Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Desarrollo, distribución y modificación del *software* libre

Esteban Natán Ramirez Escobar
Asesorado por Ing. Franklin Antonio Barrientos Luna

Guatemala, junio de 2005

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DESARROLLO, DISTRIBUCIÓN Y MODIFICACIÓN DEL *SOFTWARE* LIBRE

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA

FACULTAD DE INGENIERÍA

POR

ESTEBAN NATÁN RAMIREZ ESCOBAR

ASESORADO POR ING. FRANKLIN ANTONIO BARRIENTOS LUNA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, JUNIO DE 2005

UNIVERSIDAD DE SAN CARLOS DE GUAEMALA
FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Sydney Alexander Samuels Milson
VOCAL I	Ing. Murphy Olympo Paiz Recinos
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIO	Ing. Carlos Humberto Pérez Rodríguez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Ing. José Ricardo Morales Prado
EXAMINADOR	Ing. Edgar René Ornelis Hoil
EXAMINADOR	Inga. Virginia Victoria Tala Ayerdi
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

ACTO QUE DEDICO A

DIOS

Fuente de vida y sabiduría.

Mi amada compañera de vida

Ana Lisbeth

Con todo mi amor.

Mis padres

Marco Tulio y Marta

Gracias por su amor, cuidado y apoyo. Su ejemplo ha sido de mucha inspiración.

Mi hermana

Débora

Todo lo que hemos compartido significa mucho para mí.

Toda mi familia

Familia Ramirez

Familia Escobar

Familia Salazar Say

Por su apoyo en las distintas etapas de mi vida.

Mis amigos y hermanos espirituales

Porque han sido parte importante en este camino.

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

Desarrollo, distribución y modificación del *software* libre

Tema que me fuera asignado por la coordinación de la carrera de Ciencias y Sistemas con fecha 31 de julio de 2004.

Esteban Natán Ramirez Escobar

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

LISTA DE SÍMBOLOS

IA	Inteligencia artificial
MIT	*Massachussets Institute of Technology*
ITS	*Incompatible Timesharing System* (Sistema incompatible de tiempo compartido)
US$	Dólares estadounidenses
FSF	Free Software Foundation
DRA	Desarrollo rápido de aplicaciones
T4G	Técnicas de cuarta generación

GLOSARIO

Algoritmo Conjunto de reglas claramente definidas para la resolución de un problema. Un programa de *software* es la transcripción, en un lenguaje de programación, de un algoritmo.

ASCII *American Standard Code of Information Interchange*. Código normalizado estadounidense para el intercambio de información. Permite definir un conjunto de caracteres alfanuméricos que es utilizado para lograr la compatibilidad entre diversas aplicaciones que procesan texto.

Banners Gráfico, generalmente rectangular, que se inserta en una página *web* y tiene carácter publicitario.

Biblioteca de componentes Colección de elementos de *software* con características que los hacen reutilizables.

Browser Conocido como "navegador", es el programa utilizado para ver los contenidos de una página *web*.

BSD Familia de sistemas operativos desarrollados a partir de la versión 4.4 BSD-lite de la Universidad de Berkeley.

CASE	*Computer Aided Software Engineering* (o ingeniería de *software* asistida por computadora) es un conjunto de herramientas automatizadas para conseguir la generación automática de programas con base en una especificación a nivel de diseño.
Copyleft	Instrumento legal utilizado por los desarrolladores de *software* libre para evitar que se convierta en *software* no libre algún programa derivado de *software* libre.
DBMS	*Data Base Management System* (o sistema de administración de base de datos), es el componente de una base de datos que realiza la administración del sistema.
Debian	Sistema operativo desarrollado como *software* libre que utiliza el *kernel* de Linux.
Direcciones IP	Dirección de un determinado recurso (servidor, cliente, *router*, red) dentro de Internet.
Encriptación	Proceso que utiliza la criptografía para proteger la información de accesos no autorizados.
FAQs	*Frequently Asked Questions*, es un listado de respuestas a las preguntas más frecuentes sobre un determinado tema.
Freeware	*Software* que se distribuye gratuitamente, pero no permite la redistribución ni da acceso al código fuente del programa.

FTP	*File Transfer Protocol*. Es el protocolo de Internet utilizado para el acceso a estructuras de directorios remotas y la transferencia de archivos.
GCC	Compilador desarrollado como *software* libre. Es el más común dentro del ambiente del *software* libre.
GNU	GNU's *Not Unix* (GNU no es Unix, en español) es un proyecto que pretende desarrollar tanto *software* como sea necesario para poder usar una computadora sin utilizar *software* no libre.
GNU Emacs	Editor de texto muy popular en el ambiente del *software* libre.
GPL	GNU *General Public Licence* (o licencia pública general de GNU, en español) es la licencia más popular para la distribución de *software* libre.
Hacker	Término para designar a alguien con talento, conocimiento e inteligencia, especialmente relacionado con las operaciones de computadora, las redes, los problemas de seguridad y similares, con gran interés por la resolución de problemas.
HTML	*Hipertext Transfer Markup Language*, es el lenguaje estándar para la elaboración de páginas *web*.

HTTP	*Hipertext Transfer Protocol*, es un protocolo de Internet que permite la transferencia de información en archivos de texto, gráficos, video, audio y otros recursos multimedia.
httpd	Proceso que ejecuta las operaciones del servidor de páginas *web* en los sistemas operativos tipo Unix.
Interfaz	Elemento de transición o conexión que facilita el intercambio de datos. El teclado de una computadora, por ejemplo, es una interfaz entre el usuario y la máquina.
Internet	Sistema mundial de redes de computadoras interconectadas. Actualmente es utilizado por millones de personas en todo el mundo como una herramienta de comunicación e información.
Java	Lenguaje de programación desarrollado por Sun, el cual tiene la particularidad de poder ejecutarse tanto en ambientes Unix como Windows.
Kernel	Es el núcleo o parte principal de un sistema operativo. Se compone del conjunto de instrucciones que sirven de interfase entre las aplicaciones y el *hardware*.
LaTeX	Sistema de preparación de documentos, comúnmente utilizado en medianos o grandes documentos técnicos o científicos.

Lenguaje de programación Lenguaje con léxico y sintaxis definidos empleado para instruir a una computadora para que ejecute determinadas acciones, codificadas en dicho lenguaje.

Lenguaje ensamblador Lenguaje de bajo nivel similar al utilizado por las computadoras.

Lenguaje orientado a objetos Lenguaje de programación basado en el concepto de agrupar procesos relacionados y estructuras de datos en entes conocidos como "objetos".

Linux Sistema operativo muy popular desarrollado como *software* libre.

Mailboxes Casillas de correo electrónico. Espacio donde se almacenan los mensajes electrónicos en espera de ser recogidos por los usuarios.

Open Source Concepto similar al del *software* libre pero orientado al ambiente empresarial.

Página *web* Archivo escrito en lenguaje HTML que contiene información y enlaces a recursos de imagen, video y otras páginas *web*.

PDA O *Personal Digital Assistant*, es una computadora de tamaño suficientemente pequeño para poder ser sostenida en la mano o guardada en un bolsillo. Algunas permiten ingresar datos por medio de escritura manual.

PERL	Lenguaje utilizado para la creación de páginas *web* dinámicas.
Portar	Modificar un determinado *software* para que pueda ser utilizado en otro ambiente.
Script	Conjunto de instrucciones para ser ejecutadas en secuencia y utilizado para crear páginas *web* dinámicas.
Shareware	*Software* distribuido gratuitamente, pero exige algún pago para tener la funcionalidad completa del programa o para poder seguir usándolo después de cierto tiempo. No da acceso al código fuente.
Sistema operativo	Programa que administra los recursos de una computadora y controla a los demás programas que funcionan en dicha computadora.
Sitio *web*	Conjunto de páginas *web* afines.
***Software* empotrado**	*Software* utilizado en aparatos diferentes a las computadoras, tales como hornos de microondas, teléfonos celulares y otros.
***Software* libre**	*Software* que es distribuido con licencias que permiten su modificación y redistribución, sea ésta gratuita o no. Da acceso al código fuente.

***Software* propietario**	*Software* que limita su uso y prohibe su redistribución y modificación.
USENET	Conjunto de cientos de foros electrónicos de discusión, llamados "grupos de noticias" (*Newsgroups* en inglés). Utilizan, en su mayoría, Internet para la transmisión de los mensajes.
User friendly	Característica de un determinado *software* que le permite ser usado por una persona sin muchos conocimientos técnicos.
Yacc	Herramienta utilizada para elaborar el aspecto léxico de un lenguaje de programación, que permite crear las reglas necesarias para interpretar una sintaxis.

RESUMEN

El *software* libre es *software* que se distribuye con autorización para que pueda ser usado, copiado, examinado, modificado y redistribuido, ya sea literal o con modificaciones, gratis o mediante una remuneración económica, sin ninguna restricción.

La existencia del *software* libre ha permitido que actualmente exista gran cantidad de *software* desde sistemas operativos (como el popular Linux) hasta juegos, de gran calidad con precios muy bajos o gratis. También ha provocado el surgimiento de una comunidad de individuos que colaboran entre sí desde cualquier parte del mundo, participando como desarrolladores o verificadores de nuevo *software*.

Cualquier *software* tiene el potencial de convertirse en *software* libre, ya que esto depende únicamente de las libertades que otorgue la licencia bajo la cual es distribuido. La licencia que más se utiliza para distribuir *software* libre es la Licencia Pública General de GNU, la cual implementa un mecanismo legal conocido como *copyleft* (contrario al conocido *copyright*) para evitar que el *software* descendiente de *software* libre pueda cambiarse a *software* libre.

Las características del *software* libre han provocado el surgimiento de un modelo alterno, el modelo abierto de desarrollo. Este es un modelo colaborativo y evolutivo que produce versiones más completas cíclicamente, hace uso intensivo de la reutilización de código y se apoya fuertemente en los usuarios para hacer el proceso de pruebas y depuración.

OBJETIVOS

General

Describir el concepto de *software* libre, evaluarlo y definir su forma de producción y uso.

Específicos

1. Describir el concepto de *software* de *software* libre.

2. Comparar el *software* libre con otros tipos de *software* e indicar su importancia y uso actual.

3. Describir la metodología de desarrollo del *software* libre y compararla con otras metodologías de desarrollo más comunes.

4. Describir la forma en que el *software* libre se distribuye y cómo se aplica su licenciamiento.

5. Establecer si es posible crear un modelo de negocios rentable que desarrolle y comercialice *software* libre.

6. Dar una guía para la utilización del *software* libre y definir el tipo de aplicaciones y tipos de proyectos que pueden ser desarrolladas bajo este concepto.

INTRODUCCIÓN

De la misma manera en que el uso del Internet se ha extendido, también se ha convertido en una herramienta muy útil para la industria del *software*. El Internet ha hecho que la distribución, el comercio y aún el desarrollo de *software* hayan experimentado cambios notables. Dentro de estos cambios, seguramente el más importante es el concepto de *software* libre. El *software* libre es *software* que se distribuye con autorización para que cualquiera pueda usarlo, copiarlo, examinarlo, modificarlo y redistribuirlo, ya sea literal o con modificaciones, gratis o mediante una remuneración económica. Estas características del *software* libre tienen muchas implicaciones en la forma en que se desarrolla el *software*, la forma en que se distribuye y su licenciamiento, entre otras.

El concepto de *software* libre le ha dado un gran impulso al desarrollo de nuevos productos y su distribución masiva, en su mayoría, de forma gratuita o a precios muy bajos. También el desarrollo de los productos ha sufrido cambios impresionantes respecto a la forma en que se hace tradicionalmente, es decir, contratar a un equipo de personas para el desarrollo de un producto, para probarlo, documentarlo y distribuirlo. Con este nuevo concepto, las personas que participan en el desarrollo trabajan independientemente, en distintos lugares alrededor del mundo, enlazados únicamente por Internet y no por relaciones formales de trabajo.

Bajo este concepto se han desarrollado infinidad de nuevos productos de excelente calidad, los cuales son distribuidos por medio del Internet de forma gratuita. Un ejemplo de este tipo de productos es el sistema operativo Linux, el cual se ha convertido en un excelente competidor para los sistemas operativos tradicionales del mercado. Otro ejemplo es el sistema operativo FreeBSD, utilizado en los servidores de los sitios *web* más importantes del mundo.

En esta tesis se hace una presentación del *software* libre y se detallan varios aspectos relacionados con éste. Primeramente se describe el *software* libre y se presentan sus antecedentes, seguido por la exposición de sus implicaciones y una comparación con otros tipos de *software*. A continuación se describe la gestión de proyectos de *software* libre, la cual difiere de manera notable de la manera tradicional de gestión de proyectos. Seguidamente se presenta el modelo abierto de desarrollo, el cual es un modelo de desarrollo de *software* que se adapta a las características del *software* libre.

Se discute además lo relacionado con la comercialización del *software* libre. Aquí se evalúa la conveniencia de licenciar un producto de *software* como *software* libre. También se examinan cuáles son las condiciones que deben darse para que un producto pueda distribuirse como *software* libre y se presentan las bases legales de la distribución del mismo. Al final se presenta una parte práctica, con un ejemplo real del uso del *software* libre. En esta sección se examina la solución implementada para un determinado sitio *web* utilizando únicamente *software* libre y se evalúa esta solución.

Esta tesis está orientada a personas que se relacionen con el desarrollo, distribución, comercialización y uso del *software*. Los capítulos 2 y 3 hacen énfasis en la gestión y desarrollo del *software*; el capítulo 4 se refiere a su comercialización y distribución y el capítulo 5 se enfoca en el uso del mismo.

1. *SOFTWARE* LIBRE

1.1 Definición

El *software* libre es *software* que se distribuye con autorización para que cualquiera pueda usarlo, copiarlo, examinarlo, modificarlo y redistribuirlo, ya sea literal o con modificaciones, gratis o mediante una remuneración económica.

Se conoce como movimiento del *software* libre al grupo de personas que establece las características del *software* libre y justifica y promueve su desarrollo y distribución. Este movimiento ha establecido cuatro características esenciales del *software* libre, en sus palabras, "cuatro libertades fundamentales para los usuarios"[1]. Un programa es considerado como *software* libre si sus usuarios tienen todas estas libertades:

- La libertad de ejecutar el programa, con cualquier propósito (libertad 0).
- La libertad de estudiar cómo funciona el programa y adaptarlo a necesidades particulares (libertad 1). El acceso al código fuente es una precondición para esto.
- La libertad de distribuir copias (libertad 2).
- La libertad de modificar el programa, y liberar las modificaciones al público (libertad 3). El acceso al código fuente es una precondición para esto.[2]

Para los usuarios del *software* libre esto se traduce en la libertad de redistribuir copias, ya sea con o sin modificaciones, gratis o cobrando una cuota a cualquier persona y en cualquier lugar sin pedir o pagar permisos.

[1] Qué es el Software Libre? - El Proyecto GNU - Fundación para el Software Libre (FSF) (http://www.gnu.org/philosophy/free-sw.es.html) Septiembre del 2000.

El usuario del *software* libre también tiene la libertad de hacer modificaciones y utilizarlas de manera privada, sin mencionar que dichas modificaciones existen. Si los cambios son publicados, lo cual también queda a criterio del usuario, no es necesario avisar a nadie en particular, o de una manera específica.

Existen otras definiciones de *software* libre, sin embargo, ésta es la que establece las libertades más relevantes. Algunos autores consideran como *software* libre todo aquel que, como mínimo, permite al usuario utilizarlo sin restricciones ni necesidad de pagar y permite su redistribución no comercial[3]. Estas condiciones se cumplen para todo el *software* que en esta tesis se considera libre, pero también se cumplen para otro conjunto de *software* (como el *freeware* y el *shareware*) que en esta tesis no se considera libre.

Este otro conjunto de *software* no es relevante en esta tesis, ya que la manera en que se desarrolla, distribuye y modifica no se rige por las cuatro leyes básicas antes mencionadas. Estas leyes le dan ciertas características al *software* cuya descripción, comparación y evaluación es el objetivo de esta tesis.

En lo sucesivo, se considerará *software* libre el *software* que tenga las cuatro libertades fundamentales descritas anteriormente o aquel que tenga las mismas características que éste, al menos en lo que respecta a desarrollo, distribución y modificación.

---

[2] Ibíd.
[3] Tipos de licencias para software redistribuible libremente.
(http://www.gsyc.inf.uc3m.es/sobre/pub/novatica-mono/) Octubre del 2000

1.2 Antecedentes

1.2.1 Inicio de la industria del *software*

Al inicio de la era de la informática (por los años 60) no existía una clara distinción entre desarrolladores y usuarios. Los sistemas informáticos de ese entonces no eran tan *user friendly* como ahora y se requería de cierta experiencia técnica para operarlas, así que los desarrolladores eran al mismo tiempo los usuarios. Aunque no existía el término *software* libre, en realidad, en este ambiente todo el *software* utilizado calificaba como *software* libre. Era común la colaboración entre universidades o compañías que deseaban portar y usar un programa. Si alguien más usaba un programa interesante y poco conocido, siempre se podía pedir el código fuente para verlo, y de esa manera, entenderlo, modificarlo, o tomar ciertas partes del mismo para hacer un nuevo programa.

1.2.2 La expansión del *software* propietario

Con la propagación del uso las computadoras fuera de las universidades al hogar y la oficina, el *software* se volvió gradualmente cada vez más fácil de emplear para un usuario menos capacitado técnicamente y por lo tanto, comerciable. Surgió el comercio del *software*, que se desarrolló regido por corporaciones que usaban el modelo conocido como *software* propietario.

El *software* propietario es distribuido bajo licencias que restringen justamente lo que el *software* libre permite. De esta manera, la compañía dueña del *software* en cuestión se asegura ser la única en poder explotarlo comercialmente.

En los años 80 casi todo el *software* era propietario. Los dueños del *software* que surgieron restringieron el acceso al código fuente y por lo tanto, los usuarios ya no podían modificarlo para redistribuirlo. Esto acabó con la colaboración entre desarrolladores que no pertenecieran a la misma organización. El desarrollo de *software* pasó a ser dominio casi exclusivo de empresas que lo resguardaban celosamente e imponían una serie de restricciones valiéndose de las licencias de distribución.

En esa misma época se empezó a comercializar *hardware* que incluía su propio sistema operativo. Dicho sistema operativo era el único que podía funcionar sobre dicha plataforma, sin embargo, dicho *software* no era libre. Para poder usar las computadoras modernas de la época, como la VAX o el 68020, se debía firmar un "acuerdo de no revelar" (*nondisclosure agreement*) aún para obtener una copia ejecutable.

Esto quería decir que el primer paso para poder utilizar una computadora era comprometerse a no colaborar ni compartir código con terceros. Los dueños de *software* propietario establecieron esta prohibición y decidieron que si un cliente necesitaba algunos cambios en el *software*, quien lo haría sería la compañía vendedora.

1.2.3 El sistema GNU

1.2.3.1 Antecedentes del sistema GNU

Por esa época, Richard Stallman trabajaba en el laboratorio de Inteligencia Artificial del MIT como miembro de una comunidad que compartía el *software*. El laboratorio de IA usaba un sistema operativo denominado ITS (*Incompatible Timesharing System*) [Sistema incompatible de tiempo compartido] que los *hackers* del equipo habían diseñado y escrito en lenguaje ensamblador para la PDP-10 de Digital, una de las más grandes computadoras de la época.

El trabajo de Stallman como miembro de esta comunidad y como *hacker* de sistema en el equipo del laboratorio de IA, era mejorar este sistema. Sin embargo, la situación cambió drásticamente durante la primera parte de los años 80 cuando Digital descontinuó la serie PDP-10. Dicha máquina, elegante y poderosa en la década de los 60, no se pudo extender naturalmente a los espacios de direccionamiento más grandes que se hicieron factibles en los ochenta. Cuando el laboratorio de IA adquirió una nueva PDP-10 en 1982, sus administradores decidieron utilizar el sistema no libre de tiempo compartido de Digital en lugar de ITS.

La comunidad de *hackers* del laboratorio de IA ya había colapsado cierto tiempo antes. En 1981, la compañía derivada Symbolics había contratado a casi todos los *hackers* del laboratorio de IA, y la despoblada comunidad ya no era capaz de mantenerse a sí misma.

Debido a estos factores, a Stallman se le hizo imposible continuar su trabajo. Existía la opción de desarrollar *software* propietario para alguna compañía, pero Stallman, por razones éticas, estaba en desacuerdo con las implicaciones y consecuencias de este tipo de *software*. Así que decidió volver a iniciar una comunidad similar a la cual él había pertenecido anteriormente.

Decidió que lo primero que necesitaba era desarrollar un sistema operativo libre dada la imposibilidad de siquiera usar una computadora si no se tiene uno. Con un sistema operativo libre, podría existir una nueva comunidad de *hackers* cooperando y cualquiera podría usar una computadora sin comprometerse con el *software* propietario.

Decidió que el sistema operativo fuera compatible con Unix ya que de esta manera sería portable y los usuarios de Unix podrían cambiarse con facilidad al nuevo sistema operativo. El nombre GNU se eligió siguiendo una tradición *hacker*, como acrónimo recursivo para “GNU's Not Unix” (GNU no es Unix).

Un sistema operativo es más que un núcleo, apenas suficiente para hacer funcionar otros programas. En los años 70, todo sistema operativo digno de llamarse así incluía procesadores de órdenes, ensambladores, compiladores, intérpretes, depuradores, editores de texto, programas de correo y muchos otros. ITS los tenía, Multics los tenía, VMS los tenía, Unix los tenía, así que Stallman decidió que GNU también los incluiría.

1.2.3.2 Inicio del sistema GNU

En enero de 1984, Stallman renunció a su trabajo en MIT y comenzó a escribir *software* GNU. Como encontró tropiezos para conseguir un compilador libre, decidió que el primer programa para el proyecto GNU sería un compilador multilenguaje y multiplataforma.

Con la esperanza de no tener que escribir todo el código para el nuevo compilador, obtuvo el código fuente del compilador Pastel, pero debido a las deficiencias técnicas de este compilador que hacían imposible portarlo, desarrolló el compilador actualmente conocido como GCC.

Sin embargo, habrían de pasar varios años antes de desarrollar GCC ya que primeramente desarrolló GNU Emacs. Inició a trabajar en Emacs en septiembre de 1984 y a principios de 1985 ya empezaba a ser funcional. De esta manera permitió usar sistemas Unix para edición.

A estas alturas, surgieron más usuarios deseosos de usar Emacs y con ello la preocupación de cómo distribuirlo. Stallman lo puso a disposición del público en el servidor FTP anónimo de la computadora que usaba.

Pero en aquel tiempo mucha gente interesada en usar Emacs no tenía acceso a Internet y no podía obtener una copia por FTP, así que Stallman inició un negocio de distribución de *software* libre: se enviaba la cinta a quien la deseaba por un pago de $150. Este sistema sería el precursor de las compañías que en la actualidad se dedican a la distribución de *software* libre.

A medida que el interés en el uso de Emacs crecía, otras personas se involucraron en el proyecto GNU. Como era necesario tener fondos para proseguir, en 1985 se creó la Free Software Foundation (Fundación para el *Software* Libre, FSL), una organización de no lucrativa para el desarrollo del *software* libre. La FSL también acaparó el negocio de distribución en cinta de Emacs y más adelante lo extendió al agregar otros productos de *software* libre (tanto GNU como no-GNU) a la cinta, y con la venta de manuales libres.

1.2.3.3 Los componentes del sistema GNU

A medida que proseguía el proyecto GNU se desarrollaron o encontraron una cantidad creciente de componentes. Eventualmente se vio la utilidad de hacer una lista con los que faltaban, la cual es conocida como la lista de tareas de GNU. Dicha lista sirvió para reclutar desarrolladores para escribir las piezas faltantes. Además de los componentes Unix faltantes, se agregó a la lista otros útiles proyectos de *software* y documentación que debería tener un sistema verdaderamente completo.

En la actualidad, casi ningún componente Unix queda en la lista de tareas GNU. Esos trabajos ya han sido terminados, fuera de algunos no esenciales. Pero la lista está aún llena de proyectos considerados aplicaciones.

Como cada componente de un sistema GNU se implementó en un sistema Unix, cada uno podía correr en sistemas Unix mucho antes de que existiera un sistema GNU completo. Algunos de esos programas se hicieron populares y los usuarios comenzaron a extenderlos y transportarlos a las distintas versiones incompatibles de Unix, y algunas veces a otros sistemas también.

El proceso hizo que dichos programas fueran más potentes y atrajeran tanto fondos como contribuyentes al proyecto GNU. Pero también demoró por varios años la finalización de un sistema mínimo en funciones, a medida que el tiempo de los desarrolladores GNU se usaba para mantener esos transportes y en agregar características a los componentes existentes, en lugar de adelantar la escritura de los componentes faltantes.

1.2.3.4 El GNU HURD

En 1990, el sistema GNU estaba casi completo. El único componente importante faltante era el *kernel*. Se decidió implementar el núcleo como una colección de procesos servidores corriendo sobre Mach. Mach es un *microkernel* desarrollado en Carnegie Mellon University y luego en la University of UTA. El GNU HURD es una colección de servidores que corren sobre Mach, y se ocupan de las tareas del *kernel* Unix. Sin embargo, el inicio del desarrollo se demoró mientras se esperaba que Mach se entregara como *software* libre, tal como se había prometido.

Una razón para elegir este diseño había sido evitar lo parecía ser la parte más dura del trabajo: depurar el *kernel* sin un depurador al nivel de código fuente. Esta parte del trabajo ya había sido hecha en Mach, y se esperaba depurar los servidores HURD como programas de usuario, con GDB.

Llevó un largo tiempo hacer esto posible, y los servidores multihilo que se envían mensajes unos a otros han sido muy difíciles de depurar. Debido a esto y a otras razones, hacer que HURD trabaje sólidamente se ha tardado varios años

.

1.2.3.5 El GNU/Linux

El GNU HURD no está listo para el uso en producción. Afortunadamente, está disponible otro *kernel*. En 1991, Linus Torvalds dirigió el desarrolló de un *kernel* compatible con Unix y lo denominó Linux. Cerca de 1992, al combinar Linux con el sistema no tan completo de GNU, resultó en un sistema operativo libre completo. Es gracias a Linux que podemos ver funcionar un sistema GNU en la actualidad.

1.2.4 *Software* no-GNU

El *software* libre está íntimamente ligado con el sistema GNU. Sin embargo, existe otra gran cantidad de *software* libre aparte del sistema GNU. Existen sistemas completos, entre estos los Unix BSD libres. Estos sistemas surgieron con base en el *kernel* que evolucionó del Unix de la Universidad de Berkeley, específicamente la versión Lite 2.2.

También existe *software* que no necesariamente forma parte de un sistema completo como GNU o BSD, pero que de alguna u otra forma se puede agrupar por conjuntos con base en las licencias por medio de las cuales es distribuido.

1.2.5 El *software* libre en el mundo de los negocios

Hasta este punto de la historia, el *software* libre había sido desarrollado por aficionados voluntarios y por organizaciones sin fines de lucro. Sin embargo, no pasaría mucho tiempo sin que el *software* libre llamara la atención del mundo corporativo.

A medida que el movimiento del *software* libre se expandía, más y más empresas se dieron cuenta que podían hacer negocios utilizando este modelo de producción de *software*. Y así fue como se inició la historia de la incursión del mundo empresarial en el mundo del *software* libre.[4]

- 22 de enero de 1998: Netscape anuncia que liberará el código fuerte de su Navigator.

- 3 de febrero de 1998: se acuña el término *open source* (código fuente abierta) que es un concepto similar a *software* libre pero evita deliberadamente el sentido confrontativo que subyace en el término *free software* (*software* libre).

- 31 de marzo de 1998: el código de Navigator se hace disponible y en cuestión de horas mejoras al mismo se propagan por Internet.

- 7 de mayo de 1998: Corel Computer Corporation anuncia el Netwinder, una computadora económica que corre con Linux.

- 11 de mayo de 1998: Corel anuncia sus planes de adaptar WorPerfect y el resto de sus programas de ofimática a Linux.

- 28 de mayo de 1998: Sun Microsystems y Adaptec se unen a Linux Internacional.

- 22 de junio de 1998: IBM anuncia que utilizará el servidor libre Apache.

[4] Software libre, el negocio del siglo XXI. (http://www.baquia.com/com/20000918/art00044.print.html) Febrero de 2001

- 13 de julio de 1998: Oracle e Informix anuncian que conectarán sus bases de datos a Linux.

- 27 de enero de 1999: HP y SGI anuncian que utilizarán aporte Linux.

Esta historia aún no termina de escribirse, ya que cada vez más y más empresas de reconocido prestigio se adaptan a este modelo de negocios. Durante el año 2000 las principales compañías japonesas, entre las que se encuentran Sony, Toshiba, Fujitsu, Hitachi, NEC y Mitsubishi, anunciaron que colaborarán en la creación de un sistema operativo basado en Linux para dispositivos portátiles. El objetivo final de estas empresas es desarrollar un estándar que se pueda emplear en todo tipo de instrumentos, desde teléfonos móviles hasta PDAs.

Por otro lado, el 29 de agosto del 2000, un grupo de compañías, entre las que están Intel, IBM, Hewlett-Packard y NEC USA, anunciaron la creación de la Open Source Development Lab (OSDL), un laboratorio que pretende convertir a Linux en un sistema operativo más accesible y manejable para la gestión empresarial. Estas empresas planean invertir cientos de millones de dólares en este laboratorio, cuyo cuartel general ha sido fijado en Pórtland (Estados Unidos).

1.2.6 El *software* libre y el gobierno

El *software* libre también ha llamado la atención de los gobiernos de Estados Unidos y Europa. Durante el año 2000, el Comité de Asesoramiento en Tecnologías de la Información, integrado por destacados dirigentes de la industria y de ciertas universidades estadounidenses, elaboró un informe dirigido al presidente del país en el que recomendaba buscar métodos innovadores para escribir *software* de calidad. Ahora el comité ha dado un paso más.

Los expertos concluyeron otro documento con argumentos más radicales en el que piden al gobierno que apoye de inmediato y sin reservas la utilización de *software* libre.

La Comisión Europea también ha hecho sus propias observaciones. En julio del 2000 empezó a difundir la propuesta llamada *Puesta en práctica de la ayuda del programa de las Tecnologías de la Sociedad de Información para la investigación y lanzamiento del software libre*, que recopila datos sobre posibles actuaciones concretas para fomentar el *software* libre desde los programas IST, que financian proyectos de investigación.

1.3 Importancia y repercusión del *software* libre

Progresivamente, el *software* libre adquiere mayor protagonismo en la industria del *software*, las instituciones académicas y los gobiernos. Este protagonismo ha sido consecuencia, en gran parte, de los cambios que esta filosofía provoca y también los aportes que hace a la informática mundial. Primero, si se quiere, en forma silenciosa y por medio de un grupo reducido de personas, para posteriormente llegar a ser un tema de mucha discusión y tener un gran número de seguidores en todo el mundo y empresas promotoras.

Al hablar de cambios y aportes, se pueden considerar tres dimensiones[5], que se presentan a continuación.

1.3.1 Dimensión económica y legal

El *software* libre ha introducido cambios en la dimensión económica y legal desde el momento que propone y constituye una nueva manera de licenciar y comercializar el *software*.

[5] Sobre Open Source. (http://aula.linux.org.ar/biblioteca/sobre_open_source.htm) Febrero de 2001

Hasta antes de que el movimiento adquiriera popularidad, la manera normal en que se distribuía el *software* era cobrando por cada una de las copias que se distribuyera del programa. El código fuente constituía uno de los mayores secretos de la empresa que desarrollaba dicho programa, con el fin de evitar que otra empresa desarrollara un programa igual. Estaba totalmente restringido hacer copias del programa. Todas estas reglas comerciales, sin embargo, no son las que rigen el *software* libre. Para las empresas de *software* propietario, la forma en que el *software* libre opera constituye una anormalidad, la cual las está llevando a tomar ciertas acciones ya sea para adaptarse a esta situación o contrarrestarla.

Económicamente hablando, el *software* libre ha provocado varios cambios. El más importante es el hecho de que ahora se puede conseguir variedad de *software* de calidad por precios muchísimo más bajos que los precios cobrados por las empresas de *software* propietario. La mayoría del *software* libre se distribuye gratuitamente o por un precio generalmente muy bajo.

Como consecuencia de este cambio, ahora usuarios con menos recursos tienen acceso a la tecnología de la informática sin incurrir en violaciones a la ley. Esta situación es de enorme beneficio para países como Guatemala, donde debido a la poca capacidad de adquisición, un gran porcentaje del *software* instalado no es licenciado debidamente y como consecuencia se viola los derechos de autor.

Sin embargo, el abaratamiento del *software* no es el único cambio económico provocado por el *software* libre. También ha habido un desplazamiento de los mercados del *software*. Han surgido nuevas empresas, que venden soporte para el *software* libre y también venden el mismo *software* libre, modificado para especializarse en ciertos usuarios.

En la dimensión legal, el *software* libre ha provocado una serie de cambios, de los cuales el más importante es el surgimiento de un concepto opuesto al *copyright*, es decir, el llamado *copyleft*. Mientras el *copyright* limita la distribución de copias, el *copyleft* asegura que se puedan dar sin ningún obstáculo. Al proteger un programa con *copyleft*, quien lo hace se asegura legalmente que su programa pueda ser copiado sin ninguna restricción, actual o futura, y de igual manera las copias de su programa. Es decir, el *copyleft* se hereda conforme surgen nuevas copias de un programa.

Al igual que el cambio económico provocado por el *software* libre, el cambio en la dimensión legal provocará que más usuarios tengan acceso a *software* de gran diversidad. Esto, seguramente, permitirá llevar la tecnología de la informática a países de los llamados del tercer mundo, donde los recursos son limitados.

1.3.2 Dimensión social

El *software* libre no es simplemente un nuevo tipo de licencia, sino un fenómeno social o comunitario en un sentido amplio, sin perder de vista su influencia en la dimensión económica y tecnológica.

En el sentido social, el *software* libre ha provocado cambios en la manera que los desarrolladores y usuarios interactúan y trabajan, así como también la forma en que se enseña y aprende computación.

Tradicionalmente, las empresas que desarrollan *software* comercial forman equipos, ya sea contratando personal y/o involucrando a su propio personal para que lo haga. Las personas que integran estos equipos tienen relaciones formales de trabajo y trabajan de la manera en que la empresa se los indique. Reciben un salario y a cambio tienen que entregar cierta cantidad de código en determinada fecha.

Por su parte, los usuarios del *software*, lo compran y a cambio, si el contrato de compra lo incluye, reciben soporte para poder utilizarlo. Hasta allí termina la interacción entre los creadores del *software* y sus usuarios.

El *software* libre ha cambiado la forma en que los creadores del *software* interactúan entre sí y con los usuarios. Aunque existen las empresas que desarrollan *software* libre con equipos formales, la mayoría no es desarrollado por empresas sino por individuos que interactúan a través de Internet. Gran parte de las empresas que desarrollan *software* libre lo hacen por medio de voluntarios que contribuyen con ciertas asignaciones o elecciones de módulos por hacer.

También los usuarios toman un papel más protagónico en el mundo del *software* libre desde el momento en que pueden convertirse en desarrolladores del mismo, ya que conocen el código del programa. Pueden hacer sugerencias, descubrir errores y reportarlos y crear documentación.

Estos cambios tienen muchas implicaciones, como el hecho de que desarrollar el *software* cueste menos dinero y se obtiene un producto de mejor calidad. La disminución de los costos de desarrollo se debe a que las empresas son apoyadas por voluntarios y mucho del *software* libre ni siquiera es desarrollado por empresas sino por individuos dispersos por todo el mundo. La calidad del *software* libre se consigue, por su parte, porque puede evaluársele de mejor manera si se tiene acceso al código fuente. Un caso muy ilustrativo es el del *software* de encriptación. Si todos conocen el algoritmo de un determinado *software* de encriptación, cualquiera puede descubrir sus debilidades y eliminarlas. Así que el *software* resultante, a pesar de que todos saben cómo encripta la información, lo hace de una manera confiable.

Falta, sin embargo, mencionar una consecuencia muy importante de los cambios sociales. Esta es el traslado de los grupos de desarrollo hacia otras regiones distintas de las tradicionales. Ahora cualquiera puede participar en el desarrollo del *software* libre; solamente necesita una conexión a Internet, sin importar el lugar físico donde se encuentre. Esto ha hecho que se integren en el desarrollo de nuevos productos personas provenientes de países con poca tradición de elaboración de *software*, pero con excelentes habilidades.

Por último, como consecuencia de los cambios antes mencionados, ha surgido una especie de comunidad entre los desarrolladores de *software* libre. Una comunidad donde comparten y promueven una filosofía: la filosofía del *software* libre. Algunos van más allá al evitar todo uso de *software* que no cumpla con los requisitos del *software* libre. Esta comunidad progresivamente ha sumado más miembros hasta ser un tema de mucho interés en el contexto de la informática. Dicha comunidad se ha vuelto muy importante ya que ha hecho varios y muy buenos aportes que han sido adaptados a nivel mundial. Los ejemplos más relevantes son el sistema operativo Linux, el servidor *web* Apache, el lenguaje de programación *web* PHP y las bases de datos MySQL y PostreSQL.

1.3.3 Dimensión tecnológica

Ya se ha indicado que el *software* libre ha creado una gran cantidad de programas de excelente calidad, los cuales pueden ser nuevos e innovadores o sustitutos de *software* propietario. En el caso de los programas nuevos, el *software* libre está produciendo cambios al introducir nueva tecnología y esto a su vez permite darle nuevos usos a la computadora. En el caso de los programas sustitutos existe *software* tal como Linux o *Free*BSD, que pueden usarse como sustitutos de los sistemas operativos producidos por Microsoft; StarOffice, sustituto de Microsoft Office o PostgreSQL que es buen sustituto de los DBMS tradicionales.

Sin embargo, el cambio más importante hecho hasta el momento por el *software* libre en la dimensión tecnológica es provocar el resurgimiento de la filosofía Unix. ¿Por qué se habla del resurgimiento de la filosofía Unix? Paralelamente a la consolidación del uso de *software* propietario se popularizó el uso de del sistema Windows® de Microsoft y las aplicaciones diseñadas para dicho sistema operativo, naturalmente, son *software* propietario. Aunque los sistemas operativos y aplicaciones estilo Unix no dejaron de utilizarse, perdieron popularidad al enfrentarse al estilo *user* f*riendly* propuesto por Microsoft. También influyó en la merma de su popularidad el hecho de que surgieron sistemas operativos estilo Unix que eran *software* propietario, como es el caso de Solaris. Aunque este *software* estilo Unix era de muy buena calidad, el hecho de ser *software* propietario y la popularidad de Windows®, le disminuyó el número de sus usuarios.

Fue necesario que surgiera la filosofía del *software* libre para que esta situación cambiara. La nueva popularidad del *software* estilo Unix se debió a razones tales como el desarrollo de nuevos sistemas operativos y aplicaciones de *software* libre estilo Unix (como el sistema GNU/Linux) de buena calidad, la reducción de los precios provocada por el *software* libre y la distribución sin costo a través de Internet.

Esto tiene consecuencias para la informática mundial, ya que por primera vez existen rivales para Windows® que progresivamente adquieren mayor cuota de mercado. También ahora, gracias al *software* libre, es posible tener toda clase de *software* en una computadora, sin costo, incluyendo el sistema operativo y toda clase de aplicaciones. Es este el caso del sistema GNU/Linux. Sobre este sistema se han desarrollado varias aplicaciones de *software* libre que luego de unirlas al *kernel* Linux, dan una completa funcionalidad a una computadora sin recurrir a *software* propietario ni invertir gran cantidad de recursos económicos.

1.4 Relación del *software* libre con otros tipos de *software*

La filosofía del *software* libre se comprende mejor cuando se compara con otras clasificaciones de *software*. El *software*, dependiendo de la forma en que se distribuye, se puede clasificar en dos grandes categorías (el *software* libre y el *software* propietario). Sin embargo, puede clasificarse en varias sub-categorías como se indica a continuación. Contra estas sub-categorías se comparará el *software* libre.

1.4.1 *Software* de fuente abierta

El término *software* de *fuente abierta* es usado por algunas personas para dar a entender más o menos lo mismo que *software* libre. Sin embargo, muchos hacen diferencia entre ambos términos y tienen preferencia por alguno de los dos.

Este término surgió en una reunión efectuada el 3 de febrero de 1998 en Palo Alto, California. Las personas presentes reaccionaron al anuncio de Netscape de revelar al público el código de su bien conocido *browser*. Eric Raymond, uno de los asistentes, había sido invitado por Netscape para colaborar en el plan de liberación del código y planear las acciones posteriores.

En la reunión se decidió utilizar esta situación para mostrarle al mundo corporativo la superioridad de un *software* de desarrollo abierto. Los participantes consideraron que existía una actitud confrontativa asociada al *software* libre y que lo conveniente era crear una nueva filosofía más adaptada al mundo de los negocios. En algún momento, Richard Stallman (presentado anteriormente como el que inició el proyecto GNU) consideró la posibilidad de adoptar el término, pero luego cambió de opinión.

Posteriormente, Bruce Perens definió *software* de código abierto. Él escribió el primer bosquejo de dicho documento como 'The Debian *Free Software* Guidelines', y lo refinó usando los comentarios de los desarrolladores de Debian obtenidos en una conferencia vía e-mail de un mes de duración en junio de 1997. Luego quitó las referencias específicas a Debian del documento y creó '*Open Source Definition*': la definición del *software* de código abierto[6].

A raíz de esto, dentro de la comunidad de *hackers* se crearon dos grandes grupos. Por un lado los seguidores del término *software* libre y por el otro, los seguidores del término *software* de código fuente abierto. Sin embargo, podría decirse que ambos grupos están de acuerdo en los principios básicos, pero difieren en las recomendaciones prácticas. Para los partidarios del *software* libre, el problema radica en lo que se entiende por *software* de código abierto: *software* que permite ver su codificación, pero nada más. Aunque ellos mismos reconocen que la definición oficial del *software* de código abierto está más apegada al concepto del *software* libre[7].

Podría decirse que el *software* de código abierto está más enfocado a las organizaciones lucrativas y el *software* libre a aquellas organizaciones cuyo máximo interés no es la obtención de beneficios económicos. Sin embargo, en esencia, el *software* de código abierto y el *software* libre son lo mismo. Es más, en ocasiones ambos grupos colaboran entre sí y la mayoría del *software* de código abierto califica como *software* libre. En lo que respecta a los alcances de esta tesis, es decir, los aspectos de desarrollo, distribución y modificación, ambos tipos de *software* tienen las mismas características. Por lo que en lo sucesivo, cuando se hable de *software* libre, se incluirá el *software* de código abierto. En caso de existir diferencias se harán notar. (Ver Anexo C: la definición de *Open Source*)

---

[6] Definición de *Open Source* (http://members.nbci.com/drodrigo/es-osd.html). Septiembre del 2000.
[7] Fuente Abierta - Proyecto GNU – Fundación del Software Libre (FSF) (http://www.gnu.org/philosophy/free-software-for-freedom.es.html). Septiembre del 2000

1.4.2 *Software* de dominio público

El *software* de dominio público es *software* que no está protegido con *copyright*. Es un caso especial de *software* libre no protegido con *copyleft*, que significa que algunas copias o versiones modificadas no pueden ser libres completamente.

Algunas veces se utiliza el término **dominio público** de una manera imprecisa para decir libre o disponible gratis. Sin embargo, **dominio público** es un término legal y significa de manera precisa **sin *copyright***.

El *software* de dominio público califica como *software* libre, pero para los promotores del *software* libre con *copyleft* tiene un grave problema. Como no está regulado el proceso de copia, ya sea para permitirlo o para restringirlo, cualquiera puede hacer ambas cosas. Entonces, el *software* de dominio público puede en algún momento convertirse en *software* propietario. Para evitar esto, la mayoría del *software* libre está protegido con *copyleft*, un mecanismo legal para evitar que se establezcan restricciones para copiar el *software*.

1.4.3 *Software* protegido con *copyleft*

La forma más simple de hacer que un programa sea libre es ponerlo en el dominio público, sin derechos reservados. Esto le permite compartir el programa y sus mejoras a la gente, si así lo desean. Pero puede darse el caso de que personas conviertan el programa en *software* propietario. Ellos pueden hacer cambios, muchos o pocos, y distribuir el resultado como un producto propietario. Las personas que reciben el programa con esas modificaciones no tienen la libertad que el autor original les dio; el intermediario se las ha quitado.

El *software* protegido con *copyleft* es *software* libre cuyos términos de distribución no permiten a los redistribuidores agregar ninguna restricción adicional cuando éstos redistribuyen o modifican el *software*. Esto significa que cada copia del *software*, aun si ha sido modificado, debe ser *software* libre.

La mayoría del *software* libre está protegido mediante *copyleft*, porque se desea dar a cada usuario las libertades que el término *software* libre implica.

Para cubrir un programa con *copyleft*, primero se reservan los derechos, luego se añaden términos de distribución, los cuales son un instrumento legal que le da a quienes lo deseen los derechos a utilizar, modificar y redistribuir el código del programa o cualquier programa derivado del mismo, pero solo si los términos de distribución no son cambiados. Así, el código y las libertades se hacen legalmente inseparables.

1.4.4 *Software* libre no protegido con *copyleft*

El *software* libre no protegido con *copyleft* viene con autorización del autor para redistribuirlo y modificarlo, así como para añadirle restricciones adicionales.

Si un programa es libre pero no está protegido con *copyleft*, entonces algunas copias o versiones modificadas pueden no ser libres completamente. Una compañía de *software* puede compilar el programa, con o sin modificaciones, y distribuir el archivo ejecutable como un producto propietario de *software*.

El Sistema X Windows ilustra esto. El Consorcio X libera X11 con términos de distribución que lo hacen *software* libre no protegido con *copyleft*. Si se desea, puede obtenerse una copia que tenga esos términos de distribución y es libre. Sin embargo, hay versiones no libres también, y hay estaciones de trabajo populares y tarjetas gráficas para PC para las cuales versiones no libres son las únicas que funcionan.

1.4.5 *Software* semilibre

El *software* semilibre es *software* que no es libre, pero viene con autorización para particulares de usarlo, copiarlo, modificarlo y distribuirlo sin fines de lucro. Difiere del *software* libre en que no permite ser explotado comercialmente. PGP es un ejemplo de un programa semilibre. Este *software* no puede ser considerado libre en vista de que tiene restricciones para su distribución.

1.4.6 *Software* propietario

El *software* propietario es *software* que no es libre ni semilibre. Su uso, redistribución o modificación está prohibida, o requiere que se solicite autorización o está tan restringida que no pueda hacerla libre de un modo efectivo.

1.4.7 *Freeware*

El término *freeware* no tiene una definición clara aceptada, pero es usada comúnmente para paquetes de *software* que permiten la redistribución pero no la modificación y su código fuente no está disponible. En vista de que este tipo de *software* no tiene acceso al código fuente, no tiene todas las libertades que el *software* libre provee y no puede ser considerado como tal.

1.4.8 *Shareware*

El *shareware* es *software* que viene con autorización para redistribuir copias, pero dice que quien continúe haciendo uso de una copia o desee tener la funcionalidad completa del programa deberá pagar un cargo por licencia.

El *shareware* no es *software* libre, ni siquiera semilibre. Existen dos razones por las que no lo es: para la mayoría del *shareware*, el código fuente no está disponible; de esta manera, no se puede modificar el programa en absoluto. Además no viene con autorización para hacer una copia e instalarlo sin pagar una cantidad por licencia, ni aún para particulares involucrados en actividades sin ánimo de lucro.

1.4.9 *Software* comercial

El *software* comercial es *software* desarrollado por una entidad que tiene la intención de obtener beneficios económicos por su uso. **Comercial** y **propietario** no se refieren a lo mismo. La mayoría del *software* comercial es propietario, pero hay *software* libre comercial y hay *software* no libre no comercial.

Por ejemplo, Ada de GNU siempre es distribuida bajo los términos de la GPL de GNU y cada copia es *software* libre; pero los desarrolladores venden contratos de soporte.

2. GESTIÓN DE PROYECTOS DE *SOFTWARE* LIBRE

2.1 Gestión de proyectos de *software*

La gestión de un proyecto de *software* es el sistema de procedimientos, prácticas, metodologías y conocimientos que facilitan la planificación, organización, gestión de recursos humanos, materiales y financieros, dirección y control necesarios para que dicho proyecto termine con éxito[8].

Como proyecto se entiende la acción iniciada por una organización en la que recursos humanos, financieros y materiales se organizan de una determinada manera para realizar un trabajo único, en el que dadas unas especificaciones y dentro de limitaciones de tiempo y costo, se intenta conseguir un cambio beneficioso, medido por unos objetivos cualitativos y cuantitativos.

La complejidad de un proyecto pequeño es tal que una única persona puede entender y manejar todos los detalles referentes a su diseño y construcción. Sin embargo, todo proyecto que no permita a un individuo tener bajo control todos los detalles de dicho proyecto, necesita una adecuada gestión.

Para alcanzar el éxito en un proyecto de gran tamaño, no basta con copiar a escala las soluciones hechas para un programa pequeño. Es necesario una cuidadosa gestión y administración del proyecto.

[8] Orlando Rafael Romero Ávalos, José Luis Sánchez Lemus. **Gestión de proyectos de software, estimación y planificación de tiempos y costos de un proyecto de software.** (Guatemala: Universidad de San Carlos de Guatemala, 1998) p. 1.

La gestión de proyectos de *software* se enfoca en tres áreas: personal, problema y proceso. El manejo correcto de estas tres áreas es indispensable para lograr el éxito en proyectos de mediano o gran tamaño.

2.1.1 Personal

En todo proyecto que desee concluir con éxito, el gestor debe preocuparse por atraer, aumentar, motivar, desplegar y retener personal de calidad. Con estos objetivos en mente, los gestores del personal siguen la siguiente serie de etapas: reclutamiento, selección, gestión del rendimiento, entrenamiento, retribución, desarrollo de la carrera, diseño de la organización, diseño del trabajo, desarrollo cultural y desarrollo del espíritu de equipo.

Esta es un área crítica dentro de la gestión de proyectos de *software*. Aun cuando contar con las herramientas apropiadas es necesario, lo es aún más contar con el personal apropiado y proveerles el mejor entorno posible para que puedan producir.

2.1.2 El problema

Antes de acometer la tarea de desarrollo de un proyecto de *software* se requiere de estimaciones cuantitativas y un plan organizado. Sin embargo, esto no puede lograrse sin antes conocer los requerimientos del proyecto, sus objetivos y ámbito. Además se deben identificar todas las soluciones viables y las dificultades técnicas y de gestión.

Se debe entender de manera precisa el problema que se desea resolver. De lo contrario, existe una alta probabilidad de desarrollar una solución de menor o mayor calidad, pero para el problema equivocado.

2.1.3 El proceso

El proceso proporciona la estructura sobre la cual se construye el plan para el desarrollo del proyecto. Por un lado debe seleccionarse el modelo de proceso adecuado para el proyecto que se va a desarrollar. Este podría ser el modelo secuencial lineal, el modelo de prototipo, el modelo RAD, el modelo incremental, el modelo espiral, el modelo de ensamblaje de componentes, el modelo de métodos formales o el modelo de técnicas de cuarta generación.

Una vez escogido el modelo del proceso debe realizarse la descomposición del mismo. Es decir, se toma el conjunto de funciones del *software* y se relaciona con la estructura del proceso, creando una matriz que sirve al gestor para realizar la asignación de recursos y la planificación temporal.

2.2 Proyectos de *software* libre

Si se considera la definición de *software* libre se puede deducir que cualquier *software* desarrollado tiene el potencial para convertirse en *software* libre. Esto porque según la definición del *software* libre, lo que hace la diferencia entre un *software* libre y uno que no lo es, son únicamente los permisos o restricciones que tiene el *software* en cuestión para usarse, copiarse, examinarse, modificarse y redistribuirse.

Si una compañía propietaria de un *software* no-libre decidiera poner a disposición del público el código fuente de dicho programa y quitar todas las restricciones para usarlo, copiarlo, examinarlo, modificarlo y redistribuirlo, automáticamente este se convertiría en *software* libre.

La esencia del *software* libre no está en los modelos de desarrollo, el precio del *software*, la organización que lo desarrolla, etc., sino en los permisos que este *software* tiene. Sin embargo, la eliminación de restricciones ha permitido desarrollar un nuevo modelo de desarrollo de *software* que presenta características muy interesantes, cuya descripción y evaluación es el objetivo de esta tesis.

Este modelo de desarrollo se basa en la libertad de usar el código fuente que otro programador ha desarrollado, la libertad para cualquier programador de aportar código al desarrollo de otro programador, pero sobre todo en la existencia de una comunidad de programadores enfocada en conseguir un *software* determinado[9].

Este nuevo modelo aprovecha las características del *software* libre para adoptar un proceso de desarrollo colaborativo, con el aporte de cualquier cantidad de programadores diseminados literalmente por todo el mundo. Esta es la característica más sobresaliente de este modelo, pero existen otras que se detallan a continuación:

- El desarrollo es liderado por una organización o un individuo cuya función es ensamblar los aportes de todos los programadores y facilitar la comunicación entre los participantes.

- Los participantes en su mayoría son voluntarios que no tienen relación laboral con quienes lideran el proyecto; están geográficamente distribuidos por todo el planeta, su comunicación se basa en Internet y son considerados *hackers*.

- Pueden participar con componentes desarrollados por ellos mismos, depurando o realizando alguna tarea pendiente, necesaria para que el *software* producto tenga la funcionalidad deseada.

[9] La Catedral y el Bazar de Eric S. Raymond (http://www.sindominio.net/biblioweb/telematica/catedral.html). Noviembre del 2000.

- La motivación para el inicio y especialmente para la continuidad del proyecto es la necesidad de alguna herramienta de *software* por parte de la comunidad participante y no la consecución de beneficios económicos.

- El modelo conjuga elementos de otros modelos tales como la programación evolutiva, la creación rápida de prototipos y el ensamblado de componentes. Normalmente el *software* evoluciona al agregársele características necesarias y desechar las innecesarias. Estas funciones, en la mayoría de los casos, son prototipos que luego son ensamblados y el resultado es liberado inmediatamente para ser evaluado y mejorado por la comunidad.

- Las decisiones de diseño o de las características que se implementarán o desecharán, no son tomadas por el líder sino que simplemente se impone la opción que muestre los mejores resultados en la práctica.

- El proceso de pruebas y eventualmente el de depuración es realizado por los usuarios que tienen todo el potencial para convertirse en desarrolladores en vista de que tienen a su disposición el código fuente de los programas.

Como probablemente ya se habrá notado, el usuario del *software* libre tiene el potencial para convertirse en desarrollador del mismo y también podría estar capacitado para darle soporte al mismo, ya que tiene acceso al código fuente del programa.

Podrá pensarse que este modelo funciona exclusivamente para el desarrollo de programas pequeños, pero debe considerarse que uno de los mejores sistemas operativos de la actualidad (Linux) fue elaborado con un proceso que tenía todas las características anteriores.

2.3 Evolución de los proyectos de *software* libre

La mayoría de los proyectos de *software* libre nacen motivados por la necesidad de herramientas de *software* de un individuo u organización. Luego el desarrollo es liderado por ellos mismos ajustado a requerimientos realistas, los cuales pueden cambiar al transcurrir el tiempo. Por el contrario, los proyectos de *software* propietario surgen al tratar de satisfacer las necesidades (reales o supuestas) de terceros. Esto muchas veces provoca que se usen gran cantidad de recursos para crear programas que no se usan ya que no se necesitan.

Estas necesidades pueden ser tan variadas como la creación de una herramienta para ejecutar algún proceso repetitivo de una persona, el cual quiere optimizar. O por otro lado, podría ser la necesidad de crear una herramienta para luego distribuirla como *software* libre y de esa manera sustituir a otra herramienta similar que existe como *software* propietario.

Luego de tener claro el producto final que se desea obtener, rara vez se inicia desde cero. Eric S. Raymond dice: "Los buenos programadores saben qué escribir. Los mejores, qué rescribir (y reutilizar)"[10]. Normalmente se buscan otros proyectos similares y se selecciona los que estén mejor escritos para usarlos como plataforma inicial del desarrollo.

La tradición del mundo UNIX de compartir fuentes siempre se ha prestado a la reutilización del código (esta es la razón por la cual el proyecto GNU escogió a Linux como su sistema operativo base). Actualmente existe gran cantidad de código fuente disponible en el mundo del *software* libre, así que es casi seguro encontrar algún código de excelente calidad sobre el cual desarrollar.

[10] La Catedral y el Bazar de Eric S. Raymond (http://www.sindominio.net/biblioweb/telematica/catedral.html). Noviembre del 2000.

El siguiente paso y uno de los más importante es hacerse de una base de usuarios. Es decir, un grupo de personas que usen el *software* en sus etapas de desarrollo. Este grupo está compuesto por personas que se unen voluntariamente al proyecto por resultarle de interés y está literalmente distribuido por todo el mundo, solamente unido por Internet. Si se logra con éxito crear una base de usuarios, por un lado se tiene la seguridad de que se satisface una necesidad real y por el otro, estos usuarios pueden convertirse en excelentes colaboradores.

En el mundo del *software* libre muchos de los usuarios son *hackers* (aspecto heredado de la tradición UNIX) y esto es algo tremendamente beneficioso para el desarrollo del proyecto. Al estar disponible el código fuente del programa, estos *hackers* pueden contribuir en el desarrollo o en la depuración de los programas, diagnosticar problemas, sugerir correcciones e introducir mejoras.

Para facilitar los procesos de reclutamiento de colaboradores, interacción durante el desarrollo y distribución, se implementa una sede para el proyecto por medio de la Internet. Esto implica desarrollar un sitio *web* donde:

- cualquiera pueda obtener información acerca del proyecto e inscribirse como colaborador,
- se pueda hacer la distribución de las versiones preliminares y actualizaciones,
- se tenga un historial de las versiones liberadas para demostrar la evolución del *software*,
- se tenga acceso a documentación y soporte.,
- se pueda recibir contribuciones y
- se pueda recibir retroalimentación de los usuarios y reportes de errores.

Además se debe implementar un foro, ya sea a través de *newsgroups* o listas de correo para:

- hacer anuncios a los colaboradores,
- distribuir nuevas versiones para que sean evaluadas y
- proponer y evaluar soluciones que involucren a todo el equipo.

Durante el proceso de desarrollo se liberan periódicamente las versiones preliminares que resultan de integrar las contribuciones de los usuarios-desarrolladores e introducir las mejoras propuestas. Este proceso está a cargo del o de los líderes del proyecto, quienes evalúan las contribuciones y deciden cuáles incluir y cuáles no. Si una decisión es difícil de tomar, se pueden seguir varios pasos, pero siempre tomando en cuenta las opiniones de la base de usuarios.

El proceso de depuración se realiza en paralelo; es decir, al mismo tiempo que se liberan versiones preliminares con su código fuente disponible, los usuarios pueden identificar errores. Al tener a un grupo numeroso de personas que utilizan las versiones preliminares de un programa, los errores se hacen notorios. Como este grupo también puede observar el código de un programa, para más de alguno se hace evidente la razón del problema y alguien más puede proponer una solución. Algunas veces una misma persona realiza estas tres acciones pero lo más común es que sean tres personas diferentes: uno detecta el error, otro descubre la razón de éste y otro más lo soluciona.

Por último se da el proceso de distribución del *software*. Lo más común es implementar un sitio *web* desde el cual cualquiera pueda obtener una copia del programa, su código fuente y documentación.

Pero también proliferan las empresas que hacen negocios al distribuir *software* libre. Estas empresas preparan CDs en los que recopilan un conjunto de *software* libre junto con documentación, y los venden como cualquier otro CD. Las empresas consumidoras de *software* prefieren esta última opción, ya que de esta manera obtienen *software* de excelente calidad y el soporte que ofrece la compañía que les vendió el *software*.

2.4 Propiedad de proyectos de *software* libre

Definir la propiedad de un proyecto de *software* libre podría ser complicado, especialmente porque existe dentro de una cultura ajena a las relaciones de poder y a las restricciones legales.

La clave para conocer al propietario de un proyecto de *software* libre no está en los desarrolladores sino en la percepción de los usuarios. El propietario de un proyecto de *software* libre es aquel que es reconocido por la comunidad como el que tiene el derecho exclusivo para distribuir versiones modificadas[11]. El propietario puede ser un individuo o una organización y los parches que pone a disposición del público son reconocidos como los oficiales.

La definición del *software* libre indica que cualquiera tiene la opción de tomar el código fuente de un determinado programa libre y modificarlo. Este *software* modificado puede ser para uso propio o para ser distribuido dentro de un grupo reducido, y esto no provoca conflicto respecto a la propiedad del proyecto.

[11] Cultivando la noósfera (http://www.geocities.com/jagem/noosfera.html). Febrero de 2001.

Es cuando las modificaciones son puestas a disposición del público que la propiedad se convierte en un punto en disputa. Existen, en general, tres formas de adquirir la propiedad de un proyecto de *software* libre. La más obvia es fundar el proyecto. Cuando un proyecto tiene un único mantenedor, el cual se encuentra activo, la comunidad no cuestiona quién es el dueño.

El segundo camino es que la propiedad de un proyecto sea concedida por parte del propietario anterior a alguien más. La comunidad del *software* libre acepta que los propietarios de un proyecto tienen el derecho a pasarlo a sucesores cuando ya no están dispuestos o no pueden invertir el tiempo necesario en el desarrollo y mantenimiento del proyecto.

La tercera forma de adquirir un proyecto es asumir la responsabilidad del mismo cuando el propietario ha desaparecido o no muestra interés. La comunidad exige que quien desea hacerse cargo de un proyecto huérfano haga el intento de conocer la opinión del propietario y asegurarse de que nadie más asumió la propiedad.

Tanto en la segunda como en la tercera opción, pero más especialmente en esta última, la comunidad espera que se haga el anuncio respectivo a la base de usuarios y luego se espere para ver si no hay ninguna objeción.

Es importante hacer una observación respecto a lo anterior. Estas reglas de propiedad no han sido establecidas por nadie en especial sino que han surgido como costumbres dentro de la comunidad del *software* libre. La mayoría no está consciente de ellas pero las han seguido consistentemente y esto ha contribuido a establecer la legitimidad de los propietarios de los proyectos de *software* libre.

2.5 Gestión de proyectos de *software* libre

Los objetivos de la gestión no varían, incluso tratándose de proyectos de *software* libre. Es decir, lo que se pretende es concluir el proyecto con éxito y eso implica desarrollar una serie de actividades para planificar, organizar y liderar el proyecto. Lo que sí cambia es la manera en que se desarrollan estas actividades y eso es lo que se considerará a continuación.

2.5.1 Gestión del personal en proyectos de *software* libre

Más que contar con las mejores herramientas, el éxito de un proyecto de *software* dependerá de la calidad de la gente que colabore en el mismo. Los objetivos de la gestión del recurso humano en un proyecto son integrar al equipo personal de calidad y luego retenerlos, mientras se motivan para asegurar el máximo rendimiento.

Para entender cómo se realiza la gestión del recurso humano en los proyectos de *software* libre, es preciso estar consciente de algunas particularidades del modelo de desarrollo de *software* libre en consideración. Muy pocas personas de las que colaboran en un proyecto de *software* libre tienen un contrato formal de trabajo con los propietarios del proyecto.

Las retribuciones económicas son aún más raras. Por otro lado, los colaboradores no están concentrados en una sola geografía sino al contrario, están localizados por todo el mundo, incluso el grupo de propietarios. La única manera en que interactúan es por medio de Internet. Pero posiblemente la característica más interesante es que cualquier persona potencialmente puede participar, aun sin importar si pertenece a otra organización.

Estas características hacen que los procesos de reclutamiento y selección, entrenamiento y motivación sean diferentes a los modelos tradicionales de desarrollo.

2.5.1.1 Reclutamiento y selección

Es imposible hacer un esfuerzo por parte de los propietarios del proyecto para contactar a cientos de *hackers* alrededor del mundo para integrarlos al equipo de colaboradores. Así que la manera más fácil de hacer el reclutamiento es a la inversa: dejar que los *hackers* contacten a los propietarios del proyecto. Para esto es de mucha utilidad tener herramientas en la página *web* del proyecto que permitan a los interesados conocer el proyecto y, si están interesados, contactar a los mantenedores del mismo. De esta manera, es importante que los visitantes del sitio *web* del proyecto puedan conocer:

- la descripción del proyecto,
- sus objetivos,
- a sus propietarios-líderes,
- información técnica y, lo más importante,
- la manera en que podrían colaborar.

La manera más común de iniciar el contacto con la comunidad es suscribiéndose a la lista de correo del proyecto y antes de contribuir absorber la mayor cantidad de información posible acerca del proyecto, leyendo su página *web*.

En lo que respecta a la selección de los colaboradores, realmente no se hace una selección de personal como tal. Los coordinadores del proyecto no seleccionan a las personas que aportan sino más bien sus aportes.

Conforme se vaya desarrolla el proyecto, se puede observar y filtrar a los usuarios que hacen las mejores contribuciones y a quienes mejor conocen el proyecto para pedirles que se hagan responsables de la coordinación de alguna sub-área del desarrollo.

Es importante hacer notar que no todas las contribuciones serán aceptadas. Se espera que quien aporte tenga suficiente cantidad de conocimiento técnico. Se espera, también, que conozca las costumbres implícitas de la comunidad a la cual quiere adherirse. Esto implicará muchas veces observar durante algún tiempo la manera en que se hace el desarrollo y la estructura propia de la organización.

2.5.1.2 Entrenamiento

La instrucción explícita es muy rara en la cultura del *software* libre. La instrucción de los colaboradores de un proyecto de *software* libre es realizada sin la ayuda de nadie y se da únicamente por el propio interés del individuo. Para lograr esto, el sitio *web* del proyecto debe incluir gran cantidad de recursos educativos para que los colaboradores puedan autoinstruirse. En lo que respecta a las habilidades técnicas, se da por supuesto que el colaborador tiene el suficiente conocimiento, ya que de lo contrario debe recurrir a recursos externos al proyecto para adquirirlo.

2.5.1.3 Motivación

Este es uno de los aspectos que llama más la atención de la cultura del *software* libre. ¿Cómo se va a motivar a alguien sin darle ninguna remuneración económica?

Aun así, sin obtener beneficios económicos por sus aportes, existen cientos de miles de programadores voluntarios que dedican parte de su tiempo al *software* libre y cada día se agregan más.

Por un lado, está la motivación para adoptar la filosofía del *software* libre, es decir, cuál es la razón por la cual muchos programadores participan en la creación de excelentes herramientas de *software* y luego las ceden para que cualquiera las use, sin recibir nada a cambio.

Por el otro lado, está la motivación para participar en un proyecto de *software* libre específico, es decir, cómo se mantiene motivado al alguien para que siga participando, haciendo contribuciones de calidad a un proyecto que es de su interés.

2.5.1.3.1 Las motivaciones de la comunidad de *software* libre

Los *hackers* que participan en proyectos de *software* libre no lo hacen persiguiendo algún tipo de riqueza material. Existen maneras, sin embargo, en que la actividad en el mundo del *software* libre ayuda a algunas personas a ganar dinero. Ocasionalmente, la reputación ganada en la cultura del *software* libre puede traer consigo ofertas económicamente significativas, tal como una oferta de trabajo bien remunerado, peticiones de consultoría u ofrecimientos para ser autor de un libro. Aún así, estos efectos colaterales son raros, por lo que no ofrecen una explicación a la motivación para participar en proyectos de *software* libre.

Lo que mejor explica este fenómeno es el prestigio. En primer lugar está el prestigio que adquiere el *hacker* entre la misma comunidad de *software* libre. El reconocimiento de sus semejantes es en sí una fuente de recompensa.

En segundo lugar, este prestigio se convierte en un buen camino para atraer la atención y colaboración de otros. Para quien tiene el prestigio de ser generoso, inteligente u otras buenas cualidades, es más fácil persuadir a otras personas de que saldrán beneficiadas si se asocian con él.

Por último, este prestigio puede ser reconocido fuera de la comunidad de *software* libre y se puede ganar status en otras áreas.

¿Pero de qué manera se gana prestigio dentro de la comunidad del *software* libre? Esto puede ser explicado si se considera a la comunidad del *software* libre como una *cultura de regalos*.

Los humanos tienen tres maneras de organizar sus agrupaciones. La forma más simple es el **mando jerárquico**, pero también existe la **economía de intercambio** y la **cultura de regalos**.

Cuando existe mando jerárquico, el traspaso de los bienes escasos es controlado por una autoridad central y este proceso es respaldado por la fuerza. En los mandos jerárquicos, el prestigio es atribuido a quien tiene acceso al poder coercitivo.

Nuestra sociedad es predominantemente una economía de intercambios. El traspaso de los bienes escasos es descentralizado y se realiza a través del canje y la cooperación voluntaria. En una economía de intercambio, el prestigio es determinado por el control de las cosas para usar o canjear.

El tercer modelo es muy poco conocido, excepto para los antropólogos. La cultura de regalos son adaptaciones, no debido a la escasez sino a la abundancia. La abundancia dificulta las relaciones de mando para sustentar e intercambiar relaciones.

En la cultura de regalos, el prestigio está determinado no por lo que se controla sino por lo que se entrega. La cultura del *software* libre es una cultura de regalos. Dentro de ella no existe la escasez ya que el *software* es compartido libremente. Esta abundancia crea una situación en la que la única medida de éxito es el prestigio entre los demás.

2.5.1.3.2 La motivación en un proyecto de *software* libre

Ya se ha indicado la importancia de tener una base de usuarios para que colaboren en el desarrollo del proyecto.

Si estos usuarios están lo suficientemente motivados, se puede esperar de ellos contribuciones de calidad. Ya se ha explicado por qué hay gente dispuesta a colaborar con el *software* libre, pero ¿cómo hacer que participen en un proyecto en específico?

- Los usuarios deben ser tratados como colaboradores.
- Deben saber que son escuchados sus comentarios y retroalimentación.
- Debe buscarse su opinión en aspectos de diseño.
- Debe admitirse cuando un usuario tiene una mejor idea, procediendo a su implementación.
- Los reconocimientos por las aportaciones deben hacerse a quien corresponde y en el tiempo preciso.
- Las aportaciones de los usuarios (código, detección y corrección de errores, ideas, etc.) deben implementarse lo más pronto posible y hacerlas notorias a la comunidad.
- Debe mantenérseles constantemente informados del progreso del proyecto.
- Si es posible, debe escogerse usuarios para delegarles el mantenimiento de un área del proyecto.

2.5.1.4 Participantes

En todo proyecto de *software* libre, existen varios grupos de personas interactuando. Estos grupos pueden clasificarse de la siguiente manera[12]:

[12] La empresa ante el software libre: La empresa basada en software libre (http://oasis.dit.upm.es/~jantonio/documentos/empresa/empresa-7.html Mayo de 2001.

2.5.1.4.1 Propietarios

Los propietarios son quienes tienen el derecho exclusivo de distribuir versiones modificadas del proyecto. Pueden ser fundaciones, universidades, grupos de *hackers*, empresas o individuos.

Los propietarios, aparte de proporcionar canales fiables de venta y distribución, deben proveer los recursos tecnológicos para el desarrollo del proyecto, los recursos financieros y la cobertura legal.

2.5.1.4.2 Coordinadores

Los coordinadores (o el coordinador) son los líderes del proyecto. Es su responsabilidad mantener activo el proyecto, coordinar el trabajo de los colaboradores, manejar la comunicación y crear las versiones que serán distribuidas.

En la mayoría de los proyectos de *software* libre, la organización (o el individuo) propietaria juega el papel de la coordinación. Esto debido a que la organización se creó en torno al proyecto y este es su única razón de existencia.

Al inicio del proyecto, las habilidades técnicas son las más indispensables, pero una vez el proyecto alcanza la madurez, las habilidades de relaciones públicas y análisis de sistemas son requeridas.

Los líderes del proyecto deben ser buenos oradores y atraer la atención del público. Posiblemente les quede poco tiempo para programar, pero no pueden descuidar la supervisión del desarrollo de sus colaboradores.

2.5.1.4.3 Mantenedores

Los colaboradores-mantenedores son responsables de áreas del proyecto. Se espera que tengan un fuerte nivel técnico y capacidad de abstracción. Ellos van a realizar el trabajo de integración, a recopilar la información que les llegue de Internet y, en unión con el coordinador (atendiendo a la comunidad Internet), decidirán la política a seguir.

2.5.1.4.4 Voluntarios

Los voluntarios son la fuerza de choque del desarrollo del proyecto. Estos usuarios-desarrolladores colaboran con el código, depuración y retroalimentación. Es indispensable tener una buena base de voluntarios ya que el éxito del proyecto depende de esto. Cada voluntario debe estar permanentemente al tanto del progreso del desarrollo, debe sentirse copartícipe del proyecto. Debe atenderse a su retroalimentación, tal como sugerencias, quejas y opiniones.

2.5.1.4.5 Usuarios finales

Los usuarios finales, al contrario de los usuarios-desarrolladores, no utilizan el código fuente sino los paquetes binarios. Mientras los voluntarios buscan eficiencia, el usuario final busca funcionalidad.

Es clave darles a los usuarios un espacio para retroalimentación ya que estos hacen los comentarios más oportunos acerca de la apariencia, el modo de funcionamiento y las funcionalidades a añadir. Por otro lado, son las fuentes más seguras de ingresos económicos ya que compran documentación, asisten a cursos y solicitan soluciones a la medida.

2.5.1.5 Estructura del proyecto

Un proyecto de *software* libre puede tener tres diferentes estructuras.

El primer caso es en el que existe un propietario-líder. Es decir, solamente hay una persona a cargo del proyecto, ya sea porque él lo inició, le fue transferido por un propietario anterior o había sido abandonado y él lo recuperó. Además, esta persona es la única que hace el mantenimiento, es decir, recoge los aportes de los contribuyentes y los consolida en cada versión preliminar.

El segundo caso es donde existe un único propietario-líder y varios mantenedores. La costumbre favorece este modelo y ha funcionado en proyectos tan grandes como el desarrollo del *kernel* de Linux.

El tercer caso es en el que existe más de un propietario-líder. Esto se puede dar cuando el proyecto no lo ha iniciado o heredado una sola persona. Pero también puede darse el caso en el que a un usuario se le dé autoridad y responsabilidad sobre una parte importante del proyecto. En este caso las decisiones pueden tomarse de dos maneras. La primera es por medio de votación (como en el caso de *Apache*) y la segunda es delegando la autoridad del proyecto periódicamente entre los propietarios (como en el caso de *Perl*).

2.5.1.6 Líderes de equipos

En un proyecto de *software* libre, el surgimiento de líderes de equipos puede darse de dos maneras. Por un lado están los propietarios del proyecto. Quienes inician un proyecto o adquieren la responsabilidad de uno que ya ha sido iniciado por alguien más, inmediatamente asumen un liderazgo. Pero también pueden nombrarse líderes dentro del grupo de usuarios para coordinar el mantenimiento de una parte del desarrollo.

¿Qué habilidades se esperan de estas personas en función del rol que desarrollan?

- Capacidad de motivar a los participantes para hacer contribuciones de calidad,
- habilidades de organización para distribuir los recursos de la manera en que rindan mejor,
- habilidades de comunicación y solución de conflictos, y
- habilidades para reconocer las buenas ideas de los demás.

Nótese que no se espera que el líder forzosamente tenga habilidades de resolución de problemas técnicos. Más bien se espera que sea capaz de hacer que "varias cabezas piensen mejor que una".

Entre las actividades que un líder debe desempeñar están:

- Comunicación en dos vías con los usuarios. Por un lado, informarles del estado del proyecto y los aspectos pendientes. Por el otro, recibir sus comentarios y retroalimentación;
- recibir las contribuciones y evaluarlas para escoger las mejores,
- resolver los conflictos que pudieran surgir,
- mantener un registro de reconocimientos y hacerlo público,
- preparar los paquetes de las versiones preliminares,
- asignar los derechos de propiedad parciales, es decir, escoger usuarios y darles el control del desarrollo de algún módulo del proyecto.

2.5.1.7 Solución de conflictos

Entre los conflictos en proyectos de *software* libre podemos identificar cuatro problemas principales:

- ¿Quién realiza las decisiones del proyecto?
- ¿Quién obtiene crédito o culpabilidad por algo?
- ¿Cómo reducir la duplicación de esfuerzo y evitar que el proyecto se divida?
- ¿Qué es lo correcto, técnicamente hablando?

Estos conflictos van a ser más complejos en las estructuras donde la autoridad esté distribuida y existan derechos de propiedad parcial. Normalmente los aspectos técnicos de diseño son los que representan los riesgos más obvios de conflicto interno.

Estos conflictos son más sencillos de resolver si los líderes del proyecto entienden que a quien le sea asignado alguna parte del desarrollo, es responsable por la misma. Otra manera de resolver estos conflictos es por madurez. Si dos contribuyentes o grupos de contribuyentes tienen una disputa, y dicha disputa no puede ser resuelta bajo el criterio anterior, entonces el que ha aportado más es el que tiene prioridad.

Cuando un conflicto sobrepasa estos dos filtros, entonces debe imponerse la autoridad que el grupo de propietarios haya elegido, sea esta votación o un liderazgo rotativo.

2.5.1.8 Comunicación y coordinación

El desarrollo de un proyecto de gran tamaño causa dificultad para coordinar a los miembros del equipo. Esta dificultad es mayor cuando los colaboradores están muy dispersos geográficamente a pesar de que la tecnología de telecomunicaciones provee de dos herramientas que permiten comunicar a un grupo de personas simultáneamente sin importar su localización geográfica: video conferencia y conferencia telefónica.

Estas tecnologías no son muy efectivas para la comunicación de los equipos de proyectos de *software* libre. Por un lado, el costo económico que representan no puede ser absorbido por los proyectos. Pero por el otro lado, es imposible sincronizar a gente que se rige por distintos horarios y que normalmente tienen otras ocupaciones a las cuales dedican la mayor parte de su tiempo.

Lo que se necesitan son herramientas que sean menos costosas y que no exijan sujetarse a un horario por parte de los colaboradores, pero que permitan una excelente comunicación en dos vías. Los proyectos de *software* libre disponen de tres herramientas que cumplen con estos requisitos:

2.5.1.8.1 Listas de correo electrónico

Este es un método que permite enviar información a mucha gente al mismo tiempo de una manera sencilla y barata. Permite que todas las personas interesadas en tener algún tipo de información la reciban y puedan almacenarla para su uso posterior.

Funciona de la siguiente manera: cuando alguien muestra interés por el proyecto, puede suscribirse a la lista de correo electrónico, llenando una forma en la página *web* del proyecto, en la cual indica sus datos generales, dirección de correo electrónico e interés respecto al proyecto.

Los propietarios del proyecto clasifican las suscripciones en función de la naturaleza de los suscritos, es decir, usuarios no colaboradores que están interesados en saber acerca de nuevas versiones y recursos disponibles; usuarios contribuyentes o gente que colabora con la depuración, escribe código o hace sugerencias y por último los mantenedores de alguna parte del proyecto.

Cuando alguna información debe ser conocida por todo un grupo, se envían mensajes de correo electrónico a quienes estén suscritos en ese momento a la lista. Esta información podría ser la noticia de la liberación de una nueva versión o herramienta, la petición para corregir algunos errores de algún programa, cambios en el liderazgo, etc.

Es importante tomar en cuenta una última observación. La página *web* debe permitir darse de baja a los miembros de las listas de correo electrónico. Una manera como se puede conocer que el proyecto ya es lo suficiente maduro es cuando los colaboradores empiezan a darse de baja de las listas de correo. Esto sucede cuando ellos consideran que ya no pueden hacer ningún aporte al proyecto en vista de su madurez y pueden aportar en otra parte.

2.5.1.8.2 *Newsgroups*

Los grupos de noticias son una variedad de correo electrónico que no está muy difundida. Son espacios de intercambio de información que funcionan como una cartelera de mensajes.

La mayoría de los programas clientes de correo electrónico permiten conectarse a grupos de noticias, recibir comentarios de otras personas que participan en ellos y aportar mensajes propios.

Por medio de esta tecnología es posible leer los mensajes dejados por otros participantes, dejar un mensaje, responder al grupo con alguna opinión o responder individualmente a algún miembro[13].

[13] http://www.ciudad.com.ar/ar/portales/tecnologia/nota/0,1357,4450,00.html. Mayo de 2001.

Esta herramienta permite implementar centros de discusión. Por un lado, es posible incluir textos largos para ser analizados cuidadosamente, pero también la estructura de presentación de la información hace más sencillo el seguimiento de varias discusiones[14].

2.5.1.8.3 Páginas *web*

Las páginas *web* son una excelente herramienta para consulta. Permiten tener a disposición mayor cantidad de información que el correo electrónico o grupos de noticias. La ventaja de las páginas *web* es que quien las consulta puede seleccionar solamente la información que le interesa. Por este medio puede ponerse a disposición de la comunidad manuales, ayuda técnica, herramientas, noticias, bibliografía, etc.

Usando las listas de correo electrónico, los grupos de noticias y las páginas *web*, se tienen herramientas para distribuir información, manejar discusiones y centros de consulta.

2.5.2 El problema

El primer paso de un proyecto de *software* libre es examinar el problema. Debe hacerse un esfuerzo cuidadoso por plantearlo correctamente. Esto será de gran beneficio para todos los participantes del proyecto ya que tendrán en mente con exactitud qué es lo que están resolviendo mientras trabajan.

[14] Portal de Ciudad Internet (http://www.ciudad.com.ar/ar/portales/tecnologia/nota/0,1357,4462,00.html). Mayo de 2001.

Ahora bien, a lo largo del proyecto se debe estar atento para descubrir si el planteamiento original del problema es el correcto. La historia del *software* libre tiene varios ejemplos de proyectos que, al hacer una segunda evaluación del problema que intentan resolver, decidieron replantearlo y consiguieron grandes beneficios. Eric S. Raymond dice: "Frecuentemente, las soluciones más innovadoras y espectaculares provienen de comprender que la concepción del problema era errónea"[15].

2.5.2.1 Los objetivos del proyecto

Los proyectos de *software* libre nacen para responder a necesidades concretas y demostradas de una o varias personas. Los objetivos del proyecto, por tanto, deben apuntar a satisfacer esas necesidades. Como primer paso de la definición del problema, deben establecerse claramente estas necesidades y sus correspondientes soluciones. La implementación de estas soluciones se convierte en los objetivos del proyecto, las cuales deben ser entendidas por todos y deben poder medirse para saber cuándo ya se alcanzaron.

Desde el momento en que se decide que el *software* va a ser libre, los usuarios-colaboradores se convierten en los clientes a satisfacer. Es de suma importancia tomar en cuenta las sugerencias y peticiones de los usuarios, ya que esto puede convertirse en una gran ventaja para el proyecto. Estas sugerencias y opiniones son necesidades reales, que de satisfacerlas van a dar al *software* producido un rico mercado.

Dada la naturaleza dinámica de los proyectos de *software* libre, es posible que los objetivos cambien durante el desarrollo. Se debe estar abierto a esa posibilidad pero cuidar que no varíen mucho. De lo contrario el proyecto pierde su razón de ser y valdría la pena considerar si se continúa o no.

---

[15] La Catedral y el Bazar de Eric S. Raymond (http://www.sindominio.net/biblioweb/telematica/catedral.html). Noviembre del 2000.

2.5.2.2 El ámbito del proyecto

El siguiente paso es definir el ámbito que rodea al proyecto. Se debe entender las características del contexto en el cual funcionará el *software*, es decir, el sistema mayor al cual va a pertenecer, el mercado que va a satisfacer, los usuarios que lo van a operar.

Tras entender las características del contexto, se deben establecer las restricciones y limitaciones que se dan como consecuencia. También conviene saber con anticipación el flujo de información, es decir, cuáles datos son requeridos como entrada del sistema y qué información genera el *software*.

Los últimos aspectos del ámbito se refieren a la función y al rendimiento. Se debe conocer cuáles funciones realiza el programa para procesar la información y si existen características de rendimiento específicas.

El ámbito del proyecto debe ser comprendido en su totalidad tanto por los gestores como por el personal técnico. En consecuencia, sus enunciados deben establecerse explícitamente junto con sus limitaciones y factores de reducción de riesgo[16].

2.5.2.3 Evaluación de alternativas

El siguiente paso es considerar todas las soluciones alternativas disponibles para escoger el mejor enfoque para acometer el desarrollo del proyecto. Vale la pena revisar minuciosamente el ciberespacio con el ánimo de descubrir los proyectos similares.

[16] Orlando Rafael Romero Ávalos, José Luis Sánchez Lemus. **Gestión de proyectos de software, estimación y planificación de tiempos y costos de un proyecto de software.** (Guatemala: Universidad de San Carlos de Guatemala, 1998) p. 17.

Para cada uno de los proyectos encontrados es importante considerar qué soluciones implementan y cómo se adaptan estas soluciones al contexto definido anteriormente. Quizás se encuentre disponible otro proyecto de *software* libre que ya satisface las necesidades que dieron vida al proyecto o se pueda tomar ideas (y código) o mejor aún, evolucionar a partir del mismo.

2.5.2.4 Descomposición del problema

Como último paso antes de comenzar la planificación del proyecto se procede a descomponer el problema inicial en problemas menores más manejables.

Las funciones establecidas durante el proceso de definición del ámbito se utilizan como base para realizar la descomposición que se aplica en dos áreas principales: la funcionalidad y el proceso que se empleará para entregarse.

Cada parte puede ser aún más refinada si este proceso contribuye a hacer más fácil la planificación.

2.5.3 El proceso

Las etapas genéricas de análisis y definición, desarrollo y mantenimiento, son aplicables a todo tipo de proyecto de *software*[17]. Normalmente la dificultad en esta área de la gestión es establecer el modelo de proceso apropiado para la etapa de ingeniería. Esto es, escoger uno de los siguientes modelos más comunes:

- secuencial lineal

[17] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (3ª Edición; Editorial McGraw-Hill, 1998) p. 35.

- prototipos
- RAD
- incremental
- espiral
- ensamblaje de componentes
- métodos formales
- técnicas de cuarta generación

En los proyectos de *software* libre no se toma partido por uno de estos modelos en especial. Las características del *software* libre han permitido el surgimiento de un nuevo modelo de desarrollo: el modelo abierto. Este presenta características que se pueden observar en varios de los modelos mencionados, pero es en sí un modelo diferente y novedoso. El siguiente capítulo tiene como objetivo exponer este modelo y evaluarlo.

3. DESARROLLO DE *SOFTWARE* LIBRE

3.1 Desarrollo de *software*

El proceso de desarrollo del *software* está compuesto por tres fases genéricas: la de definición, la de desarrollo y la de mantenimiento. Estas áreas genéricas existen independientemente del área de aplicación, tamaño o complejidad del proyecto.

3.1.1 Fase de definición

En ésta se define la información que se procesará, la función y rendimiento deseados, el comportamiento esperado del sistema, las interfaces necesarias, las restricciones existentes y los criterios de validación necesarios para obtener un sistema correcto[18].

Las tareas de esta fase se pueden clasificar en tres áreas[19]:

3.1.1.1 Análisis del sistema

Dado que el *software* a desarrollar formará parte de un sistema mayor, se empieza por conocer y establecer cuáles requisitos de este sistema serán satisfechos por el *software* a desarrollar. Se investiga cuáles son los componentes del sistema mayor, es decir, qué *software*, *hardware*, personas, procesos ya existen y qué papel desempeñan.

---

[18] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 18.
[19] José Alejandro Peña Rodríguez. **Planificación del recurso humano en proyectos de ingeniería de software.** (Guatemala: Universidad Mariano Gálvez de Guatemala) p. 14.

Seguidamente se establecen los componentes (también al nivel de *hardware*, *software*, personas y procesos) que será necesario agregar como consecuencia de la implementación del *software* y cómo se dará la interacción entre los componentes ya existentes y los nuevos.

Este análisis se realiza a un alto nivel y se diferencia del análisis de requisitos en el enfoque del estudio. Mientras el análisis del sistema se enfoca en el ámbito en el cual se implementará el *software*, el análisis de requisitos se enfoca en el *software* propiamente.

3.1.1.2 Planificación

Se definen las tareas necesarias para completar el sistema y con base en esto se hace una planificación temporal, se asignan recursos (personal, equipos, herramientas de *software*, espacios físicos) y se establecen costos.

La planificación permite escoger el curso de acción más viable en lo que respecta a tiempo y costo. Como resultado de este proceso se obtiene el plan del proyecto, el cual proporciona información acerca del orden en que se realizarán las tareas, la forma en que se asignarán recursos y la disponibilidad temporal de los mismos.

El plan del proyecto debe comunicar el ámbito y los recursos a los gestores del proyecto, al personal técnico y a los clientes. Debe además definir los riesgos y sugerir técnicas para prevenirlos. Aparte de definir el coste y la agenda, debe proporcionar el enfoque global del desarrollo para toda le gente involucrada en el proyecto[20].

[20] José Alejandro Peña Rodríguez. **Planificación del recurso humano en proyectos de ingeniería de software.** (Guatemala: Universidad Mariano Gálvez de Guatemala) p. 53.

3.1.1.3 Análisis de los requisitos

Se enfoca especialmente en el *software* a desarrollar, para el cual se especifica la función, rendimiento y el comportamiento de los programas, se indican las interfaces entre elementos y se establecen las restricciones de diseño[21].

Como resultado de este proceso se obtiene una lista de requisitos, los cuales pueden corresponder a una de las siguientes tres categorías[22]:

- **Requisitos normales**. Aquellos que se declaran como metas y objetivos del producto, los cuales le darán su distintivo.

- **Requisitos esperados**. Son los que son implícitos y aplican a cualquier software, tales como la facilidad de interacción, buen funcionamiento, fiabilidad y otros.

- **Requisitos innovadores**. Son los que van más allá de las expectativas normales y hacen que el producto sea más satisfactorio de lo esperado.

3.1.2 Fase de desarrollo

En la fase de desarrollo se define el diseño de las estructuras de datos, se especifica cómo se implementará la función como una arquitectura del *software*, cómo se implementarán los detalles procedimentales, y cómo se caracterizarán las interfaces. También se traduce el diseño en un lenguaje de programación y se hace el proceso de prueba. Esta fase consiste en tres tareas técnicas: diseño del *software*, generación de código y prueba del *software*.

---

[21] Juan Carlos Soria Oliva. **Comparación del desarrollo de sistemas de software utilizando la metodología tradicional versus la aplicación de prototipos rápidos estructurados.** (Guatemala: Universidad de San Carlos de Guatemala: 1992) p. 5.

3.1.2.1 Diseño del *software*

El diseño es el núcleo técnico del proceso del desarrollo del *software*, en el cual se traducen los requerimientos en un conjunto de representaciones que describen la estructura de datos, la arquitectura del sistema, el procedimiento algorítmico y las características de las diferentes interfaces[23].

El diseño del *software* se realiza en dos pasos. En primer lugar, el diseño preliminar donde los requerimientos se traducen en datos y la arquitectura del *software*. El segundo paso se enfoca hacia los refinamientos de la representación arquitectónica, lo que conduce a una estructura de datos detallada y a representaciones algorítmicas[24].

La **estructura de datos** es una representación de la relación lógica entre los elementos individuales de datos. La organización y complejidad de la estructura de datos son ilimitadas, sin embargo, existen ciertas estructuras clásicas que son la base para construir estructuras más complejas, tales como los elementos escalares, los vectores secuenciales y n-dimensionales y las listas enlazadas[25].

La **arquitectura del software** alude a la estructura global del *software* y la manera en que ésta proporciona integridad conceptual a un sistema. En su forma más simple, la arquitectura es la estructura jerárquica de los componentes del programa (módulos), la manera de interactuar de estos componentes y la estructura de datos usada por estos componentes[26].

---

[22] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 18.
[23] José Alejandro Peña Rodríguez. **Planificación del recurso humano en proyectos de ingeniería de software.** (Guatemala: Universidad Mariano Gálvez de Guatemala) p. 9.
[24] Juan Carlos Soria Oliva. **Comparación del desarrollo de sistemas de software utilizando la metodología tradicional versus la aplicación de prototipos rápidos estructurados.** (Guatemala: Universidad de San Carlos de Guatemala: 1992) p. 8-9.
[25] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 237-238.
[26] Ibid., p. 235

El **procedimiento algorítmico** define la jerarquía de control. Se centra en los detalles de procesamiento de cada módulo individualmente. Debe incluir las secuencias de acontecimientos, puntos exactos de decisión, operaciones repetitivas e incluso la organización / estructura de datos[27].

El **diseño de interfaz** describe cómo se comunica el *software* consigo mismo, con los sistemas que operan con él y con los operadores que lo emplean. Una interfaz implica un flujo de información; por lo tanto, los diagramas de flujos de datos y control proporcionan la información necesaria para el diseño de la interfaz[28].

3.1.2.2 Generación de código

Durante el proceso de codificación se traduce el diseño elaborado previamente a una serie de instrucciones para ser ejecutadas por la computadora en un determinado orden[29].

3.1.2.3 Pruebas del *software*

Luego de haber terminado la codificación, la siguiente actividad es asegurarse que el *software* creado se comporta de la manera esperada. Si no sucediera de esa manera, es necesario descubrir las razones de este comportamiento no deseado, el cual podría deberse a defectos existentes en la función, lógica o implementación[30].

---

[27] Ibid., p. 238
[28] Ibid., p. 230
[29] José Alejandro Peña Rodríguez. **Planificación del recurso humano en proyectos de ingeniería de software.** (Guatemala: Universidad Mariano Gálvez de Guatemala) p. 15.
[30] José Alejandro Peña Rodríguez. **Planificación del recurso humano en proyectos de ingeniería de software.** (Guatemala: Universidad Mariano Gálvez de Guatemala) p. 15.

Las pruebas pueden agruparse en 5 clases[31]:

- **Prueba de procedimientos**. La prueba de procedimientos o unidades es el nivel básico donde se prueban los procedimientos que componen un módulo.

- **Prueba de módulos**. Dado que un módulo se compone de varias funciones que interactúan entre sí, debe asegurarse que esta cooperación funciona correctamente. Este proceso es más efectivo si el módulo se prueba como una unidad aislada sin la presencia de otros módulos del sistema.

- **Prueba del sistema**. Esta es la prueba de integración. En esta etapa se van uniendo los módulos progresivamente y se evalúa su interacción. El objetivo de esta etapa es probar las interfaces y cuando el sistema ha sido integrado completamente, se evalúa si proporciona las funciones especificadas en los requisitos.

- **Prueba de aceptación**. En esta prueba, el *software* se introduce en el ambiente donde va a operar y se prueba con datos reales. En esta etapa se descubren errores en la definición de requisitos, ya que estos pueden no reflejar las características y rendimiento reales esperados por los usuarios.

Como parte del proceso de prueba debe darse el proceso de depuración. Este consiste en detectar la causa y localización del error y corregir el código incorrecto (en la fase de mantenimiento). Esto implica, definitivamente, hacer pruebas otra vez.

3.1.3 Fase de mantenimiento

[31] Ian Sommerville. **Ingeniería de software.** (Editorial Addison-Wesley Iberoamericana) p. 192-193.

Esta fase debe su existencia al cambio que va asociado con la corrección de errores, las adaptaciones requeridas a medida que evoluciona el entorno del *software* y mejoras que deben ser incluidas.

Esta fase vuelve a aplicar los pasos de las fases de definición y desarrollo, pero en el contexto del *software* ya existente. Existen cuatro tipos de cambios por los cuales se realiza el mantenimiento[32]:

- **Corrección**. El mantenimiento correctivo modifica el *software* para corregir sus defectos.

- **Adaptación**. Con el paso del tiempo, es muy probable que cambie el entorno original en el cual se desarrolló el *software*. El mantenimiento adaptivo modifica el *software* para ajustarlo a los cambios en el entorno.

- **Mejora**. Conforme se usa el *software*, los usuarios pueden descubrir funciones adicionales que van a producir beneficio. El mantenimiento perfectivo modifica el *software* para implementar requisitos adicionales a los originales.

- **Prevención**. El *software* de computadora se deteriora debido al cambio, por esto el mantenimiento preventivo modifica el *software* para que se pueda corregir, adaptar y mejorar más fácilmente.

3.2 Modelos de desarrollo

[32] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 237-238.

Durante el proceso del desarrollo de *software* se incorpora una estrategia, la cual depende de la naturaleza del proyecto y de la aplicación, los métodos y las herramientas a utilizarse, así como de los controles y entregas que se requieren. Esta estrategia se conoce como **modelo de proceso** o **paradigma de ingeniería del software**.

A continuación se exponen diferentes modelos de proceso.

3.2.1 Modelo secuencial lineal

Es llamado algunas veces **ciclo de vida clásico** o **modelo en cascada**. Sugiere un enfoque sistemático, secuencial del desarrollo del *software* que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento. Aunque la propuesta original incluía ciclos para retroalimentación, la mayoría de organizaciones que aplica este modelo lo hacen como si fuera estrictamente lineal.

3.2.1.1 Actividades

Modelado según el ciclo de ingeniería convencional, el modelo secuencial lineal acompaña a las siguientes actividades.

3.2.1.1.1 Ingeniería y modelado de sistemas

El trabajo comienza estableciendo requisitos de todos los elementos del sistema y asignando al *software* algún subgrupo de estos requisitos. Esta visión del sistema es esencial cuando el *software* se debe interconectar con otros elementos como *hardware*, personas y bases de datos.

La ingeniería y el análisis de sistemas acompañan a los requisitos que se recogen en el nivel de sistema con una pequeña parte de análisis y diseño. La ingeniería de información acompaña a los requisitos que se recogen en el nivel estratégico de empresa y en nivel del área de negocio.

3.2.1.1.2 Análisis de los requisitos del *software*

El proceso de reunión de requisitos se intensifica y se centra especialmente en el *software*.

Para comprender la naturaleza de los programas a construirse, debe comprenderse el dominio de la información del *software*, así como la función requerida, comportamiento, rendimiento e interconexión. Estos datos se documentan y se repasa los requisitos del sistema y del *software*.

3.2.1.1.3 Diseño

El diseño es realmente un proceso de muchos pasos que se centra en cuatro atributos distintos de un programa: estructura de datos, arquitectura del *software*, representaciones de interfaz y detalle procedimental. El proceso de diseño traduce requisitos en una representación del *software* que se pueda evaluar por calidad antes de que comience la generación de código. Al igual que los requisitos, el diseño se documenta y se hace parte de la configuración del *software*.

3.2.1.1.4 Generación de código

El diseño se traduce en forma legible por la máquina. El paso de generación de código lleva a cabo esta tarea. Si se lleva a cabo el diseño en una forma detallada, la generación de código se realiza mecánicamente.

3.2.1.1.5 Pruebas

Una vez se ha generado código, comienzan las pruebas del programa. Un proceso de prueba se centra en los procesos lógicos internos del *software*, asegurando que todas las sentencias se han comprobado, y en los procesos externos funcionales; es decir, la realización de pruebas para la detección de errores y asegurar de que la entrada definida produzca resultados reales de acuerdo con los resultados requeridos.

3.2.1.1.6 Mantenimiento

El *software* indudablemente sufrirá cambios después de ser entregado al cliente. Se producirán cambios porque se han encontrado errores, porque debe adaptarse para acoplarse a los cambios de su entorno externo o porque el cliente requiere mejoras funcionales o de rendimiento. El mantenimiento vuelve a aplicar cada una de las fases precedentes a un programa ya existente y no a uno nuevo.

3.2.1.2 Evaluación

El modelo secuencial lineal es el paradigma más antiguo y más extensamente utilizado en la ingeniería del *software*. Sin embargo, existen ciertos problemas con su aplicación:

- Los proyectos reales raramente siguen el modelo secuencial que propone el modelo. Aunque el modelo lineal puede acoplar interacción, lo hace indirectamente. Como resultado, los cambios pueden provocar confusión cuando el equipo del proyecto comienza.

- Es muy difícil conocer todos los requisitos al inicio. El modelo lo requiere y tiene dificultades a la hora de acomodar la incertidumbre natural al comienzo de muchos proyectos.

- No es posible tener versiones de trabajo de los programas hasta que el proyecto esté muy avanzado. Un grave error puede ser desastroso si no se detecta hasta que se revisa el programa.

Cada uno de estos problemas es real. Sin embargo, el paradigma del ciclo de vida clásico tiene un lugar definido e importante en el trabajo de la ingeniería del *software*. Pese a tener debilidades, es significativamente mejor que un enfoque hecho al azar para el desarrollo del *software*[33].

3.2.2 El modelo de construcción de prototipos

Un cliente a menudo define un conjunto de objetivos generales para el *software*, pero no identifica los requisitos detallados de entrada, procesamiento o salida. En otros casos, el responsable del desarrollo del *software* puede no estar seguro de la eficacia de un algoritmo, de la capacidad de adaptación de un sistema operativo o de la forma en que debería tomarse la interacción hombre-máquina. En estas y otras situaciones de incertidumbre, un paradigma de construcción de prototipos puede ofrecer el mejor enfoque.

3.2.121 Actividades

[33] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 22-24.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales del *software*, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición.

Entonces aparece un **diseño rápido**, el cual se centra en una representación de esos aspectos del *software* que serán visibles para el usuario/cliente. El diseño rápido lleva a la construcción de un prototipo.

El prototipo es evaluado por el cliente/usuario y utilizado para refinar los requisitos del *software* a desarrollar. La interacción ocurre cundo el prototipo satisface las necesidades del cliente, a la vez que permite que el desarrollador comprenda mejor lo que se quiere hacer.

Lo ideal sería que el prototipo sirviera como un mecanismo para identificar los requisitos del *software*. Si se construye un prototipo de trabajo, el desarrollador intenta hacer uso de los fragmentos del programa ya existentes o aplica herramientas que permiten generar rápidamente programas de trabajo.

Cuando ha terminado el proceso de identificar los requisitos, el sistema construido seguramente no se podrá utilizar, ya que será lento, demasiado grande o torpe en su uso. Por lo tanto el siguiente paso es comenzar de nuevo y construir una versión rediseñada que resuelva todos estos problemas.

3.2.2.2 Evaluación

El desarrollo de prototipos puede presentar los siguientes problemas:

- El cliente ve lo que parece ser una versión de trabajo del *software*, sin saber que con la urgencia de hacerlo funcional no se ha tenido en cuenta la calidad del *software* global o la facilidad de mantenimiento a largo plazo. Cuando se informa de que el producto se debe construir otra vez para que se puedan mantener los niveles altos de calidad, el cliente no lo entiende y pide que se apliquen "unos pequeños ajustes" para que se pueda hacer el prototipo un producto final. Con demasiada frecuencia, la gestión del desarrollo de *software* es muy lenta.

- El desarrollador a menudo hace compromisos de implementación para hacer que el prototipo funcione rápidamente. Se puede utilizar un sistema operativo o lenguaje de programación inadecuado simplemente porque está disponible y porque es conocido. Un algoritmo eficiente se puede implementar simplemente para demostrar la capacidad. Después de algún tiempo, el desarrollador debe familiarizarse con estas selecciones, y olvidarse de las razones por las que son inadecuadas. La selección menos ideal ahora es una parte integral del sistema.

Aunque pueden surgir problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería de *software*. La clave es definir las reglas del juego al comienzo, es decir, el cliente y el desarrollador deben ponerse de acuerdo en que el prototipo se construya para servir como un mecanismo de definición de requisitos. Entonces se descarta (al menos en parte), y se realiza la ingeniería del *software* con una visión hacia la calidad y la facilidad de mantenimiento[34].

3.2.3 El modelo DRA

[34] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 24-25.

El desarrollo rápido de aplicaciones (DRA) es un modelo de proceso del desarrollo del *software* lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. El modelo DRA es una adaptación a **alta velocidad** del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando el enfoque de construcción basado en componentes. Si se comprenden bien los requisitos y se limita el ámbito del proyecto, el proceso DRA permite al equipo de desarrollo crear un sistema completamente funcional dentro de períodos cortos de tiempo.

Si una aplicación de gestión puede modularse de forma que permita completarse cada una de las funciones principales en menos de tres meses, es un candidato del DRA. Cada una de las funciones pueden ser afrontadas por un equipo DRA diferente y ser integradas en un solo conjunto.

3.2.3.1 Actividades

Cuando se utiliza para aplicaciones de sistemas de información, el enfoque DRA comprende las siguientes fases:

3.2.3.1.1 Modelado de gestión

El flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Adónde va la información? ¿Quién la procesa?

3.2.3.1.2 Modelado de datos

El flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar el sistema al cual va a pertenecer el *software*. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos.

3.2.3.1.3 Modelado del proceso

Los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir o recuperar un objeto de datos.

3.2.3.1.4 Generación de aplicaciones

El DRA asume la utilización de técnicas de cuarta generación. En lugar de crear el *software* con lenguajes de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas automáticas para facilitar la construcción del *software*.

3.2.3.1.5 Pruebas y entrega

Como el proceso DRA enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.

3.2.3.2 Evaluación

Al igual que todos los modelos de proceso, el enfoque DRA tiene inconvenientes.

- Para proyectos grandes aunque por escalas, el DRA requiere recursos humanos suficientes como para crear el número correcto de equipos DRA.

- El DRA requiere clientes y desarrolladores comprometidos en las rápidas actividades necesarias para completar un sistema en un marco de tiempo abreviado.

No todos los tipos de aplicaciones son apropiados para el DRA. Si un sistema no se puede modularizar adecuadamente, la construcción de los componentes necesarios para DRA será problemático.

Si está en juego el alto rendimiento, y se va a conseguir el rendimiento convirtiendo interfaces en componentes de sistemas, el modelo DRA puede que no funcione. DRA no es adecuado cuando los riesgos técnicos son elevados. Esto ocurre cuando una nueva aplicación hace uso de tecnologías nuevas, o cuando el nuevo *software* requiere un alto grado de interoperatividad con programas de computadoras ya existentes[35].

3.2.4 El modelo incremental

El modelo incremental combina elementos del modelo lineal secuencial (aplicados repetitivamente) con la filosofía interactiva de construcción de prototipos. Este modelo aplica secuencias lineales de forma sorprendente de la misma forma que progresa el tiempo en el calendario. Cada secuencia lineal produce un incremento del *software*.

3.2.4.1 Actividades

[35] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 25-26.

Cuando se utiliza un modelo incremental, el primer incremento es a menudo un **producto esencial**. Es decir, se afrontan los requisitos básicos, pero muchas funciones suplementarias (algunas conocidas, otras no) quedan sin extraer. El cliente utiliza el producto central (o sufre revisión detallada). Como un resultado de utilización o evaluación, se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto central a fin de cumplir mejor las necesidades del cliente y la entrega de funciones y características adicionales. Este proceso se repite siguiendo la entrega de cada incremento, hasta que se elabore el producto completo.

3.2.4.2 Evaluación

El modelo de proceso incremental, como la construcción de prototipos y otros enfoques evolutivos, es interactivo por naturaleza. Pero a diferencia de la construcción de prototipos, el modelo incremental se centra en la entrega de un producto operacional en cada incremento. Los primeros incrementos son versiones “desmontadas” del producto final, pero proporcionan la capacidad que sirve al usuario y también proporciona una plataforma de evaluación por parte del usuario.

El desarrollo incremental es particularmente útil cuando la dotación de personal no está disponible para una implementación completa durante la fecha límite de gestión que se haya establecido para el proyecto. Los primeros incrementos se pueden implementar con menos personas. Si el producto central es bien recibido, se puede añadir más personal (si se requiere) para implementar el siguiente incremento. Además, los incrementos se pueden planear para gestionar riesgos técnicos. Así, si un sistema requiere la disponibilidad de ciertos recursos cuya fecha de entrega sea incierta, podría planificarse los primeros incrementos de modo que se evite la utilización de estos recursos, pero se entregue una funcionalidad parcial a los usuarios sin un retraso exagerado[36].

3.2.5 El modelo en espiral

El modelo en espiral es un modelo de proceso de *software* evolutivo que acompaña la naturaleza interactiva de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial. Se proporciona el potencial pare el desarrollo rápido de versiones incrementales de *software*.

En el modelo espiral, el *software* se desarrolla en una serie de versiones incrementales. Durante las primeras iteraciones, la versión incremental podría ser un modelo en papel o un prototipo. Durante las últimas iteraciones, se producen versiones cada vez más completas de ingeniería del sistema.

3.2.5.1 Actividades

El modelo en espiral se divide en un número de actividades estructurales también llamadas **regiones de tareas**. Generalmente existen entre tres y seis regiones de tareas.

[36] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 26-27.

- **Comunicación con el cliente**. Las tareas requeridas para establecer comunicación entre el desarrollador y el cliente.

- **Planificación**. Las tareas requeridas para definir recursos, tiempo y otras informaciones relacionadas con el proyecto.

- **Análisis de riesgos**. Las tareas requeridas para evaluar riesgos técnicos y de gestión.

- **Ingeniería**. Las tareas requeridas para construir una o más representaciones de la aplicación.

- **Construcción y adaptación**. Las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario.

- **Evaluación del cliente**. Las tareas requeridas para obtener la reacción del cliente según la evaluación de las representaciones del *software* creadas durante la etapa de ingeniería e implementadas durante la etapa de implementación.

Cada una de las regiones está poblada por una serie de tareas que se adaptan a las características del proyecto que va a emprenderse. Para proyectos pequeños, el número de tareas y su formalidad es bajo. Para proyectos mayores y más críticos, cada región contiene tareas que se definen pare lograr un nivel más alto de formalidad.

Cuando empieza este proceso evolutivo, el equipo de ingeniería del *software* gira alrededor de la espiral en la dirección de las agujas del reloj, comenzando por el centro. El primer circuito de la espiral produce el desarrollo de una especificación de productos. Los siguientes pasos en la espiral se podrán utilizar para desarrollar un prototipo y progresivamente versiones más sofisticadas del *software*. Cada paso de la región de planificación produce ajustes en el plan del proyecto. El coste y la planificación se ajustan según la reacción del cliente. Además, el gestor del proyecto ajusta el número planificado de iteraciones requeridas para completar el *software*.

3.2.5.2 Evaluación

El modelo en espiral es un enfoque realista del desarrollo de sistemas y de *software* a gran escala. Como el *software* evoluciona, a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los niveles evolutivos. El modelo en espiral utiliza la construcción de prototipos como mecanismo de reducción de riesgos, pero lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto. Mantiene el enfoque sistemático de los pasos sugeridos por el ciclo de vida clásico, pero lo incorpora al marco de trabajo interactivo que refleja de forma más realista el mundo real. El modelo en espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica adecuadamente, debe reducir los riesgos antes de que se conviertan en problemáticos.

Pero al igual que otros paradigmas, el modelo en espiral no es la panacea. Requiere una considerable habilidad para la evaluación del riesgo y cuenta con esta habilidad para el éxito. Si un riesgo importante no es descubierto y gestionado, indudablemente surgirán problemas. Finalmente, el modelo en sí mismo es relativamente nuevo y no se ha utilizado tanto como los paradigmas lineales secuenciales o de construcción de prototipos. Todavía tendrán que pasar muchos años antes de que se determine con absoluta certeza la eficacia de este nuevo e importante paradigma[37].

3.2.6 El modelo de ensamblaje de componentes

La tecnología de objetos proporciona el marco de trabajo técnico para un modelo de proceso basado en componentes para la ingeniería del *software*. El paradigma de orientación a objetos enfatiza la creación de clases que encapsulan tanto los datos como los algoritmos que se utilizan para manejar los datos. Si se diseñan y se implementan adecuadamente, las clases orientadas a objetos son reutilizables por las diferentes aplicaciones y arquitecturas de sistemas basados en computadoras.

El modelo de ensamblaje de componentes incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza y exige un enfoque interactivo para la creación del *software*. Sin embargo, el modelo ensamblador de componentes configura aplicaciones desde componentes preparados de *software* (a veces llamados **clases**).

3.2.6.1 Actividades

La actividad de la ingeniería comienza con la identificación de clases candidatas. Esto se lleva a cabo examinando los datos que se manejarán por parte de la aplicación y el algoritmo que aplicará para seguir el tratamiento.

---

[37] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 28-29.

Los datos y los algoritmos correspondientes se empaquetan en una clase. Las clases creadas en los proyectos de ingeniería del *software* anteriores se almacenan en una *biblioteca de clases* o *depósito*. Una vez identificadas las clases candidatas, la biblioteca de clases se examina para determinar si estas clases ya existen. En caso de que así fuera, se extraen de la biblioteca y se vuelven a utilizar.

Si una clase candidata no reside en la biblioteca, se aplican métodos orientados a objetos. Se compone así la primera interacción de la aplicación a construirse, mediante las clases extraídas de la biblioteca y las clases nuevas construidas para cumplir las necesidades únicas de la aplicación. El flujo del proceso vuelve a la espiral y volverá a introducir por último a la iteración ensambladora de componentes a través de la actividad de ingeniería[38].

3.2.6.2 Evaluación

El modelo ensamblador de componentes lleva a la reutilización del *software*, y la reutilización proporciona beneficios a los ingenieros del *software*. Según estudios de reutilización, QSM Associates, Inc. informa que el ensamblaje de componentes lleva una reducción del 70 por ciento del tiempo de ciclo de desarrollo, un 84 por ciento del coste del proyecto y un índice de productividad del 26.2, comparado con la norma de la industria del 16.9.

Aunque estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda que el ensamblaje de componentes proporciona ventajas significativas para los ingenieros del *software*.

[38] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 29-30.

3.2.7 El modelo de desarrollo concurrente

El modelo de proceso concurrente se puede representar en forma de esquema como una serie de actividades técnicas importantes, tareas y estados asociados a ellas. Por ejemplo, la actividad de **ingeniería** definida para el modelo en espiral se lleva a cabo invocando las tareas siguientes: modelado de construcción de prototipos y/o análisis, especificación de requisitos y diseño.

Por ejemplo, la actividad **análisis** se puede encontrar en uno de los estados destacados anteriormente en cualquier momento dado. De forma similar, otras actividades se pueden representar de una forma análoga. Todas las actividades existen concurrentemente, pero residen en estados diferentes.

El modelo de proceso concurrente define una serie de acontecimientos que dispararán transiciones de estado a estado para cada una de las actividades de la ingeniería del *software*. Por ejemplo, durante las primeras etapas del diseño, no se contempla una inconsistencia del modelo de análisis. Esto genera la corrección del modelo de análisis de sucesos, que disparará la actividad de análisis del estado **hecho** al estado **cambios en espera**.

En realidad, el modelo de proceso concurrente es aplicable a todo tipo de desarrollo de *software* y proporciona una imagen exacta del estado actual de un proyecto. En vez de confinar las actividades de ingeniería del *software* a una secuencia de sucesos, define una red de actividades. Todas las actividades de la red existen simultáneamente con otras. Los sucesos generados dentro de una actividad dada o en algún otro lugar de la red de actividad inician las transiciones entre los estados de una actividad[39].

[39] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 30-31.

3.2.8 El modelo de métodos formales

El modelo de métodos formales acompaña a un conjunto de actividades que conducen a la especificación matemática del *software* de computadora. Los métodos formales permiten que un ingeniero de *software* especifique, desarrolle y verifique un sistema basado en computadora aplicando una notación rigurosa y matemática. Algunas organizaciones de desarrollo del *software* actualmente aplican una variación de este enfoque llamado **ingeniería del software de sala limpia**.

Cuando se utilizan métodos formales durante el desarrollo, proporcionan un mecanismo para eliminar muchos de los problemas que son difíciles de superar con paradigmas de la ingeniería del *software*. La ambigüedad, lo incompleto y la inconsistencia se descubren y se corrigen más fácilmente, no mediante una revisión a propósito para el caso, sino mediante la aplicación de análisis matemático. Cuando se utilizan métodos formales durante el diseño, sirven como base para la verificación de programas y por consiguiente permiten que el ingeniero del *software* descubra y corrija errores que no se pudieron detectar de otra manera.

3.2.8.1 Evaluación

Aunque todavía no hay un enfoque establecido, los modelos de métodos formales ofrecen la promesa de un *software* libre de defectos. Sin embargo, se ha hablado de una gran preocupación sobre su aplicabilidad en un entorno de gestión:

- El desarrollo de modelos formales actualmente es bastante caro y lleva mucho tiempo.

- Se requiere un estudio caro porque pocos responsables del desarrollo de *software* tienen los antecedentes necesarios para aplicar métodos formales.

- Es difícil utilizar los modelos como un mecanismo de comunicación con clientes que no tienen muchos conocimientos técnicos.

No obstante, es posible que el enfoque a través de métodos formales tenga más partidarios entre los desarrolladores de *software* que deben construir *software* de mucha seguridad y entre los desarrolladores que pasan grandes penurias económicas al aparecer errores de *software*[40].

3.2.9 Técnicas de cuarta generación

El término **técnicas de cuarta generación** abarca un amplio espectro de herramientas de *software* que tienen algo en común: todas facilitan al ingeniero del *software* la especificación de algunas características del *software* de alto nivel. Luego, la herramienta genera automáticamente el código fuente basándose en la especificación del técnico. Cada vez parece más evidente que cuanto mayor sea el nivel en el que se especifique el *software*, más rápido se podrá construir el programa. El paradigma T4G para la ingeniería del *software* se orienta hacia la posibilidad de especificar el *software* usando formas de lenguaje especializado o notaciones gráficas que describan el problema que hay que resolver en términos que lo entienda el cliente. Actualmente un entorno para el desarrollo de *software* que soporte el paradigma T4G puede incluir todas o algunas de las siguientes herramientas: lenguajes no procedimentales de consulta a bases de datos, generación de informes, manejo de datos, interacción y definición de pantallas, generación de códigos, capacidades gráficas de alto nivel y capacidades de hoja de cálculo.

[40] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 31-32.

Inicialmente muchas de estas herramientas estaban disponibles pero solo para ámbitos de aplicación muy específicos. Pero actualmente los entornos T4G se han extendido a todas las categorías de aplicaciones del *software*.

3.2.9.1 Actividades

Al igual que con otros paradigmas, T4G comienza con el paso de reunión de requisitos. Idealmente, el cliente describe los requisitos que a continuación son traducidos a un prototipo operativo. Sin embargo, en la práctica no se puede hacer eso. El cliente puede que no esté seguro de lo que necesita, puede ser ambiguo en la especificación de hechos que le son conocidos y puede que no sea capaz o no esté dispuesto a especificar la información en la forma en que puede utilizar una herramienta T4G. Por esta razón, el diálogo cliente-desarrollador descrito por los otros paradigmas es una parte esencial del enfoque T4G.

Para aplicaciones pequeñas, se puede ir directamente desde el paso de recolección de requisitos al paso de implementación, usando un lenguaje de cuarta generación no procedimental (L4G). Sin embargo, es necesario un mayor esfuerzo para desarrollar una estrategia de diseño para el sistema incluso si se utiliza un L4G. El uso de T4G sin diseño (para grandes proyectos) causará las mismas dificultades (poca calidad, mantenimiento pobre, mala aceptación del cliente) que se encuentran cuando se desarrolla *software* mediante los enfoques convencionales.

La implementación mediante L4G permite al que desarrolla el *software* centrarse en la representación de los resultados deseados, que es lo que se traduce automáticamente en un código fuente que produce dichos resultados. Obviamente, debe existir una estructura de datos con información relevante y a la que el L4G pueda acceder rápidamente.

Para transformar una implementación T4G en un producto, el que lo desarrolla debe dirigir una prueba completa, desarrollar con sentido una documentación y ejecutar el resto de las actividades de integración requeridas en los otros paradigmas de ingeniería del *software*. Además, el *software* desarrollado con T4G debe ser construido de modo que facilite la realización del mantenimiento de forma expeditiva.

3.2.9.2 Evaluación

Al igual que todos los paradigmas del *software*, el modelo T4G tiene ventajas e inconvenientes. Los defensores aducen reducciones drásticas en el tiempo de desarrollo del *software* y una mejora significativa en la productividad de la gente que construye el *software*. Los detractores aducen que las herramientas actuales de T4G no son más fáciles de utilizar que los lenguajes de programación, que el código fuente producido por tales herramientas es "ineficiente" y que el mantenimiento de grandes sistemas de *software* desarrollados mediante T4G, es cuestionable.

Hay algún mérito en lo que se refiere a las indicaciones de ambos lados y es posible resumir el estado actual de los enfoques T4G:

- El uso de T4G ha crecido considerablemente en la última década y ahora es un enfoque viable para muchas de las diferentes áreas de aplicación. Junto con las herramientas de **ingeniería de software asistida por computadora** (*CASE*) y los generadores de código, T4G ofrece una solución fiable a muchos problemas de *software*.

- Los datos recogidos en compañías que usan T4G parecen indicar que el tiempo requerido para producir *software* se reduce mucho para aplicaciones pequeñas y de tamaño medio y que la cantidad de análisis y diseño para las aplicaciones pequeñas, también se reduce.

- Sin embargo, el uso de T4G para grandes trabajo de desarrollo de *software* exige el mismo o más tiempo de análisis, diseño y prueba (actividades de ingeniería de *software*), perdiéndose así un tiempo sustancial que se ahorra mediante la eliminación de codificación.

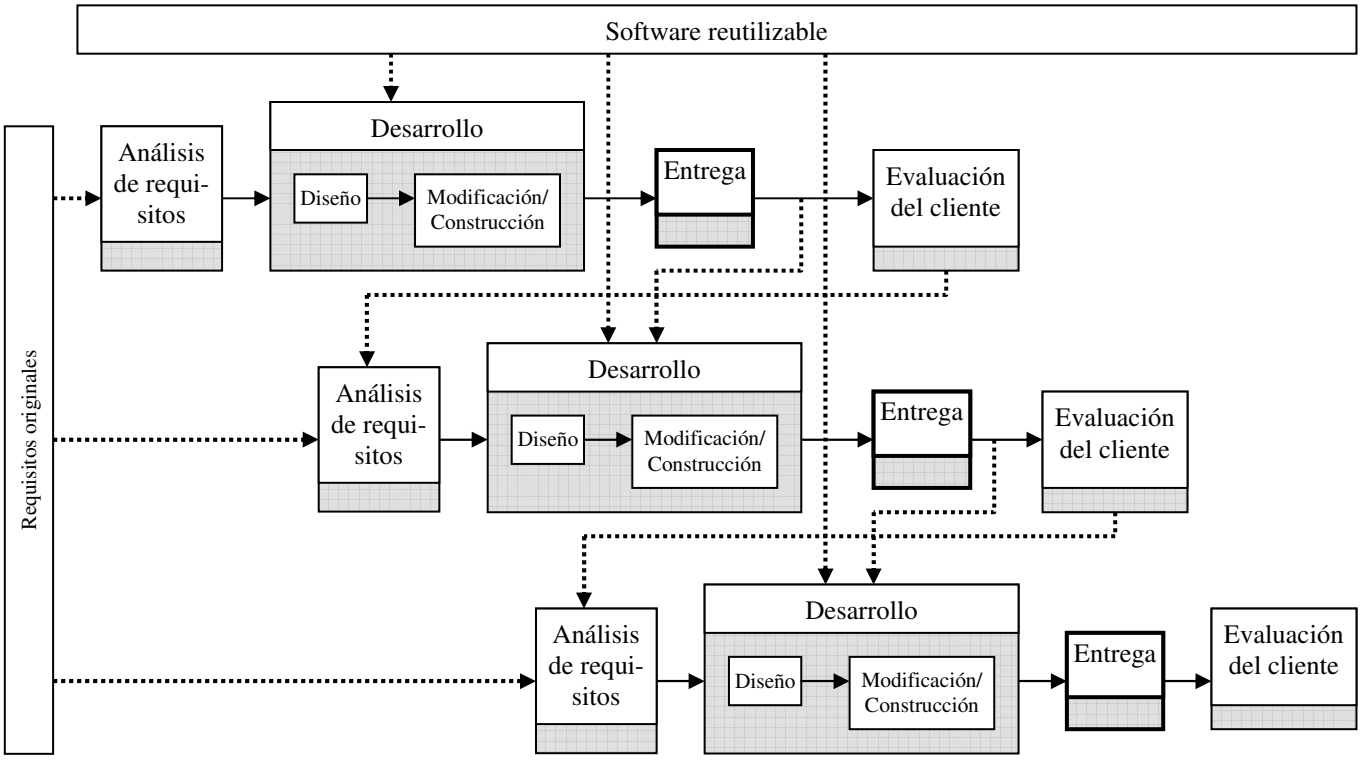
Las técnicas de cuarta generación ya se han convertido en una parte importante del desarrollo de *software*. Cuando se combinan con enfoques de ensamblaje de componentes, el paradigma T4G se puede convertir en el enfoque dominante hacia el desarrollo del *software*.

3.3 El modelo abierto de desarrollo

El modelo abierto de desarrollo, a diferencia del modelo lineal secuencial, es un modelo evolutivo. Se adapta a la naturaleza del *software* la cual implica cambios a lo largo de la vida del *software*. Implementa un desarrollo iterativo mediante el cual se producen versiones más completas de ciclo en ciclo.

Este modelo combina elementos de algunos de modelos expuestos anteriormente. Del modelo en espiral se obtiene el enfoque de un proceso evolutivo y cíclico y se incluye también el enfoque de reutilización de código del modelo de ensamblaje de componentes. Del modelo incremental, el enfoque de entregas periódicas mejoradas; y del modelo de construcción de prototipos, el enfoque de interacción con el cliente para definir los requisitos. A estas características habría que agregarle una última que es una extensión de la interacción con el cliente: el proceso de pruebas (y por consiguiente el de depuración) es realizado en su mayor parte con la evaluación del cliente, es decir, posterior a la entrega de una versión del *software*. En la Figura 1 se muestra el proceso seguido por el modelo.

Figura 1. Modelo abierto de desarrollo



3.3.1 Actividades

El modelo aplica secuencias incrementales que incluye análisis de requisitos, desarrollo, entregas y evaluación del cliente, donde cada secuencia depende de una secuencia anterior que le provee de retroalimentación y de una versión anterior desarrollada.

El primer paso del proceso es el análisis de los requisitos. Al inicio del proyecto se definen un conjunto de requisitos que, en conjunto, son la razón por la cual se crea el proyecto. Conforme el proyecto avanza y con base en la retroalimentación del cliente estos requisitos son modificados o nuevos requisitos son agregados.

La siguiente etapa del modelo es la de desarrollo, que está compuesta a su vez por el proceso de diseño y la modificación/construcción. El *software* resultante de esta etapa puede tener sus orígenes en cualquiera de las siguientes fuentes: *software* desarrollado por terceros que se reutilizará, el *software* proveniente de la secuencia anterior o de la codificación hecha en la propia secuencia. De esta manera, el *software* puede resultar de una modificación de *software* ya existente o de la construcción de nuevo *software*.

El siguiente paso es poner el *software* a disposición del cliente. No se espera que la entrega esté libre de errores, ya que la prueba más intensa se realiza posterior a la entrega. El tiempo entre entregas generalmente es muy corto, aunque esto depende del criterio de los desarrolladores. Existen proyectos que entregan una versión por mes como también existen proyectos que durante el desarrollo intenso liberan una versión por día, como ha sucedido en ocasiones con Linux.

El último paso del ciclo es la evaluación del cliente. Por un lado, se evalúa la funcionalidad del *software* y de este proceso pueden surgir sugerencias que debidamente retroalimentadas afectan el análisis de los requisitos de la siguiente secuencia. Por otro lado, se prueba el *software* para descubrir los errores que podría tener. Como el código fuente del *software* está disponible, es posible realizar también el proceso de depuración. Estas observaciones son retroalimentadas para que se hagan las modificaciones correspondientes en la etapa de desarrollo de la siguiente secuencia.

3.3.1.1 Análisis de requisitos

El proceso de análisis de requisitos implica en un principio entender el problema y luego representar y entender el dominio de la información de dicho problema, definir las funciones que debe realizar el *software* y representar el comportamiento del *software*[41].

[41] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 188.

Este proceso genera una lista de requisitos que deben implementarse en el *software* a desarrollar, los cuales pueden clasificarse como requisitos normales, esperados o innovadores (ver Sección 3.1.1.3). Esta lista se clasifica por prioridades para que el equipo de desarrollo tenga claro en qué orden se implementarán.

Dada que todas las aplicaciones de *software* son realmente procesadoras de datos, es importante representar y entender el dominio de la información. El *software* procesa datos como tales, pero también procesa eventos que representan algún aspecto del control del sistema. El examen del dominio de la información se da desde tres puntos de vista:

- **El contenido de la información**. Representa los objetos individuales de datos y control que componen alguna colección mayor de información a la que transforma el *software*. Estos objetos pueden relacionarse con otros, por lo que es necesario indicar esta relación.

- **El flujo de la información**. Representa cómo cambian los datos y el control a medida que se mueven dentro del sistema. Las transformaciones que se aplican a los datos son funciones que debe realizar un programa. Los datos y control que se intercambian entre dos funciones definen la interfaz de dichas funciones.

- **La estructura de la información**. Representa la organización interna de los elementos de datos o control. Esta es la base para elaborar el diseño de las estructuras de datos en la siguiente fase del proceso.

El segundo paso del proceso de análisis de requisitos es definir las funciones que debe realizar el *software*. Genéricamente las funciones del *software* pueden clasificarse como de entrada, procesamiento o salida. Para estas funciones debe estudiarse y definirse lo que se desea que realicen sin hacer énfasis en cómo deben realizarlo.

El último paso del análisis de requisitos es definir el comportamiento del *software*. Es decir, cómo el *software* responde a acontecimientos externos. Un programa siempre se encuentra en un determinado estado, un modo de comportamiento que cambia cuando ocurre algún evento. Debe representarse los estados del *software* y los acontecimientos que causan los cambios de estado.

El modelo abierto de desarrollo se plantea un conjunto de requisitos iniciales. Estos son los requisitos originales que los propietarios del proyecto se plantearon al inicio y que impulsaron la creación del proyecto. A estos requisitos se les asocia una prioridad y esta determina si son implementados en las primeras iteraciones o no. Luego, a lo largo del desarrollo, los clientes pueden indicar otros requisitos que desean que se implementen. Estos son también son analizados, priorizados e implementados.

El modelo abierto de desarrollo incluye una fase de análisis de requisitos en cada iteración debido a que estos pueden cambiar y aumentar, por lo que es necesario estudiarlos y definir cuáles serán implementados para la siguiente entrega del *software*.

3.3.1.2 Desarrollo

El proceso de desarrollo consiste en una serie de actividades que pueden agruparse en dos áreas: el diseño y la modificación/construcción. Ambas están fuertemente ligadas a la reutilización de *software* desde el mismo inicio del proyecto. El siguiente paso, luego de definir los requisitos iniciales del *software*, es la búsqueda de *software* reutilizable que pueda servir de base sobre la cual continuar el desarrollo.

Sin embargo, la reutilización de *software* no se da solamente en la primera iteración del proceso de desarrollo. Cada nueva iteración implica hacer uso del *software* producto de la iteración anterior y constantemente se agrega código de terceros que, luego de ser evaluado, se determina que tiene la funcionalidad deseada.

El diseño entonces puede ser de dos tipos: el diseño como tal del *software* que se construirá en la iteración, y el rediseño del *software* que se reutilizará. De igual manera, durante la codificación pueden tenerse dos enfoques: modificación y construcción. Durante la modificación puede introducirse cambios al *software* reutilizado para adaptarlo, corregirlo y mejorarlo. Por otro lado, la construcción implica la creación de nuevo *software*.

3.3.1.2.1 Reutilización de *software*

La reutilización de *software* es una de las características fundamentales del modelo de desarrollo abierto. Básicamente adopta el mismo enfoque del modelo de ensamblaje de componentes, es decir, la búsqueda de *software* previamente desarrollado que pueda utilizarse para los propósitos del proyecto.

Sin embargo, la manera en que se da la reutilización de *software* es diferente de un modelo a otro. La razón de esto es que se está trabajando con *software* libre, por lo que es posible obtener y modificar el *software* de terceros. Estas dos características (usar el *software* de terceros y modificarlo) no se reconocen en el modelo de ensamblaje de componentes. Este modelo especifica que es necesaria una biblioteca de componentes propios, que son los únicos que pueden ser usados. Debido a la existencia de *software* libre, los desarrolladores no tienen que limitarse a usar el *software* propio sino que tienen a su disposición gran cantidad de *software*, que pueden usar prácticamente sin ninguna restricción.

Al contrario del modelo de ensamblaje de componentes, el *software* reutilizado no se usa **encapsulado**. El *software* libre permite modificar los componentes reutilizados para adaptarlos a las necesidades del proyecto. Es importante hacer notar que permite disminuir la ineficiencia o complejidad que podría surgir como consecuencia de incluir dentro del proyecto *software* que ha sido optimizado para otros fines.

La reutilización de *software* da ciertas ventajas al modelo abierto de desarrollo. Por un lado, mejora la calidad del *software*. Cuando se utiliza *software* desarrollado por terceros, este normalmente ya ha sido probado y utilizado en producción. Esto asegura que dicho *software* tiene una baja proporción de errores. Pero los beneficios no son solamente para el proyecto sino también para los desarrolladores originales, en caso de encontrarse algún error al usar el *software* y notificarles.

Además la reutilización del *software* permite desarrollar aplicaciones con menos esfuerzo y en menor tiempo. Permite usar diseños que han sido probados en la práctica y que han tenido buenos resultados. Por último producen una disminución en los costos.

3.3.1.2.2 Diseño

Luego del análisis de requisitos, cada iteración contempla un proceso de diseño. A este proceso de diseño hay que agregarle una variable debida a la reutilización de *software*. Esto implica que ya se trabaja sobre un *software* que ha pasado por un proceso de diseño anterior concebido por los desarrolladores originales del *software*, que pueden ser los miembros del equipo que desarrollaron la iteración anterior o personas ajenas al proyecto.

El diseño del *software* debe traducir los requisitos analizados anteriormente en una representación del *software* tomando en cuenta que ya existe un *software* previamente diseñado. Por lo tanto, se debe efectuar un diseño para la construcción que se hará en la iteración y un rediseño para el *software* reutilizable, el cual deberá establecer la guía para hacer su modificación.

El diseño debe implementar todos los requisitos explícitos analizados en la fase anterior, como también los implícitos a todo *software* tales como buen funcionamiento, fiabilidad y otros.

Es importante considerar que el diseño debe quedar documentado y debe constituirse la guía para quienes construyen el código y lo prueban, pero especialmente para ser utilizado en iteraciones posteriores.

En este sentido, el diseño debe estructurarse para permitir cambios y presentar uniformidad, de tal manera que sea más fácil de comprender en su totalidad.

El proceso de diseño, al igual que el de codificación, es abierto. Por tal motivo se involucra a los colaboradores del proyecto. Normalmente el diseño se descompone en segmentos más pequeños que son delegados a colaboradores, quienes son responsables de su parte. Sin embargo, el diseño del nivel superior involucra a más personas, entre las cuales debe lograrse un consenso. Es importante estar abiertos a las sugerencias de colaboradores que pueden enriquecer este proceso con sus experiencias.

3.3.1.2.3 Modificación/Construcción

El siguiente avance se da en la codificación. Como base para este proceso se tiene el diseño previamente elaborado y algún *software* que será reutilizado. Por un lado, se tiene la modificación del *software*. Esta se da cuando se trabaja con un *software* de terceros que se usará como punto de partida en una determinada área del proyecto o se acoplará para agregar cierta funcionalidad.

En este caso, el *software* ha sido desarrollado con otro propósito y es necesario adaptarlo a los fines del proyecto. Pero también de iteración en iteración se trabaja con *software* propio. Luego de que una versión ha sido probada por los usuarios, pueden darse algunas sugerencias que es conveniente implementar. También, esta modificación incluye la corrección de errores detectados por los usuarios. Todo esto supone eliminar código, hacer cambios cambiarlo o agregar algo que no esté implementado.

En lo referente a la construcción de *software*, esta se da cuando no es posible utilizar algún *software* previo, debido a que se quiere implementar alguna función novedosa o se quiere introducir alguna nueva tecnología.

Tanto la modificación como la construcción se apoyan en la colaboración. De esta manera, es posible recibir aportaciones de varias personas que participan en el proyecto por medio de Internet. Si es necesaria alguna construcción, puede ponerse a disposición de los colaboradores su descripción y designar a alguien para su desarrollo. Por otro lado, la disposición del código fuente del *software* libre permite que los colaboradores participen en la modificación. Aun cuando normalmente las modificaciones son solicitadas y descritas, en algunos casos, por iniciativa propia, los colaboradores pueden analizar algún código y hacerle mejoras que luego son introducidas al proyecto.

3.3.1.3 Entrega

La entrega reúne el fruto del esfuerzo hecho en las fases anteriores en un producto funcional y capaz de realizar trabajo útil. Este producto es empaquetado y puesto a disposición del cliente para que sea usado y evaluado.

La periodicidad de las entregas depende en gran medida del grado de interacción que exista con los usuarios. Si el proyecto depende en gran medida de los usuarios para el desarrollo, las entregas serán más frecuentes.

Linux tiene un desarrollo extremadamente colaborativo y durante los períodos de desarrollo intenso ha llegado a hacer una entrega diaria. Algunos líderes de proyectos consideran imprudente hacer entregas sin antes haber sido probadas exhaustivamente y pueden dejar pasar mucho tiempo entre entregas. Sin embargo, este modelo asume que los usuarios contribuirán para probar el *software* y realizar su depuración, así que mientras más pronto llegue el *software* a sus manos, más rápidamente se corregirán los errores potenciales.

Cada nueva versión incluye una serie de cambios que pueden ser desde corrección de errores hasta el rediseño total de una parte importante del proyecto. Los cambios pueden pertenecer a diferentes áreas del proyecto independientes entre sí, pero que al momento de preparar la entrega merecen ser incluidos.

Es importante preparar una versión formal para ser entregada, especificando claramente cuáles cambios han sido hechos a partir de la versión anterior. Cada versión debe ser acompañada por su correspondiente código fuente y las herramientas necesarias para ser instalada.

Luego, todo el paquete debe hacerse disponible a los usuarios por los medios que se consideren más convenientes. Los paquetes históricos también deben estar disponibles para que los colaboradores puedan hacer comparaciones.

3.3.1.4 Evaluación del cliente

La última fase de cada iteración es la que corresponde a los usuarios. En esta etapa se da el proceso de prueba y depuración. Aun cuando el *software* es probado mientras se desarrolla, el modelo especifica que el proceso más extenso se da después de la entrega y por parte de los usuarios.

Esto es posible debido a la disponibilidad del código fuente asociada al *software* libre. Esto representa una serie de ventajas, tales como que en caso de existir un error, este será descubierto, será establecida su causa y será solucionado por alguien y no necesariamente la misma persona. Esto acelera el proceso de prueba y asegura la calidad del *software* debido a que ha sido usado en un ambiente real.

3.3.1.4.1 Prueba del *software*

La prueba del *software* es un elemento crítico para la garantía de la calidad del *software* y representa una revisión final de las especificaciones, del diseño y de la codificación. La prueba es un proceso de ejecución del programa con la intención de descubrir errores. Este proceso debe hacerse tomando como base los requisitos del *software* ya que, en última instancia, los errores son los defectos del *software* que le impiden cumplir con sus requisitos[42].

El proceso de prueba generalmente parte de las funciones hasta llegar al sistema total, con la finalidad de disminuir la posibilidad de dejar un error sin descubrir. Lo común es que las pruebas de las funciones sean realizadas por quienes hicieron el código, ya que la prueba del sistema requiere la integración del trabajo producido por varios grupos

En el caso del modelo abierto del desarrollo, gran parte de los errores son descubiertos de manera descendente, es decir, al hacer uso de la totalidad del sistema. Luego de establecer la existencia de un error debido al comportamiento e incorrecto del *software*, este es rastreado hasta llegar a los niveles inferiores.

[42] Roger S. Pressman. **Ingeniería del software, un enfoque práctico.** (4ª Edición; Editorial McGraw-Hill, 1998) p. 301, 302.

3.3.1.4.1 Depuración

Aunque están muy relacionadas, la prueba y la depuración son dos cosas distintas. La prueba es el proceso de establecer la existencia de errores en un programa y la depuración es el proceso de localizar donde están localizados los errores y corregirlos.

La primera etapa del proceso depuración es la más trabajosa, como todo proceso de búsqueda. La segunda etapa suele ser más fácil, siempre y cuando sea un error de codificación. El problema se complica si es un error de diseño o una mala interpretación de los requisitos, lo cual podría implicar hacer un rediseño.

Como ya se indicó, el descubrimiento de la existencia de un error, su localización y su corrección puede ser hecho por diferentes personas. Por ello es importante la retroalimentación de estas tres situaciones. Luego del aviso de la existencia de un error, un grupo de usuarios puede dedicarse a localizarlo y corregirlo. Si esto escapa de sus posibilidades, deben dar el correspondiente aviso para que alguien más indique los cambios que deben hacerse para corregirlo. Por último, estos cambios deben ser notificados a quienes participan en la codificación para que sean incluidos en la próxima versión del *software*.

3.3.2 Evaluación

El modelo abierto de desarrollo presenta muchas ventajas y elimina muchos de los problemas que tienen los otros modelos descritos anteriormente. Sin embargo, su eficacia podría variar dependiendo el tipo de aplicación que se desarrolle, como a continuación se examina.

3.3.2.1 *Software* de sistemas

El *software* de sistemas es un conjunto de programas que han sido escritos para servir a otros programas. Algunos, como compiladores, editores y utilidades de gestión de archivos, procesan estructuras de información complejas pero determinadas. Otras aplicaciones, tales como ciertos componentes del sistema operativo, utilidades de manejo de periféricos y procesadores de telecomunicaciones de sistemas, procesan.

En cualquier caso, el área del *software* de sistemas se caracteriza por una fuerte interacción con el *hardware* de la computadora, una gran utilización por múltiples usuarios, una operación concurrente que requiere una planificación, una compartición de recursos, estructuras de datos complejas y múltiples interfaces externas.

El modelo abierto de desarrollo se adapta muy bien a las características de este *software*, ya que requiere de mejoras constantes y la implementación de nueva tecnología. Por otro lado, este *software* tiene una amplia base de usuarios con gran capacidad técnica, lo que los hace aptos para el proceso de evaluación y depuración que propone el modelo abierto de desarrollo.

3.3.2.2 *Software* de tiempo real

El *software* que mide, analiza y controla sucesos del mundo real conforme ocurren, se denomina de tiempo real. Entre los elementos del *software* de tiempo real se incluyen: un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y un componente de monitorización que coordina todos los demás componentes, de forma que pueda mantenerse la respuesta en un tiempo real (típicamente en el rango de 1 milisegundo a 1 minuto).

El *software* de tiempo real es muy especializado, ya que una aplicación desarrollada solo responde a las necesidades de un caso en especial. No cuenta con una base amplia de usuarios que estén dispuestos hacer el proceso de evaluación y depuración. Sí puede ser necesario un proceso evolutivo e interactivo para la determinación de los requisitos, por lo que el modelo abierto de desarrollo podría ser usado en parte. Lo que sí es posible es la reutilización de *software* al existir ciertos componentes que tienen algunas funciones necesarias.

3.3.2.3 *Software* de gestión

El procesamiento de información comercial constituye la mayor de las áreas de aplicación del *software*. Los **sistemas discretos** como nóminas, cuentas por cobrar o por pagar e inventarios han evolucionado hacia el *software* de sistemas de información de gestión, que accede a una o más bases datos grandes que contienen información comercial. Las aplicaciones en esta área reestructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones. Además de las tareas convencionales de procesamiento de datos, las aplicaciones de *software* de gestión también realizan cálculo interactivo tal como el procesamiento de transacciones en puntos de venta.

El *software* de gestión es desarrollado específicamente para una organización. Cada aplicación solamente satisface los requisitos de esta organización y no puede ser transportada a otra organización. Este *software* es desarrollado internamente o contratando algún equipo cuyo producto no es usado por nadie más. Sin embargo, es un *software* que necesita mucho mantenimiento y puede hacer uso de la reutilización.

3.3.2.4 *Software* de ingeniería y científico

El *software* de ingeniería y científico está caracterizado por los algoritmos de **manejo de números**. Las aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática. Sin embargo, las nuevas aplicaciones del área de ingeniería/ciencia se han alejado de los algoritmos convencionales numéricos. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas, han comenzado a coger características del *software* de tiempo real e incluso del *software* de sistemas.

El uso de este *software* generalmente no es público. Es muy complejo y necesita ciertos conocimientos especializados. El desarrollo es muy cuidadoso por lo que no es posible para cualquiera participar en su desarrollo. Estas características podrían limitar el uso del modelo abierto de desarrollo.

3.3.2.5 *Software* empotrado

Los productos inteligentes se han convertido en algo común en casi todos los mercados de consumo e industriales. El *software* empotrado reside en memoria de sólo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo. El *software* empotrado puede ejecutar funciones muy limitadas y curiosas tales como el control de las teclas de un horno microondas o suministrar una función significativa y con capacidad de control, tales como funciones digitales en un automóvil para el control de gasolina, sistemas de frenado, etc.

El *software* empotrado es desarrollado por los fabricantes de equipos. Depende del *hardware* y la estructura de éste normalmente es muy restringida. Por otro lado, este *software* no evoluciona tanto, así que el modelo abierto de desarrollo es poco aplicable.

3.3.2.6 *Software* de computadoras personales

El mercado del *software* de computadoras personales está en auge. El procesamiento de textos, las hojas de cálculo, los gráficos por computadora, multimedia, entretenimientos, gestión de bases de datos, aplicaciones financieras, de negocios y personales, y redes o acceso a bases de datos externas, son algunas de los cientos de aplicaciones.

Este es el *software* más público que existe y se adapta en gran manera al modelo abierto de desarrollo. Tiene mucha evolución, una base grande de usuarios y sus funciones pueden ser implementadas por gran cantidad de programadores.

3.3.2.7 *Software* de inteligencia artificial

El *software* de inteligencia artificial hace uso de algoritmos no numéricos para resolver problemas complejos, para los que no son adecuados el cálculo o el análisis directo. Actualmente el área más activa de la inteligencia artificial es la de sistemas expertos, también llamados sistemas basados en el conocimiento. Sin embargo, otras áreas de aplicación para el *software* de inteligencia artificial es el reconocimiento de patrones (imágenes y voz), la prueba de teoremas y los juegos. En los últimos años se ha desarrollado una nueva rama del *software* de inteligencia artificial llamada redes neuronales artificiales. Una red neuronal simula la estructura de proceso del cerebro (las funciones de la neurona biológica) y a la larga pueden llevar a una clase de *software* que pueda reconocer patrones complejos y a aprender de "experiencia" pasada.

El *software* de inteligencia artificial requiere conocimientos más allá de programación. Sin embargo, estos conocimientos son de las ciencias de la computación y no de otras, como sucede con el *software* científico. El modelo abierto de desarrollo puede proveer de grandes beneficios al ser usado en este tipo de *software*.

4. COMERCIALIZACIÓN Y DISTRIBUCIÓN DE *SOFTWARE* LIBRE

Hasta este punto se han mostrado los grandes beneficios del *software* libre para asegurar la calidad y el desarrollo eficiente de *software*. También se ha demostrado que el *software* libre da grandes ventajas a los usuarios de *software*. Sin embargo, desde el punto de vista empresarial, se plantea la interrogante acerca de la rentabilidad de este tipo de *software*.

En este capítulo se responderá a este cuestionamiento y se explicarán los aspectos legales que dan soporte a la distribución del *software* libre.

4.1 La ilusión de la manufactura

Los programas de computación, como cualquier otra herramienta o bien de capital, tienen dos clases distintas de valor económico: el valor de uso y el valor de venta. El valor de uso de un programa es su valor económico como herramienta. El valor de venta de un programa es su valor como una mercancía vendible.

Cuando se trata de razonar acerca de la economía de producción del *software*, erróneamente se asume un **modelo de fábrica**. Es decir, se tiende a asumir que el *software* tiene las características de valor de un bien manufacturado típico. Este razonamiento se basa en dos premisas que no son ciertas del todo:

1. La mayor parte del tiempo del desarrollador se paga por el valor de venta.

2. El valor de venta del *software* es proporcional a su costo de desarrollo (el costo de los recursos necesarios para reproducir el *software* funcionalmente) y a su valor de uso.

La primera premisa se puede contradecir al observar que el *software* escrito para ser vendido es apenas una pequeña parte de la totalidad de *software* desarrollado. La gran mayoría del *software* es producido para uso propio de las organizaciones que lo desarrollan y no para ser vendido. Esto incluye el *software* financiero y de bases de datos que toda mediana o gran compañía necesita hacer. Incluye también código técnico especializado como los *device drivers*, que no se venden. También se incluye el *software* empotrado que se encuentran en una gran variedad de máquinas, desde aviones hasta tostadoras.

La mayoría de este código está muy integrado a su ambiente, de tal manera que su reutilización o copia es muy dificultosa. De esta manera, cuando el ambiente cambia, es necesario hacer mucho trabajo para mantener el *software* funcionando. Este mantenimiento compone la mayoría del trabajo (más del 75%) por el que se le paga a un programador. Por esto la mayoría de los recursos se gastan en escribir y mantener programas que no son vendidos.

La segunda premisa puede objetarse al examinar el comportamiento real de los consumidores. Hay algunos bienes para los cuales esta relación es válida (antes de la depreciación) tales como comida, vehículos y herramientas. También hay bienes intangibles para los cuales el valor de venta se ajusta fuertemente al costo de desarrollo y reemplazo, tal como los derechos para reproducir música, mapas o bases de datos. Estos bienes pueden retener o incluso aumentar su valor de venta después de que su productor ha desaparecido del mercado.

En contraste, cuando el productor de *software* sale del mercado (o incluso si el *software* es descontinuado), el mayor precio que los consumidores pagarán por él rápidamente cae hasta casi cero, sin importar su valor teórico o el costo de desarrollo de un bien funcionalmente equivalente. Este comportamiento revela que el precio que un consumidor pagará se ve efectivamente disminuido por el valor futuro esperado del servicio prestado por el productor del *software*, tal como mejoras, actualizaciones y proyectos siguientes.

Dicho de otra manera, el *software* es una industria de servicios que opera bajo la persistente pero infundada ilusión de que es una industria de manufactura. Normalmente se tiende a creer otra cosa. Simplemente puede ser porque la pequeña porción de la industria del *software* que vende, es también la única que promociona y publicita su producto. Además, algunos de los productos más visibles y más altamente promocionados son breves.

Es importante notar que la ilusión de la manufactura alienta estructuras de precios erróneas. Si más del 75% del costo del ciclo de vida de un proyecto de *software* típico se utilizará en mantenimiento, depuración y extensiones, entonces la política común de cargar un precio alto de adquisición y una cuota de soporte relativamente baja o nula lleva a resultados que no conforman a ninguna de las partes.

Los consumidores pierden porque los incentivos en el modelo de fábrica se vuelven en contra del servicio competente ofrecido por un productor. Si los ingresos del productor vienen de vender agregados, más esfuerzo hará para construir agregados y promocionarlos. El servicio al cliente no será un centro de ganancias, sino que se convertirá en un espacio para ofrecer cosas inservibles y tomará los suficientes recursos sólo para evitar las críticas de un pequeño grupo de clientes.

Pero también los productores pierden. Mantener un soporte indefinidamente continuo con un precio fijo, solo es posible en un mercado que se expande lo suficientemente rápido como para cubrir el soporte y los costos del ciclo de vida en los que se incurrió por el *software* vendido ayer, con las entradas del *software* que se venderá mañana. Una vez que el mercado madura y las ventas decaen, los productores no tendrán otra opción que cortar los gastos, dejando huérfano al producto.

Sea que esto se haga explícitamente (al descontinuar el producto) o implícitamente (al dificultar el soporte) los clientes se trasladarán a la competencia, ya que el producto ha perdido su valor futuro esperado. En corto plazo, se puede escapar de esta trampa haciendo que las versiones que reparan errores aparezcan como nuevos productos con un nuevo precio, pero los consumidores rápidamente se cansan de esto. En el largo plazo, el único modo de escapar es no tener competidores.

¿Cómo se maneja esto si la industria del *software* no corresponde al modelo de fábrica? Para manejar la estructura de costo real de un ciclo de vida de *software* eficiente, es necesaria una estructura de precios fundada en los contratos de servicios, suscripciones y un intercambio continuo de valor entre el productor y el cliente. Bajo las condiciones buscadoras de eficiencia del mercado libre, esta clase de estructura de precios será la que la mayor parte de la industria madura del *software* seguirá al final[43].

4.2 Los ingresos del *software* libre

En vista de lo indicado anteriormente, se observa que al cambiar del *software* no libre al *software* libre, el valor de venta se ve afectado, no así el valor de uso. Entonces la rentabilidad del *software* libre se basa en el uso efectivo de su valor esperado de uso[44].

[43] La ilusión de la manufactura (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node4.html). Mayo de 2001.
[44] Modelos de valor de venta Indirecto (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node8.html). Mayo de 2001.

Se podría suponer que al liberar un producto de *software* se podría perder la ventaja competitiva, ya que los competidores podrán hacer uso de dicho *software* sin incurrir en los gastos de desarrollo. Esto es cierto, pero solamente en parte. Lo que se debe considerar seriamente es si las ganancias obtenidas por liberar el paquete exceden las pérdidas por el beneficio que obtiene la competencia[45]. A continuación se detallan cuáles podrían ser estas ganancias[46].

4.2.1 Posicionamiento en el mercado

En este modelo, se usa *software* libre para crear o mantener una posición de mercado para un *software* propietario que genera un flujo directo de ingresos. En la variante más común, el uso de *software* cliente libre genera ventas de *software* servidor, ingresos por subscripciones o publicidad asociada a un portal.

Netscape Communications Inc., seguía esta estrategia cuando liberó el *browser* Mozilla al principio de 1998. El lado del cliente de su negocio estaba al 13% y caía cuando Microsoft lanzó Internet Explorer.

Abriendo las fuentes del todavía ampliamente popular navegador Netscape, Netscape le negó a Microsoft la posibilidad de un monopolio de *browser*s. Esperaban que la colaboración de la comunidad del *software* libre acelerara el desarrollo y depuración del *browser*, y esperaban que Internet Explorer estaría confinado a jugar un rol secundario, y prevendría que definieran exclusivamente HTML.

[45] Razones para cerrar las fuentes (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node7.html). Mayo de 2001.
[46] Modelos de valor de venta indirecto (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node10.html). Mayo de 2001.

La estrategia funcionó. En noviembre de 1998 Netscape comenzó a recuperar participación de mercado. Al tiempo que Netscape fue adquirido por AOL al principio de 1999, la ventaja competitiva de mantener Mozilla era suficientemente clara como para que uno de los primeros compromisos públicos de AOL fuera continuar apoyando el proyecto Mozilla, aunque todavía estaba en estado alfa.

4.2.2 Asegurar el futuro

Este modelo es para fabricantes de *hardware*. Las presiones del mercado han forzado a las compañías de *hardware* a escribir y mantener *software* (desde *device drivers* hasta herramientas de configuración, al nivel de sistemas operativos completos), que no es un centro de ganancias. Es un costo, a menudo sustancial.

En esta situación, liberar el *software* es una opción clara. No hay ninguna ganancia que perder. Lo que el fabricante gana es una cantidad muy grande de desarrolladores, una respuesta más rápida y flexible a las necesidades del usuario, y una mayor confiabilidad a través de las revisiones. Probablemente también ganará una mayor lealtad de parte de sus clientes, al realizar las adecuaciones de *software* que ellos necesitan.

Los productos de *hardware* tienen una vida finita de producción y soporte. Después de eso, los clientes están por su propia cuenta. Pero si tienen acceso al código de los *drivers* y pueden modificarlos de acuerdo a sus necesidades, habrá más clientes satisfechos que mostrarán lealtad a la compañía. Un buen ejemplo de adoptar este modelo fue la decisión de Apple Computer de abrir el código de Darwin, la base de su sistema operativo MacOSX, a mediados de marzo de 1999.

4.2.3 Expansión del mercado de servicios

En este modelo, se abre el código del *software* para crear una posición de mercado, no para *software* cerrado, sino para servicios. Esto es lo que hacen RedHat y otros distribuidores de Linux. Lo que realmente están vendiendo no es el *software*, sino el valor agregado por el ensamblaje y pruebas de un sistema operativo que está garantizado. Otros elementos de su proposición de valor incluyen soporte de instalación gratuito y la provisión de opciones para contratos de soporte continuo.

El efecto constructor de mercado del *software* libre puede ser extremadamente poderoso, especialmente para compañías que están inevitablemente en una posición de servicios. Un caso instructivo es el de Digital Creations, una casa de diseño de sitios *web* que comenzó en 1998 y se especializa en bases de datos complejas y sitios transaccionales. Su mayor herramienta es objeto de publicidad que ha pasado por varios nombres y encarnaciones, pero ahora se llama Zope. Cuando Digital Creations buscaba capital, los consultores que evaluaron cuidadosamente el nicho prospectivo de mercado, su gente y sus herramientas, recomendaron que la empresa abriera el código de Zope.

Desde el punto de vista de los estándares de la industria del *software*, esta decisión se ve como una jugada poco inteligente. La sabiduría convencional opina que el producto más importante de una compañía no debe ser regalada bajo ninguna circunstancia. Pero los consultores tenían dos razones. Una es que la base de Zope refleja el cerebro y las habilidades de la gente de Digital Creations. La segunda es que Zope generaría más valor como constructor de mercado que como herramienta secreta.

Para ver esto, comparemos dos escenarios diferentes. En el convencional, Zope permanece como el arma secreta de Digital Creations. Estipulemos que es un arma muy efectiva. Como resultado, la firma será capaz de despachar alta calidad en cortos tiempos, posiblemente.

Sería fácil satisfacer a los clientes, pero será difícil construir una base de clientes para comenzar. En cambio, los consultores dijeron que abriendo el código de Zope sería publicidad crítica para Digital Creations.

Se esperaba que los clientes que evaluaran Zope consideraran más eficiente contratar a los expertos que desarrollar experiencia Zope internamente. Un directivo de Zope ha confirmado que la estrategia del *software* libre ha "abierto muchas puertas que no se hubieran abierto de otra manera". Los clientes potenciales respondieron a la lógica de la situación y Digital Creations está prosperando.

4.2.4 Comercialización de accesorios

En este modelo, se venden accesorios para *software* libre. En un bajo nivel, se venden tazas y otros recuerdos. En un alto nivel, documentación de calidad profesional.

O'Reilly Associates, publicadores de muchos excelentes volúmenes de referencia sobre *software* libre, es un buen ejemplo de una compañía de accesorios. O'Reilly contrata y soporta a *hackers* conocidos de la cultura del *software* libre (como Larry Wall y Brian Behlendorf) como un modo de construir su reputación en el mercado elegido.

4.2.5 Liberar el futuro, vender el presente

En este modelo se distribuye *software* como binarios y fuentes con una licencia cerrada, pero que incluye una fecha de expiración en las previsiones de cierre. Por ejemplo, se podría escribir una licencia que permita la libre redistribución, prohíba el uso comercial sin una cuota, y garantice que el *software* se regirá por licencia GPL un año después de su fecha de lanzamiento, o si el productor desaparece.

Bajo este modelo, los clientes se aseguran que el producto es adaptable a sus necesidades, porque tienen las fuentes. El producto es a prueba de futuro, ya que la licencia garantiza que una comunidad de *software* libre puede retomar el producto si la compañía original deja de existir.

Como el precio de venta y el volumen están basados en estas expectativas de los clientes, la compañía gozará de ingresos que no hubiese obtenido si hubiera distribuido su producto exclusivamente en forma cerrada. Más aún, el viejo código llevado a licencia GPL recibirá una gran cantidad de revisión, arreglos y características menores, lo cual quita una parte de ese 75% de mantenimiento por parte de sus creadores.

La principal desventaja de este modelo es que su naturaleza cerrada tiende a inhibir la revisión y participación en las primeras fases del ciclo de vida, que es precisamente cuando más se las necesita.

4.2.6 Liberar el *software*, vender la marca

Este es un modelo de negocios especulativo. Se abre el código de una tecnología de *software*, se retiene una suite de pruebas o un conjunto de criterios de compatibilidad y se le vende a los usuarios una marca, certificando que su implementación de tecnología es compatible con todas las demás que lleven esa marca.

4.2.7 Liberar el *software*, vender el contenido

Este es otro modelo de negocios especulativo. Supone algo parecido a un servicio de suscripción. El valor no está en el *software* cliente ni en el servidor, sino en proveer información confiable. Entonces se abre el código de todo el *software* y se venden suscripciones al contenido. Cuando los *hackers* traduzcan el *software* cliente a nuevas plataformas y lo mejoren de diversas formas, el mercado automáticamente se expande.

4.3 Liberar o no liberar

Tras revisar los modelos de mercado que permiten el desarrollo libre, ahora podemos aproximarnos a la pregunta general acerca de cuándo tiene sentido (económicamente) usar un modelo abierto y cuándo conviene ser cerrado. Primero conviene aclarar cuáles son las ganancias de cada estrategia.

La aproximación cerrada permite recolectar rentas de secretos, pero, desde otro punto de vista, cierra las puertas a la posibilidad de una verdadera revisión independiente. La aproximación abierta establece las condiciones para una revisión independiente, pero no deja tomar ganancias de los secretos.

La ganancia de tener secretos está bien entendida ya que, tradicionalmente, los modelos de negocios de *software* se han construido a su alrededor. Hasta hace poco tiempo, las ganancias de las revisiones independientes no estaban bien entendidas. Sin embargo, desarrollos como el del sistema operativo Linux han enseñando que la revisión independiente, tal como se da en el modelo abierto de desarrollo, es el único método escalable de obtener alta calidad y confiabilidad.

Entonces, en un mercado competitivo, los clientes que busquen alta confiabilidad y calidad recompensarán a los productores de *software* que se dirijan hacia implementar *software* libre y que descubran cómo mantener un flujo de entradas de dinero en los mercados de servicios, valor agregado y soporte asociados al *software*.

Una ganancia igualmente importante de abrir el código es su utilidad como un medio de propagar estándares abiertos y construir mercados alrededor de ellos. El crecimiento dramático de Internet se debe en gran parte a que nadie es el poseedor del protocolo TCP/IP y nadie ha cerrado propietariamente los protocolos base de Internet.

Ningún consumidor de *software* elegiría racionalmente encerrarse en un monopolio controlado por el productor dependiendo de fuentes cerradas, si existe una alternativa libre de calidad aceptable. Este argumento toma más fuerza cuando el *software* se torna más crítico para el negocio del consumidor. Mientras más vital es, menos tolerará el consumidor ser controlado por una organización externa.

Finalmente, una ganancia importante para el consumidor del *software* libre es que este tipo de *software* es a prueba de futuro. Si las fuentes son abiertas, el consumidor tiene algunos recursos si el productor desaparece. Esto puede ser particularmente importante para asuntos relacionados con el *hardware*, ya que éste tiende a tener ciclos de vida cortos, pero el efecto es más general y se traduce en un aumento de valor para el *software* de fuentes abiertas.

4.3.1 Condiciones básicas para liberar *software*

Cuando las ganancias de los secretos son más altas que las de abrir las fuentes, es económicamente sensato mantener las fuentes cerradas. Pero cuando los retornos de abrir las fuentes son más altos que las ganancias proporcionadas por los secretos, es sensato moverse hacia la apertura de las fuentes.

En si misma, esta es una observación trivial. Se convierte en no trivial cuando las ganancias provenientes del *software* libre son más difíciles de medir y predecir que la renta proveniente de los secretos y que dicha ganancia es generalmente muy subestimada.

Aun así, podemos esperar que liberar algún *software* tenga ganancias cuando la confiabilidad, estabilidad y escalabilidad son críticas y cuando la corrección del diseño y la implementación no puede ser efectivamente verificada por otros medios distintos a la revisión independiente.

El deseo racional de un consumidor de evitar quedar atado a un productor monopólico incrementará su interés en el *software* libre (y, por lo tanto, el valor competitivo de mercado de los productores que implementen *software* libre) cuando el *software* es más crítico para el consumidor. Entonces, otro criterio empuja hacia la apertura de las fuentes cuando el *software* es un bien de capital crítico del negocio.

En lo que respecta al área de las aplicaciones, se ha observado que el modelo abierto de desarrollo crea efectos de confianza y situaciones en las que tanto el productor como el consumidor obtienen beneficios que, a través del tiempo, tienden a atraer más consumidores.

Frecuentemente es mejor tener una pequeña parte de ese mercado que se expande rápidamente, que una parte mayor de un mercado cerrado y estancado. De acuerdo a esto, y para *software* de infraestructura, es muy probable que la liberación del *software* tenga una mayor ganancia a largo plazo que una jugada de fuentes cerradas (para lucrar con la propiedad intelectual). Podemos completar esta lógica observando que liberar el *software* parece ser más exitoso generando mayores retornos que *software* cerrado cuando se trata de *software* que establece una infraestructura común de comunicaciones y cómputos.

Finalmente, debemos notar que los productores de servicios únicos o altamente diferenciados tienen mucho más temor que sus métodos sean copiados por competidores que los productores de servicios para los cuales los algoritmos críticos y las bases de conocimiento están ampliamente entendidas. Por esto, la liberación de *software* dominará cuando los métodos clave (o equivalentes funcionales) formen parte de un conocimiento común de ingeniería.

Desde la otra cara de la moneda, la liberación del *software* parece ser la última opción para compañías que tienen una posesión única de una tecnología de *software* generadora de valor, la cual es relativamente poco sensible a las fallas, puede ser verificada por medios distintos a la revisión independiente, no es crítica para el negocio y no tendría su valor incrementado substancialmente por efectos de red.

Es conveniente liberar el *software* cuando:

- La confiabilidad, estabilidad y escalabilidad son críticas.
- La corrección del diseño y la implementación sólo pueden ser verificadas por medio de revisiones independientes entre colegas.
- El *software* es crítico para el control del negocio por parte del consumidor.
- El *software* establece una infraestructura común de comunicaciones y cómputos.
- Los métodos clave (o equivalentes funcionales de ellos) forman parte de conocimientos comunes de ingeniería.

4.3.2 El momento adecuado para liberar el *software*

La historia del juego Doom ilustra como la presión del mercado y la evolución del producto pueden cambiar las magnitudes de ganancias entre fuentes abiertas y cerradas.

Cuando Doom fue lanzado a fines de 1993, su modo de animación de tiempo real, en primera persona, lo hacía algo totalmente único. No sólo era por el impacto visual de la técnica, sino que por muchos meses nadie podía darse cuenta como podía ser logrado en los microprocesadores de baja potencia de la época.

Esos secretos valían una seria ganancia. Además, la potencial ganancia de abrir las fuentes eran bajas ya que como un juego en solitario, el *software* incurría en costos de fallas tolerablemente bajos, no era terriblemente difícil de verificar, no era crítico para el negocio de ningún cliente y no se beneficiaba de los efectos de red. Era económicamente racional mantener a Doom en una estrategia de fuentes cerradas.

Sin embargo, el mercado alrededor de Doom no se mantenía quieto. Competidores inventaron equivalentes funcionales de sus técnicas de animación, y otros juegos similares comenzaron a aparecer. Cuando estos juegos se comenzaron a devorar la porción de mercado de Doom, el valor de ganancia de los secretos se desvaneció.

Además, los esfuerzos para expandir esa porción de mercado trajeron nuevos desafíos tecnológicos, tales como mejor confiabilidad, más características de juego, una mayor base de usuarios y múltiples plataformas. Con el advenimiento de los partidos multijugador y los servicios de juegos Doom, el mercado comenzó a desplegar ciertos efectos de red substanciales.

Todas estas tendencias aumentaron las ganancias de abrir las fuentes. En algún punto las curvas de ganancias se cruzaron y fue económicamente razonable liberar el código de Doom y cambiar el rumbo hacia hacer dinero en negocios secundarios, como antologías de escenarios de juegos. Y en un tiempo después de ese punto, la transición ocurrió.

Doom es un caso interesante de estudio porque no se trata ni de un sistema operativo, ni de un *software* de comunicaciones o red; por lo tanto, está bastante lejos de los ejemplos comunes y obvios del *software* libre.

De hecho, el ciclo de vida de Doom, completo con punto de cruce de curvas, puede comenzar a tipificar la naturaleza del *software* de hoy, en el cual las comunicaciones y la computación distribuida crean serios problemas de robustez/confiabilidad/escalabilidad que sólo son accesibles a través de la revisión entre colegas y frecuentemente cruzan los límites entre ambientes técnicos y entre actores que compiten.

Si las tendencias actuales continúan, el desafío central de la tecnología de *software* y de la administración de productos será saber cuándo liberar, para explotar el efecto de revisión entre colegas y capturar altos retornos en servicios y otros mercados secundarios. Existen incentivos de ingresos demasiado obvios como para dejar pasar el punto de cruce de curvas con mucho error en cualquier dirección.

La razón para decir que este es un tema serio es que tanto el grupo de usuarios como el grupo de talentos disponibles para ser reclutados como colaboradores para una determinada categoría de producto son limitados. Si dos productores son el primero y segundo en liberar las fuentes de código que compite en una función relativamente parecida, el primero atraerá la mayor cantidad de usuarios y la mayor cantidad de codesarrolladores (y los más motivados), mientras el segundo tomará lo que quede[47].

4.4 Las licencias del *software* libre

Si después de la evaluación correspondiente se decide que la mejor opción es liberar un determinado *software*, el siguiente paso es definir la base legal bajo la cual se hará la distribución.

[47] Cuando ser abierto, cuando ser cerrado (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node11.html). Mayo de 2001.

Para esto, se establece una licencia que se constituye el marco legal que delimita exactamente los derechos que los usuarios tienen sobre el *software* en cuestión. Por ejemplo, en la mayoría de los programas propietarios la licencia priva al usuario de los derechos de copia, modificación, préstamo, alquiler, uso en varias máquinas, etc. De hecho, las licencias suelen especificar que la propietaria del programa es la empresa editora del mismo, que simplemente vende derechos restringidos de uso del mismo.

En el mundo del *software* libre, la licencia bajo la que se distribuye un programa también va a ser de gran importancia. Normalmente, las condiciones de las licencias de *software* libre son el resultado de un compromiso entre varios objetivos hasta cierto punto contrapuestos. Entre ellos pueden citarse los siguientes:

- Garantizar algunas libertades básicas (de redistribución, de modificación, de uso) a los usuarios.

- Asegurar algunas condiciones impuestas por los autores (cita del autor en trabajos derivados, por ejemplo).

- Procurar que los trabajos derivados sean también *software* libre.

Los autores pueden elegir proteger su *software* con distintas licencias según el grado con que quieran cumplir cada uno de estos objetivos, y los detalles que quieran asegurar. De hecho, el autor puede (si así lo desea) distribuir su *software* con licencias diferentes por canales (y con precios) diferentes. Por lo tanto, el autor de un programa suele elegir con mucho cuidado la licencia bajo la que lo distribuye. Y los usuarios, y especialmente quien redistribuya o modifique el *software*, deben estudiar con cuidado su licencia.

Afortunadamente, aunque cada autor podría utilizar una licencia diferente para sus programas, en realidad casi todo el *software* libre usa una de las licencias más habituales (GPL, LGPL, Artistic, "estilo" BSD, "estilo" Netscape, etc.), algunas veces con ligeras variaciones. Para simplificar aún más las cosas, en los últimos tiempos han surgido organizaciones que promueven marcas que garantizan que todo el *software* que cubren está distribuido bajo licencias que aseguran ciertas condiciones razonables y simples de entender. Un ejemplo notable de esto es la marca Open Source.

En los siguientes apartados se analizarán algunas de las licencias bajo las que distribuye habitualmente el *software* libre[48].

4.4.1 GPL (Licencia Pública General de GNU)

La licencia pública general de GNU (GPL, según las iniciales de su nombre en inglés, General Public License) es la licencia bajo la cual se distribuye el *software* del proyecto GNU. Sin embargo, hoy día pueden encontrarse gran cantidad de *software* no relacionado con el proyecto GNU pero distribuido bajo la GPL, tal como el *kernel* Linux. La GPL se diseñó cuidadosamente para promover la producción de más *software* libre. Por ello, prohibe explícitamente algunas acciones sobre el *software* que podrían llevar a la integración de *software* protegido por la GPL en programas propietarios. La GPL usa como base legal la legislación sobre *copyright*, haciendo un uso muy interesante de ella, ya que se usa el *copyright* para promover la distribución de *software* que garantiza mucha más libertad a los usuarios que los trabajos habitualmente protegidos por *copyright*. Por lo tanto, algunas veces se dice que el *software* cubierto por la GPL está *copylef*, un interesante juego de palabras en inglés. (Ver Anexo A: la licencia pública general de GNU)

[48] Licencias, licencias, licencias (http://projects.openresources.com/libresoft-notes/libresoft-notes-es/node7.html). Mayo de 2001.

Las principales características de la GPL son las siguientes:

- Permite la redistribución binaria, pero sólo si se garantiza también la disponibilidad del código fuente. Esta disponibilidad puede garantizarse bien mediante su distribución simultánea, o mediante el compromiso de distribución a solicitud de quien recibe la versión binaria.

- Permite la redistribución fuente (y obliga a ella en caso de distribución binaria).

- Permite las modificaciones sin restricciones (siempre que el trabajo derivado quede también cubierto por la GPL).

- La integración completa sólo es posible con *software* cubierto por la GPL.

4.4.2 LGPL (GPL para bibliotecas)

La licencia pública de GNU para bibliotecas (LGPL, según las iniciales de su nombre en inglés, *Library General Public License*) fue diseñada por la Free Software Foundation para proteger las bibliotecas que se desarrollaban en el proyecto GNU, pero que quería que pudieran ser enlazadas con programas propietarios.

Las principales características de la LGPL son las siguientes:

- Cubre las bibliotecas del proyecto GNU y otro *software*.

- Diseñada para permitir el uso de bibliotecas libres con *software* propietario (por ejemplo, en el caso de un compilador libre).

- Es idéntica a la GPL cuando se redistribuye la biblioteca como tal.

- Permite la integración con cualquier otro *software*. En este caso, no hay prácticamente limitaciones.

4.4.3 BSD (Berkeley *Software* Distribution)

La licencia BSD cubre, entre otro *software*, las entregas de BSD (Berkeley *Software* Distribution). Estas fueron hechas por el CSRG (Computer Science Research Group) de la Universidad de California en Berkeley. Las entregas de BSD fueron la forma en que el CSRG distribuía su trabajo y las contribuciones de muchos otros programadores alrededor del sistema operativo Unix. De hecho, Unix era un sistema operativo propietario, y por ello durante mucho tiempo los usuarios de las entregas de BSD necesitaban también una licencia Unix. Las entregas de BSD se usaron como la base de muchos sistemas operativos propietarios, como SunOs de Sun Microsystems o Ultrix de DEC. Todos los sistemas derivados de las entregas de BSD componen la rama BSD del árbol Unix. La otra gran rama, normalmente llamada Sistema V, deriva más directamente del código hecho en AT&T.

Afortunadamente, a principios de los años 90 el CSRG hizo varias entregas que afirmaban que no incluían código propietario. Después de algunos litigios con los dueños de la licencia Unix se llegó a un acuerdo con la entrega de BSD-Lite, que fue reconocida como completamente libre de código propietario. Esa entrega fue el origen de NetBSD, FreeBSD y OpenBSD. La entrega de BSD-Lite fue el último trabajo realizado por el CSRG antes de desaparecer.

La licencia BSD es un buen ejemplo de una licencia permisiva, que casi no impone condiciones sobre lo que un usuario puede hacer con el *software*, incluyendo cobrar a los clientes por distribuciones binarias, sin la obligación de incluir el código fuente.

Probablemente esta licencia, además de la excelencia técnica del *software*, fue una de las razones principales para su uso en tantos sistemas propietarios derivados de Unix durante los años 80. Los principales puntos que establece la licencia son:

- Se permite la redistribución, uso y modificación del *software*.

- Las distribuciones deben incluir copias literales de la licencia, anuncio de *copyright* y una **negación de responsabilidad** (*disclaimer*).

- Debe incluirse reconocimiento del origen del *software* (la Universidad de California) en cualquier anuncio.

Para resumir, los redistribuidores pueden hacer casi cualquier cosa con el *software*, incluyendo usarlo para productos propietarios. Los autores sólo quieren que su trabajo sea reconocido. Es importante darse cuenta que este tipo de licencia no incluye ninguna restricción orientada a garantizar que los trabajos derivados sigan siendo libres. De hecho, muchos sistemas operativos derivados de versiones de BSD han sido distribuidos como *software* no libre. Puede argumentarse que esta licencia asegura "verdadero *software* libre", en el sentido que el usuario tiene libertad ilimitada con respecto al *software*, y puede decidir incluso redistribuirlo como no libre. Otras opiniones están más orientadas a destacar que este tipo de licencia no contribuye al desarrollo de más *software* libre, incluso cuando es básicamente la versión libre original redistribuida como un producto propietario. (Ver Anexo B: la licencia BSD)

4.4.4 Otras licencias

- *Artistic license*, usada como una de las licencias de distribución de Perl (la otra es la GPL).

- NPL (Licencia Pública de Netscape), incluye ciertos privilegios para el primer autor.

- X Windows System (X Consortium) License

- X*Free*86 Project License

- Licencia de Tcl/Tk

5. USO DEL *SOFTWARE* LIBRE

Como última parte de esta investigación se ha hecho un estudio sobre el uso del *software* libre. En este estudio se describe cómo implementar una solución completa aprovechando las ventajas que da el *software* libre. Específicamente se examina la implementación del sitio *web* *Open*Re*source*S.com, un centro de información sobre *software* libre, diseñado para manejar gran cantidad de información recibida tanto automática como manualmente y para que el sistema de información interno gestione todos los asuntos relativos a la publicidad que constituye su principal fuente de financiamiento.

La temática cubierta por el sitio *web* hizo que se considerara el uso de *software* libre como opción en su construcción, resultando además ser la más adecuada a las necesidades del mismo. Este estudio describe las herramientas utilizadas para la construcción de este sitio y el resultado final de cuando todas trabajan juntas coordinadamente. También se discute la idoneidad de este tipo de *software* para el diseño de sistemas de publicación electrónica.

Entre las herramientas utilizadas pueden destacarse el servidor *web* Apache, el lenguaje PHP utilizado para generar dinámicamente páginas HTML, el buscador htdig, la herramienta GLOBAL de generación de código HTML a partir de archivos escritos en diferentes lenguajes de programación, el gestor de foros w-agora, o el generador de estadísticas analog. Para cada una de ellas se describe en primer lugar su funcionalidad, se resume la historia y estado actual de las mismas y se analizan las posibilidades de su interrelación.

5.1 La publicación electrónica

Los primeros sistemas de publicación electrónica distribuidos se centraban en la publicación de información estática, o al menos de costosa actualización. Por ejemplo, WAIS (Wide Area Information Servers) permitía realizar búsquedas en ficheros previamente indexados o gopher una navegación jerárquica basada en directorios y ficheros. El WWW (World Wide *Web* o simplemente *web*) comenzó de forma similar, como un conjunto de páginas con un formato normalizado (HTML) en el que además de la propia información podían existir enlaces a otras páginas.

En el caso particular del *web*, en los primeros momentos las páginas se creaban *ad-hoc*, es decir, sin reutilizar la información ya existente en otros formatos digitales. Las herramientas básicas para la creación de estos documentos eran editores de texto adaptados para generar HTML. Cuando la información requería modificaciones (por ejemplo, en actualizaciones), se reeditaban individualmente las páginas necesarias. Todo ello suponía grandes costes de actualización, gran número de errores, y en general una falta casi absoluta de dinamicidad en los servidores de información. Aún hoy día muchos centros de información basados en *web* están construidos con base en este modelo.

La demanda de servicios ágiles y dinámicos, donde la información pueda adaptarse a las necesidades de cada usuario particular, ha supuesto la aparición de diversas herramientas que, combinadas, permiten producir páginas HTML a partir de datos en otros formatos. Así, existen programas que utilizan como datos códigos fuente de lenguajes de programación, formatos propietarios (como MS-Word), o formatos públicos (como ASCII o LaTeX). Pero también se pueden crear páginas automáticamente a partir de servicios como el correo electrónico o los grupos de USENET, generarlas automáticamente a partir de plantillas en función del tipo de usuario o del momento de conexión, integrarlas con bases de datos o en general con el sistema de información de la empresa, etc.

Por otra parte, el mundo de Internet ha utilizado generalmente *software* libre. En el caso particular del WWW, programas como Mosaic, el precursor de los navegadores *web* actuales, o el servidor *web* más usado actualmente, Apache, se han creado bajo la cobertura del *software* libre.

En este estudio se describe la arquitectura utilizada para crear el sitio *web* Open Resources, que reune una serie de características que lo hacen muy interesante:

- **Dinamicidad**, ya que está diseñado para crecer rápidamente, incorporando información tanto automáticamente (recibida de listas de correo, grupos de noticias) como manualmente (artículos escritos por expertos en distintas áreas).

- **Interactividad**, ya que proporciona búsquedas, permite la participación de los visitantes del sitio, etc.

- **Gestión comercial**, para el control de visitas, gestión de *banners*, *accounting*, etc.

- **Gestión de la apariencia**, que ha permitido independizar el contenido (la información) de su apariencia y generar las páginas dinámicamente.

- **Utilización de software libre en todo el sistema**, lo que le aporta cualidades como la fiabilidad, la rápida detección de errores y la facilidad para su corrección, su coste, etc.

5.2 *Software* libre en la publicación electrónica

¿Por qué es conveniente utilizar *software* libre a la hora de construir un sitio *web*? La respuesta se puede construir a partir de las siguientes ventajas, que son genéricas del *software* libre, pero lo son especialmente cuando se aplican al desarrollo de un sitio *web*.

5.2.1 Calidad del *software*

La disponibilidad sin trabas del código fuente de un programa permite verificar realmente la calidad del mismo. De igual modo, los errores pueden detectarse y corregirse mucho antes. Esto es especialmente importante para un sistema de publicación electrónica basado en el *web*, donde el mal funcionamiento de programas básicos significa la imposibilidad de acceso a la información (con el consiguiente perjuicio para los usuarios).

5.3.1 Mejora rápida del *software*

Con el modelo del *software* libre los programas pueden desarrollarse incrementalmente realizando modificaciones a programas, a su vez resultado de modificaciones anteriores. Esto, unido a la cantidad de programadores que pueden llegar a involucrarse en determinados proyectos de *software* libre, hace que la velocidad de desarrollo de las aplicaciones libres pueda ser mucho más alta que la de las propietarias.

Para el caso de un sitio *web* como Open Resources, el uso de *software* libre supone la garantía de utilizar *software* muy probado (reutilizado de productos o versiones diferentes) y a la vez de fácil y rápida actualización (las mejoras incorporadas y las erratas detectadas están disponibles inmediatamente).

5.3.1 Simplicidad de prueba

Cuando se usa *software* libre es posible realizar pruebas con muchos programas en un corto periodo de tiempo. Se buscan en el *web*, se consulta sobre ellos en listas especializadas, y se bajan de la red los que se desea probar. Las pruebas pueden ser muy completas, e incluir modificaciones si son necesarias para personalizar el programa.

Una vez elegido el *software* más adecuado para una tarea dada, el tiempo de puesta en producción es también muy corto, y no incluye fases imprescindibles en el caso de *software* propietario, como la adquisición o la negociación de una licencia especial.

Este proceso tan simple y flexible permite evaluar rápidamente herramientas que de otra forma ni siquiera se considerarían. El poder hacerlo permite utilizar tecnología de punta y proporcionar rápidamente nuevos servicios, algo que en el campo de los servicios de información en *web* es absolutamente imprescindible. Por ejemplo, en el caso de Open Resources se evaluaron herramientas como crit para añadir comentarios a documentos HTML ya existentes, visibles al navegar a través de una interfaz determinada.

5.3 El *web* de Open Resources

El sitio Open Resources está dirigido a programadores involucrados en proyectos de *software* libre, usuarios de este *software* y, en general, cualquier usuario informático interesado en esta forma de entender el negocio de la producción y distribución de *software*. Por tanto el diseño de la arquitectura debe estar enfocado a obtener un sitio *web* atractivo, fácil de mantener y de ampliar.

5.4 El diseño del sistema

Como en cualquier otro sistema de publicación, el núcleo es la información que se va a publicar. En el caso de Open Resources, dicha información es de diferentes tipos (código de programas, artículos, mensajes de correo, etc.), está generada por diferentes tipos de autores (colaboradores diarios que escriben noticias, regulares que escriben artículos o esporádicos que envían mensajes de correo electrónico), y tiene diferentes periodos de validez (las noticias más cortas que los artículos, por ejemplo). Por todo ello fue difícil elegir un formato común para toda la información.

En principio, el formato HTML podría considerarse como la opción natural' para la información no generada de forma automática (los mensajes de correo, el código fuente, etc. utilizan sus formatos específicos y se han usado las herramientas descritas posteriormente para convertirlos en HTML), pues es en el que se distribuye dicha información. Sin embargo, elegir esta opción significaría ligar el contenido, la información en sí misma, al formato. Para evitarlo se decidió emplear otro formato, que fuese estándar y pudiese ser utilizado en cualquier plataforma. Estas restricciones descartaban cualquier formato binario, dejando solo los formatos basados en ASCII. Sin embargo, el ASCII plano no permite dotar a los documentos de estructura (distinguir títulos, secciones, etc.) por lo que se barajaron diversas opciones: SGML, LaTeX, etc. Al final, la elección fue LaTeX a causa de la existencia previa de herramientas como GSyC-doc que facilitaban su conversión en HTML de forma muy controlada y flexible.

Las diferentes fuentes de información del sistema que convergen en el HTML que el servidor enviará a Internet son las siguientes:

- **HTML**, archivos con código directamente escrito en HTML como por ejemplo la gestión de los banners de los mayoristas de publicidad.

- **LaTeX**, los artículos, noticias y en general toda la información generada por los trabajadores o colaboradores de Open Resources. Usando la herramienta GSyC-doc estos ficheros se convertirán a HTML.

- **Código fuente**, en diferentes lenguajes (como C o Java), que serán convertidos en ficheros HTML por la herramienta GLOBAL.

- **E-mail**, los mensajes de correo electrónico, recibidos de las listas de correo a las que está suscrito el sistema, se convierten también en páginas HTML usando la herramienta mhonarc.

- **http**, el protocolo que se usa para transferir las páginas HTML desde el servidor hasta el navegador del usuario. Dicho protocolo permite también el traslado de información en sentido contrario, mecanismo empleado en Open Resources para realizar los foros de debate.

A partir de estas fuentes se genera la información que se almacenará en el sitio *web*. Lo usual es alimentar con ella directamente al servidor, en este caso Apache. Es decir, el conjunto de información generado suele ser directamente el que se muestra en Internet.

Sin embargo, para hacer más modificable la estructura de las páginas, insistiendo en la idea de separar contenidos de presentación, se decidió generar las páginas finales dinámicamente. Para ello se parte de unas estructuras HTML ya construidas (cabeceras, menús, publicidad, etc.) en las que se inserta la información. Para hacerlo se empleó el lenguaje de programación PHP, que fue diseñado con ese objetivo.

Por otro lado, para realizar un seguimiento del sistema se utiliza una herramienta de análisis de los archivos históricos del servidor *web*.

Finalmente, la gestión de publicidad se realiza con dos mecanismos básicos:

- Todas las páginas del sitio se sirven mediante una plantilla escrita en PHP, que incluye, además del código HTML correspondiente a cada página, llamadas a programas que insertan la publicidad dinámicamente. Estos programas están escritos también en PHP, y permiten funcionalidades típicas como rotación de anuncios o selección del anuncio a mostrar según la sección.

- Para llevar cuenta de las veces que se sirve cada anuncio se utilizan los archivos históricos. Básicamente, se utiliza el mismo *software* para llevar la contabilidad de las páginas servidas y para llevar la contabilidad de los anuncios servidos. A partir de esta información se puede facturar a los anunciantes, y contrastar con sus propios ficheros históricos (cuando los tienen).

5.5 *Software* usado en OpenResources.com

Open Resources fue construido con el requisito de ser un sitio de alta disponibilidad, con pocas fallas y capaz de servir a gran cantidad de usuarios. Las herramientas informáticas utilizadas para conseguirlo se describen a continuación.

5.5.1 GNU/Linux

GNU/Linux es el sistema operativo libre de mayor difusión, probablemente el segundo sistema operativo más utilizado en ordenadores personales tras MS-Windows.

Es un sistema operativo tipo Unix, multi-tarea y multi-usuario. Además, dispone de una gran cantidad de *software*, desde el específicamente dedicado a los servicios de Internet hasta el *software* tradicional en ordenadores personales (hojas de cálculo, editores, etc.)

De entre las distribuciones de GNU/Linux existentes en la actualidad, se ha elegido Debian, que resulta ser una de las más habituales, muy estable, y la que incluye el mayor número de paquetes *software*.

5.5.2 Apache

El proyecto Apache es un esfuerzo para desarrollar y mantener un servidor HTTP para varios sistemas operativos modernos, como UNIX o Windows NT. Su objetivo es proporcionar un servidor seguro, eficiente y extensible.

La historia de Apache comenzó en febrero de 1995. Entonces el servidor *web* más popular en Internet era el *deamon* de HTTP desarrollado por Rob McCool en el National Center for Supercomputing Applications (NCSA), en la Universidad de Illinois en Urbana-Champaign.

Sin embargo, su desarrollo se había paralizado, por lo que un grupo de administradores (a través del correo electrónico) se pusieron de acuerdo para coordinar los parches que relizaban. Utilizando como base NCSA httpd 1.3 añadieron algunos de los parches más habituales, los probaron en sus servidores y realizaron la primera versión oficial de Apache (la 0.6.2) en abril de 1995. Después de un completo rediseño, la migración a numerosas plataformas y una serie de pruebas intensivas, en diciembre de 1995 vio la luz la versión 1.0 de Apache. En menos de un año se convirtió en el servidor más utilizado en Internet. En mayo de 1999 la estadística de Netcraft afirma que el 57% de los servidores *web* de Internet usan Apache (el 61% si se añaden sus derivados) lo que le hace el servidor más usado.

Open Resources utiliza Apache como servidor y por tanto constituye el núcleo central del *software* utilizado, al que alimentan o con el que se enlazan el resto de los elementos descritos en esta sección.

5.5.3 PHP

PHP es un lenguaje de tipo *script*, integrado con HTML, que se ejecuta en los servidores HTTP (por ejemplo en Apache) y del que existen intérpretes para múltiples plataformas.

Según la encuesta de Netcraft en febrero de 1999 se usaba en 345,549 dominios y en 101140 direcciones IP. Tiene la particularidad de integrarse de forma muy eficiente con Apache en forma de módulo.

El sitio Open Resources genera las páginas HTML dinámicamente, utilizando PHP para construirlas a partir de plantillas pre-configuradas. De esta forma, la gestión de la apariencia, de la publicidad, etc. se puede realizar de forma independiente de los contenidos.

5.5.4 GSyC-doc

GSyC-doc es el nombre que engloba un conjunto de herramientas escritas en Perl para convertir diversos documentos, fundamentalmente pensado para convertir documentos escritos en LaTeX, en documentos HTML. Estas herramientas proporcionan mecanismos para enlazar los documentos, configurar su apariencia, actualizarlos parcialmente manteniendo la coherencia de los enlaces, etc.

GSyC-doc se ha desarrollado y usado sobre la distribución Debian de GNU/Linux, aunque debería funcionar (probablemente sin modificaciones) sobre la mayoría de sistemas operativos (especialmente los tipo Unix).

GSyC-doc se basa en las siguientes herramientas:

- GNU Make.
- LaTeX2HTML.
- Perl.
- TeX (y LaTeX). Usando la distribución teTeX, aunque cualquier distribución moderna de TeX y LaTeX debería servir.

El formato básico de la información generada en Open Resources es LaTeX, por lo que la herramienta GSyC-doc constituye una parte fundamental a la hora de generar y configurar el HTML que se distribuye finalmente a través del servidor.

5.5.5 htdig

htdig es un sistema completo para indexar y realizar búsquedas sobre un pequeño dominio o sobre una intranet. Este sistema no está pensado para reemplazar los sitios de búsqueda e indexación tradicionales de Internet como Lycos, Infoseek, Webcrawler o AltaVista, sino para cubrir las necesidades de búsquedas en una empresa, en el servidor de una universidad o incluso sobre solamente una determinada sección de un sitio *web*. htdig se desarrolló en la Universidad de San Diego (de ahí parte de su nombre) como mecanismo para buscar en los varios servidores del campus.

5.5.6 global

GLOBAL es un sistema de etiquetado de código fuente que funciona de la misma forma sobre diversas plataformas.

Los lenguajes de programación que soporta son C, Yacc y Java. Una de sus características más interesantes es que permite localizar una función determinada entre distintos ficheros de código fuente, mostrando el código con una sola pulsación de ratón.

Además permite localizar elementos rápidamente (tanto su definición como referencias a ellos), detectar repetidos, tratar un árbol de código fuente con sus subdirectorios como una única entidad, manejar expresiones regulares en las búsquedas (POSIX 1003.2), etiquetar el código, etc.

Uno de los fundamentos del *software* libre es precisamente poder acceder al código fuente. Open Resources quiere facilitar el acceso al código fuente de cualquier programa libre, dándole formato con GLOBAL.

5.5.7 w-agora

w-agora es un programa escrito en PHP para instalar, gestionar y configurar foros de debate basados en *web*. Permite instalar foros, BBS, libros de visitas y todos los elementos relacionados; puede además ser adaptado con facilidad para distribuir informaciones, anuncios, difundir actualizaciones de *software*, publicar FAQs, permitir que los usuarios carguen información en el servidor, etc.

w-agora constituye el mecanismo básico de interacción de los visitantes de Open Resources, y les permite añadir noticias, comentar los artículos, hacer sugerencias, etc.

5.5.8 mhonarc

mhonarc es un programa escrito en Perl, lo que hace que pueda usarse en multitud de plataformas, cuyo propósito es convertir mensajes de correo electrónico (RFC 822 y MIME) en HTML. Además mhonarc puede traducir carpetas o mailboxes tipo Unix en código HTML, eliminar o añadir mensajes a ficheros HTML creados por el propio mhonarc, generar índices de los mensajes, utilizar filtros para los mensajes, etc. Todo ello con la facilidad añadida de adaptar su presentación.

Open Resources ofrece como uno de sus múltiples servicios la posibilidad de consultar algunas de las listas de correo más útiles para la comunidad del *software* libre, permitiendo a sus visitantes navegar por ellas en formato HTML, para lo cual se emplea mhonarc.

5.5.4 analog

analog es un programa para analizar los archivos históricos de servidores *web*. Con él se puede conocer qué páginas son las más populares, de qué países proceden los visitantes, quiénes han tratado de seguir enlaces que no existen y desde dónde, etc. Además los resultados pueden visualizarse a través de un visor de HTML.

analog es un *software* muy configurable (permite elegir entre más de 27 tipos de informes en 26 idiomas distintos) y se puede utilizar en multitud de plataformas aparte de GNU/Linux (Windows (3.1, 95 y NT), DOS, Mac, diferentes tipos de Unix (TM), etc.) y entiende el formato de varios servidores *web* (Microsoft IIS, Netscape, WebSTAR, Netpresenz) aparte de los formatos estándares (NCSA, W3, etc.).

5.5 Funcionamiento del sistema y conclusiones

El número de visitas a este sitio *web* está creciendo de forma apreciable desde su lanzamiento en enero de 1999. Ello demuestra que utilizar solamente *software* libre es viable (el sitio realmente se ha podido construir), eficiente (con un simple PC se espera atender cientos de peticiones concurrentes), robusto (como demuestra el tiempo de funcionamiento sin fallas) y desde luego adaptable (pues tanto la organización física, como el aspecto gráfico pueden ser modificados con suma facilidad).

Según los comentarios de los creadores y administradores del sitio, aunque la decisión inicial de utilizar *software* libre fue motivada fundamentalmente por la temática del sitio *web*, la decisión ha probado ser acertada. Toda la funcionalidad requerida ha podido ser realizada usando *software* libre. En muchos casos, de hecho, puede decirse que ha podido ser realizada gracias al *software* libre, ya que no existen herramientas propietarias adecuadas para muchas de las tareas. Por otra parte, el nivel de flexibilidad de uso de las herramientas libres ha permitido adaptarlas completamente a las necesidades del sitio. En general, puede decirse que el sitio *web* se ha beneficiado extraordinariamente del uso de *software* libre[49].

[49] Software libre en la publicación electrónica: el ejemplo de Open Resources (http://www.it.uc3m.es/~pubelec99/actas/doc/open_resources.html). Mayo de 2001.

CONCLUSIONES

1. El *software* libre es aquel que se distribuye con la autorización de usarlo, copiarlo, examinarlo, modificarlo y redistribuirlo, literal o modificado, gratis o mediante una remuneración económica. Establece la libertad de correr el programa, con cualquier propósito, la libertad de estudiar cómo funciona el programa y adaptarlo a necesidades particulares, la libertad de distribuir copias, de modificar el programa y liberar las modificaciones al público.

2. El *software* libre ha provocado una serie de cambios en las dimensiones económica, legal, social y tecnológica, en vista de que gracias a él, ahora es posible usar *software* sin problemas legales a un bajo precio. Se ha formado una comunidad mundial de desarrolladores que colaboran entre sí y ha provocado el resurgimiento de las tecnologías estilo Unix, con sus acostumbrados altos estándares.

3. El modelo abierto de desarrollo surgió como una respuesta a las características del *software* libre. Es un modelo colaborativo y evolutivo que produce versiones más completas cíclicamente, hace uso intensivo de la reutilización de código, incluso de terceros, y se apoya fuertemente en los usuarios para hacer el proceso de pruebas y depuración. Evita muchos de los inconvenientes de otros modelos como la imposibilidad de conocer todos los requisitos inicialmente, la necesidad de contar con equipos de trabajo establecidos, imposibilidad de entregar *software* funcional en cada ciclo y otros.

4. El *software* libre se distribuye amparándose en alguna de varias licencias que establecen claramente cuáles son los derechos que tienen los usuarios sobre el *software* en cuestión. De estas licencias, la más utilizada es la GPL, la cual implementa un mecanismo legal para evitar que el *software* desarrollado a partir de un *software* libres se convierta posteriormente en *software* no libre.

5. Es posible tener ingresos al comercializar *software* libre, aunque estos son difíciles de medir ya que se refieren a aspectos tales como ganar posición en el mercado, venta de servicios asociados y asegurar el mantenimiento a bajo costo en el futuro. No siempre conviene liberar el *software*, pero existirán casos donde es claramente la mejor opción, en cuyo caso es importante hacerlo en el momento preciso.

6. Las aplicaciones candidatas a desarrollarse como *software* libre deben ser aquellas en las cuales la confiabilidad, estabilidad y escalabilidad son críticas, establecen una infraestructura de comunicaciones y cómputos, sus componentes claves son de dominio público y necesitan una revisión independiente para asegurar su calidad.

RECOMENDACIONES

1. En vista la calidad del *software* libre y de la disponibilidad de su código fuente, las instituciones educativas deberían basar en éste su proceso de enseñanza-aprendizaje, extendiendo de esta manera las opciones de recursos educativos con la última tecnología sin incurrir en gastos excesivos.

2. Los desarrolladores guatemaltecos interesados en crear productos de calidad aceptable mundialmente deben considerar como una de las mejores alternativas el modelo abierto de desarrollo, especialmente si las tecnologías utilizadas son de dominio público (tales como las usadas en la infraestructura de Internet) ya que de esta manera ampliarán sus recursos de colaboración con las ventajas que estos tienen.

3. Las empresas comercializadoras de *software* deberían examinar sus productos con el objetivo de descubrir para cuáles es mejor liberarlos en lugar mantenerlos restringidos, en vista del crecimiento de su cuota de mercado, el mejoramiento de la calidad de su producto o el crecimiento de su reputación como empresa que podrían tener.

4. Los usuarios de *software* con recursos limitados tienen en el *software* libre una gran cantidad de aplicaciones que pueden utilizar en lugar de aquellas que no pueden comprar y que aún así utilizan ilegalmente. En el caso de los usuarios con demandas de calidad, estabilidad, escalabilidad y cuyo *software* no puede limitarse a los cambios que la empresa desarrolladora hace periódicamente, definitivamente la mejor opción es el *software* libre.

BIBLIOGRAFÍA

1. Cuando ser abierto, Cuando ser cerrado (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node11.html). Mayo de 2001.

2. Cultivando la noósfera (http://www.geocities.com/jagem/noosfera.html). Febrero de 2001.

3. Definición de *Open Source* (http://members.nbci.com/drodrigo/es-osd.html). Septiembre de 2000.

4. Fuente Abierta - Proyecto GNU – Fundación del *Software* Libre (FSF) (http://www.gnu.org/philosophy/free-*software*-for-freedom.es.html). Septiembre de 2000.

5. http://www.ciudad.com.ar/ar/portales/tecnologia/nota/0,1357,4450,00.html. Mayo de 2001.

6. La Catedral y el Bazar de Eric S. Raymond (http://www.sindominio.net/biblio*web*/telematica/catedral.html). Noviembre de 2000.

7. La empresa ante el *software* libre: La empresa basada en *software* libre (http://oasis.dit.upm.es/~jantonio/documentos/empresa/empresa-7.html). Mayo de 2001.

8. La ilusión de la manufactura (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node4.html). Mayo de 2001.

9. Licencias, licencias, licencias (http://projects.*openresources*.com/libresoft-notes/libresoft-notes-es/node7.html). Mayo de 2001.

10. Modelos de valor de venta indirecto (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node8.html). Mayo de 2001.

11. *Open Source*: los programas íntegros. (http://www.baquia.com/com/legacy/8512.html). Febrero de 2001.

12. Peña Rodríguez, José Alejandro. Planificación del recurso humano en proyectos de ingeniería de *software*. Tesis Ing. en Sistemas. Guatemala: Universidad Mariano Gálvez de Guatemala.

13. Portal de Ciudad Internet (http://www.ciudad.com.ar/ar/portales/tecnologia/nota/0,1357,4462,00.html). Mayo de 2001.

14. Pressman, Roger S. **Ingeniería del *software*, un enfoque práctico.** 3ª Edición. s. l.: Editorial McGraw-Hill, 1998.

15. Pressman, Roger S. **Ingeniería del *software*, un enfoque práctico.** 4ª Edición. s. l.: Editorial McGraw-Hill, 2000.

16. Qué es el *software* libre? - El Proyecto GNU - Fundación para el *Software* Libre (FSF) (http://www.gnu.org/philosophy/free-sw.es.html). Septiembre de 2000.

17. Razones para cerrar las fuentes (http://members.nbci.com/drodrigo/magic-cauldron/es-magic-cauldron/node7.html). Mayo de 2001.

18. Romero Ávalos, Orlando Rafael y José Luis Sánchez Lemus. Gestión de proyectos de *software*, estimación y planificación de tiempos y costos de un proyecto de *software*. Tesis Ing. en Sistemas. Guatemala: Universidad de San Carlos de Guatemala, 1998.

19. Sobre *open source*. (http://aula.linux.org.ar/biblioteca/sobre_*open_source*.htm). Febrero de 2001.

20. *Software* libre en la publicación electrónica: el ejemplo de *Open Resources* (http://www.it.uc3m.es/~pubelec99/actas/doc/*open_resources*.html). Mayo de 2001.

21. *Software* libre, el negocio del siglo XXI. (http://www.baquia.com/com/20000918/art00044.print.html). Febrero de 2001.

22. Sommerville, Ian. **Ingeniería de *software*.** s. l.: Editorial Addison-Wesley Iberoamericana.

23. Soria Oliva, Juan Carlos. Comparación del desarrollo de sistemas de *software* utilizando la metodología tradicional versus la aplicación de prototipos rápidos estructurados. Tesis Ing. en Sistemas. Guatemala: Universidad de San Carlos de Guatemala: 1992.

24. Tipos de licencias para *software* redistribuible libremente. (http://www.gsyc.inf.uc3m.es/sobre/pub/novatica-mono/). Octubre de 2000.

ANEXO A

La licencia pública general de GNU

Esta es la conocida GNU *Public License* (GPL) versión 2 (de junio de 1.991) que cubre la mayor parte del *software* de la Free Software Foundation, y muchos más programas. La traducción al castellano ha sido revisada por Richard Stallman, pero no tienen ningún valor legal, ni ha sido comprobada de acuerdo a la legislación de ningún país en particular.

Las versiones traducidas no son oficiales. En términos legales, la versión original (en inglés) de la GPL es la que especifica los términos de distribución actuales para los programas GNU.

La razón de porqué la FSL no aprueba como oficialmente válidas estas traducciones es porque chequearlas sería difícil y costoso (necesitando la ayuda de abogados bilingües en otros países). Por otro lado, podrían introducirse errores que provocarían que el propósito de la licencia no sea entendido correctamente.

Se permite publicar traducciones de la GPL o la LGPL en otros idiomas, con la condición de que (1) se identifique las traducciones como no oficiales, para informar a la gente que ellas legalmente no cuentan como sustitutas de la versión auténtica, y (2) se acuerda instalar cambios cuando se requiera[50].

[50] http://www.gnu.org/copyleft/copyleft.es.html#translations (http://www.gnu.org/copyleft/copyleft.es.html - translations). Julio de 2001.

Versión en español

Preámbulo

Las licencias que cubren la mayor parte del software están diseñadas para quitarle a usted la libertad de compartirlo y modificarlo. Por el contrario, la Licencia Pública General de GNU pretende garantizarle la libertad de compartir y modificar software libre, para asegurar que el software es libre para todos sus usuarios. Esta Licencia Pública General se aplica a la mayor parte del software del la Free Software Foundation y a cualquier otro programa si sus autores se comprometen a utilizarla. (Existe otro software de la Free Software Foundation que está cubierto por la Licencia Pública General de GNU para Bibliotecas). Si quiere, también puede aplicarla a sus propios programas.

Cuando hablamos de software libre, estamos refiriéndonos a libertad, no a precio. Nuestras Licencias Públicas Generales están diseñadas para asegurarnos de que tenga la libertad de distribuir copias de software libre (y cobrar por ese servicio si quiere), de que reciba el código fuente o que pueda conseguirlo si lo quiere, de que pueda modificar el software o usar fragmentos de él en nuevos programas libres, y de que sepa que puede hacer todas estas cosas.

Para proteger sus derechos necesitamos algunas restricciones que prohiban a cualquiera negarle a usted estos derechos o pedirle que renuncie a ellos. Estas restricciones se traducen en ciertas obligaciones que le afectan si distribuye copias del software, o si lo modifica.

Por ejemplo, si distribuye copias de uno de estos programas, sea gratuitamente, o a cambio de una contraprestación, debe dar a los receptores todos los derechos que tiene. Debe asegurarse de que ellos también reciben, o pueden conseguir, el código fuente. Y debe mostrarles estas condiciones de forma que conozcan sus derechos.

Protegemos sus derechos con la combinación de dos medidas:

1. Ponemos el software bajo copyright y

2. le ofrecemos esta licencia, que le da permiso legal para copiar, distribuir y/o modificar el software.

También, para la protección de cada autor y la nuestra propia, queremos asegurarnos de que todo el mundo comprende que no se proporciona ninguna garantía para este software libre. Si el software se modifica por cualquiera y éste a su vez lo distribuye, queremos que sus receptores sepan que lo que tienen no es el original, de forma que cualquier problema introducido por otros no afecte a la reputación de los autores originales.

Por último, cualquier programa libre está constantemente amenazado por patentes sobre el software. Queremos evitar el peligro de que los redistribuidores de un programa libre obtengan patentes por su cuenta, convirtiendo de facto el programa en propietario. Para evitar esto, hemos dejado claro que cualquier patente debe ser pedida para el uso libre de cualquiera, o no ser pedida.

Los términos exactos y las condiciones para la copia, distribución y modificación se exponen a continuación.

Licencia Pública General de GNU
Términos y condiciones para la copia, distribución y modificación

0. Esta Licencia se aplica a cualquier programa u otro tipo de trabajo que contenga una nota colocada por el tenedor del copyright diciendo que puede ser distribuido bajo los términos de esta Licencia Pública General. En adelante, ``Programa'' se referirá a cualquier programa o trabajo que cumpla esa condición y ``trabajo basado en el Programa'' se referirá bien al Programa o a cualquier trabajo derivado de él según la ley de copyright. Esto es, un trabajo que contenga el programa o una proción de él, bien en forma literal o con modificaciones y/o traducido en otro lenguaje. Por lo tanto, la traducción está incluida sin limitaciones en el término ``modificación''. Cada concesionario (licenciatario) será denominado ``usted''.

Cualquier otra actividad que no sea la copia, distribución o modificación no está cubierta por esta Licencia, está fuera de su ámbito. El acto de ejecutar el Programa no está restringido, y los resultados del Programa están cubiertos únicamente si sus contenidos constituyen un trabajo basado en el Programa, independientemente de haberlo producido mediante la ejecución del programa. El que esto se cumpla, depende de lo que haga el programa.

1. Usted puede copiar y distribuir copias literales del código fuente del Programa, según lo has recibido, en cualquier medio, supuesto que de forma adecuada y bien visible publique en cada copia un anuncio de copyright adecuado y un repudio de garantía, mantenga intactos todos los anuncios que se refieran a esta Licencia y a la ausencia de garantía, y proporcione a cualquier otro receptor del programa una copia de esta Licencia junto con el Programa.

Puede cobrar un precio por el acto físico de transferir una copia, y puede, según su libre albedrío, ofrecer garantía a cambio de unos honorarios.

2. Puede modificar su copia o copias del Programa o de cualquier porción de él, formando de esta manera un trabajo basado en el Programa, y copiar y distribuir esa modificación o trabajo bajo los términos del apartado 1, antedicho, supuesto que además cumpla las siguientes condiciones:

a. Debe hacer que los ficheros modificados lleven anuncios prominentes indicando que los ha cambiado y la fecha de cualquier cambio.

b. Debe hacer que cualquier trabajo que distribuya o publique y que en todo o en parte contenga o sea derivado del Programa o de cualquier parte de él sea licenciada como un todo, sin carga alguna, a todas las terceras partes y bajo los términos de esta Licencia.

c. Si el programa modificado lee normalmente órdenes interactivamente cuando es ejecutado, debe hacer que, cuando comience su ejecución para ese uso interactivo de la forma más habitual, muestre o escriba un mensaje que incluya un anuncio de copyright y un anuncio de que no se ofrece ninguna garantía (o por el contrario que sí se ofrece garantía) y que los usuarios pueden redistribuir el programa bajo estas condiciones, e indicando al usuario cómo ver una copia de esta licencia. (Excepción: si el propio programa es interactivo pero normalmente no muestra ese anuncio, no se requiere que su trabajo basado en el Programa muestre ningún anuncio).

Estos requisitos se aplican al trabajo modificado como un todo. Si partes identificables de ese trabajo no son derivadas del Programa, y pueden, razonablemente, ser consideradas trabajos independientes y separados por ellos mismos, entonces esta Licencia y sus términos no se aplican a esas partes cuando sean distribuidas como trabajos separados. Pero cuando distribuya esas mismas secciones como partes de un todo que es un trabajo basado en el Programa, la distribución del todo debe ser según los términos de esta licencia, cuyos permisos para otros licenciatarios se extienden al todo completo, y por lo tanto a todas y cada una de sus partes, con independencia de quién la escribió.

Por lo tanto, no es la intención de este apartado reclamar derechos o desafiar sus derechos sobre trabajos escritos totalmente por usted mismo. El intento es ejercer el derecho a controlar la distribución de trabajos derivados o colectivos basados en el Programa.

Además, el simple hecho de reunir un trabajo no basado en el Programa con el Programa (o con un trabajo basado en el Programa) en un volumen de almacenamiento o en un medio de distribución no hace que dicho trabajo entre dentro del ámbito cubierto por esta Licencia.

3. Puede copiar y distribuir el Programa (o un trabajo basado en él, según se especifica en el apartado 2, como código objeto o en formato ejecutable según los términos de los apartados 1 y 2, supuesto que además cumpla una de las siguientes condiciones:

a. Acompañarlo con el código fuente completo correspondiente, en formato electrónico, que debe ser distribuido según se especifica en los apartados 1 y 2 de esta Licencia en un medio habitualmente utilizado para el intercambio de programas, o

b. Acompañarlo con una oferta por escrito, válida durante al menos tres años, de proporcionar a cualquier tercera parte una copia completa en formato electrónico del código fuente correspondiente, a un coste no mayor que el de realizar físicamente la distribución del fuente, que será distribuido bajo las condiciones descritas en los apartados 1 y 2 anteriores, en un medio habitualmente utilizado para el intercambio de programas, o

c. Acompañarlo con la información que recibiste ofreciendo distribuir el código fuente correspondiente. (Esta opción se permite sólo para distribución no comercial y sólo si usted recibió el programa como código objeto o en formato ejecutable con tal oferta, de acuerdo con el apartado b anterior).

Por código fuente de un trabajo se entiende la forma preferida del trabajo cuando se le hacen modificaciones. Para un trabajo ejecutable, se entiende por código fuente completo todo el código fuente para todos los módulos que contiene, más cualquier fichero asociado de definición de interfaces, más los guiones utilizados para controlar la compilación e instalación del ejecutable. Como excepción especial el código fuente distribuido no necesita incluir nada que sea distribuido normalmente (bien como fuente, bien en forma binaria) con los componentes principales (compilador, kernel y similares) del sistema operativo en el cual funciona el ejecutable, a no ser que el propio componente acompañe al ejecutable.

Si la distribución del ejecutable o del código objeto se hace mediante la oferta acceso para copiarlo de un cierto lugar, entonces se considera la oferta de acceso para copiar el código fuente del mismo lugar como distribución del código fuente, incluso aunque terceras partes no estén forzadas a copiar el fuente junto con el código objeto.

4. No puede copiar, modificar, sublicenciar o distribuir el Programa excepto como prevé expresamente esta Licencia. Cualquier intento de copiar, modificar sublicenciar o distribuir el Programa de otra forma es inválida, y hará que cesen automáticamente los derechos que te proporciona esta Licencia. En cualquier caso, las partes que hayan recibido copias o derechos de usted bajo esta Licencia no cesarán en sus derechos mientras esas partes continúen cumpliéndola.

5. No está obligado a aceptar esta licencia, ya que no la ha firmado. Sin embargo, no hay hada más que le proporcione permiso para modificar o distribuir el Programa o sus trabajos derivados. Estas acciones están prohibidas por la ley si no acepta esta Licencia. Por lo tanto, si modifica o distribuye el Programa (o cualquier trabajo basado en el Programa), está indicando que acepta esta Licencia para poder hacerlo, y todos sus términos y condiciones para copiar, distribuir o modificar el Programa o trabajos basados en él.

6. Cada vez que redistribuya el Programa (o cualquier trabajo basado en el Programa), el receptor recibe automáticamente una licencia del licenciatario original para copiar, distribuir o modificar el Programa, de forma sujeta a estos términos y condiciones. No puede imponer al receptor ninguna restricción más sobre el ejercicio de los derechos aquí garantizados. No es usted responsable de hacer cumplir esta licencia por terceras partes.

7. Si como consecuencia de una resolución judicial o de una alegación de infracción de patente o por cualquier otra razón (no limitada a asuntos relacionados con patentes) se le imponen condiciones (ya sea por mandato judicial, por acuerdo o por cualquier otra causa) que contradigan las condiciones de esta Licencia, ello no le exime de cumplir las condiciones de esta Licencia. Si no puede realizar distribuciones de forma que se satisfagan simultáneamente sus obligaciones bajo esta licencia y cualquier otra obligación pertinente entonces, como consecuencia, no puede distribuir el Programa de ninguna forma. Por ejemplo, si una patente no permite la redistribución libre de derechos de autor del Programa por parte de todos aquellos que reciban copias directa o indirectamente a través de usted, entonces la única forma en que podría satisfacer tanto esa condición como esta Licencia sería evitar completamente la distribución del Programa.

Si cualquier porción de este apartado se considera inválida o imposible de cumplir bajo cualquier circunstancia particular ha de cumplirse el resto y la sección por entero ha de cumplirse en cualquier otra circunstancia.

No es el propósito de este apartado inducirle a infringir ninguna reivindicación de patente ni de ningún otro derecho de propiedad o impugnar la validez de ninguna de dichas reivindicaciones. Este apartado tiene el único propósito de proteger la integridad del sistema de distribución de software libre, que se realiza mediante prácticas de licencia pública. Mucha gente ha hecho contribuciones generosas a la gran variedad de software distribuido mediante ese sistema con la confianza de que el sistema se aplicará consistentemente. Será el autor/donante quien decida si quiere distribuir software mediante cualquier otro sistema y una licencia no puede imponer esa elección.

Este apartado pretende dejar completamente claro lo que se cree que es una consecuencia del resto de esta Licencia.

8. Si la distribución y/o uso de el Programa está restringida en ciertos países, bien por patentes o por interfaces bajo copyright, el tenedor del copyright que coloca este Programa bajo esta Licencia puede añadir una limitación explícita de distribución geográfica excluyendo esos países, de forma que la distribución se permita sólo en o entre los países no excluidos de esta manera. En ese caso, esta Licencia incorporará la limitación como si estuviese escrita en el cuerpo de esta Licencia.

9. La Free Software Foundation puede publicar versiones revisadas y/o nuevas de la Licencia Pública General de tiempo en tiempo. Dichas nuevas versiones serán similares en espíritu a la presente versión, pero pueden ser diferentes en detalles para considerar nuevos problemas o situaciones.

Cada versión recibe un número de versión que la distingue de otras. Si el Programa especifica un número de versión de esta Licencia que se refiere a ella y a ``cualquier versión posterior'', tienes la opción de seguir los términos y condiciones, bien de esa versión, bien de cualquier versión posterior publicada por la Free Software Foundation. Si el Programa no especifica un número de versión de esta Licencia, puedes escoger cualquier versión publicada por la Free Software Foundation.

10. Si quiere incorporar partes del Programa en otros programas libres cuyas condiciones de distribución son diferentes, escribe al autor para pedirle permiso. Si el software tiene copyright de la Free Software Foundation, escribe a la Free Software Foundation: algunas veces hacemos excepciones en estos casos. Nuestra decisión estará guiada por el doble objetivo de de preservar la libertad de todos los derivados de nuestro software libre y promover el que se comparta y reutilice el software en general.

AUSENCIA DE GARANTÍA

11. Como el programa se licencia libre de cargas, no se ofrece ninguna garantía sobre el programa, en todas la extensión permitida por la legislación aplicable. Excepto cuando se indique de otra forma por escrito, los tenedores del copyright y/u otras partes proporcionan el programa ``tal cual'', sin garantía de ninguna clase, bien expresa o implícita, con inclusión, pero sin limitación a las garantías mercantiles implícitas o a la conveniencia para un propósito particular. Cualquier riesgo referente a la calidad y prestaciones del programa es asumido por usted. Si se probase que el Programa es defectuoso, asume el coste de cualquier servicio, reparación o corrección.

12. En ningún caso, salvo que lo requiera la legislación aplicable o haya sido acordado por escrito, ningún tenedor del copyright ni ninguna otra parte que modifique y/o redistribuya el Programa según se permite en esta Licencia será responsable ante usted por daños, incluyendo cualquier daño general, especial, incidental o resultante producido por el uso o la imposibilidad de uso del Programa (con inclusión, pero sin limitación a la pérdida de datos o a la generación incorrecta de datos o a pérdidas sufridas por usted o por terceras partes o a un fallo del Programa al funcionar en combinación con cualquier otro programa), incluso si dicho tenedor u otra parte ha sido advertido de la posibilidad de dichos daños.

FIN DE TÉRMINOS Y CONDICIONES

Cómo aplicar estos términos a sus nuevos programas.

Si usted desarrolla un nuevo Programa, y quiere que sea del mayor uso posible para el público en general, la mejor forma de conseguirlo es convirtiéndolo en software libre que cualquiera pueda redistribuir y cambiar bajo estos términos.

Para hacerlo, añada los siguientes anuncios al programa. Lo más seguro es añadirlos al principio de cada fichero fuente para transmitir lo más efectivamente posible la ausencia de garantía. Además cada fichero debería tener al menos la línea de ``copyright" y un indicador a dónde puede encontrarse el anuncio completo.

<una línea para indicar el nombre del programa y una rápida idea de qué hace.>

Copyright (C) 19aa <nombre del autor>

Este programa es software libre. Puede redistribuirlo y/o modificarlo bajo los términos de la Licencia Pública General de GNU según es publicada por la Free Software Foundation, bien de la versión 2 de dicha Licencia o bien (según su elección) de cualquier versión posterior.

Este programa se distribuye con la esperanza de que sea útil, pero SIN NINGUNA GARANTÍA, incluso sin la garantía MERCANTIL implícita o sin garantizar la CONVENIENCIA PARA UN PROPÓSITO PARTICULAR. Véase la Licencia Pública General de GNU para más detalles.

Debería haber recibido una copia de la Licencia Pública General junto con este programa. Si no ha sido así, escriba a la Free Software Foundation, Inc., en 675 Mass Ave, Cambridge, MA 02139, EEUU.

Añada también información sobre cómo contactar con usted mediante correo electrónico y postal.

Si el programa es interactivo, haga que muestre un pequeño anuncio como el siguiente, cuando comienza a funcionar en modo interactivo:

Gnomovision versión 69, Copyright (C) 19aa nombre del autor

Gnomovision no ofrece ABSOLUTAMENTE NINGUNA GARANTÍA. Para más detalles escriba ``show w''.

Esto es software libre, y se le invita a redistribuirlo bajo ciertas condiciones. Escriba ``show c'' para más detalles.

Los comandos hipotéticos ``show w" y ``show c" deberían mostrar las partes adecuadas de la Licencia Pública General. Por supuesto, los comandos que use pueden llamarse de cualquier otra manera. Podrían incluso ser pulsaciones del ratón o elementos de un menú (lo que sea apropiado para su programa).

También deberías conseguir que su empleador (si trabaja como programador) o tu Universidad (si es el caso) firme un ``renuncia de copyright" para el programa, si es necesario. A continuación se ofrece un ejemplo, altere los nombres según sea conveniente:

Yoyodyne, Inc. mediante este documento renuncia a cualquier interés de derechos de copyright con respecto al programa Gnomovision (que hace pasadas a compiladores) escrito por Pepe Programador.

<firma de Pepito Grillo>, 20 de diciembre de 1996> Pepito Grillo, Presidente de Asuntillos Varios.

Esta Licencia Pública General no permite que incluya sus programas en programas propietarios. Si su programa es una biblioteca de subrutinas, puede considerar más útil el permitir el enlazado de aplicaciones propietarias con la biblioteca. Si este es el caso, use la Licencia Pública General de GNU para Bibliotecas en lugar de esta Licencia[51].

[51] Licencia Pública GNU (http://projects.openresources.com/libresoft-notes/libresoft-notes-es/node106.html). Mayo de 2001.

ANEXO B

La Licencia BSD

Esta es la licencia aplicada a las distribuciones de software del Computer Science Research Group, de la Universidad de California en Berkeley.

Versión en español

Se permite su redistribución tanto en forma de código fuente como en forma binaria, con o sin modificaciones, con tal de que se cumplan las siguientes condiciones:

1.Redistribuciones del código fuente deben retener el aviso de copyright declarada arriba, esta lista de condiciones y la denegación de garantía que sigue.

2.Redistribuciones en forma binaria deben reproducir el aviso de copyright declarada arriba, esta lista de condiciones y la denegación de garantía que sigue en la documentación y/u otros materiales que acompañen la distribución.

3.Todo material de promoción que mencione características o el uso de este software debe mostrar el siguiente reconocimiento: Este producto incluye software desarrollado por la Universidad de California, Berkely y sus contribuidores.

4.No se permite ni el uso del nombre de la Universidad ni el uso de los nombres de sus contribuidores para apoyar o promover productos derivados de este software sin previo permiso específico por escrito.

LOS REGENTES Y CONTRIBUIDORES PROVEEN ESTE SOFTWARE "TAL Y COMO ESTÁ", Y DENEGA CUALQUIER GARANTÍA, YA SEA EXPRESA O IMPLÍCITA, INCLUYENDO SIN LIMITACIÓN LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD Y APTITUD PARA UN PROPÓSITO ESPECÍFICO. NI LOS REGENTES NI SUS CONTRIBUIDORES SERÁN EN NINGÚN CASO RESPONSABLES POR PERJUICIOS, YA SEAN DIRECTOS, INDIRECTOS, INCIDENTALES, ESPECIALES, PUNITIVOS O CONSECUENTES (INCLUYENDO SIN LIMITACIÓN LA ADQUISICIÓN DE BIENES O SERVICIOS DE SUSTITUCIÓN; PÉRDIDA DE USO, DATOS O GANANCIAS; O INTERRUPCIÓN DE NEGOCIOS), SEA CUAL SEA SU CAUSA Y BAJO CUALQUIER TEORÍA DE RESPONSABILIDAD, YA SEA EN CONTRATO, RESPONSABILIDAD ESTRICTA O ENTUERTO (INCLUYENDO DE NEGLIGENCIA O DE CUALQUIER OTRA MANERA) QUE SURJA DE CUALQUIER MANERA DEL USO DE ESTE SOFTWARE, AÚN EN EL CASO DE HABER SIDO AVISADO DE LA POSIBILIDAD DE TAL PERJUICIO[52].

[52] Licencia BSD (http://www.eldemonio.org/bsdl.html). Julio de 2001.

ANEXO C

La Definición de Open Source

Versión en español

Open source no sólo significa acceso al codigo fuente. Las condiciones de distribución de un programa open-source deben cumplir con el siguiente criterio:

1. Libre Redistribución

La licencia no debe restringir a nadie vender o entregar el software como un componente de una distribución de software que contenga programas de distintas fuentes. La licencia no debe requerir royalty ni ningun tipo de cuota por su venta.

2. Código Fuente

El programa debe incluir el código fuente, y se debe permitir su distribución tanto como código fuente como compilado. Cuando de algun modo no se distribuya el código fuente junto con el producto, debera proveerse un medio conocido para obtener el código fuente sin cargo, a través de Internet. El código fuente es la forma preferida en la cual un programador modificará el programa. No se permite el código fuente deliberadamente confundido (obfuscation). Tampoco se permiten formatos intermedios, como la salida de un preprocesador, o de un traductor.

3. Trabajos Derivados

La licencia debe permitir modificaciones y trabajos derivados, y debe permitir que estos se distribuyan bajo las mismas condiciones de la licencia del software original.

4. Integridad del Código Fuente del Autor.

La licencia puede restringir la distribución de codigo fuente modificado sólo si se permite la distribución de "patch files" con el código fuente con el propósito de modificar el programa en tiempo de construcción. La licencia debe permitir explícitamente la distribución de software construido en base a código fuente modificado. La licencia puede requerir que los trabajos derivados lleven un nombre o número de versión distintos a los del software original.

5. No Discriminar Personas o Grupos.

La licencia no debe hacer discriminación de personas o grupos de personas.

6. No Discriminar Campos de Aplicación.

La licencia no debe restringir el uso del programa en un campo especifico de aplicación. Por ejemplo, no puede restringir su uso en negocios, o en investigación genética.

7. Distribución de la Licencia.

Los derechos concedidos deben ser aplicados a todas las personas a quienes se redistribuya el programa, sin necesidad de obtener una licencia adicional.

8. La Licencia No Debe Ser Específica a un Producto.

Los derechos aplicados a un programa no deben depender de la distribución particular de software de la que forma parte. Si el programa es extraido de esa distribución y usado o distribuido dentro de las condiciones de la licencia del programa, todas las personas a las que el programa se redistribuya deben tener los mismos derechos que los concedidos en conjunción con la distribucion original de software.

9. La Licencia No Debe Contaminar Otro Software.

La licencia no debe imponer restricciones sobre otro software que es distribuido junto con el. Por ejemplo, la licencia no debe insistir en que todos los demás programas distribuidos en el mismo medio deben ser software open-source[53].

[53] Licencia BSD (http://www.eldemonio.org/bsdl.html). Julio de 2001.