunos

casos será necesario determinar si se mezclaron ramas. Esto abre nuevas interrogantes que, en proyectos grandes, puede ser de interés estudiar. En el estudio se debe considerar y ajustar los errores que pueden presentar los repositorios, particularmente, aquellos que han sido migrados de un SCM a otro.

El análisis de las ramas puede ser un proceso lento. La herramienta CVSAnalY tiene algoritmos para seguir el desarrollo de las ramas, sin embargo, es un proceso que puede tardar muchas horas ó días para preparar los datos de sólo un repositorio, como el del programa *evolution*.

Dado que el desarrollo de la línea principal puede presentar información valiosa y a la complejidad computacional del análisis de las ramas, se descartó en este estudio. No obstante, aquí se plantea como etapa de refinamiento para un trabajo futuro.

Capítulo 4

Procesamiento, resultados e interpretaciones

En el presente capítulo se describe el proceso efectuado en donde se integran las mayores etapas definidas en el capítulo anterior (Preparación de datos y Análisis y refinamiento) y los resultados obtenidos. Para mayor comprensión, la etapa de análisis y refinamiento se ha subdividido en Procesamiento y Análisis.

En la figura 4.1 se muestra esquemáticamente cada etapa.

1. Preparación de los Datos. Se procede a la recuperación de los datos desde el repositorio *git* y se obtiene el registro de acuerdo a los criterios especificados por el investigador. Los criterios están descritos como una tabla en un archivo separado por comas (CSV). El registro (*log*) sirve de entrada en la siguiente etapa.

2. Procesamiento. Utilizando los criterios en conjunto con el registro de cambios, se procede a extraer los datos asociados a cada cambio. Se obtiene información como autores, fechas, archivos modificados, líneas agregadas y modificadas, a su vez, se determina información extra, como es el tipo de archivo involucrado en cada cambio. Los valores obtenidos se vacían como archivos CSV y en una base de datos, para su posterior análisis. Además, se revisan los resultados para determinar si es necesario refinar los datos según las indicaciones de la sección 3.6.

3. Análisis. Se procede a explorar los datos, realizando agrupaciones y el lenguaje *R*[23] para programar el análisis estadístico y generación de gráficos,

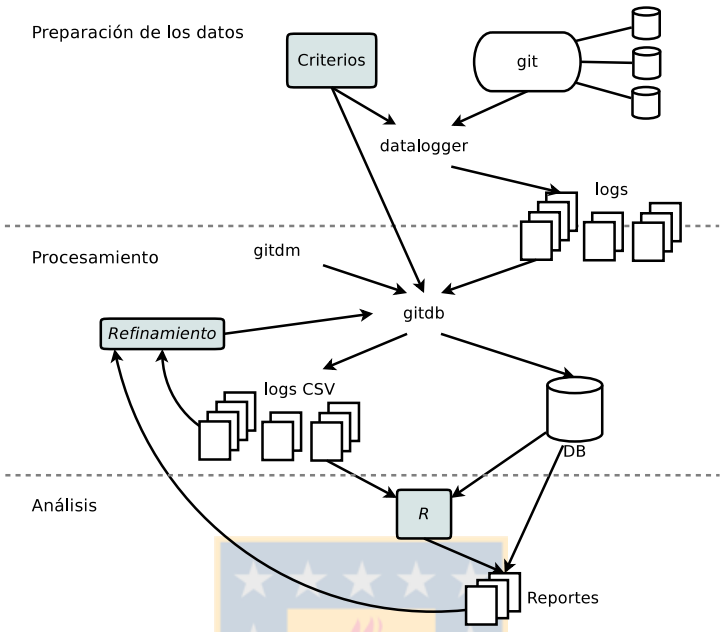

Figura 4.1: Separación de etapas en el procesamiento de los datos

así como la obtención de reportes mediante el uso directo de SQL sobre el motor de base de datos. De acuerdo a la información analizada, es necesario realizar un nuevo refinamiento y volver a iterar a partir de la etapa de procesamiento.

Las etapas de preparación de los datos y procesamiento se realizaron en un equipo del Laboratorio de Biofísica Molecular de la Universidad de Concepción, el cual cuenta con dos procesadores AMD Opteron 2376 Quad–Core, de 2.3Ghz, 8GB de memoria y un disco SATA de 150GB y sistema operativo Linux (kernel 2.6.31). Cabe notar que en el procesamiento se ocuparon sólo 2 núcleos de los 8 disponibles. En dicho equipo se utilizó *git* 1.6.5.2, *Postgresql* 8.4 y *Python* 2.6.4.

Anteriormente, se hicieron pruebas en distintos equipos en donde se pudo determinar que el cuello de botella se producía primero al usar *cvsanaly* en vez de *gitdm*, y posteriormente al usar filtros para procesar la salida de *git log* directamente con *gitdm*. Además, de mejoras en el código fuente, con la separación de las tareas se pudo reducir el tiempo de preparación y procesamiento de más de 144 horas[1] a 13 minutos, y posteriormente a 8 minutos.

[1]La cantidad de horas precisas se desconoce, por cuanto se abortó el proceso pasados 6 días en 2 oportunidades.

El análisis se hizo en un computador portátil con procesador Intel Core 2 (U7600) de 1.2Ghz, 2GB de memoria RAM, con sistema operativo Linux (kernel 2.6.31), y los programas *Postgresql* 8.3.8 y *R* 2.9.2.

4.1. Preparación de los datos

El archivo principal de datos contiene el grupo de repositorios a estudiar, que se encuentran alojados en un mismo servidor. El formato del archivo es simple, como se muestra en la tabla 4.1, puede ser editado en una planilla electrónica y almacenado como un archivo con valores separados por comas (CSV). El archivo puede contener un número arbitrario de filas y columnas respetando que:

		fecha *1*	fecha *2*	...	fecha *n*
módulo	*tarball*	versión *1*	versión *2*	...	versión *n*
módulo *a*	*a*	etiqueta *a-1*	etiqueta *a-2*	...	etiqueta *a-n*
módulo *b*	*b*	etiqueta *b-1*	etiqueta *b-2*	...	etiqueta *b-n*
módulo *c*	*c*		etiqueta *c-2*	...	etiqueta *c-n*
...	...	...	...	...	...
módulo *m*	*m*	etiqueta *m-1*	etiqueta *m-2*	...	etiqueta *m-n*

Tabla 4.1: Formato del archivo que describe las versiones, módulos y etiquetas

1. Las dos primeras columnas son fijas: la primera columna contiene el nombre del repositorio de un módulo (por ejemplo *gconf*) y la segunda el nombre de la versión empaquetada como *tarball*[2], (por ejemplo, *GConf*). Existe esta separación para los casos en que el nombre del repositorio y su correspondiente *tarball* difieren.

2. La primera fila contiene las fechas en que se ha entregado cada versión;

3. La segunda fila, a partir de la tercera columna, contienen las versiones que se desean estudiar.

4. A partir de la tercera fila comienzan los datos de cada módulo y la etiqueta asociada en el repositorio cuando se entregó una versión. En el caso que no se haya etiquetado, se puede llenar con el *hash* que identifica al último *commit* efectuado antes de empaquetar el programa asociado con dicha versión.

[2]En UNIX y sus derivados, se le llama *tarball* a un archivo empaquetado con la utilidad *tar*, el cual puede estar o no comprimido con *gzip* o *bzip2*. Es la forma habitual de distribuir el código fuente de los programas.

		03/03/1999	25/05/2000	02/04/2001	27/06/2002	...	23/09/2009
repo_name	tarball	1.0	1.2	1.4	2.0	...	2.28
gconf	GConf			GCONF_1_0_0	GCONF_1_2_0	...	2.28.0
at-spi	at-spi				gnome-2-0-bp	...	AT_SPI_1_28_0
atk	atk				ATK_1_0_2	...	ATK_1_28_0
gail	gail				GAIL_0_16	...	
glib	glib	GLIB_1_2_2	GLIB_1_2_8	GLIB_1_2_9	GLIB_2_0_4	...	2.22.0
gtk+	gtk+	GTK_1_2_2	GTK_1_2_8	GTK_1_2_9	GTK_2_0_5	...	2.18.0
gtk-doc	gtk-doc				1b101d..2e9c77	...	GTK_DOC_1_11
...	...	...	...	...	...	...	...

Tabla 4.2: Extracto del archivo utilizado para describir las versiones, módulos y etiquetas

Se eligió este formato porque es compacto, fácil de procesar computacionalmente y facilita la edición a las personas. Además, será tan completo como el investigador lo estime conveniente para su estudio, lo cual permite el llenado con pocos datos para familiarizarse con el procedimiento e ir completando a medida que se requiere.

En la tabla 4.2 se muestra un extracto del archivo utilizado en este estudio, el cual es confeccionado manualmente indicando los módulos (repositorios *git*) que se desea estudiar. Se puede apreciar que las etiquetas no siempre siguen un mismo patrón, incluso entre versiones de un mismo módulo, como es el caso de *gconf*, *gtk+* o *glib*. En módulos como *gtk-doc* no existe una etiqueta para la versión 0.9 de dicho producto, y que figura como la entregada para GNOME 2.0[3], por lo que se indica el *hash*[4] del historial de *git* que corresponde a la entrega de *gtk-doc* 0.9.

También es posible apreciar que hay módulos que aparecieron con GNOME 2.0, como es el caso de *at-spi*, *atk*, *gail*[5] y *gtk-doc*. En todos esos casos, se debe dejar en blanco las celdas que corresponden a las versiones para las cuales no existe programa. Así mismo, se puede ver que *gail* no aparece bajo la versión GNOME 2.28. El programa que procesa estos datos es capaz de detectarlo y sólo procesará los datos explícitamente definidos.

Para analizar los cambios entre una versión y otra se analiza el historial entre dos etiquetas contiguas, mientras que para la primera versión de un programa se analiza

[3]Disponible en `http://ftp.gnome.org/pub/GNOME/desktop/2.0/2.0.0/sources/`.

[4]El *hash* completo es una cadena de números en hexadecimal de 40 dígitos. El valor que corresponde a dicha celda es es *1b101d645406a1e86f24a2eb25e46e87b62e9c77*, por razones de espacio se redujo para simbolizarlo.

[5]*at-spi*, *atk* y *gail* son bibliotecas de accesibilidad y que efectivamente se incorporaron en GNOME 2.0.

a partir del inicio del historial del programa hasta la etiqueta de la versión. Por ejemplo, para conocer los cambios que hubo en *glib* entre las versiones de GNOME 1.0 y 1.2, se procesa el historial entre las etiquetas GLIB_1_2_2 y GLIB_1_2_8, lo cual se obtiene un registro de la siguiente forma:

```
$ git log -M -C --numstat GLIB_1_2_2..GLIB_1_2_8
commit 9cf514b19813d2b9ce56d811d1594fbed458e24d
Author: Tim Janik <timj@src.gnome.org>
Date:   Thu May 25 01:47:57 2000 +0000

   grrrr, 1.2.8 is taht

1       1       ChangeLog
1       1       ChangeLog.pre-2-0
1       1       ChangeLog.pre-2-10
1       1       ChangeLog.pre-2-12
1       1       ChangeLog.pre-2-2
1       1       ChangeLog.pre-2-4
1       1       ChangeLog.pre-2-6
1       1       ChangeLog.pre-2-8

commit 9abb741dd99acede9464b4fc51258013002b655a
Author: Tim Janik <timj@gtk.org>
Date:   Thu May 25 01:46:55 2000 +0000

...
```

Así mismo, para conocer los cambios en GNOME 1.0, se procesa el historial desde el inicio de *glib* hasta GLIB_1_2_2. El registro obtenido, debe ser procesado para extraer información, el cual además del identificador, autor, fecha y comentario, también incluye el detalle con el nombre de cada archivo modificado, con el número de líneas añadidas y eliminadas. A partir del nombre del archivo, es posible inferir su tipo y discriminar si se trata de código, documentación para el usuario, documentación para el programador, traducción, imagen, sonido u otro.

A partir de datos como la tabla 4.2, se obtiene el registro de cada repositorio y cada versión. Si hubiera 27 módulos, cada uno con 18 versiones, se obtendrán 486 registros que procesar. Aunque el proceso de llenado de la tabla es manual, se automatizó la obtención y procesamiento del registro para ordenar los datos a estudiar.

4.2. Procesamiento

El procesamiento consiste en utilizar como entrada los archivos que guardan el registro de cada módulo y versión de acuerdo a los criterios definidos para el estudio. Para ello se emplean las utilidades programadas para tales efectos y que se encuentran públicamente disponibles en un repositorio en *gitorious*[6]. En dicho repositorio también se encuentran las versiones modificadas de los programas que fueron utilizados en algún momento del estudio y que posteriormente fueron descartadas. También se encuentra una versión extendida de *gitdm* para permitir concretar este estudio, permitiendo su uso en futuros estudios que involucren varios repositorios y versiones. De la misma forma, se encuentra las utilidades programadas que permiten relacionar los criterios del estudio, los archivos de entrada, poblamiento de la base de datos y facilitar la generación de reportes.

Como se muestra en la figura 4.1 *gitdb* es el programa empleado para procesar los archivos con los registros de entrada, dado los criterios definidos por el investigador y presentar los datos de la forma esperada por *gitdm* y guardar los resultados en formato CSV y cargarlos en la base de datos.

En el procesamiento se encontraron algunos errores, atribuibles al proceso de migración entre un sistema de control de versiones a otros, ya sea desde CVS a Subversion o desde Subversion a *git*. Uno de los errores encontrados fue la incorrecta identificación de algunos desarrolladores. Por ejemplo, mediante el siguiente comando es posible encontrar los errores que ilustran la situación:

```
$ git log --all | grep '^Author: [0-9].*'
Author: 11:03:30  Tim Janik <timj@imendio.com>
Author: 13:28:23  Tim Janik <timj@imendio.com>
...
Author: 05:27:43 2000  Owen Taylor <otaylor@redhat.com>
...
Author: 3 <jrb@redhat.com>
...
```

El resultado esperado es el nombre del desarrollador y su correo electrónico, o en el peor caso, sólo su correo electrónico. Sin embargo, en algunos registros se obtienen como nombres elementos que corresponde a la hora, año y/u otros números. Este error no produce variaciones en la determinación de los archivos involucrados en cada cambio, así como tampoco tiene incidencia en la obtención de las líneas

[6]El repositorio de las herramientas utilizadas se encuentran en http://gitorious.org/mining-tools/.

añadidas y/o eliminadas, sí puede afectar la determinación de colaboradores únicos en un módulo.

Otro error encontrado fue la detección inapropiada de algunas etiquetas. Es una práctica habitual etiquetar un cambio en el momento que se libera una versión, para posterior referencia del instante del tiempo en que se liberó un programa y los cambios involucrados. Por la forma de trabajar Subversion, cuando se crea una etiqueta, se realiza una copia con el estado del repositorio. Ello explica que el tamaño de los repositorios de Subversion sea mucho más grande que con CVS o *git* para un mismo proyecto, tal como fue señalado en la sección 3.2. Luego, en el proceso de conversión de Subversion a git, dichas copias fueron consideradas como código nuevo, en circunstancias que se trata del mismo código y en donde sólo se debió añadir una etiqueta.

Cuando el repositorio de un proyecto es grande, la copia de todo su código y considerarlo como nuevo provoca cambios en los resultados. El error con el cual se detectó este problema involucraba más de 2 millones de líneas de código. Dado que el etiquetado en los cambios están acompañados de un patrón común en los comentarios, la solución a este problema pasó por añadir una condición especial en el procesamiento para aquellos cambios que responden a esta condición.

Los errores en el proceso de migración puede ser producto de la poca experiencia o conocimiento con las herramientas. A medida que se profundiza el conocimiento de las nuevas herramientas es posible darse cuenta que la migración se pudo haber realizado por caminos alternativos o bien detectar fallos que en el momento de la migración pasaron desapercibidos o darle la importancia necesaria a ciertos patrones de comportamiento.

También es necesario considerar que la migración se realiza con las herramientas disponibles en un instante y su nivel de madurez. Con esta retroalimentación es que los proyectos FOSS mejoran. Finalmente, y aunque se puedan detectar ciertos fallos, la presión por utilizar una herramienta que ofrece mejores prestaciones para los desarrolladores, puede influir en aceptar estos errores en la historia de los proyectos. No en vano, es posible corregirlos cuando se escrutan, tal cual ha ocurrido en este estudio.

4.3. Análisis

La figura 4.2 muestra la frecuencia y cantidad de cambios que se aplicado semanalmente en *GNOME Plaftorm* desde que se inició el proyecto el 15 de agosto de 1997 hasta el 23 de septiembre de 2009, fecha en que se entregó la versión 2.28 (ver

tabla 2.1 para el detalle de las fechas). A diferencia del estudio de Kroah-Hartman, Corbet y McPherson sobre el núcleo de Linux[11], para obtener números significativos es necesario agruparlos por semana, en circunstancias que la cantidad de aportes al núcleo de Linux se puede dividir en días e incluso horas. Sin embargo, hay que tener en cuenta que *GNOME Platform* es un subconjunto de GNOME, y por consiguiente, la perspectiva es distinta a la del núcleo de Linux.

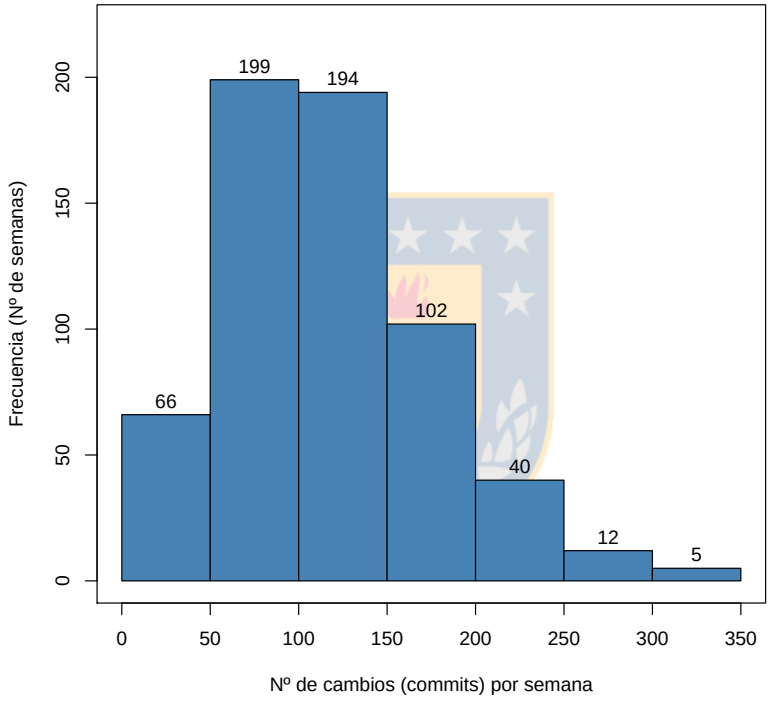

Figura 4.2: Histograma de la frecuencia cambios semanales desde el inicio de GNOME hasta la versión 2.28

En el histograma se aprecia que en 393 semanas[7]se han aplicado entre 50 y 150 cambios (*commits*), en donde se agrupa la mayor cantidad de contribuciones. Dado la forma del histograma, la mejor medida de tendencia central es la mediana, cuyo valor da una esperanza de 110 cambios por semana para *GNOME Platform*.

Además, el histograma muestra que, excepcionalmente, han habido 5 semanas que

[7]Obtenidos de la suma de 199 y 194.

se aplicado entre 300 y 350 cambios, es decir, semanas con una actividad de desarrollo tres veces mayor que la esperanza del proyecto. Esto también se puede apreciar en la figura 4.3, en el cual se encuentran graficados en detalle los cambios por semana.

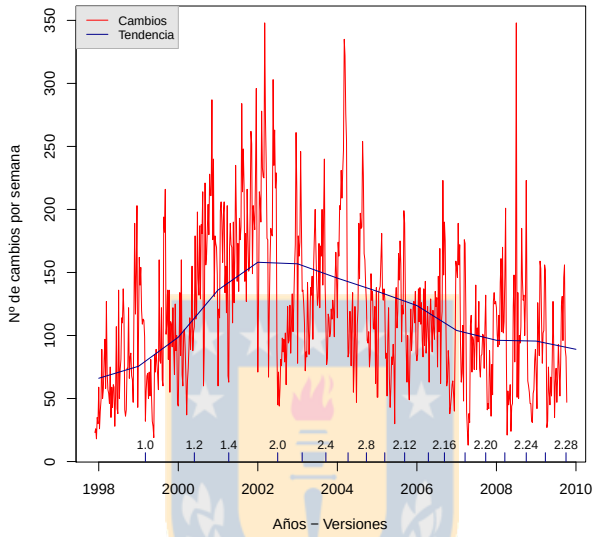

Figura 4.3: Número de cambios por semana a través del tiempo

En la figura 4.3 se puede ver la mayor actividad general ocurrió antes de liberar las versiones 2.0, 2.6 y 2.24. De manera similar, se puede apreciar un aumento de actividad en las fechas próximas a cada versión, el cual también se ve claro para todas las versiones, exceptuando las versiones 1.2 y 2.20 en donde el fenómeno se encuentra presente, pero no es tan evidente.

También es posible apreciar un aumento de actividad entre las versiones 1.0 y 1.2, antes de llegar al año 2000. Ello se explica por la entrega de la versión 1.0.53, conocida como *October GNOME* y que fue considerada la primera versión verdaderamente estable. La fecha de entrega de la versión 1.0.53 fue el 12 de octubre de 1999.

La línea azul, muestra la tendencia calculada con el algoritmo *super smooth* de Friedman[8] para los cambios por semana. Con dicha línea se ve una fuerte alza desde el inicio del proyecto hasta la versión 2.0. Posteriormente, se observa un descenso continuado, con una mayor pendiente entre las versiones 2.12 y 2.18.

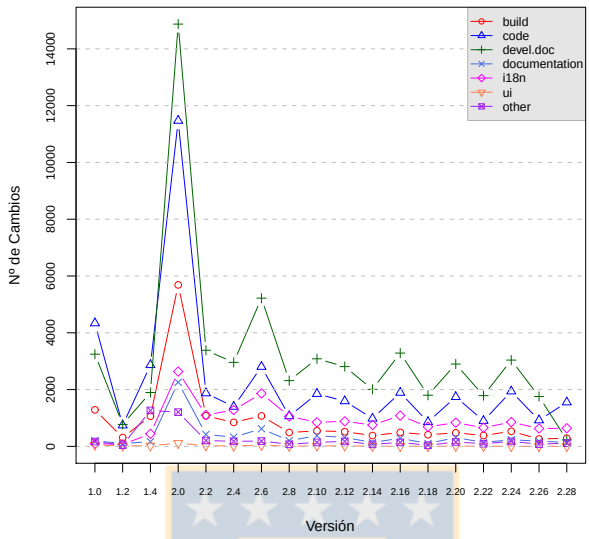

Figura 4.4: Número de cambios (*commits*) por tipo de contribución en las versiones de *GNOME Platform*

Al graficar el número de cambios por tipo de contribución y sin separar por semanas, como se muestra en la figura 4.4, se observa una forma de la curva que sugiere lo mismo que la predicción del algoritmo *super smooth*. Los datos utilizados para esta gráfica se encuentran en la tabla 4.4.

En la tabla 4.3 se describe la clasificación utilizada para agrupar los tipos de archivo encontrados en cada repositorio.

El incremento en el desarrollo entre las versiones 1.2 y 2.0 se puede explicar separadamente:

1. Entre 1.2 y 1.4 aparecieron nuevos actores en el desarrollo de GNOME. En 1999 se formaron las empresas Eazel y Hexlicode, quienes proveyeron a GNOME 1.4 del administrador de archivos Nautilus y del cliente de correo electrónico Evolution, respectivamente. Ambos programas trajeron consigo un mayor desarrollo de las bibliotecas subyacentes, así como mayor dedicación de programadores, ya que ambas empresas contrataron programadores de la comunidad para trabajar tiempo completo en GNOME[8].

[8]Ver "*The Story of the GNOME project*" en http://primates.ximian.com/∼miguel/gnome-

Tipo	Detalle
build	Construcción del software
code	Código fuente
devel-doc	Documentación de desarrollo
documentation	Documentación de usuario
i18n	Traducciones
ui	Interfaz de usuario
other	Imágenes, sonidos y desconocidos

Tabla 4.3: Clasificación de las contribuciones

2. Entre 1.4 y 2.0 se explica fundamentalmente por la reescritura del código, coincidiendo con uno de los predicados de Brooks[4] que indica que todo proyecto debe estar preparado para comenzar de nuevo. El cambio en el primer dígito no es al azar, no sólo indica que no hay compatibilidad hacia atrás sino que también marca una nueva era en la concepción del escritorio[9].

Adicionalmente, se debe considerar que en la versión 2.0, se incorporaron nuevas características, como la de accesibilidad para discapacitados. Lo cual comenzó a gestarse con el anunció de Sun Microsystems de crear el *Accessibility Technologies Lab*[10] para trabajar en conjunto con la recién creada Fundación GNOME[11].

Es necesario destacar que el 15 de mayo de 2001, la empresa Eazel dejó de operar[12], tras liberar Nautilus 1.0.3. Es importante, porque con Eazel ocurrieron varios hechos que muestran la fortaleza de un proyecto FOSS frente a este tipo de adversidades[13]: la licencia GPL de Nautilus permitió que el proyecto continuara, la calidad del código y la integración de Eazel con la comunidad de desarrolladores de GNOME.

history.html.

[9] La discusión en donde se define conceptualmente lo que se desea de GNOME en el futuro comienza en febrero de 2001 en http://mail.gnome.org/archives/gnome-hackers/2001-February/msg00235.html, con especial énfasis en los mensajes de Havoc Pennington en http://mail.gnome.org/archives/gnome-hackers/2001-February/msg00241.html y http://mail.gnome.org/archives/gnome-hackers/2001-February/msg00280.html.

[10] http://www.highbeam.com/doc/1G1-65345095.html

[11] http://mail.gnome.org/archives/foundation-list/2000-July/msg00176.html y http://mail.gnome.org/archives/foundation-list/2000-August/msg00125.html.

[12] http://mail.gnome.org/archives/gnome-hackers/2001-May/msg00203.html.

[13] Ver la discusión iniciada en http://mail.gnome.org/archives/gnome-hackers/2001-May/msg00203.html y continuada en http://mail.gnome.org/archives/gnome-hackers/2001-May/msg00215.html.

Ver.	build	code	dev-doc	doc	i18n	ui	other	Total
1.0	1.286	4.338	3.247	196	95	53	170	9.385
1.2	312	742	773	102	70	17	43	2.059
1.4	1.062	2.866	1.896	165	442	21	1.265	7.717
2.0	5.687	11.471	14.872	2.257	2.635	113	1.205	38.240
2.2	1.074	1.873	3.384	421	1.104	25	206	8.087
2.4	845	1.399	2.958	325	1.281	15	178	7.001
2.6	1.072	2.804	5.220	622	1.864	39	187	11.808
2.8	488	1.065	2.316	202	1.055	8	74	5.208
2.10	550	1.850	3.084	375	849	18	141	6.867
2.12	517	1.593	2.810	314	882	19	177	6.312
2.14	381	982	2.009	138	752	5	73	4.340
2.16	491	1.891	3.286	283	1.083	7	141	7.182
2.18	410	859	1.801	102	690	2	39	3.903
2.20	476	1.741	2.899	309	835	10	156	6.426
2.22	384	901	1.787	141	658	3	94	3.968
2.24	530	1.938	3.039	234	855	4	174	6.774
2.26	260	920	1.756	181	628	0	79	3.824
2.28	286	1.552	230	144	639	2	113	2.966
Total	16.111	40.785	57.367	6.511	16.417	361	4.515	142.067

Tabla 4.4: Número de cambios (*commits*) realizados por tipos de archivo en cada versión de *GNOME Platform*

Al disponer de un gráfico separado por tipo de contribución, se observa que la documentación de desarrollo lidera los cambios en la versión 2.0 y siguientes, con altos y bajos, pero siempre describiendo la misma gráfica que los cambios en el código, con una fuerte diferencia a partir de la versión 2.26, en donde decrece fuertemente.

En contraste a la cantidad de cambios en la documentación de desarrollo, se observa, tanto en la figura 4.4 como en la tabla 4.4, que la documentación al usuario es casi 10 veces menor, y que los cambios en la interfaz de usuario es cercana a cero; ambos fenómenos se explican por la naturaleza de *GNOME Platform*, lo cual se explica por tratarse de un conjunto de bibliotecas cuyo fin es proveer una API a los desarrolladores de programas.

4.3.1. Colaboradores del proyecto

Otro tipo de análisis para conocer la evolución de un proyecto es la cantidad de colaboradores que aportan con su esfuerzo al desarrollo del proyecto a través del tiempo. En la figura 4.5 se muestra el número de colaboradores que contribuye en cada tipo de contribución.

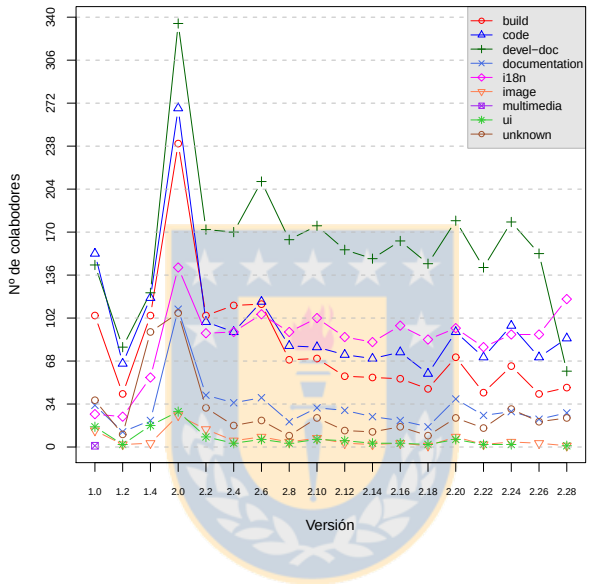

Figura 4.5: Desarrolladores de *GNOME Platform* por tipo de contribución a través de las versiones

Los valores mostrados en la gráfica no deben ser sumados para determinar la cantidad total de colaboradores en GNOME a través del tiempo, dado que hay desarrolladores que colaboran en más de un área, y por consiguiente, al sumarlos se podrían contabilizar erróneamente. Para ello es necesario analizarlos por separado, como se muestra en la figura 4.6.

Comparando las figuras 4.5 y 4.6, se puede observar que en la versión 2.0 hay bastante colaboradores que contribuyen en más de un área, dado que la suma de aportes supera los 600 y la cantidad real es de 353. Además, la cantidad de programadores se mantiene entre 68 y 102, y el número total de colaboradores se mantiene entre los 148 y 187, con variaciones en las versiones 1.2, 2.0, 2.6 y 2.28.

Resulta interesante contrastar el número de colaboradores en la versión 1.2 con respecto a la tendencia en los cambios realizados por semana de la figura 4.3. A

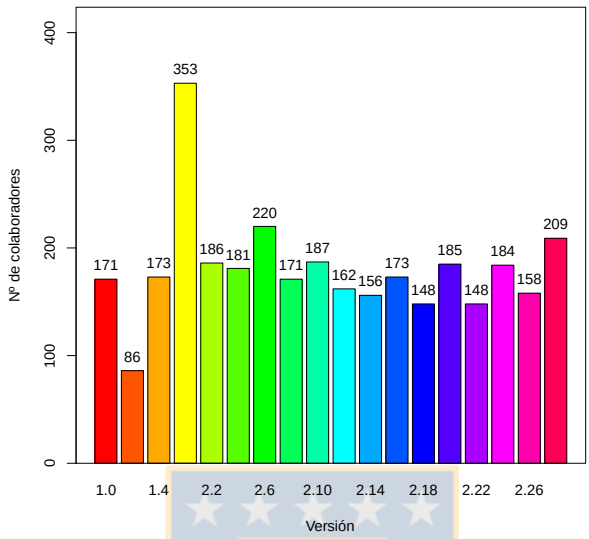

Figura 4.6: Número de desarrolladores de *GNOME Platform* a través de las versiones

pesar que la cantidad de colaboradores entre las versión 1.0 y 1.2 bajó casi a la mitad, la actividad se continuó aumentando. Ello se explica porque el nivel de contribuciones no es lineal al número de colaboradores y en donde un grupo de desarrolladores realiza gran parte de las contribuciones. En la medida que exista un grupo de tales desarrolladores contribuyendo en forma activa, el proyecto seguirá evolucionando. Aunque sí un aumento significativo en desarrolladores produce un impacto positivo en las contribuciones, como ocurrió para la versión 2.0.

4.3.2. Evolución del código fuente

En la figura 3.2 del capítulo 3 se mostraron las interrelaciones de las bibliotecas que componen *GNOME Platform*. En dicho capítulo se indicó que a partir de la versión 2.8 comenzó el esfuerzo para consolidar funcionalidades en un conjunto menor de bibliotecas, recomendando utilizar aquellas que continuarían su desarrollo y desalentando el uso de aquellas destinadas a desaparecer en el futuro.

Entonces se pueden separar las bibliotecas entre aquellas recomendadas y aquellas que no lo son. Debido a la notable diferencia en el tamaño de éstas, se separaron a

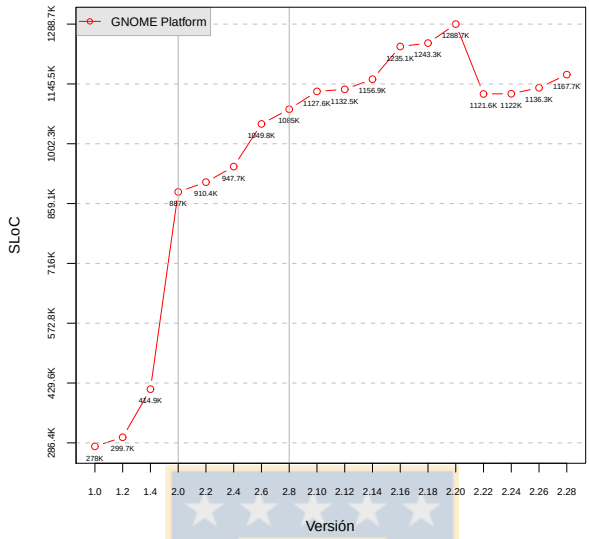

Figura 4.7: Evolución del tamaño del código fuente de *GNOME Platform*

su vez en subgrupos por su tamaño, para facilitar la visualización en los gráficos y entender mejor su evolución.

En el aspecto general, se observa en la figura 4.7 el crecimiento global de *GNOME Platform.* Una vez más, queda en evidencia la importancia de la versión 2.0 en la evolución del objeto de estudio. En la figura se muestra una línea vertical para resaltar las versiones 2.0 y 2.8.

De acuerdo a la figura 4.7 y a la tabla 4.5, entre las versiones 1.4 y 2.0 el tamaño se duplicó, vale decir que, a pesar de reescribir las bibliotecas se añadió bastante código nuevo, desde entonces, se aprecian períodos con más cambios seguidos por períodos de asentamiento. Por ejemplo, entre 2.0 y 2.6 el ritmo de crecimiento no fue tan acelerado, cambiando entre 2.6 y 2.8, algo similar volvió a ocurrir hasta la versión 2.18.

En 2.22 se observa una caída en el tamaño del código fuente de 167.152 líneas de código fuente, lo cual representa un 60 % del tamaño de 1.0. Este fenómeno se explica porque el desarrollo de las bibliotecas *libxml2* y *libxslt* llega sólo hasta 2.20, momento en el cual dejaron de ser consideradas dentro de *GNOME Platform*. Las funcionalidades de *libxml2* y *libxslt* son mucho más amplias, y hoy en día se considera parte de la funcionalidad básica ofrecida por todos los sistemas a los

Versión	Líneas de código
1.0	277.971
1.2	299.728
1.4	414.887
2.0	887.043
2.2	910.414
2.4	947.717
2.6	1.049.823
2.8	1.084.971
2.10	1.127.635
2.12	1.132.533
2.14	1.156.899
2.16	1.235.134
2.18	1.243.327
2.20	1.288.721
2.22	1.121.569
2.24	1.122.040
2.26	1.136.303
2.28	1.167.721

Tabla 4.5: Líneas de código de *GNOME Platform*

cuales está orientado a GNOME. Este hecho queda en evidencia al observar la figura 4.9, más adelante.

En las figuras 4.9 y 4.10 se aprecia la separación de las líneas de código por módulos. Se ha excluido de las gráficas el módulo *gnome-mime-data* el cual es demasiado pequeño y altera las escalas de las gráficas sin proveer mayor información.

En la figura 4.10 se observa que el módulo *gail*[14] se detiene en la versión 2.22, esto se debe a que fue integrado dentro de *gtk+*[15], lo cual también se puede observar en el incremento entre 2.22 y 2.24 de la figura 4.9. Para entender la integración es necesario comprender el contexto en que se encontraba *gail* en *GNOME Platform*. Como se explicó previamente, en GNOME 2.0 se añadió el soporte para discapacitados, mediante las bibliotecas *at-spi*[16], *atk* y *gail*. *at-spi* provee una interfaz para comunicarse con herramientas de asistencia, tales como lector de pantalla o

[14] *gail* es el acrónimo de *GNOME Accessbility Implementation Library*.

[15] https://bugzilla.gnome.org/show_bug.cgi?id=169488.

[16] *at-spi* es el acrónimo de *Assistive Technology Service Provider Interface*. Al escribirlo en minúsculas es para referirse a la biblioteca, mientras en mayúsculas al acrónimo en sí.

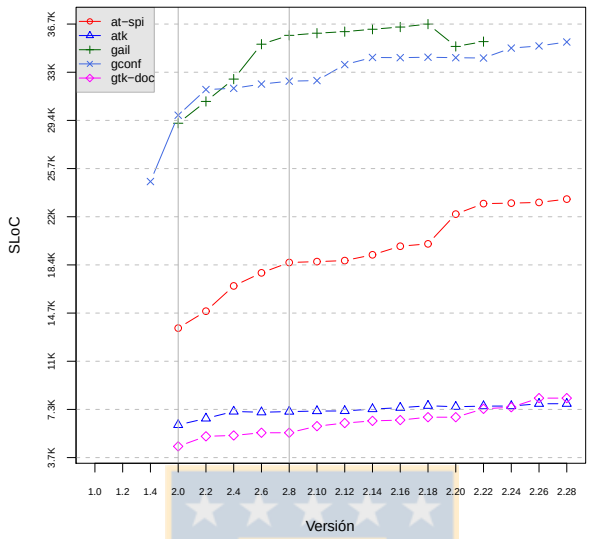

Figura 4.8: Evolución del tamaño de las bibliotecas más pequeñas y activas

un magnificador; esto se hace mediante el uso de CORBA (*ORBit2*) con interfaces definidas en IDL (*libIDL*). *atk* [17] describe un conjunto de interfaces que soportan AT-SPI a nivel de interfaz gráfica de usuario y es independiente de la implementación, esto es, se puede utilizar como base para construir widgets en GTK+, Motif o Qt; y *gail* proveía la implementación de dichos *widgets* en GTK+. En el desarrollo de la versión 2.0 se mantuvo como biblioteca independiente porque había seguridad que pudiera estar implementada correctamente para esa fecha, y a partir de 2.6 se consideró integrarla[18], proceso que finalmente se cumplió en la versión 2.24.

En la figura 4.11 se presentan las bibliotecas más grandes de *GNOME Platform*, destacándose *glib* y *gtk+*. *glib* es la biblioteca que provee funcionalidades generales, independiente de la interfaz gráfica así como un sistema de objetos en C, mientras que *gtk+* provee los *widgets* para construir interfaces gráficas de usuario. Se puede observar que a partir de la versión 2.0, el tamaño del código fuente de *gtk+* tiene períodos contiguos de estabilidad seguidos por uno de mayor incremento, esto se debe a que el ciclo de desarrollo de *gtk+* es de 9 meses[19], a diferencia del ciclo de desarrollo de 6 meses en que se entrega una nueva versión de GNOME,

[17] *atk* es el acrónimo de *Accesibility Toolkit.*

[18] https://bugzilla.gnome.org/show_bug.cgi?id=169488.

[19] http://www.gtk.org/plan/2.10/.

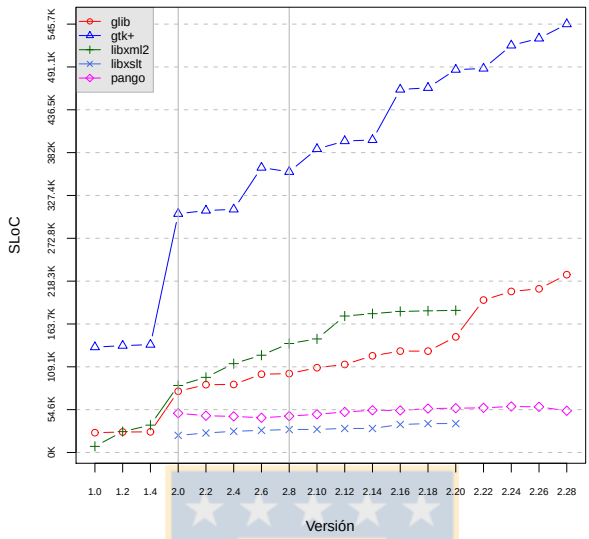

Figura 4.9: Evolución del tamaño de las bibliotecas más grandes y activas

y por consiguiente de *GNOME Platform*. La tercera biblioteca de mayor tamaño y activa es *pango*, la cual provee todo el manejo de tipografías y procesamiento de texto en un ambiente gráfico, con soporte UTF-8, el cual fue incorporado en la versión 2.0 y cuyo crecimiento es lento pero sostenido, lo cual se explica por la funcionalidad delimitada que tiene y que fue incorporada en su versión inicial.

En la figuras 4.10 y 4.11 se encuentra graficada la evolución de las bibliotecas declaradas como no recomendadas o desaconsejadas, las cuales se encuentra separadas para mejorar la visualización. En la figura4.10 se observa que a partir de la versión 2.0, el tamaño tiene poca variación, lo que se acentúa luego de la versión 2.8. La excepción es *libgnome*, la cual tiene un fuerte descenso entre la versión 2.2 y 2.4. Cabe recordar que a partir de la versión 2.8 se inicia la separación entre *GNOME Platform* y *GNOME Desktop*; y, además, en la versión 2.14 se definió el proyecto Ridley para consolidar las bibliotecas, y por consiguiente, se desaconseja utilizar las bibliotecas que debieran ser integradas en otras bibliotecas mayores.

En la versión 2.8 se desaconsejó el uso de *libgnomeprint* y *libgnomeprintui*, cuya funcionalidad fue integrada a *gtk+* en la versión 2.12, lo cual se puede ver en la figura 4.11, en donde el último desarrollo se realizó hasta la versión 2.10.

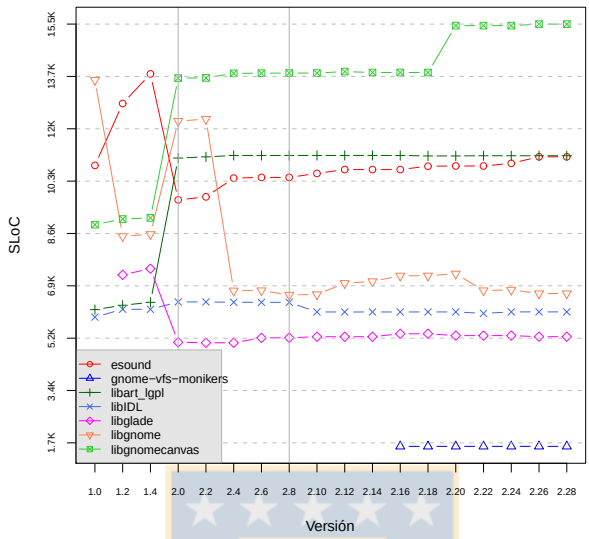

Figura 4.10: Evolución del tamaño de las bibliotecas no recomendadas y pequeñas

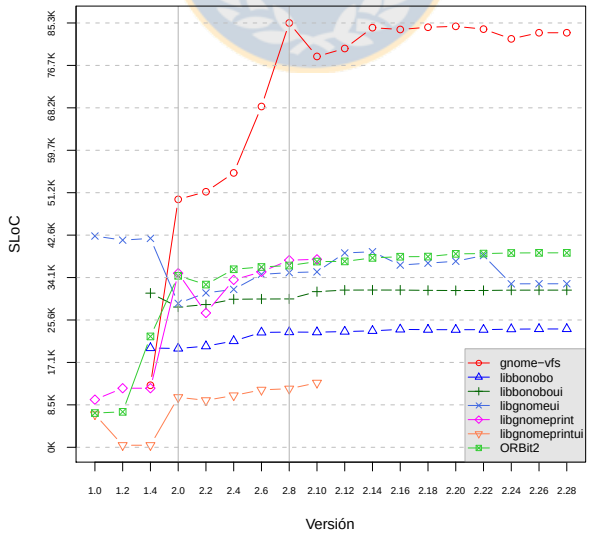

Figura 4.11: Evolución del tamaño de las bibliotecas no recomendadas y grandes

A partir el proyecto Ridley, se desaconsejó el uso de *ORBit2*, *audiofile*, *esound*, *libIDL*, *libart_lgpl*, *libbonobo*, *libbonobui*, *libgnome*, *libgnomeui* y *libgnomecanvas*. Algunas funcionalidades, como *libart_lgpl* fueron ofrecidas por bibliotecas externas mientras que otras fueron descartadas o bien reimplementadas en *gtk+* o *glib*.

A partir de la versión 2.28 se desaconsejó el uso de *libglade* y *gnome-vfs*. En ambos casos fueron reemplazados por las implementaciones de *GtkBuilder* y *GIO*, disponibles en *gtk+* y *glib*, respectivamente. En el caso de *gnome-vfs*, lo que se hizo fue llevar la implementación de un sistema de archivos virtual a un nivel más básico de las bibliotecas, sin depender siquiera de bibliotecas de interfaz gráfica, dado que la funcionalidad lo amerita; es así como se eliminan más de 85.000 líneas de código y se reimplementan en *glib*.

Mención especial merece el módulo *gnome-vfs-monikers*, el cual fue introducido en la versión 2.16 y su uso se desaconsejó en la versión 2.18.

Cuando se hace la separación entre *GNOME Platform* y otros productos que componen GNOME, lo que se busca es separar aquellas funcionalidades que deben ser más estables a largo plazo, en particular la API. Se puede observar en las cuatro figuras (4.8, 4.9, 4.10 y4.11) que, salvo *gnome-vfs-monikers*, no se introdujo ningún módulo nuevo, sino que ha ocurrido todo lo contrario. A su vez, al desarrollo de las bibliotecas se ha detenido y sólo se corrigen fallos en el software, se entregan nuevas versiones para mantener compatibilidad hacia atrás a nivel de ABI y API, con ciertas excepciones como *libgnomeprint, libgnomeprintui* y *gail* que ya no se entregan como módulos individuales.

Parnas[20] sugiere, entre las medidas geriátricas para tratar la edad del software, la amputación de una sección de código que no vale la pena mantener o salvar, ya sea por su obsolescencia, dificultad de mantener o bien por la frecuencia requerida de mantención.

Este proceso se observa en el desarrollo de distintas versiones de GNOME, en donde se desaprueba el uso futuro de ciertas bibliotecas con miras a su completa remoción del proyecto. Esta política se utiliza para bibliotecas completas, con lo cual se disminuyen las dependencias a trozos de código que ya no es atractivo de mantener o que nuevas bibliotecas las superan en funcionalidades y mantenibilidad.

Sin embargo, existen partes del código que si bien se desaprueban, aún continúan viviendo dentro del código fuente. Si bien existen parámetros de compilación[20], aún molesta su presencia para revisar código activo. Dado que existe un compromiso de compatibilidad hacia atrás a nivel de API/ABI en las versiones de una

[20]Por ejemplo, #define GTK_DISABLE_DEPRECATED

misma serie, no es posible eliminarlas tan fácilmente. Es por ello que se prepara una nueva serie de versiones en las cuales se realice limpieza del código que se arrastra solamente por compatibilidad hacia atrás.

Un cambio mayor de versión, por ejemplo de 2.30 a 3.0, permitiría amputar –en términos de Parnas– aquel código que se arrastra por compatibilidad hacia atrás.

Otra de las recomendaciones de Parnas[20] que se cumple, esta vez en la planificación del proyecto, es la disposición de un plan de retiro del software. Aunque en estricto rigor no se trata del fin de un proyecto, sino la garantía de mantención definida a priori de las versiones consideradas estables. En el ciclo de desarrollo de 6 meses, la versión estable cuenta con 3 versiones de correcciones de fallos; luego se libera una nueva versión del escritorio, y la versión estable anterior deja de recibir mantención oficial; repitiéndose el ciclo.

4.3.3. Distribución de lenguajes de programación

En la tabla 4.6 se puede ver que el principal lenguaje de programación de *GNOME Platform* es C, aumentando su cuota de participación a medida que avanzan las versiones. Allí se puede comparar el crecimiento en líneas de código entre las versiones 1.0, 2.0 y 2.28; así como la proporción que ocupa cada lenguaje en todos los módulos que componen *GNOME Platform*.

	Versiones					
	1.0		2.0		2.28	
Lenguaje	SLoC	%	SLoC	%	SLoC	%
Ansi C	260.748	93.80 %	857.924	96.72 %	1.144.416	97.97 %
Perl	1.020	0.37 %	7.021	0.79 %	10.177	0.87 %
Python			5.459	0.62 %	4.405	0.38 %
Shell	13.712	4.93 %	11.609	1.31 %	4.087	0.35 %
Yacc	1.680	0.60 %	2.340	0.26 %	1.948	0.17 %
C preprocessor			1.007	0.11 %	1.720	0.15 %
Lex	292	0.11 %	796	0.09 %	564	0.05 %
Assembler			445	0.05 %	457	0.04 %
Lisp	61	0.02 %	199	0.02 %	137	0.01 %
C-Shell	86	0.03 %	117	0.01 %	117	0.01 %
Awk	372	0.13 %	105	0.01 %	103	0.01 %
Php			21	0.00 %	21	0.00 %

Tabla 4.6: Distribución por lenguajes de programación en *GNOME Platform*

La participación de otros lenguajes distintos de C, se da por el hecho de contar con herramientas adicionales, como para la construcción de documentación, cuyo código permanece relativamente estático a través del tiempo. En cambio, las bibliotecas con mayor desarrollo son *glib* y *gtk+*, las cuales se encuentran escritas en C y debido al ritmo de crecimiento a través del tiempo, se explica como aumenta la participación de C como lenguaje en la *GNOME Platform*.

Debido a que la parte del código se encuentra escrito en C, y a que existen implementaciones de la métricas de complejidad del software de McCabe[17] y Halstead[17] para estudiar precisamente este lenguaje, lo cual se puede realizar en un trabajo futuro. Sin embargo, Herraiz[14, 12, 13] indica que dichas métricas no son necesariamente mejores que las líneas de código fuente (SLoC).

Capítulo 5

Conclusiones

Se estudió la evolución de la plataforma de desarrollo de GNOME (*GNOME Platform*). Para estudiar otros módulos, paquetes o conjuntos de éstos, ya está disponible la infraestructura disponible para llevarlo a cabo. No obstante, es trabajo del investigador definir los paquetes o programas que conforman un módulo, así como las marcas dentro del repositorio que identifican cada versión. Se provee de una herramienta semi–automática para facilitar su búsqueda, pero dado que el marcado de versiones es un proceso humano, es necesario revisar la correctitud y existencia de las marcas.

Una de las promesas elaboradas por Bird y otros[3] que indica que es posible convertir los repositorios desde cualquier a SCM a *git,* manteniendo el historial de sus ramas, mezclas y etiquetas, en forma intacta, no siempre se cumple. En aquellos proyectos en que se realizó la migración de un SCM a otro, sin el debido cuidado, puede provocar daños en el registro histórico, como es el caso de la migración del proyecto GNOME desde Subversion a *git*. No obstante, es posible adoptar medidas que permiten corregir las distorsiones provocadas. Además, si se dispone de la historia antigua, existe la posibilidad de comparar las situaciones en donde se ha producido un error, pero la historia del proyecto en el nuevo repositorio permanecerá sucia a menos que se intervenga directamente.

La simplificación y generalización del proceso es el aporte del esta tesis. La simplificación, por cuanto se puede indicar el lugar en donde estarán los repositorios, así como llenar una tabla en donde se indiquen el nombre de los programas (repositorios) y las marcas o *changesets* que identifican a una versión, así como las versiones que se desean estudiar. El formato de almacenamiento es simple, el cual puede ser editado con una planilla electrónica o con un editor de textos. Y a partir de ello, se puede comenzar a analizar los datos. Respecto a la generalización,

el procesamiento presentado es independiente del proyecto al cual se aplique, en donde se requiere que los repositorios a estudiar se manejen con *git* como SCM.

Los resultados obtenidos pueden ser mejorados aumentando la precisión en la determinación de identidades. El código actual tiene la limitación que determina las identidades al procesar un archivo o registro, comenzando el proceso desde cero en el proceso siguiente.

Al finalizar este estudio, se puede concluir que los módulos estudiados en GNOME evolucionan de forma heterogénea, aún tratándose de aquellos que en donde existe un compromiso de estabilidad a nivel de API/ABI. La longevidad varía de acuerdo a la madurez alcanzada, en algunos casos los módulos desaparecen (como *libbonobo* y *libbonoboui*) mientras que otros son integrados a una biblioteca mayor (como *gail* en *gtk+*). Por otro lado, el nivel de mantención del código varía entre módulos, en algunos casos (como *libart_lgpl*) luego de alcanzar un alto nivel de desarrollo para la versión 2.0 fue abandonado sin presentar evolución hasta la fecha, en donde se mantiene sólo para garantizar compatibilidad hacia atrás. Módulos como *gtk+* y *glib* se muestran con bastante actividad, aún con variaciones en el número general de colaboradores.

Este proyecto se puede extender para estudiar más módulos de GNOME o de otros proyectos, así como aplicar nuevas métricas al código, llamar otros programas externos y/o realizar otro tipo de análisis con los datos ya recolectados.

Se plantea como trabajo futuro el estudio de los cambios que produce en un proyecto los cambios en el equipo de desarrollo, así como la manera en que afecta las prácticas de desarrollo la incorporación de nuevos colaboradores. A su vez, estudiar como afecta el cambio de un sistema de control de versiones en la productividad de un proyecto de software.

Dado que se estudia la evolución de un software, también es posible estudiar los comentarios que se escriben en el código fuente y determinar correlaciones entre el momento en que se escribe el código y el momento en que se realiza la documentación y por quién es realizada. Si además se considera un proyecto conformado por varios subproyectos, es posible estudiar si la documentación o carencia de ella, afecta en el uso de la API por otros programas.

Se puede extender el estudio para determinar el impacto de programas para incentivar la incorporación de nuevos colaboradores, como *Google Summer of Code*, en GNOME, en particular la incidencia en el código fuente y el nivel retención que presenta el proyecto. Esto permitiría aplicar medidas correctivas en el proyecto, así como justificar el uso correcto de los recursos. Si se aplica a un grupo amplio de proyectos, es posible enfocar los esfuerzos por parte de los patrocinadores.

Bibliografía

[1] B. Appleton. Software Configuration Management. http://www.cmcrossroads.com/cgi-bin/cmwiki/view/CM/SoftwareConfigurationManagement, June 2009. 2.3.1

[2] M. Bar and K. Fogel. *Open source development with CVS*. Paraglyph Press, 2003. 2.4.2.1

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. Devanbu. The promises and perils of mining git. In *MSR '09: Proceedings of the Sixth Working Conference on Mining Software Repositories*. IEEE Computer Society, 2009. 3.4, 3.4, 3.5.3, 5

[4] F. P. Brooks Jr. The Mythical Man-Month. *Datamation*, pages 44–52, 1974. 2

[5] S. Chacon. *Pro Git*. Apress, Inc., New York, USA, 2009. 3.2, 3.4

[6] M. de Icaza, E. Lee, F. Mena, and T. Tromey. The GNOME Desktop Environment. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 38–38, Berkeley, CA, USA, 1998. USENIX Association. 2.1, 2.1.1, 2.2, 2.4.1

[7] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005. 2.3.5, 2.4.2

[8] J. H. Friedman. A variable span scatterplot smoother. Technical Report 5, Laboratory for ComputationalStatistics. Stanford University, 1984. 4.3

[9] D. M. Germán. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance*, 16(6):367–384, 2004. 2.1

[10] J. M. González-Barahona and G. Robles. Free software engineering: A field to explore. *Upgrade*, 4(4):49–54, 2003. 1.2

[11] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing,and Who is Sponsoring It: An August 2009 Update. Technical report, The Linux Foundation, Aug. 2009. 3.5.4, 4.3

[12] I. Herraiz. *A statistical examination of the properties and evolution of libre software*. PhD thesis, Universidad Rey Juan Carlos, 2008. http://purl.org/net/who/iht/phd. 1.2, 2.2, 4.3.3

[13] I. Herraiz. A statistical examination of the evolution and properties of libre software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, pages 439–442. IEEE Computer Society, 2009. 4.3.3

[14] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, pages 21–28. IEEE Computer Society, 2007. 4.3.3

[15] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Determinism and evolution. In *Msr '08: proceedings of the 2008 international working conference on mining software repositories*, pages 1–10, New York, NY, USA, 2008. ACM. 1.1

[16] Ionannis Antoniades, Jesus M. González-Barahona, Santiago Dueñas, Gregorio Robles, Stefan Koch, Ksenia Fraczek, and Anis Hadzisalihovic. Design of retrieval system. Technical report, FLOSSMetrics Consortium, 15 Apr. 2007. 3.5.2

[17] S. H. Kan. *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, September 2002. 4.3.3

[18] M. M. Lehman and M. Lehman. Laws of software evolution revisited. In *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996. 1.2

[19] B. O'Sullivan. *Mercurial: the definitive guide*. O'Reilly Media, Inc., July 2009. 2.3.1

[20] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. Texto clásico. 3.5.2, 4.3.2

[21] H. Pennington. *GTK+/GNOME Application Development*. New Riders Publishing, Carmel, IN, USA, 1999. 2.1.1

[22] C. Pilato, B. Collins-Sussman, and B. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., April 2008. 2.4.2, 2.4.2.1

[23] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0. 3

[24] E. S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. O'Reilly & Associates, Inc., revised edition, 2001. 1.1, 2.3

[25] G. Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, 2006. 2.2

[26] G. Robles, J. J. Amor, J. M. González-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *IWPSE*, pages 165–174, 2005. 3.5.1

[27] G. Robles and J. M. González-Barahona. Developer identification methods for integrated data from various sources. In *MSR*, 2005. 3.6.1

[28] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9, New York, NY, USA, 2006. ACM. 3.5.1

[29] G. Robles, S. Koch, and J. M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Edinburg, Scotland, UK, 2004. 3.5.1

[30] G. Robles-Martínez, J. M. González-Barahona, J. Centeno-González, V. Matellan-Olivera, and L. Rodero-Merino. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, pages 111–115, Portland, Oregon, 2003. 1.1, 3.5.1

[31] I. Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. 1.2

[32] D. Woods and G. Guliani. *Open Source for the Enterprise: Managing Risks Reaping Rewards*. O'Reilly Media, Inc., 2005. 2.1

Apéndice A

Glosario

ABI. Application Binary Interface (Interfaz Binaria de Aplicación)

API. Application Programming Interface (Interfaz de Programación de Aplicación).

FLOSS. Free Libre Open Source Software.

FOSS. Free Open Source Software.

git. Sistema de control de versiones distribuido, el cual es Software Libre.

GNOME. Ambiente de escritorio, que provee una interfaz gráfica de usuario que se ejecuta en un sistema operativo y está compuesto exclusivamente de Software Libre. Es el acrónimo de GNU Network Object Model Environment.

KDE. Ambiente de escritorio, al igual de GNOME, que originalmente utilizaba bibliotecas que no eran Software Libre. Es el acrónimo de K Desktop Environment, aunque originalmente fue Kool Desktop Environment.

mercurial. Sistema de control de versiones distribuido, el cual es Software Libre.

OSS. Open Source Software.

RCS. Revision Control System.

SCM. Software Configuration Management.

SCM. Source Code Management.

VCS. Version Control System.