

Control de revisiones

Versión 1.0

José E. Marchesi (jemarch@gnu.org)

Control de revisiones, versión 1.0

Copyright © 2004 José E. Marchesi

Se permite la copia y distribución de este documento, tal cual, siempre y cuando se incluya esta notificación de copyright.

Índice General

1	Introducción.....	1
2	Parches y transportabilidad	3
2.1	Motivación.....	3
2.2	El concepto de parche	3
2.3	Elaboración de parches con <code>diff</code>	5
2.4	Aplicación de parches con <code>patch</code>	9
3	Control de revisiones con RCS.....	13
3.1	Almacenamiento de revisiones: parches sucesivos.....	13
3.2	El sistema de control de revisiones RCS	15
4	Control de revisiones con CVS	16
4.1	Motivación	16
4.2	Conceptos generales	17
4.3	Administración de CVS	20
4.3.1	Estructura de un repositorio	20

1 Introducción

Tradicionalmente el desarrollo de cualquier paquete software estaba restringido a un grupo muy reducido de desarrolladores, que compartían muchas convenciones y compromisos. Estos desarrolladores pertenecían al mismo departamento o laboratorio, y disponían de estándares y modos de desarrollo exclusivos y propios. Con el advenimiento del Software Libre y su modelo de desarrollo distribuido, donde una gran cantidad de desarrolladores de muy diferentes condiciones y usualmente repartidos a lo largo y ancho del planeta desarrollan conjuntamente, se hizo necesaria la utilización de herramientas que potenciaran la comunicación y coordinación entre ellos. Indudablemente una buena comunicación entre los integrantes de un proyecto y la adopción de convenciones comunes se vuelve mas difícil a medida que el grupo de desarrollo crece y se dispersa.

Algunas de estas herramientas toman la forma de documentos donde se establecen los principios de desarrollo que todos los desarrolladores deben seguir. Un ejemplo son las GCS¹. Por otra parte, para conseguir un nivel de comunicación aceptable que pueda responder a las necesidades de los desarrolladores, se utilizan herramientas tales como listas de correo, entornos web colaborativos², grupos de noticias y sistemas de tickets. Además, también se desarrollaron herramientas específicas para facilitar las tareas específicas de desarrollo en colaboración, como la elaboración y aplicación de parches (*diff* y *patch*), el control de versiones (*rcs* y *cvs*), la gestión de una base de datos de bugs (*Bugzilla*), etc. El siguiente esquema resume los tipos de herramientas para el desarrollo distribuido, y algunos ejemplos concretos.

- Documentos
 - Estándares y convenciones
 - Políticas comunes de desarrollo
 - Normas de releasing
 - Etc
- Herramientas para la comunicación
 - Listas de correo (*mailman*)
 - Entornos web colaborativos (*wikis*)
 - Grupos de noticias (*news*)
 - Sistemas de gestión de tickets (*rt*)
- Herramientas de desarrollo software
 - Elaboración y aplicación de parches (*diff* y *patch*)
 - Control de revisiones (*RCS*)
 - Control de revisiones mejorado (*CVS*)
 - Gestión de base de datos de bugs (*Bugzilla*)

De entre todos los tipos de herramientas nos centraremos en las de desarrollo software, y en concreto las relativas a los parches y el control de revisiones. En las siguientes secciones aprenderemos a hacer parches, distribuirlos y aplicarlos. Examinaremos someramente el

¹ GNU Coding Standards

² Conocidos como “wikis”

sistema de control de revisiones RCS, que puede entenderse como un “super parche”. A continuación entraremos a fondo a examinar el sistema de control de revisiones CVS (descendiente y ampliación del anterior), el concepto de repositorio centralizado, de copia de trabajo, etc. Dentro del ámbito del CVS primero prestaremos atención a los aspectos de administración (cómo crear un repositorio, gestionar los usuarios, sus permisos, etc) y después, mediante un ejemplo, a la utilización de un repositorio CVS ya instalado y correctamente configurado. Terminaremos con un pequeño apartado más retórico, que intentará dejar clara una de las máximas de los sistemas de control de revisiones: *Ninguna herramienta de desarrollo colaborativo puede sustituir a la comunicación entre los desarrolladores.*

2 Parches y transportabilidad

2.1 Motivación

19:30, sábado. Recluído en casa por un tiempo infame que parece encontrarse a gusto en Madrid, y, en fin, no tener demasiada prisa por irse a otra parte. Tarde lluviosa, oscura, fría y asquerosa. La tarde perfecta para hackear algo, me digo, lo que sea. Me pongo las zapatillas de casa, preparo algo de café, y me siento confortablemente delante del ordenador. La comodidad es importante en estos casos: la silla no debe ser demasiado alta ni demasiado baja, la luz debe ser buena pero no tan fuerte como para reflejar en el monitor. El café debe ser suave pero no aguado, y un cenicero vacío y generoso (compañero indispensable del hacker fumador) debe estar accesible y no demasiado cerca del teclado. No debe olvidarse el taco de hojas en blanco y el lápiz, siempre a mano para garabatear listas y esquemas que ayuden a centrar la mente en el combate contra algoritmos complicados. En fin, todo listo para una sesión de hacking, esperemos provechosa, pero siempre instructiva y agradable. Me siento con ánimos y poderoso, y decido con valentía: hoy toca autoconf. El monstruo de la portabilidad, hacedor de maravillas (y tolerado únicamente por ello) suele ser un hueso duro de roer. Llevo una temporada adaptando un programa para utilizar autoconf, y estoy decidido a completar hoy el port. En fin, navego en mi Emacs por buffers Dired hasta llegar al directorio que contiene el programa y comienzo a trabajar. Despacio, con cuidado, con cariño, sustituyo el antiguo sistema de configuración, añadiendo ficheros, borrando otros, sacando esto de aquí y lo otro de allá. Finalmente obtengo en el directorio una versión que utiliza únicamente autotools para configuración, compilación e instalación. Éxito, pues. No veo el momento de enviar al mantenedor del programa la nueva configuración, anticipando su satisfacción por un trabajo tan bien hecho. Rápidamente empaqueto el directorio con el programa modificado en un fichero tgz, lo cuelgo en mi web personal, y envío un correo electrónico a la lista de desarrollo del programa, proclamando la mejora, y donde encontrarla. En pocas horas recibo la respuesta del mantenedor del programa, que me deja consternado: si, la mejora es patente, oportuna y muy útil. Pero, por decirlo en pocas palabras, no dispone de tiempo (ni de ganas) para extraer mis cambios de la distribución modificada que colgué en la web. El correo termina con un ruego y una palabra llena de misticismo, que ya había leído en otras ocasiones en listas de desarrollo: "unidiff". ¿Qué ha fallado?, me pregunto. Claramente, no he sabido rematar adecuadamente la tarea. Extraigo rápidamente una lección de lo sucedido: una correcta distribución de un hack es tan importante como su elaboración.

2.2 El concepto de parche

Consideremos la distribución de fuentes de un programa cualquiera que hemos bajado de internet. Dichas fuentes estarán organizadas en forma de una jerarquía de directorios y ficheros. Contendrán tanto ficheros fuente escritos en algún lenguaje, como imágenes, ficheros con documentación, instrucciones de instalación, etcétera. Supongamos además que vamos a realizar una serie de modificaciones sobre dicho programa (añadirle alguna característica, mejorar otra ya existente, bugfixing, etc). En este caso hipotético estamos en una situación típica de desarrollo de Software Libre, en la que intervienen dos roles claramente diferenciados:

- El mantenedor del programa (muy posiblemente el autor principal u original del mismo) que se encarga de determinar las líneas generales de su desarrollo, tal y como el versionado. El mantenedor es el que posee la copia maestra de las fuentes del programa que pasan a ser de forma periódica los *releases*, o liberaciones de las versiones. A esta copia maestra se la suele llamar en argot de desarrollo el *mainstream* del programa. De esta forma, es el mantenedor el que aplica los cambios que considera adecuados a este mainstream, haciéndolo evolucionar hacia nuevas versiones del programa.
- Los desarrolladores son personas que colaboran (de forma esporádica o continuada) en el desarrollo del proyecto correspondiente. Su participación puede ser mas o menos comprometida. Los desarrolladores no tienen por qué ser escritores de software, sino que también hay hackers de documentación, hackers de interfaz, hackers artistas, músicos, etc. Los desarrolladores estables del proyecto mantienen una estrecha comunicación entre ellos, y también con el mantenedor del programa.

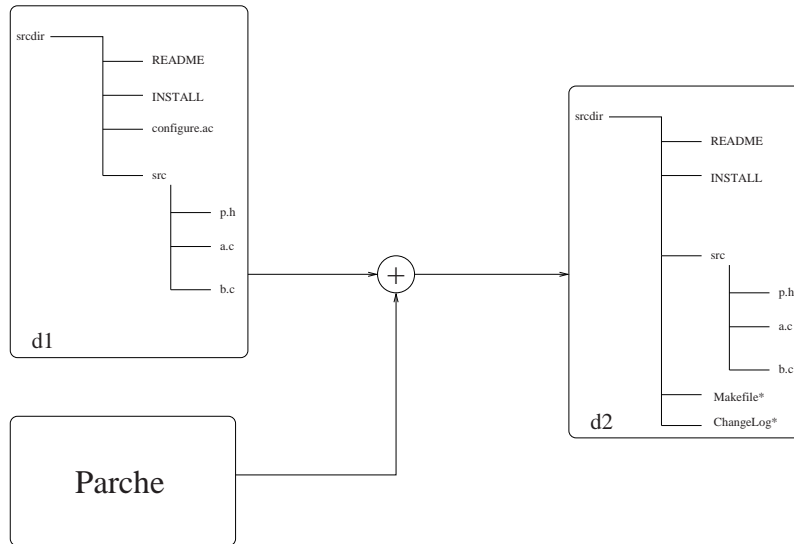
Volviendo a nuestra tarea como desarrollador, dependiendo de lo que vamos a hacer es claro que implicará cambios en el contenido de alguno de los ficheros de la distribución, y tal vez cambios en la propia estructura de directorios. Sea como fuere, una vez realizados los cambios obtendremos una distribución modificada distinta de la original. Ahora se nos plantea un problema: suponiendo que no somos el mantenedor del programa, ¿cómo podemos hacerle llegar dichos cambios, de modo que los incorpore a la versión principal o *mainstream*?. Existen dos formas principales de hacerlo:

- Distribuir la distribución modificada completa.
- Distribuir únicamente nuestras modificaciones, de modo que aplicadas al mainstream resulte nuestra distribución modificada.

Ambas formas de revertir nuestras modificaciones al mantenedor del mainstream disfrutan de ciertas ventajas y adolecen de determinados inconvenientes. Una medida útil para compararlas es cómo se distribuye el trabajo entre el mantenedor del mainstream y el desarrollador que ha efectuado los cambios. En la primera opción (el desarrollador envía al mantenedor la distribución completa con sus modificaciones) está claro que el desarrollador está cargando al mantenedor con un trabajo extra: determinar cuales fueron los cambios introducidos por el desarrollador. Esta tarea habitualmente consume mucho tiempo y es fácil cometer errores, sobre todo si el mantenedor recibe modificaciones de muchos desarrolladores. La segunda opción (enviar únicamente las modificaciones concretas) requiere de mas trabajo por parte del desarrollador y facilita la tarea del mantenedor. Enviando los cambios concretos, es fácil para el mantenedor evaluar dichos cambios y decidir si incorporarlos al mainstream. En la inmensa mayoría de los proyectos (en los que la relación entre mantenedores y desarrolladores es de pocos a muchos) se establece como obligatoria la segunda opción, asumiendo que de esta forma el mantenedor puede hacer mejor su trabajo y, como consecuencia, el desarrollo del programa en su conjunto sale favorecido.

Ahora bien, a la expresión de los cambios concretos sobre la distribución original se la llama, de forma general, *parche*. Obsérvese que un parche no es una distribución, sino mas bien las modificaciones que hay que realizar sobre la distribución original para obtener la modificada. Desde este punto de vista, un parche es una transformación. La *aplicación* de un parche sobre una distribución, entonces, da como resultado una nueva distribución: la modificada. Esto queda explicado en forma esquemática en la siguiente figura, en donde la aplicación de un parche sobre una distribución de fuentes denominada **d1** da lugar a otra

distribución **d2**. Los nombres de ficheros seguidos de un asterisco en **d2** denotan un fichero modificado o creado, con respecto a **d1**.



Luego la aplicación del parche anterior implica las siguientes modificaciones sobre **d1**:

- Se han modificado los ficheros fuente ‘`srcdir/src/p.h`’ y ‘`srcdir/src/b.c`’.
- Se han creado los ficheros ‘`srcdir/ChangeLog`’ y ‘`srcdir/Makefile`’.
- Se ha borrado el fichero ‘`srcdir/configure.ac`’.

Evidentemente, el parche debe contener toda la información necesaria para, durante su aplicación, realizar todos los cambios que desea el desarrollador. El mantenedor del programa recibirá el parche y lo aplicará al mainstream (que tomará el papel de **d1** en el gráfico anterior) si lo cree conveniente. También es importante notar que el parche contiene información tanto de los cambios en la estructura de ficheros de la distribución (borrar un fichero, añadir un directorio, etc) como las modificaciones al contenido de los propios ficheros (añadir una línea, borrarla, etc).

El lector se dirá: “si, vale, ya se lo que es un parche. Es una transformación... Pero, ¿qué aspecto tienen? ¿Existe algún tipo de herramientas para generarlos y aplicarlos?”. En la siguiente sección aprenderemos a utilizar la herramienta unix **diff** para elaborar parches.

2.3 Elaboración de parches con diff

Como vimos en la sección anterior, un parche contiene dos tipos de información para hacer posible su aplicación:

- Información de estructura, como borrado de ficheros, creación de nuevos directorios, etc.
- Información de contenido, esto es, qué se ha cambiado en el contenido de los ficheros.

Ambos tipos de información pueden obtenerse en forma de parche mediante la utilización de la herramienta GNU **diff**. En principio nos centraremos en obtener parches para el contenido de un solo fichero. Posteriormente, añadiremos información de estructura.

Como su propio nombre indica, el comando `diff` sirve para encontrar diferencias entre dos ficheros de texto. `diff` no se limita a determinar si dos ficheros son distintos, sino que resume las diferencias y se las presenta al usuario. Como veremos, podemos indicar a `diff` si dicha información va a ser leída por un humano, y por tanto debe ser fácilmente legible, o va a ser procesada por otro programa, en cuyo caso la información facilitada debe ser fácilmente procesable de forma automática, aun a riesgo de perder legibilidad. Una buena forma de comenzar a aprender `diff` es examinando una serie de ejemplos.

Supongamos que tenemos dos ficheros, `'a.txt'` y `'b.txt'`.

- Contenido de `'a.txt'`:

```
Join us now
and share the software
You will be free, hacker,
you will be free
```

- Contenido de `'b.txt'`:

```
Join us now
and share the software
You'll be free, hacker,
you'll be free
```

Claramente el contenido de `'b.txt'` difiere del de `'a.txt'` en las líneas tercera y cuarta. Si ejecutamos `diff` y le proporcionamos los ficheros anteriores, su respuesta es:

```
jemarch@MalditoBastardo:~/tmp$ diff a.txt b.txt
3,4c3,4
< You will be free, hacker,
< you will be free
---
> You'll be free, hacker,
> you'll be free
jemarch@MalditoBastardo:~/tmp$
```

Veamos como podemos interpretar el informe que nos da `diff`. La primera línea hace referencia a la localización de las diferencias en los ficheros, y de qué tipo son. En este caso, con `3,4c3,4` nos indica que las líneas tercera y cuarta de `'a.txt'` han sido cambiadas (de ahí la `c`) en las líneas tercera y cuarta de `'b.txt'`. A continuación, nos muestra como eran las líneas tercera y cuarta en el primer fichero (líneas precedidas del caracter `<`) y las mismas modificadas en el segundo fichero (líneas precedidas del caracter `>`). La línea de tres guiones `---` separa ambos tipos de información. Luego tenemos que `diff` nos proporciona una cabecera que describe la localización y el tipo de cambio (en este caso, `3,4c3,4`) y a continuación la descripción del cambio en sí.

Veamos lo que ocurre cuando *dispersamos* las modificaciones en el fichero:

Nuevo contenido de `'b.txt'`:

```
Join us Now
and share the software
You will be free, hacker,
you'll be free
```

Ahora las líneas distintas son la primera y la cuarta. Veamos qué nos dice `diff` ante esta nueva situación:

```
jemarch@MalditoBastardo:~/tmp$ diff a.txt b.txt
1c1
< Join us now
---
> Join us Now
4c4
< you will be free
---
> you'll be free
jemarch@MalditoBastardo:~/tmp$
```

Ahora `diff` nos proporciona dos cabeceras y dos descripciones de los cambios.

`diff` proporciona la información de los cambios en secciones, cada una compuesta de una cabecera y de un cuerpo. La cabecera identifica donde están localizados los cambios expresados en el cuerpo, y de qué tipo es el cambio. Si llamamos `f1` y `f2` al primer y segundo ficheros pasados a `diff` en línea de comandos, podemos esquematizar la forma de una sección como sigue:

```
<DondeEnF1><TipoCambio><DondeEnF2>
<Contenido en f1>
---
<Contenido en f2>
```

Cada una de estas secciones recibe el nombre en argot de *chunk*. De los ejemplos vistos parece que `diff` intenta concentrar la información de los cambios en el menor número posible de chunks.

Veamos qué ocurre si hacemos otro tipo de modificación a `'b.txt'`: en lugar de modificar el contenido de una línea, vamos a borrarla. Además, añadiremos una línea mas.

Nuevo contenido de `'b.txt'`:

```
Join us now
You will be free, hacker,
you'll be free
forever
```

Como era de esperar, `diff` nos responde con dos chunks:

```

jemarch@MalditoBastardo:~/tmp$ diff a.txt b.txt
2d1
< and share the software
4a4
> forever
jemarch@MalditoBastardo:~/tmp$

```

El tipo de modificación de las cabeceras de los chunks ya no son una letra **c**. En el primer chunk (que especifica la modificación: “se ha borrado la segunda línea del contenido del fichero ‘a.txt’”) el tipo de modificación es **d** (de delete), y en el segundo chunk (que especifica la modificación: “se ha añadido una nueva línea, que pasa a ser la cuarta, al contenido del fichero ‘a.txt’”) el tipo de modificación es **a** (de addition).

Vemos que es relativamente sencillo interpretar la información que nos proporciona **diff**. Basta recordar que se compone de distintos chunks, cada uno de ellos formado por una cabecera y un cuerpo descriptivo. Sin embargo, podemos plantearnos: ¿es la salida de **diff** lo suficientemente descriptiva como para ser utilizada como un parche?. Recordemos que la aplicación de un parche involucra al fichero original (en este caso, ‘a.txt’) y al parche. Ambos deben proporcionar la información suficiente como para obtener el fichero modificado (en este caso, ‘b.txt’). La respuesta es afirmativa. Sin embargo, las salidas de **diff** anteriores no son fácilmente procesables por programas. Esto es así porque están pensadas para ser leídas y fácilmente interpretadas por humanos, tal y como lo hemos hecho a lo largo de los ejemplos. Es posible comunicar a **diff** que vamos a utilizar su salida pasándosela a otro programa, y que por tanto ésta debe ser fácilmente procesable, aun a riesgo de perder legibilidad. Esto se hace pasándole la opción en línea de comandos **-u** (que significa *Unidiff*). Veamos qué aspecto tiene la salida de una aplicación de **diff** a los ficheros anteriores con sus contenidos originales (los del primer ejemplo):

```

jemarch@MalditoBastardo:~/tmp$ diff -u a.txt b.txt
--- a.txt Sat Jun 19 21:36:42 2004
+++ b.txt Sat Jun 19 22:08:19 2004
@@ -1,4 +1,4 @@
  Join us now
  and share the software
-You will be free, hacker,
-you will be free
+You'll be free, hacker,
+you'll be free
jemarch@MalditoBastardo:~/tmp$

```

Ahora la cabecera del chunk es notoriamente distinta. En primer lugar, incluye información acerca de ambos ficheros, como sus nombres y su fecha de última modificación. A continuación sigue especificando las líneas afectadas en el primer y segundo fichero, respectivamente. Aparte del hecho de utilizar los caracteres **-** y **+** en lugar de **<** y **>**, la diferencia más notable es la inclusión en este último formato de *información de contexto*. En efecto, aparte de las dos líneas modificadas en ‘a.txt’, también se incluyen las dos primeras líneas del fichero. La cantidad de contexto que se incluye en los parches puede regularse mediante opciones en línea de comando. Por defecto, **diff** considera que un contexto adecuado es un

entorno de cuatro líneas por encima y por debajo del contenido del chunk correspondiente. En nuestro ejemplo solo hay dos líneas por encima del chunk y ninguna por debajo. La inclusión del contexto sirve para minimizar la posibilidad de confusión a la hora de aplicar el parche cuando el mainstream y nuestra copia original no coinciden, como veremos en la siguiente sección.

Resumiendo:

- Para generar un parche utilizamos el comando `diff`, pasándole como argumento, en orden, el fichero original y su contrapartida modificada.
- La información proporcionada por `diff` es una serie de cero o mas chunks o secciones, que especifican cambios localizados en el fichero original.
- `diff` siempre trata de generar el menor número de chunks posible.
- Podemos indicar a `diff` qué tipo de salida deseamos mediante una opción en línea de comando:
 - Si no especificamos nada, `diff` genera una descripción de los cambios destinada a ser interpretada por humanos.
 - Si especificamos la opción `-u`, `diff` genera una descripción de los cambios en formato *Unidiff*, que está especialmente diseñada para ser aplicada como parche. Este formato incluye información de contexto y de los ficheros involucrados.
- El formato mas apropiado para ser utilizado como parche es el *Unidiff*.

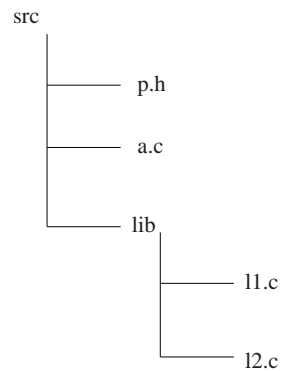
En la siguiente sección aprenderemos a utilizar la herramienta de aplicación de parches: GNU `patch`. `patch` recibe como argumentos el fichero original y un parche generado con `diff`, y recrea el fichero modificado.

2.4 Aplicación de parches con patch

En la sección anterior aprendimos a generar parches utilizando el comando `diff`. Recordemos la estructura de un parche Unidiff:

```
--- <Nombre F1> <Fecha Ultima Modificación F1>
+++ <Nombre F2> <Fecha Ultima Modificación F2>
<CHUNK1 ....>
<CHUNK2 ....>
...
<CHUNKn ....>
```

Algo importante a tener en cuenta es que podemos concatenar en un solo fichero varios parches, para obtener un parche que afecta a varios ficheros en una estructura de directorios. Por ejemplo, si tenemos la siguiente estructura de ficheros y directorios:



Podemos hacer un parche múltiple con modificaciones a los ficheros `src/a.c` y `src/lib/11.c` de la siguiente forma:

- Obtenemos el parche de `src/a.c`:

```

jemarch@MalditoBastardo:~/tmp$ diff -u src/a.c src/na.c > a.patch
jemarch@MalditoBastardo:~/tmp$ cat a.patch
--- src/a.c Sat Jun 19 22:41:42 2004
+++ src/na.c Sat Jun 19 22:42:35 2004
<CHUNKS ...>
jemarch@MalditoBastardo:~/tmp$
  
```

- De forma similar, obtenemos el parche de `src/lib/11.c`:

```

jemarch@MalditoBastardo:~/tmp$ diff -u src/lib/11.c src/lib/n11.c
> 11.patch
jemarch@MalditoBastardo:~/tmp$ cat 11.patch
--- src/lib/11.c Sat Jun 19 22:41:56 2004
+++ src/lib/n11.c Sat Jun 19 22:42:39 2004
<CHUNKS ...>
jemarch@MalditoBastardo:~/tmp$
  
```

- Por último, simplemente concatenamos los parches para obtener el parche múltiple:

```

jemarch@MalditoBastardo:~/tmp$ cat a.patch l1.patch
> modificaciones.patch
jemarch@MalditoBastardo:~/tmp$ cat modificaciones.patch
--- src/a.c Sat Jun 19 22:41:42 2004
+++ src/na.c Sat Jun 19 22:42:35 2004
<CHUNKS de a ...>
--- src/lib/l1.c Sat Jun 19 22:41:56 2004
+++ src/lib/nl1.c Sat Jun 19 22:42:39 2004
<CHUNKS de b ...>
jemarch@MalditoBastardo:~/tmp$

```

Es de destacar que antes de la primera cabecera del parche puede ir cualquier texto que no cumpla su formato. Esto es extremadamente útil a la hora de reportar parches en un correo electrónico, donde el mensaje electrónico en sí puede ir antes del parche, y éste al final. De hecho, `patch` ignorará cualquier texto del parche que figure antes de la primera cabecera.

Ahora supongamos que, satisfechos de nuestras modificaciones a ‘`src/a.c`’ y ‘`lib/l1.c`’, deseamos enviar al mantenedor del programa nuestros cambios. Procedemos montando un parche múltiple y se lo enviamos por correo electrónico. El mantenedor, al recibir el parche, lo primero que hace es leerlo y evaluarlo, para asegurarse de que por una parte implemente la mejora declarada, y por otra no introduzca nuevos errores. En este momento puede decidir aplicar el parche al mainstream o rechazarlo¹. Supongamos que ha optado por aplicar el parche. ¿Qué debe hacer?. El mantenedor posee el mainstream y nuestro parche. Procede utilizando el comando `patch`, que recibe como parámetro el parche a aplicar:

```

jemarch@MalditoBastardo:~/mainstream$ patch -p0 < modificaciones.patch
patching file src/a.c
patching file src/lib/l1.c
jemarch@MalditoBastardo:~/mainstream$

```

`patch` obtiene información del parche múltiple sobre qué ficheros debe actuar. La opción en línea de comando `-p0` especifica que debe utilizarse el path completo de los nombres de ficheros presentes en las cabeceras de los parches. O dicho de otro modo, que el directorio donde se ejecuta `patch` es relativamente el mismo que desde el que se hizo el diff. El mantenedor podría haber optado por:

```

jemarch@MalditoBastardo:~/mainstream/src$ patch -p1 < modificaciones.patch
patching file a.c
patching file lib/l1.c
jemarch@MalditoBastardo:~/mainstream/src$

```

Obsérvese que los paths de los ficheros a parchear han “perdido” un directorio. El resultado, no obstante, es el mismo, ya que `patch` ha sido ejecutado en ‘`src/`’.

Podemos resumir el funcionamiento de `patch`:

¹ A los bugs introducidos por un parche que escapan a la atención del mantenedor y se hacen patentes una vez aplicados en el mainstream se los conoce como “regressions”, en argot de desarrollo

1. Desecha el texto presente antes de la primera cabecera de parche.
2. Por cada parche presente en el parche múltiple:
 1. Busca el fichero a parchear.
 2. Lo parchea utilizando la información presente en el parche.
3. Desecha el texto presente después del contenido del último parche hasta el fin del parche múltiple.

En ocasiones, la copia original desde la que el desarrollador realizó el parche no coincide con el mainstream sobre el que el mantenedor aplica el parche. Esto puede suceder por varios motivos, el principal de los cuales es que haya pasado un tiempo desde que el desarrollador obtuvo su copia original, y el mantenedor haya aplicado otros parches desde entonces. En este caso, `patch` trata de aplicar el parche de todos modos, utilizando la información de contexto presente en el parche. En el caso de que detecte incongruencias insalvables, `patch` reporta un “conflicto”, y deja en manos del mantenedor qué hacer a continuación. El lector puede preguntarse hasta qué punto `patch` es capaz de parchear ficheros con incongruencias. Tal vez sorprendentemente, la respuesta es que `patch` puede solucionar la mayoría de las incongruencias, siempre y cuando la información de contexto sea lo suficientemente representativa. Como es obvio, dicha capacidad está casi por completo condicionada por la “calidad de representación” del contexto presente en el parche. Si, por ejemplo, el fichero a parchear contiene mucho código repetitivo (resultado de un “cortapega”), `patch` puede resultar confundido y reportar un conflicto: no tiene suficiente información de contexto para aplicar el parche.

Como veremos, las herramientas de control de revisiones hacen que los desarrolladores puedan obtener copias frescas del mainstream con mucha facilidad, y de ese modo reducir las situaciones de incongruencia en su mayor parte. Por lo general, la aplicación de parches con incongruencias es algo peligroso y muy delicado. `patch` no garantiza una aplicación correcta del parche en ese caso, e incluso puede reportar una aplicación de un parche como satisfactoria, cuando en realidad se ha visto confundido y corrompido el contenido del fichero resultante. Por el contrario, cuando no hay incongruencias el resultado de `patch` siempre será correcto.

3 Control de revisiones con RCS

3.1 Almacenamiento de revisiones: parches sucesivos

Supongamos que estamos desarrollando un script `sh` de gestión de usuarios, llamado ‘`gesusuarios.sh`’. A medida que vamos añadiendo facilidades y arreglando problemas obtenemos revisiones sucesivas del script. Una de las formas más habituales de distinguir revisiones es mediante la utilización de números de versión. Nos planteamos el problema de como almacenar las distintas revisiones de nuestro script. Una forma evidente sería tener un “directorio de revisiones”, conteniendo las distintas revisiones sucesivas del script:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ ls
gesusuarios-0.1.sh  gesusuarios-0.4.sh  gesusuarios-0.6.sh
gesusuarios-1.1.sh  gesusuarios-1.3.sh  gesusuarios-0.2.sh
gesusuarios-0.5.sh  gesusuarios-1.0.sh  gesusuarios-1.2.sh
gesusuarios-2.0.sh
jemarch@MalditoBastardo:~/tmp/gesusuarios$
```

De este modo, cuando el desarrollador desea obtener una revisión determinada, simplemente la copia de este directorio. Cuando desea hacer alguna mejora, toma la última revisión, trabaja sobre ella, e introduce una nueva. El problema crónico de esta forma de almacenar revisiones salta a la vista: la gran redundancia de los datos almacenados. Las diferencias entre una revisión y la siguiente pueden ser grandes o pequeñas, pero casi siempre son pequeñas en relación al contenido del fichero. El desperdicio de espacio en disco es realmente vergonzoso. Mas problemas se hacen evidentes cuando consideramos para qué sería útil utilizar el directorio de revisiones. Aparte de las operaciones evidentes de introducir una nueva revisión o rescatar una anterior, son de gran utilidad para el desarrollador las siguientes:

- Obtener diferencias entre dos revisiones cualesquiera del fichero. Si utilizamos nuestro directorio de revisiones, podemos llevar a cabo esta operación mediante invocaciones a `diff`. Por ejemplo:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ diff gesusuarios-0.6.sh
gesusuarios-1.0.sh
```

- Revertir una de las revisiones. Esta operación es extremadamente útil. Dadas n revisiones r_1, r_2, \dots, r_n y dado que $r_1 < r_2 < \dots < r_n$, *revertir* la revisión r_i consiste en “restar” a todas las revisiones r_j tal que $r_j > r_i$ las diferencias entre r_{i-1} y r_i .

O dicho de otro modo, se trata de eliminar a r_i de la cadena de revisiones, teniendo en cuenta dicho impacto en todas las revisiones posteriores. Aunque puede realizarse con combinaciones de `diff` y `patch`, no es en absoluto trivial.

Dado que llevar a cabo las operaciones anteriores mediante invocaciones sucesivas a `diff` y `patch` resultaría complejo, repetitivo, y por tanto dado a errores, podríamos implementar las operaciones en un script de gestión de nuestro directorio de revisiones. Como parámetros,

el script obtendrá la operación deseada (crear una nueva revisión, obtener diferencias, revertir una o mas revisiones, etc) y las revisiones sobre las que se lleva a cabo. Si llamáramos ‘gesrev’ al script, podríamos utilizarlo de la siguiente forma:

- Obtener la revisión 0.2:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ gesrev or 0.2
```

donde `or` es la operación ObtenerRevisión.

- Obtener las diferencias entre las revisiones 0.2 y 1.0:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ gesrev od 0.2 1.0
```

donde `od` es la operación ObtenerDiferencias.

- Revertir la revisión 0.2:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ gesrev rv 0.2
```

donde `rv` es la operación ReVertir.

- Etc...

En definitiva, habríamos construido nuestro propio sistema de control de revisiones (o de versiones, que identifican a las revisiones), compuesto del directorio de revisiones y el script `gesrev` que implementa las operaciones sobre el mismo. Sin embargo, podríamos pensar en optimizar nuestro sistema de control de revisiones. El problema de la redundancia de información puede solventarse almacenando en el directorio de revisiones únicamente los cambios realizados a la revisión anterior. Recordando que podemos almacenar de forma concisa las diferencias entre dos ficheros en un parche generado por `diff`, nuestro directorio de revisiones quedaría de la siguiente forma:

```
jemarch@MalditoBastardo:~/tmp/gesusuarios$ ls
gesusuarios.sh      gesusuarios-0.1.sh,v  gesusuarios-0.2.sh,v
gesusuarios-0.3.sh,v gesusuarios-0.4.sh,v  gesusuarios-1.0.sh,v
gesusuarios-1.1.sh,v gesusuarios-1.5.sh,v  gesusuarios-2.0.sh,v
jemarch@MalditoBastardo:~/tmp/gesusuarios$
```

donde ‘`gesusuarios.sh`’ es el script original, y los ficheros terminados en ‘`,v`’ son los ficheros `diff` que, aplicados en cadena con `patch`, dan lugar a la revisión correspondiente. Obsérvese que esta forma de almacenar las revisiones no conlleva pérdida de información y sin embargo es extremadamente compacta. El proceso de obtener una revisión determinada sería el siguiente:

1. Aplicar a ‘`gesusuarios.sh`’ el parche correspondiente a la primera revisión. El resultado es un fichero temporal ‘`gesusuarios.sh-resultado`’.
2. Hasta llegar a la revisión deseada...
 1. Aplicar al último fichero temporal ‘`gesusuarios.sh-resultado`’ el parche de esta revisión.

3. El fichero temporal corresponde a la revisión deseada. Retornarlo como resultado.

Un problema de esta forma de almacenar revisiones consiste en que debido a la naturaleza en cadena del proceso de construir las revisiones, si por alguna razón se estropea uno de los ficheros `diff`, automáticamente los que le siguen se vuelven inutilizables. Para minimizar este riesgo, podríamos pensar en almacenar todos los parches correspondientes a las revisiones en un solo fichero junto al script original. Y ya que estamos, podemos añadir metainformación al fichero para identificar los números de versiones de las revisiones, el nombre de las personas que llevan a cabo las operaciones, un *changelog*, etc. En definitiva, la estructura del fichero quedaría:

```
<META INFORMACION>
<PARCHE REVISION N>
<META INFORMACION>
<PARCHE REVISION N-1>
<META INFORMACION>
...
<META INFORMACION>
<PARCHE REVISION 1>
<META INFORMACION>
<FICHERO ORIGINAL>
```

Con las modificaciones pertinentes al script `gesrev`, habríamos construido un sistema de control de revisiones bastante razonable y robusto. Obsérvese que nuestro directorio de revisiones original se ha convertido en un solo fichero, siendo más portable, más robusto y más cómodo de utilizar. Tanto a la forma de directorio como a la forma de fichero que almacena revisiones se la denomina *repositorio* de revisiones.

3.2 El sistema de control de revisiones RCS

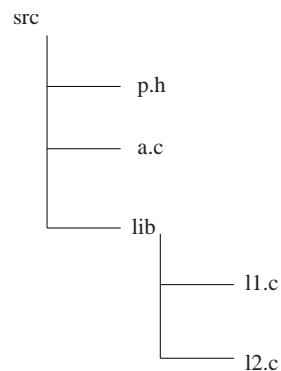
En la sección anterior vimos como podemos utilizar el concepto de parche y las herramientas `diff` y `patch` para construir repositorios de revisiones, usualmente identificadas por números de versión. Partiendo de una solución obvia pero con muchos problemas, fuimos refinando hasta llegar a un sistema que utilizaba un solo fichero para almacenar todas las revisiones, y un script que implementaba las operaciones típicas como la creación de una nueva revisión, obtención de diferencias entre dos revisiones cualesquiera, revertir revisiones, etc. En esta sección estudiaremos un sistema de control de revisiones de ese estilo llamado RCS que ofrece las operaciones anteriores y unas cuantas más.

4 Control de revisiones con CVS

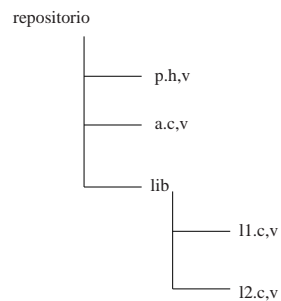
4.1 Motivación

En el capítulo anterior aprendimos a utilizar RCS para gestionar revisiones. Una de las características principales de dicho gestor de revisiones es que actúa sobre ficheros individuales manteniendo un fichero de revisiones sucesivas RCS de cada uno de ellos. El donde poner dichos ficheros RCS depende del criterio del desarrollador. Vimos que lo habitual es mantener un directorio por cada proyecto (al que llamamos “repositorio” del proyecto) con los ficheros RCS de sus ficheros.

El hecho de que RCS trabaje sobre ficheros individuales dificulta la gestión de proyectos compuestos no solo de varios ficheros, sino de una jerarquía de directorios mas o menos compleja. Eso implica que, pese a que el desarrollador puede obviar la complejidad de gestionar revisiones de un fichero, irremediamente debe lidiar con los ficheros de un proyecto en su conjunto. Por ejemplo, recordemos el proyecto del capítulo primero, cuya estructura de directorios puede verse en la siguiente figura.



Si utilizáramos RCS para gestionar las revisiones de este proyecto, deberíamos montar un repositorio para albergar los ficheros RCS de los ficheros ‘p.h’, ‘a.c’, ‘11.c’ y ‘12.c’. También es indispensable recordar la estructura de directorios del proyecto. Una forma de hacerlo es estructurar nuestro repositorio de la misma forma que la copia de trabajo, sustituyendo los ficheros fuente por sus ficheros RCS:



A primera vista esto puede parecer satisfactorio, pero cuando consideramos operaciones a nivel de repositorio (y no de fichero) las cosas se complican. Para verlo, supongamos que almacenamos el repositorio anterior en ‘~/repositorio’, y deseamos obtener en ‘~/proyecto’

una copia de trabajo de toda la estructura, con las últimas revisiones de cada uno de los ficheros. Procederíamos mediante los siguientes comandos.

```
jemarch@MalditoBastardo:~$ mkdir proyecto
jemarch@MalditoBastardo:~$ cd proyecto
jemarch@MalditoBastardo:~/proyecto$ mkdir lib
jemarch@MalditoBastardo:~/proyecto$ co ~/repositorio/p.h,v
/home/jemarch/repositorio/p.h,v --> p.h
revision 1.1
done
jemarch@MalditoBastardo:~/proyecto$ co ~/repositorio/a.c,v
/home/jemarch/repositorio/a.c,v --> a.c
revision 1.1
done
jemarch@MalditoBastardo:~/proyecto$ cd lib
jemarch@MalditoBastardo:~/proyecto/lib$ co ~/repositorio/lib/l1.c,v
/home/jemarch/repositorio/lib/l1.c,v --> l1.c
revision 1.1
done
jemarch@MalditoBastardo:~/proyecto/lib$ co ~/repositorio/lib/l2.c,v
/home/jemarch/repositorio/lib/l2.c,v --> l2.c
revision 1.1
done
```

Si a continuación nos ponemos a trabajar sobre algún fichero, debemos llevar la cuenta de ellos para hacer posteriormente un *check in* de las modificaciones. En conjunto, esta forma de trabajar parece presentar muchos inconvenientes. Piénsese, por ejemplo, en el trabajo que conllevaría obtener una copia de trabajo con revisiones concretas de cada uno de los ficheros, no necesariamente las últimas. ¡Habría que recordarlas!

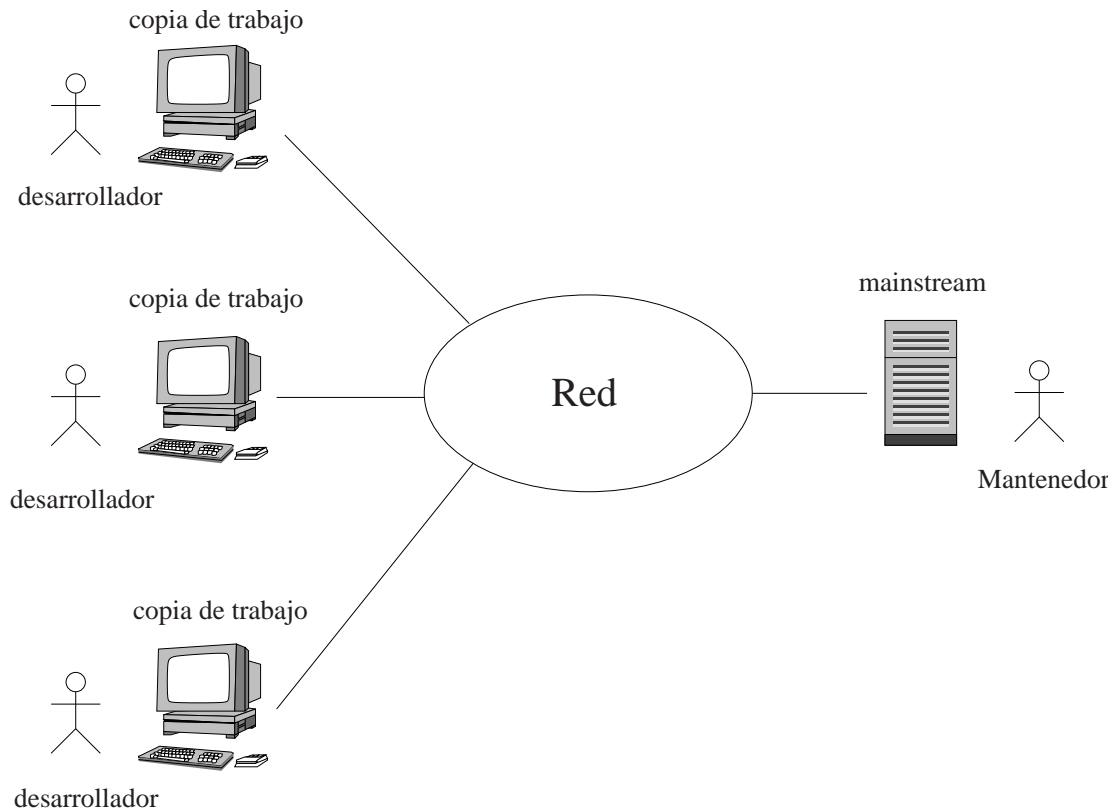
Con estas carencias de RCS en mente, Dick Grune desarrolló unos scripts `sh` que, utilizando RCS internamente, ofrecían funcionalidades para trabajar con este tipo de repositorio de ficheros RCS estructurados en directorios. Al resultado lo denominó `CVS`. Pese a que actualmente no queda nada de esa implementación original, el espíritu sigue siendo el mismo: superar la gestión orientada a fichero de RCS y adoptar un modelo más orientado al proyecto en su conjunto. Como veremos, `CVS` nos permite gestionar infinidad de aspectos a nivel de proyecto, como releases, branches, tags, etcétera. Además, `CVS` flexibiliza el modo de tratar con varios desarrolladores simultáneos, superando así el rígido modelo de cerrojo de RCS.

En las siguientes secciones examinaremos en profundidad el sistema `CVS`. Al principio nos centraremos en los aspectos de administración (¿cómo montar un repositorio `CVS`?) y a continuación en los aspectos de utilización por parte de desarrolladores.

4.2 Conceptos generales

Recordemos el material expuesto en el capítulo uno. Los dos roles principales involucrados en el desarrollo de software colaborativo son el mantenedor del programa y los desarrolladores. El mantenedor conserva la copia maestra de la estructura de fuentes del programa

en cuestión, que denominamos *mainstream*. Cuando un desarrollador desea trabajar sobre la estructura de fuentes, obtiene una copia privada y opera sobre ella. Una vez codificada y probada la mejora, la reenvía al mantenedor en forma de parche. El mantenedor entonces decide si incorporar los cambios al *mainstream*. La siguiente figura muestra esta situación tan típica en desarrollo colaborativo.



Es de esperar que el mantenedor utilice algún tipo de sistema de control de revisiones para almacenar el *mainstream*. De esta forma, podemos distinguir dos formas distintas de encarnación de las fuentes del programa:

- El *mainstream* o copia maestra, siempre almacenada mediante un sistema de control de revisiones con el fin de conservar historiales y posibilidad de recuperar cualquier estado intermedio de las fuentes. Debido a ello a las copias maestras a menudo se las denomina *repositorios*.
- Las copias privadas de los desarrolladores, llamadas *copias de trabajo*, sobre las que se efectúa el desarrollo propiamente dicho.

Una copia de trabajo puede estar o no *al día* con respecto al *mainstream*. Supongamos que un desarrollador (llamémosle **A**) obtiene una copia de trabajo en un momento dado. Una vez se ponga a trabajar y modifique cualquier archivo, es claro que su copia de trabajo no está al día respecto al repositorio: se han efectuado cambios en la copia de trabajo. Por otra parte, supongamos que posteriormente otro desarrollador **B** obtiene a su vez una copia de trabajo, efectúa cambios, y los envía al mantenedor para que ponga al día el *mainstream* antes que **A** haya modificado su propia copia de trabajo. En este caso se dice

que el mainstream no está al día con respecto a la copia de trabajo de **A**: se han realizado cambios en el repositorio.

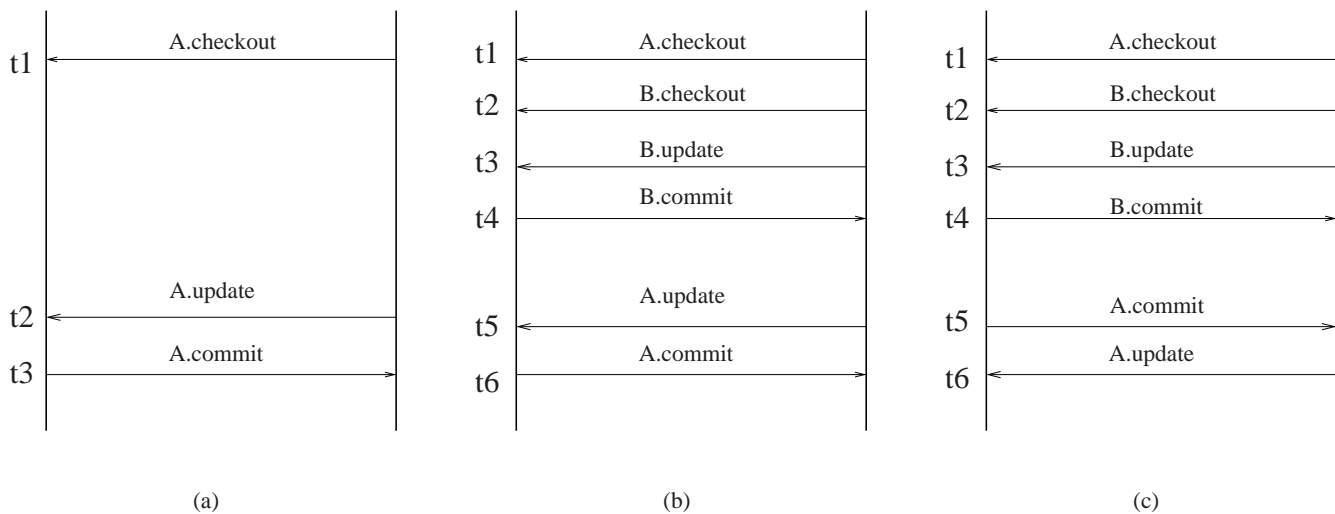
Para formalizar estas y otras situaciones en las que puede encontrarse el repositorio y las n copias de trabajo existentes en un momento dado, definimos las siguientes operaciones realizables por un desarrollador:

Checkout Es la operación mediante la cual un desarrollador obtiene una copia de trabajo del repositorio. En el instante en el que termina el checkout su copia de trabajo está al día respecto al repositorio.

Commit Es la operación mediante la cual se pone al día el repositorio con respecto a la copia de trabajo. Al igual que en el caso del checkout, en el instante en que termina el commit el repositorio está al día con respecto a la copia de trabajo.

Update Es la operación mediante la cual se pone al día la copia de trabajo con respecto al repositorio. Es importante no confundir esta operación con la de commit.

Veamos algunos ejemplos.



La figura (a) muestra la secuencia de operaciones típicas por parte de un desarrollador. Éste obtiene una copia de trabajo mediante un checkout. A continuación trabaja con el. Esto implica que a partir de t_1 el repositorio no estará al día con la copia de trabajo. Cuando el desarrollador termina sus modificaciones, debe revertir sus modificaciones al repositorio (esto es, poner al día al repositorio respecto a su copia de trabajo) haciendo un commit. Sin embargo, antes efectúa un update. ¿Por qué?. No olvidemos que en nuestro modelo *repositorio/copias de trabajo* es posible que haya mas de un desarrollador trabajando con el mismo repositorio. Por tanto, es posible que cuando el desarrollador **A** haga commit el contenido del repositorio haya cambiado con respecto a t_1 . Esa es la razón de que haga un update antes del commit. El update pone al día la copia de trabajo con respecto al repositorio, en el caso de que este último haya cambiado desde que realizara el checkout. En (a) no ha habido ninguna interferencia por parte de otro desarrollador, pero podría haberlo habido.

Esta situación de acceso simultáneo al repositorio se muestra en la figura (b). De nuevo el desarrollador **A** ha obtenido una copia de trabajo mediante checkout. Esta vez, no obstante, otro desarrollador llamado **B** obtiene su propia copia de trabajo. Este desarrollador termina antes que **A** y revierte su trabajo de forma correcta al repositorio (hace update y commit). A partir de este instante el repositorio no está al día respecto de la copia de trabajo de **A**. Cuando **A** termina sus modificaciones hace un update. En este momento se pone al día su copia de trabajo con respecto al repositorio, conteniendo éste las modificaciones realizadas por **B**. Luego en la copia de trabajo actualizada de **A** estarán sus cambios mezclados con los que ha introducido **B**. De esta forma, cuando **A** hace commit se crea una nueva revisión en el repositorio que incorpora tanto los cambios de **B** como los de **A**.

Por su parte, la figura (c) nos muestra una situación peligrosa: **A** ha hecho commit antes del update. ¿Qué puede ocurrir?. Como **A** no ha puesto al día su copia de trabajo antes de revertir sus cambios al repositorio, el sistema de control de revisiones confrontará la revisión previamente realizada por **B** con la de **A**. ¡Pero esta última no tiene en cuenta los cambios realizados por **B**!. Lo normal en estos casos es que el contenido del fuente se corrompa y no tenga sentido. Además, generalmente es causa de conflictos en el proceso de mezcla.

Todo esto nos lleva a enunciar una regla de oro: **siempre se debe hacer un *update* antes de un *commit*, aun cuando tengamos la certeza de que el repositorio no ha sido modificado por nadie mas**. Es una costumbre mas que saludable, que puede ahorrarnos muchos problemas en la utilización cotidiana de CVS.

Una vez asimilados los conceptos de *repositorio*, *copia de trabajo*, *checkout*, *commit* y *update*, lo que queda es examinar cómo los implementa CVS.

4.3 Administración de CVS

4.3.1 Estructura de un repositorio

El repositorio es donde se encuentra toda la información relativa a los ficheros bajo control de revisiones. Cada uno de los módulos contiene una estructura de ficheros que conforma el contenido del módulo. Puede considerarse un *módulo* como la estructura de ficheros de un proyecto, puesto bajo control de revisiones. La siguiente figura muestra la estructura de un repositorio CVS con n módulos.

