PROYECTO EUROPA
UNIVERSIDAD DE CÁDIZ

Análisis y Diseño de Algoritmos

Herramientas auxiliares libres

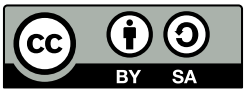
Curso 2008-2009

Autores:

Inmaculada Medina Bulo inmaculada.medina@uca.es
Francisco Palomo Lozano francisco.palomo@uca.es
Manuel Palomo Duarte manuel.palomo@uca.es

Acción de Innovación Docente:

Código IE-26 en el Registro de Actividades de Innovación Docente



Índice general

Índice de cuadros

Índice de figuras

Índice de listados

1

Introducción

Este documento pretende recopilar información necesaria para programar software usando una plataforma libre GNU/Linux.

Supondremos que el lector posee algunos conocimientos básicos sobre sistemas operativos. En particular, supondremos que sabe entrar en el sistema y manejar una terminal Linux.

2 Edición de código

2.1 Editores de texto frente a procesadores de texto

Es importante diferenciar entre un *editor de texto* y un *procesador de texto*. Un editor de texto produce «texto sencillo» sin códigos especiales de control, ni otra información más que la necesaria para representar la secuencia de caracteres que conforma el fichero editado. Por contra, un procesador de texto puede producir texto con elementos especiales de formato que hacen que el fichero obtenido resulte inapropiado para su utilización posterior por parte de las herramientas que empleamos habitualmente en programación. De hecho, el formato de la representación interna de los ficheros que producen los procesadores de texto suele estar muy alejado del texto sencillo. No es raro encontrar ficheros binarios, comprimidos, XML, etc.

Además, tradicionalmente, los formatos producidos por cada fabricante de procesadores de texto han sido incompatibles entre sí. Sólo recientemente se han realizado esfuerzos serios de estandarización internacional de formatos de documentos ofimáticos. Sin embargo, los formatos de texto sencillo poseen muchos más elementos en común y existen diversos estándares internacionales desde hace décadas, por lo que resulta notablemente más sencillo su empleo y transformación.

Es cierto que los procesadores de texto suelen presentar la posibilidad de guardar o exportar el texto editado como texto sencillo, eliminando la información extra de formato. No obstante, ya que están principalmente orientados a la producción de cartas y otros documentos ofimáticos, y no a la confección de programas de ordenador, no resultan adecuados para la edición de código.

Por lo tanto, es importante emplear un editor de texto en lugar de un procesador de texto a la hora de editar código fuente. Existen multitud de editores de texto

y los sistemas GNU/LINUX suelen venir equipados de serie con un gran número de ellos.

2.2 Editores de código

Un *editor de código* no es más que un editor de texto especializado que posee funciones específicas para facilitar el trabajo con código fuente. Algunas de las funciones que puede proporcionar un editor tal son:

Sangrado automático Al emplear la tecla ⇥ o la tecla ↵ la línea de código se sangra automáticamente. Esto es útil porque en ocasiones permite descubrir de manera anticipada posibles errores de sintaxis provocados por el olvido de algún delimitador o terminador. Este tipo de errores normalmente se traduce en un sangrado deficiente.

Resaltado de sintaxis Se muestra el código destacando los diferentes elementos, por ejemplo, con colores. Se distinguen así fácilmente las palabras reservadas, identificadores, cadenas, etc.

Autocompletado Se completan automáticamente algunos elementos conforme se escribe, o bien se ofrece la posibilidad de hacerlo. Por ejemplo, al escribir la palabra reservada correspondiente al comienzo de un bucle, el editor puede completar automáticamente el resto de la estructura del bucle, con lo que el programador pasa directamente a rellenar el esqueleto resultante. Otra posibilidad es que, al escribir, se presente una predicción de la entrada que el programador puede aceptar o, si sigue escribiendo, rechazar. También es posible que las principales estructuras sintácticas estén disponibles a través de un menú, lo que acelera la escritura de código y evita ciertos errores comunes.

Ayuda contextual Se integran, por ejemplo, los manuales en línea de las principales bibliotecas del lenguaje con el propio editor, de manera que se pueda consultar la ayuda de una función con una simple pulsación.

Referencias cruzadas Se genera un índice de referencias cruzadas para navegar con comodidad a lo largo de los distintos módulos que componen un programa. Por ejemplo, esto permite a partir de una llamada a función visitar su definición de manera sencilla, o comprobar rápidamente en qué otros módulos se utiliza.

Estructuración en bloques Se muestra la estructura de bloques de cada unidad de código, con la posibilidad de ocultarlos o mostrarlos individualmente.

Obviamente, no todos los editores de código poseen todas estas funciones o las tienen activadas por omisión. Del mismo modo, existen editores que incorporan otras extensiones. Según los editores se dediquen en exclusiva a un lenguaje o permitan trabajar con varios, distinguimos entre *editores específicos* y *editores multilenguaje*.

2.3 Entornos integrados de desarrollo

Existen también diversos *entornos integrados de desarrollo* (IDE). Estos entornos integran herramientas de naturaleza muy diversa bajo una misma interfaz, para facilitar su uso por parte del programador. Entre estas herramientas, dependiendo del IDE, pueden encontrarse algunas de las siguientes:

Editor La mayoría poseen un editor de código, normalmente con resaltado de sintaxis y algún tipo de ayuda contextual.

Gestor de proyectos Es normal que se trabaje con más de un fichero. Para ello suele crearse un proyecto al que se añaden los distintos ficheros. Normalmente se calculan automáticamente las dependencias entre los módulos a fin de minimizar el esfuerzo de recompilación.

Compilador Desde el IDE es fácil proceder a la compilación de un módulo individual o, más comúnmente, a la *reconstrucción* de todo el proyecto.

Depurador La existencia de un *depurador* integrado permite la ejecución de un programa paso a paso, el establecimiento de *puntos de control*, la inspección de variables, etc.

Perfilador Pocos IDE proporcionan acceso a un *perfilador*, que es una herramienta que permite obtener estadísticas sobre la ejecución de distintas partes de un programa y facilita la detección de problemas de rendimiento.

Control de versiones Algunos IDE permiten colocar los ficheros o proyectos bajo la supervisión de un *sistema de control de versiones*, lo que permite mantener un histórico de la evolución del programa, recuperar versiones anteriores, encontrar fácilmente las últimas modificaciones que se realizaron a un determinado fichero, etc.

Pruebas Algunos IDE proporcionan acceso a un *marco de pruebas*, por lo general, para realizar *pruebas unitarias*; éstas consisten normalmente en pruebas de caja negra que se realizan sobre las funciones.

Si bien los IDE se emplean en la producción profesional de programas de ordenador, no son tan adecuados desde el punto de vista didáctico cuando se pretende que el estudiante adquiera soltura con las distintas herramientas que se necesitan a lo largo de todo el proceso.

Por lo tanto, no aconsejamos su empleo, al menos, en tanto en cuanto no se manejen con soltura las herramientas básicas que intervienen en el proceso y se entienda cómo éstas se relacionan entre sí. Esto tiene el efecto beneficioso para el estudiante de proporcionarle una sólida fundación para el estudio de otras materias donde es indispensable conocer las interioridades de los compiladores y otras herramientas afines. Además, sin duda ayuda a comprender mejor el porqué de algunos errores.

2.4 GNU Emacs

GNU Emacs es un editor multilenguaje programable con un potente sistema de macros y un gran número de utilidades y extensiones disponibles. Aunque en sus versiones actuales pueden encontrarse interfaces gráficas, originariamente Emacs fue diseñado para terminales alfanuméricas. Debido a su concepción original, posee un sistema extraordinariamente potente de manejo a través de combinaciones de teclas. Los programadores versados en Emacs suelen emplear casi exclusivamente el teclado en lugar del ratón y del sistema de menús para ejecutar con increíble rapidez todo tipo de tareas.

En este sentido, Emacs comparte la filosofía del editor Vi, aunque es más sencillo de manejar y bastante más potente, ya que es completamente programable.[1] Emacs puede también funcionar como terminal, gestor de ficheros, navegador de hipertexto, cliente de FTP, cliente de correo, etc. También puede trabajarse con Emacs mediante una arquitectura cliente-servidor y lograr un sinfín de cosas más. En broma hay quien incluso le llama «el sistema operativo Emacs» y existen personas que se encuentran tan a gusto trabajando con él que rara vez salen de Emacs para realizar alguna tarea. No obstante, incluso las versiones mejoradas de Vi, como Vim, que poseen varias extensiones, son más ligeras y rápidas de arrancar que

[1] Emacs posee su propio lenguaje de programación. Se trata de Emacs Lisp, un dialecto de Lisp orientado a la manipulación de texto.

Emacs, lo que puede suponer una ventaja en aquellos sistemas donde los recursos estén muy limitados.

Desde una sesión de Emacs pueden crearse varios *marcos* o ventanas independientes. A su vez, cada ventana puede dividirse horizontal o verticalmente en varios *buffers* o zonas separadas. Notablemente, puede haber más de una ventana o buffer visitando un mismo fichero. Esto resulta muy útil, ya que permite, por ejemplo, editar una parte de un programa a la vez que se está viendo otra (con la posibilidad de cortar y pegar fácilmente de una zona a otra).

En la figura 2.1 puede apreciarse una sesión de Emacs desde la que se ha intentado compilar el fichero C++ que se muestra en el buffer superior. El resultado de la compilación aparece en el buffer inferior. De hecho, hay un error de sintaxis y el programador ha ordenado a Emacs que indique el lugar en el que se ha producido. Puede apreciarse que, puesto que el *buffer activo* contiene un fichero escrito en C++, Emacs lo ha detectado e incluso muestra una entrada en el menú específica para este lenguaje.

Las entradas que aparecen en el menú y en la barra de herramientas de Emacs son contextuales y van cambiando conforme al *modo de edición* del buffer activo, que en principio está determinado por el tipo de fichero que contiene.

2.5 Kate

Kate es un editor multilenguaje con sangrado automático, resaltado de sintaxis y estructuración de bloques. Posee también opciones de búsqueda y remplazo de texto que resultan a la vez potentes y fáciles de utilizar, así como un sistema de búsqueda de archivos.

Kate permite definir *marcadores*. La estructuración de bloques se realiza a través de un tipo especial de *marcador plegable*. Otra característica útil de Kate es que permite emplear una terminal en una ventana del editor.

En la figura 2.2 puede apreciarse una sesión de Kate en la que aparecen tres ventanas: una ventana de documentos que muestra los ficheros cargados en la sesión, una ventana que muestra un fragmento de código C++ perteneciente a uno de los ficheros y una terminal desde donde se ha compilado el programa.

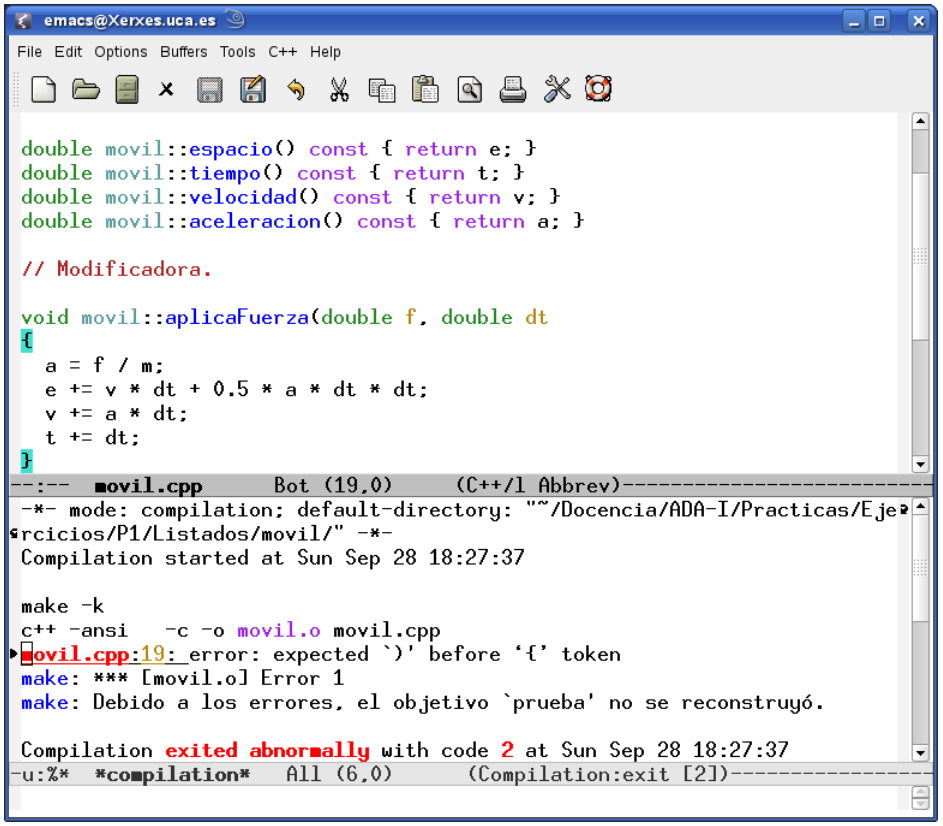


Figura 2.1: Editor GNU Emacs con interfaz gráfica GTk.

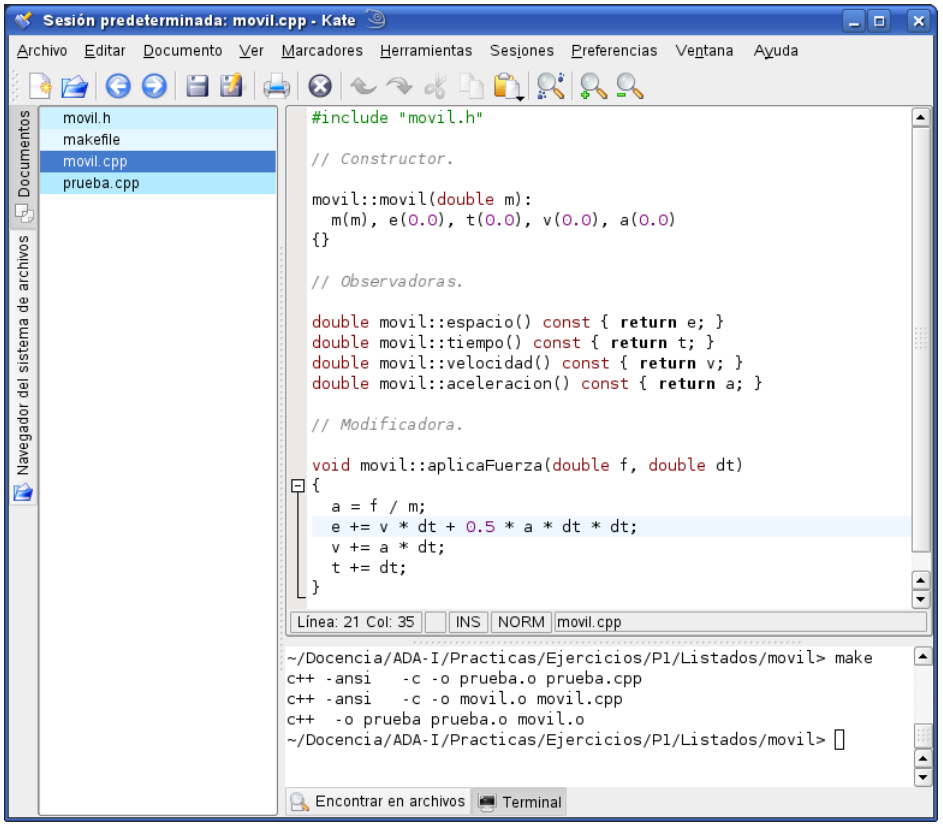


Figura 2.2: Editor Kate.

3

Recodificación

3.1 Conjuntos de caracteres y codificación

Para escribir código en cualquier lenguaje de programación o, en un marco más general, para escribir texto, son necesarias dos cosas:

1. Un *juego de caracteres* o *conjunto de caracteres* que describa los símbolos de los que disponemos para la escritura, a los que denominaremos *caracteres*, y un código distinto para cada uno de ellos.

2. Una *codificación* que especifique cómo representar cada código de carácter en una computadora.

En general, el diseño de juegos de caracteres y de codificaciones es una tarea compleja que ha sido desarrollada tradicionalmente por organismos de estandarización nacionales e internacionales y, en mayor o menor medida, por empresas o consorcios.

Existen normas que sólo fijan juegos de caracteres, mientras que otras se encargan de sus codificaciones. No obstante, es muy común que una misma norma recoja tanto juegos de caracteres como sus codificaciones. Es por ello que, a menudo, se confunden ambas cosas, si bien es importante distinguir entre ambos conceptos.

Los juegos de caracteres suelen asignar códigos a los caracteres de manera sencilla, por ejemplo, haciendo corresponder a cada carácter su posición en una tabla. En los casos más simples, una codificación puede consistir en representar directamente los códigos de los caracteres en la computadora.[1]

[1] Por ejemplo, supongamos que un juego de caracteres describe 256 caracteres mediante una tabla cuyas posiciones se numeran correlativamente de 0 a 255 y que el código de cada carácter

Sin embargo, una codificación puede ser mucho más compleja, ya que existen detalles técnicos importantes que pueden influir en ella. Existen codificaciones de longitud fija y de longitud variable, según cada carácter ocupe el mismo espacio una vez codificado o éste pueda variar en función del carácter en cuestión. La longitud del código puede ser inferior o superior a una mínima unidad de información prefijada, normalmente un octeto, codificándose los caracteres en múltiplos de dicha unidad. Cuando es menor, se suelen emplear bits adicionales de relleno. Cuando es mayor, se suele fijar un orden entre las unidades que lo componen.[2]

Históricamente, casi desde que florecieron los lenguajes de programación de alto nivel, tuvo una gran difusión en programación el estándar ASCII (American Standard Code for Information Interchange). Publicado por el USASI (United States of America Standards Institute) en 1968, este estándar especifica un juego de caracteres con una codificación de 7 bits diseñado originalmente para el inglés americano.

Posteriormente, ASCII, que es un estándar americano, fue adoptado como base para un estándar internacional por ISO, al que se denominó ISO 646, que definía varios juegos de caracteres, también de 7 bits, sustituyendo el carácter de dólar por otros símbolos monetarios y diversos caracteres de puntuación por otros caracteres alfabéticos de distintos idiomas al objeto de cubrir mejor las lenguas alemana, francesa, española, entre otras.

Luego, surgieron los juegos de caracteres ISO 8859, de 8 bits, de los cuáles los más empleados en Europa occidental son los ISO 8859-1 e ISO 8859-15.

En teoría, el código C++ puede escribirse empleando la norma ISO 10 646 y, en concreto, la codificación UTF-8 definida en ella, véase [ISO, 2000], que es de longitud variable. Esta norma es equivalente al estándar UNICODE [UNICODE, 2006] e incluye cualquier lengua escrita humana (y algunas más, véanse [UNICODE, 2008 *a*, *b*]).

Por ejemplo, podríamos emplear π como identificador. Aquí el glifo π representa un carácter alfabético, del alfabeto griego, que ocupa la posición 960 (03C0 en hexadecimal) entre los caracteres definidos por la citada norma. En concreto, se trata del carácter «letra pi minúscula griega».

---

se corresponde con su posición en la tabla. Una codificación puede especificar que cada código de carácter se represente en la máquina con 8 bits, en binario natural.

[2] Un juego de caracteres codificable en 7 bits puede emplear, finalmente, 8, especificándose, por ejemplo, que su bit más significativo sea 0. En un juego de caracteres codificable en 16 bits puede colocarse el octeto más significativo antes que el menos significativo (*big endian*) o al contrario (*little endian*).

No obstante, en la práctica esto depende del compilador que estemos empleando. Actualmente, el compilador de GNU C++ sólo permite emplear ASCII.

3.2 Codificación en los sistemas GNU/LINUX

Los sistemas GNU/LINUX permiten al usuario trabajar en distintas lenguas y disponen de varios sistemas de codificación para representar texto escrito en ellas. Dentro de estos sistemas de codificación, los dos más importantes que utilizaremos son:

- UTF-8 (una codificación de Unicode e ISO 10646 UCS)
- Latin 1 (ISO 8859-1)

Normalmente, la codificación inicial dependerá de los ajustes iniciales de idioma y teclado especificados durante la instalación del sistema operativo, así como de la propia distribución empleada. Los sistemas GNU/LINUX más modernos suelen venir configurados para emplear inicialmente UTF-8. Podemos ver la «localización cultural» de nuestro sistema ejecutando la orden `locale` desde una terminal:

```
> locale
LANG=es_ES.UTF-8
LC_CTYPE="es_ES.UTF-8"
LC_NUMERIC="es_ES.UTF-8"
LC_TIME="es_ES.UTF-8"
LC_COLLATE="es_ES.UTF-8"
LC_MONETARY="es_ES.UTF-8"
LC_MESSAGES="es_ES.UTF-8"
LC_PAPER="es_ES.UTF-8"
LC_NAME="es_ES.UTF-8"
LC_ADDRESS="es_ES.UTF-8"
LC_TELEPHONE="es_ES.UTF-8"
LC_MEASUREMENT="es_ES.UTF-8"
LC_IDENTIFICATION="es_ES.UTF-8"
LC_ALL=
```

Por ejemplo, en este caso, nuestra localización cultural indica que empleamos español (`es`) de España (`ES`) con codificación UTF-8.

Podemos modificar la localización cultural modificando la variable de entorno LANG. Por ejemplo, si empleamos el intérprete de órdenes `bash`, podríamos ejecutar:

```
LANG=es_ES.ISO8859-1
```

3.3 El problema de la codificación

Si trabajamos con un mismo fichero en sistemas con codificaciones distintas y con editores que no preservan la codificación original es normal que acabemos con un fichero cuya codificación no sea la que deseamos.

Esto puede manifestarse de diversas formas. Problemas con los acentos, problemas con el sangrado, aparición de caracteres extraños al final de la línea, etc.

En una terminal podemos intentar averiguar la codificación de un fichero con la orden `file`. Por ejemplo, si disponemos de un fichero llamado `prueba` codificado con Latin 1 obtendríamos lo siguiente:

```
> file prueba
prueba: ISO-8859 text
```

Con la opción `-i` obtendríamos la información en formato MIME:

```
> file -i prueba
prueba: text/plain; charset=iso-8859-1
```

Hay que tener en cuenta que algunas codificaciones son indistinguibles, por lo que no siempre obtendremos el resultado deseado. Una vez averiguada la codificación de un fichero podemos intentar recodificarlo.

3.4 Tabuladores

Existen editores que, cuando se pulsa la tecla ⇥ insertan un *carácter tabulador* en el texto. En su representación visual o impresa un carácter tal se presenta habitualmente como una secuencia de espacios, hasta la siguiente *parada de tabulación*. La idea es permitir de manera sencilla la alineación vertical de texto. Por ejemplo, un editor puede estar configurado para poseer una parada de tabulación cada ocho columnas.

No obstante, la posición de las paradas de tabulación puede depender del editor, visor o programa de impresión utilizado. Cada herramienta puede estar configurada para sustituir cada carácter tabulador por un número distinto de espacios, con los consiguientes problemas de presentación. Si esto ocurre, una solución es eliminar tabuladores en favor de espacios.

Normalmente, los editores poseen opciones de configuración que permiten que la edición se lleve a cabo sin tabuladores, es decir, insertando espacios. De todos modos, cuando sea necesario, podemos sustituir los tabuladores por espacios empleando la orden `expand`, que nos permite especificar las paradas de tabulación. Por ejemplo, para sustituir los tabuladores por espacios con paradas de tabulación cada cuatro columnas haríamos:

```
> expand -t 4 prueba > prueba-sin-tabuladores
```

También podemos especificar una lista de paradas de tabulación:

```
> expand -t 4,6 prueba > prueba-sin-tabuladores
```

Para hacer lo contrario, es decir, sustituir espacios por tabuladores, podemos emplear la orden `unexpand`.

3.5 Finales de línea

Existen tres convenios principales muy extendidos a la hora de representar los finales de línea en un fichero de texto:

Mac Se emplea fundamentalmente en los sistemas de Apple. Cada final de línea consta de dos caracteres: un carácter de nueva línea, seguido de uno de retorno de carro.

DOS Se emplea fundamentalmente en los sistemas de Microsoft. Cada final de línea consta de dos caracteres: un carácter de retorno de carro, seguido de uno de nueva línea.

Unix Se emplea en sistemas Unix y derivados, como GNU/LINUX. Cada final de línea consta de un único carácter: un carácter de nueva línea.

```
> dos2unix *.cpp *.h
> dos2unix -k *
```

La opción `-k` (o `--keepdate`) preserva la fecha de modificación de los ficheros.

3.6 Acentos y otros caracteres especiales

Podemos traducir un fichero de una codificación a otra con distintos programas. Uno de los más sencillos es `recode`:

- Paso de Latin 1 a UTF-8:

```
> recode latin-1..utf-8 ficheros
```

- Paso de UTF-8 a Latin 1:

```
> recode utf-8..latin-1 ficheros
```

En lugar de `latin-1` y `utf-8` podemos emplear las abreviaturas `l1` y `u8`.

Hay que tener en cuenta que hay recodificaciones seguras y otras que no lo son. Por ejemplo, siempre se puede pasar un fichero en Latin 1 a UTF-8 y es seguro deshacer dicho cambio, es decir, volver a pasar el fichero resultante a Latin 1.

Sin embargo, en general, un fichero en UTF-8 no se puede pasar a Latin 1. Por ejemplo, un fichero UTF-8 puede contener caracteres del ruso o del japonés que no se pueden representar en Latin 1.

3.7 Recodificación desde Emacs

En el editor de texto GNU Emacs, la codificación que se está empleando se puede observar en la primera parte de la línea de modo de Emacs, como se describe en el cuadro 3.1.

Normalmente, si no se ha especificado lo contrario, Emacs intentará emplear al editar un fichero la codificación del entorno en el que se ejecuta, es decir, la que

`--:--`	Por decidir Sin codificación especial. Esto aparece, por ejemplo, cuando el fichero está vacío o, en general, cuando no contiene acentos ni otros símbolos especiales que no puedan representarse en ASCII (ISO 646).
`-u:--`	UTF-8 Permite representar prácticamente todas las lenguas conocidas, así como una gran cantidad de símbolos especiales, científicos, técnicos, etc.
`-1:--`	Latin 1 (ISO 8859-1) Apropiado para las lenguas de Europa occidental, principalmente.
`-0:--`	Latin 9 (ISO 8859-15) Muy parecido a Latin 1, pero incorpora el símbolo € del euro en lugar del símbolo ¤ de moneda, así como otras modificaciones menores.

Cuadro 3.1: Codificaciones en la línea de modo.

aparece en nuestra localización cultural. Al principio, cuando el fichero está vacío, aparecerá `--:--` en la línea de modo. Sin embargo, conforme vayamos introduciendo caracteres y guardándolos en el fichero, su codificación podrá variar. Por ejemplo, si la codificación del entorno es UTF-8 e introducimos caracteres específicos de esta codificación, la línea de modo mostrará `-u:--`.

Si se desea, puede cambiarse el entorno del lenguaje antes de comenzar a editar. Por ejemplo, para cambiar a Latin 1: `C-x RET l latin-1`.[3] Otra posibilidad es emplear `M-x set-language-environment RET latin-1`.[4] Por último, podemos recurrir al menú:

```
Options → Mule (Multilingual Environment)
        → Set Language Environment
        → European
        → Latin-1
```

Puede cambiarse de codificación explícitamente desde dentro de Emacs con `C-x RET f` o bien, alternativamente, `M-x set-buffer-file-coding-system`. También puede emplear el menú:

```
Options → Mule (Multilingual Environment)
        → Set Coding Systems
        → For Saving This Buffer (C-x RET f)
```

En cualquier caso nos preguntará la codificación deseada. Podemos contestar, por ejemplo, con alguna de las siguientes: `utf-8`, `latin-1` o `latin-0`.

En todo caso, es necesario guardar para hacer efectivo el cambio de codificación. No obstante, para que esto sea posible, el fichero debe contener exclusivamente caracteres representables con la codificación especificada; de lo contrario, la línea de modo volverá a indicar la codificación original.

Al cargar un fichero, Emacs intenta averiguar en qué está codificado. Sin embargo, cabe la posibilidad de que no lo haga correctamente. El problema es que algunas codificaciones pueden ser indistinguibles, a falta de otra información externa. Por ejemplo, tanto Latin 1 como Latin 9 son codificaciones que contienen 256 símbolos, cada uno de los cuales se representa mediante 8 bits haciéndole corresponder como código un número único entre 0 y 255. El código que en Latin 1

[3] En notación Emacs, esto significa que hay que pulsar las teclas Ctrl y X al mismo tiempo, luego ↵, L, escribir `latin-1` y, por último, pulsar ↵.

[4] Que equivale a pulsar las teclas Alt y X al mismo tiempo, escribir el nombre de la orden (puede emplearse el tabulador para completar), luego ↵, escribir `latin-1` y, por último, ↵.

corresponde a ¤, en Latin 9 corresponde a €. Si editamos un fichero en Latin 9, al cargarlo, Emacs lo interpretará como Latin 1 y veremos ¤ en lugar de €. No es problema de Emacs, es un problema inherente a estas codificaciones.

En tales casos, no cabe otra posibilidad que decirle a Emacs en qué codificación está el fichero antes de cargarlo o, una vez cargado, utilizar `C-x RET r` o también `M-x revert-buffer-with-coding-system`. También se puede emplear el menú:

```
Options → Mule (Multilingual Environment)
        → Set Coding Systems
        → For Reverting This File Now (C-x RET r)
```

4 Traducción y ejecución

4.1 Entornos de desarrollo

Un *entorno de desarrollo* de software es un conjunto de herramientas que se emplean en el proceso de desarrollo de software. Esto incluye, en el sentido amplio del término, a la *plataforma de desarrollo*: es decir, el sistema operativo y el hardware empleado. Existen distintos tipos de entornos:

Tradicionales Constan de un conjunto de herramientas independientes que normalmente se usan en la etapa de implementación y en la de prueba. Ejemplos de estas herramientas son:

- Terminales
- Editores
- Intérpretes
- Compiladores
- Enlazadores
- Depuradores
- Perfiladores
- Generadores de referencias cruzadas
- Sistemas de ayuda a la recompilación
- Sistemas de control de versiones
- Sistemas de prueba de programas

Integrados Agrupan herramientas existentes en un entorno tradicional en una única herramienta. Suelen incluir sus propias bibliotecas de componentes reutilizables y también facilitar la creación de *interfaces gráficas de usuario*[1].

CASE El nombre proviene de sus siglas en inglés: *Computer Assisted Software Engineering* y han sufrido una gran evolución. En su estado actual, suelen ser entornos integrados que han sido extendidos para cubrir total o parcialmente el proceso de desarrollo y no exclusivamente la etapa de implementación. Generalmente facilitan el empleo de una metodología de desarrollo de software específica.

No nos dedicaremos aquí a explicar ningún entorno de desarrollo particular, ya que nuestro objetivo es únicamente describir en suficiente detalle cómo se produce un ejecutable a partir de código fuente escrito en lenguaje C++. Por otro lado, este proceso es prácticamente común a todos los entornos y, aunque permanezca oculto al desarrollador, conocerlo permite comprender mejor el porqué de ciertos errores de programación.

A estos efectos, supondremos que se emplea un entorno de desarrollo tradicional formado por un sistema operativo tipo UNIX, un editor y un compilador de C++. Concretamente, los autores suelen emplear un sistema operativo GNU/LINUX, el editor GNU Emacs y el compilador GNU C++; todas herramientas pertenecientes al proyecto GNU (*GNU's Not Unix*), de la FSF (*Free Software Foundation*).

4.2 Cómo dar las órdenes

El proceso de traducción nos lleva del fichero de texto con el *código fuente* al fichero binario con el *código ejecutable*. Supongamos que ya se dispone del código fuente y que hay que obtener el ejecutable.

En todos los sistemas LINUX existe, al menos, un compilador de C++. Éste es, por lo general, el compilador GNU C++ y será el empleado en los ejemplos que aparecerán a continuación. Normalmente se ejecuta mediante la orden `c++`, o también `g++`.

El *fichero fuente* es un fichero de texto que crearemos con nuestro editor favorito. Supongamos que hemos creado nuestro primer programa en el fichero `hola.cpp` y que disponemos de una terminal; la orden

[1] IGU (en inglés, GUI: *Graphical User Interface*).

```
> c++ hola.cpp
```

compilará este código para producir el ejecutable en el fichero `a.out`; para ejecutarlo simplemente escribiremos su nombre:

```
> a.out
¡Hola a todos!
```

Si ahora escribimos otro programa y damos la orden análoga para compilarlo, el nuevo fichero ejecutable `a.out` se escribirá sobre el antiguo, que se perderá. Para evitar esto el programa traductor, `c++`, dispone de una opción que nos permite darle otro nombre al fichero de salida. Esta opción es `-o` *fichero*; por lo tanto, otra forma de compilar es

```
> c++ -o hola hola.cpp
```

o bien

```
> c++ hola.cpp -o hola
```

En los sistemas GNU/LINUX un fichero puede tener cualquier nombre y éste puede contener cualquier carácter salvo el separador de directorios, la barra `/` (y el `\0`). Es costumbre que el *fichero ejecutable* tenga el mismo nombre que el fuente, pero sin extensión, y el nombre del fichero con código fuente en C++ acabe en los caracteres `.cpp` (aunque también puede acabar en `.cc` o `.cxx`). Y recuerde que las mayúsculas y las minúsculas se consideran caracteres diferentes.

4.3 Etapas en la traducción

En realidad, `c++` no es exactamente el compilador, aunque para abreviar digamos «el compilador `c++`». Más bien, es un programa *maestro* o *conductor* que simplemente se va a encargar de bifurcar procesos ejecutando una serie de programas, entre los que está el verdadero compilador, que son los que van a hacer realmente la tarea.

Además `c++` se encargará de gestionar las opciones de la línea de órdenes y sabrá a qué programa hay que pasárselas, creará y borrará ficheros intermedios temporales, etc.

Puede preguntarse por qué la traducción completa no se hace en un único programa. La respuesta es que es mucho más fácil para el fabricante hacerlo en pasos. Además el programa responsable de cada paso puede ser utilizado por separado y ser así aprovechado por otro compilador.

El esquema de la figura 4.1 indica de manera simplificada los pasos que se siguen en la traducción y las opciones que intervienen. Conceptualmente podemos imaginarnos que la traducción tiene lugar en esos pasos, aunque algunos fabricantes, por ejemplo, incluyan el preprocesador y el compilador en un mismo programa o se efectúe la compilación y la optimización en varias etapas.

Preprocesado El *preprocesador* es un programa que «prepara» el código fuente para el compilador. Nos ahorra muchísima tarea, pues gracias a él podemos incluir en el código otros ficheros (llamados *cabeceras*), definir símbolos que nos aclararán el programa (*macros*), compilar partes del código condicionalmente, etc. También se encarga de unir líneas partidas y cadenas de caracteres literales adyacentes, quitar los comentarios, sustituir *secuencias de escape*, etc.

Las líneas que comienzan con el carácter *sostenido* (#) son interpretadas por el preprocesador, no por el compilador.

La opción `-E` preprocesa el código fuente y presenta el resultado en la salida estándar. Una extensión de fichero `.cpp` indica código fuente sin preprocesar, por lo que `c++` le pasa el preprocesador en primer lugar. Éste es el caso más normal.

Compilación El *compilador* propiamente dicho actúa ahora sobre el código preprocesado; en algunos sistemas el preprocesador y el compilador son un mismo programa, pero todo ocurre *como si* fueran dos distintos. La compilación puede suceder en un paso o en varios.

Dependiendo de las opciones pasadas a `c++` puede haber una[2] etapa de *optimización* de código, donde el compilador intenta eliminar posibles redundancias, almacenar ciertas variables en registros, etc., produciendo un mejor

[2] O varias. La «optimización», más correcto sería decir «mejora», de código es un proceso complejo que puede requerir la realización de diversas tareas en distintos momentos de la traducción, incluso tras el ensamblado.

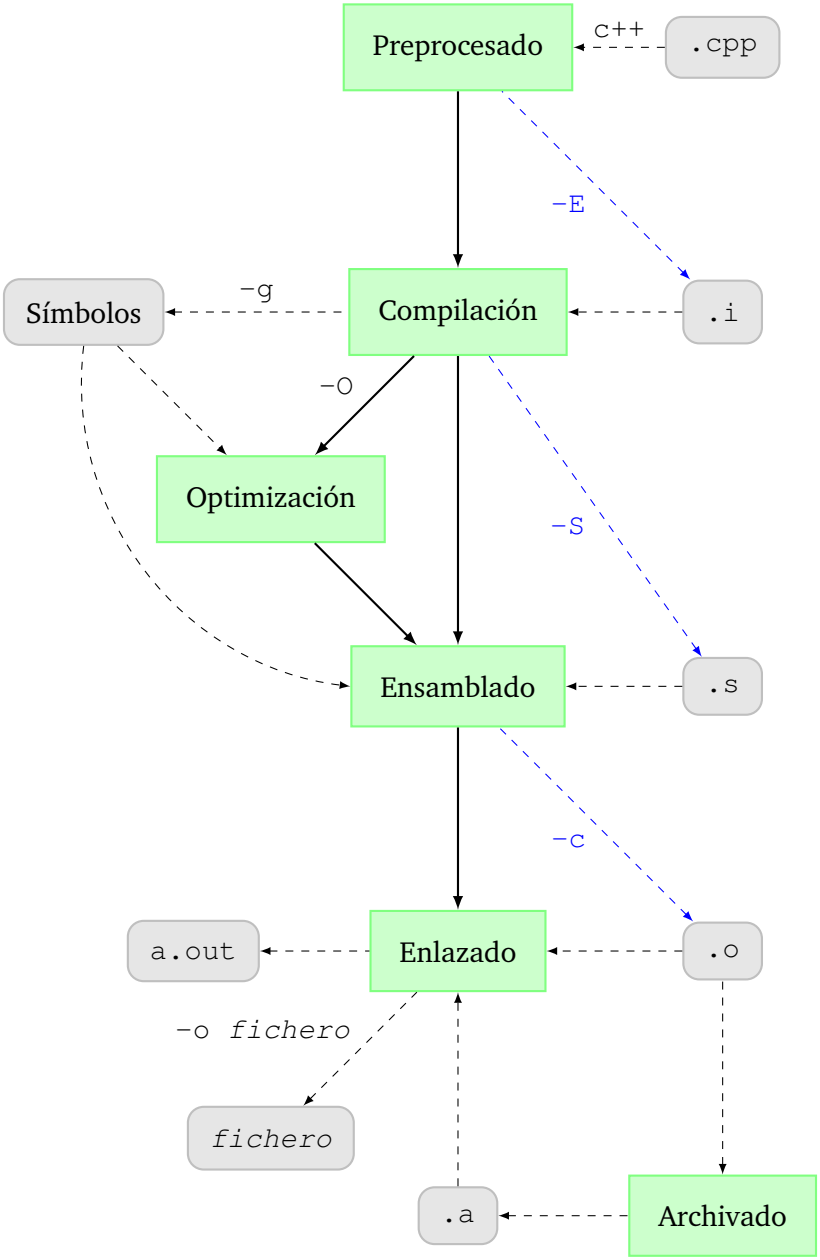


Figura 4.1: Etapas en la traducción de un programa.

código ensamblador. Esto se consigue con la opción `-O` (*o* mayúscula), que puede ir seguida de un dígito para expresar distintos grados de optimización.

También es muy importante la opción `-g`, que hace que en el código resultante se incluya una tabla de símbolos que podrá ser interpretada más tarde por un programa llamado *depurador*, que nos permite controlar la ejecución del programa interactivamente y descubrir errores ocultos. No es recomendable mezclar las opciones `-O` y `-g`, y algunos compiladores no lo permiten siquiera.

Una extensión de fichero `.i` indica código fuente ya preprocesado, por lo que `c++` no le pasa el preprocesador, sino que lo compila directamente.

Ensamblado Aunque en algunos sistemas es normal que el compilador ya produzca código máquina directamente, otras veces hay otro paso intermedio y lo que se produce es código en ensamblador, que ya es un lenguaje de bajo nivel. Esto hace que el compilador sea más fácil de realizar, puesto que el *ensamblador* ya está hecho y se aprovecha. Además podemos ver el código ensamblador y modificarlo si sabemos y nos interesa afinar mucho.

Con la opción `-S` obtenemos un fichero de extensión `.s` con el código fuente en ensamblador. Asimismo, si a `c++` le pasamos un fichero con extensión `.s`, supone que contiene código ensamblador y le pasa el programa ensamblador directamente.

La opción `-c` deja el código máquina resultante del ensamblado en un fichero con extensión `.o` llamado *módulo objeto*, que ya es código máquina pero no es ejecutable aún. Esta opción es muy importante, pues nos permite la *compilación por separado*.

Enlazado En todos los sistemas existe una separación en este punto. El enlazado se efectúa aparte por otro programa. Éste se llama *enlazador* o *editor de enlaces*. Como su nombre indica, enlaza el módulo objeto producido por el ensamblador con otros módulos que le podamos indicar, y con los módulos correspondientes de las *bibliotecas* de funciones, que son archivos formados por módulos objeto con funciones ya compiladas. Sin que le digamos nada, `c++` se encarga de llamar al enlazador pasándole el archivo correspondiente a las funciones de la biblioteca estándar de C++ y otras muchas del sistema, de uso común.

El resultado de todo es el fichero `a.out` con el código máquina ejecutable. Ya se ha dicho que con la opción `-o` *fichero* podemos cambiarle el nombre.

5

Control de dependencias

5.1 GNU Make

La utilidad GNU Make está especialmente diseñada para ayudar a rehacer trabajos monótonos que deban repetirse cada vez que ciertos ficheros sean modificados. Ésta es, precisamente, la situación que nos encontraremos en muchas ocasiones.

Para ello, se describen en un fichero una serie de *reglas*. Cada regla consta de *objetivos*, *dependencias* y *órdenes*. Sólo los objetivos son obligatorios en una regla, y se separan de las dependencias por el carácter «:». Cada línea que describe las dependencias de uno o varios objetivos puede ir seguida de una serie de órdenes sangradas obligatoriamente con un tabulador.

Cuando se requiera la realización de un objetivo, basta ejecutar `make`, que comprobará si alguna de sus dependencias ha sido modificada con posterioridad al objetivo (esto es, si el objetivo es más antiguo). En tal caso, el objetivo deberá rehacerse y sus órdenes ejecutarse, no sin antes comprobar y rehacer las dependencias del mismo que sean necesarias.

No todos los objetivos son necesariamente ficheros: existen «objetivos falsos», que no están asociados a ningún fichero, y que se emplean para desencadenar una serie de acciones (por ejemplo, la limpieza del directorio de trabajo). Los objetivos falsos *all*, *clean* y *clean-all* son tan comunes en los *makefiles* que no se suelen traducir sus nombres.

Al principio, se definen una serie de variables a cuyos valores se accede mediante $. Algunas de ellas, como **CXX** y **CXXFLAGS**, son especiales: `make` las utiliza en sus «reglas implícitas». Por ejemplo, una de las reglas implícitas de `make` dice que un módulo objeto se obtiene a partir de su fuente .cpp compilando con $(**CXX**)

y con las opciones $(**CXXFLAGS**). Por lo tanto, `make` conoce esta información y no es necesario suministrársela.

Existen otras variables especiales. Así, **$@** es el objetivo que se está tratando, $^, todas sus dependencias, y $< su primera dependencia.

Los valores de las variables se pueden cambiar desde fuera. Por ejemplo, para compilar de manera que luego se pueda emplear el depurador se haría:

```
> make CXXFLAGS=-g
```

Supongamos que hemos incluido una variable llamada OPTIMIZACION, que inicialmente vale 0, para poder cambiar fácilmente el nivel de optimización del compilador. Para GNU C++, 0 indica «sin optimización», y 3, «optimización plena». Por ejemplo, si ya se ha compilado el código para llevar a cabo un experimento sin optimización y, posteriormente, se desea recompilarlo para volver a realizar el experimento con optimización plena, puede hacerse:

```
> make clean
> make OPTIMIZACION=3
```

Para ampliar información sobre GNU Make recomendamos consulte su manual o ABURRUZAGA GARCÍA [2008], ambos disponibles gratuitamente en la red.

5.2 Otras herramientas

Existen otras herramientas que pueden hacer labores similares a GNU Make, entre ellas destacamos:

cmake es una herramienta más potente que GNU Make. No compila el software directamente, sino que genera ficheros para compilarlo en la plataforma correspondiente (GNU Make en el caso de GNU/Linux).

cook al igual que Make, indica cómo compilar un software a partir de su fuente. Se basa en la indicación de cómo generar cada fichero destino mediante recetas, de ahí su nombre.

Perforce Jam es otro sistema similar a GNU Make, pero cuyo fichero de configuración usa un lenguaje imperativo y listas. Además permite la conexión con *autoconf*.

premake es un software para generar ficheros que compilen automáticamente un software en una plataforma dada. Trabaja con código C, C++, y C#, y es capaz de generar ficheros para Visual Studio y GNU Make entre otros sistemas.

6 Calculadoras

6.1 GNU bc

GNU bc es una calculadora programable de precisión arbitraria. A continuación se muestra un ejemplo de sesión.

```
> bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.

# Los cálculos se realizan con los decimales que indique «scale».

scale
20

# Obsérvese que no se redondea.

1 / 6
.16666666666666666666

# Podemos calcular números muy grandes.

2^32
4294967296
2^64
18446744073709551616
2^128
340282366920938463463374607431768211456
```

```
# Siempre podemos utilizar el resultado previo, que se guarda en «.».

.^2
11579208923731619542357098500868790785326998466564056403945758400791\
3129639936

# Disponemos del seno, coseno y arcotangente.

pi = 4 * a(1)
pi
3.14159265358979323844

s(pi)
.00000000000000000002

c(pi)
-1.00000000000000000000

# Disponemos de la exponencial y del logaritmo neperiano.

e = e(1)
e
2.71828182845904523536

l(e)
.99999999999999999999

# Ejemplo de cálculo con la raíz cuadrada.

phi = (1 + sqrt(5)) / 2
phi
1.61803398874989484820
```

Nótese que `bc` no redondea, sino que trunca. Se pueden crear variables y agrupar expresiones empleando paréntesis.

Ilustraremos la programación de la calculadora con un ejemplo sencillo pero bastante completo. Consideremos el siguiente algoritmo, que comprueba si un vector v de n elementos está ordenado.

Para analizar su tiempo promedio supondremos que los n elementos de v son distintos y que sus $n!$ permutaciones son equiprobables. Esto equivale, en la práctica, a suponer una distribución uniforme de los datos de entrada. Se toma como

```
i ← 1
ordenado ← ⊤
mientras i < n ∧ ordenado
        i ← i + 1
        ordenado ← v[i] ≤ v[i + 1]
```

Listado 6.1: Algoritmo que comprueba si un vector está ordenado.

operación crítica la comparación entre elementos del vector, es decir, el tiempo de este algoritmo para un vector v se define como el número de veces que se ejecuta la comparación $v[i] \leq v[i+1]$.

Bajo estas hipótesis, puede demostrarse que el tiempo promedio que se obtiene para un vector de tamaño $n > 1$ viene dado por:

$$\bar{t}(n) = \sum_{i=1}^{n-1} \frac{1}{i!} \tag{6.1}$$

y puede aproximarse asintóticamente mediante $e - 1$.

El programa que describiremos a continuación permite calcular dichos tiempos hasta un n dado, así como el error absoluto de su estimación asintótica. En primer lugar, definimos la función factorial y la función que calcula el tiempo promedio a partir de la fórmula 6.1.

```
/* Función factorial */

define factorial (n) {
  auto p, i
  p = 1
  for (i = 1; i <= n; ++i) {
    p *= i
  }
  return p
}
```

```
/* Tiempo promedio */

define tiempo(n) {
  auto s, i
  s = 0
  for (i = 1; i < n; ++i) {
    s += 1 / factorial(i)
  }
  return s
}
```

A continuación, definimos una función para redondear un número con los decimales que se indiquen, ya que no deseamos obtener los resultados truncados. El redondeo se realiza «al más cercano».

```
/* Redondeo al más cercano */

define redondeo(x, p) {
  auto tmp

  /* Suma 5 en la posición del (p + 1)−ésimo dígito decimal */

  x += 5 / 10 ^ (p + 1)

  /* Trunca a p dígitos */

  tmp = scale
  scale = p
  x /= 1
  scale = tmp
  return x
}
```

Por último, solicitamos la precisión y el tamaño de vector. Como resultados presentamos los tiempo promedio y los correspondientes errores absolutos de la

aproximación asintótica para cada tamaño de vector hasta el indicado.

```
/* Precisión */

"precisión = "
p = read()
print "\n"

/* BC trunca, no redondea: emplearemos dígitos de guarda */

scale = p + 3

/* Número de elementos */

"n = "
n = read()
print "\n"

/* Resultados */

for (i = 1; i <= n; ++i) {
  espacios = length(n) - length(i)

  /* Tiempo promedio del algoritmo */

  print "t(", i, ")"
  for (j = 0; j < espacios; ++j) {
    " "
  }
  " = "
  t = tiempo(i)
  print redondeo(t, p), "\t\t"

  /* Error absoluto de la aproximación asintótica */

  print "e(", i, ")"
  for (j = 0; j < espacios; ++j) {
    " "
  }
  " = "
```

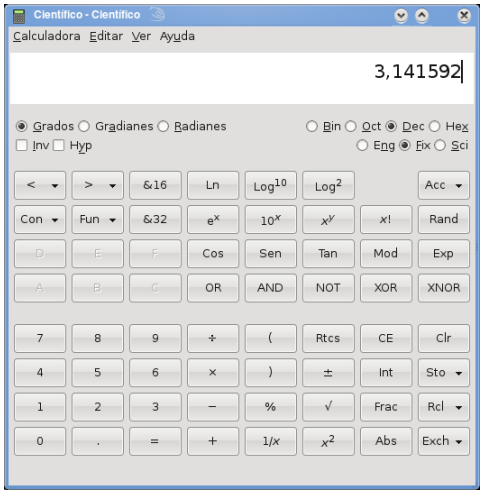


Figura 6.1: Calculadora gcalctool en modo científico.

```
  e = e(1) − 1 − t
  redondeo(e, p)
}
quit
```

6.2 Calculadoras gráficas

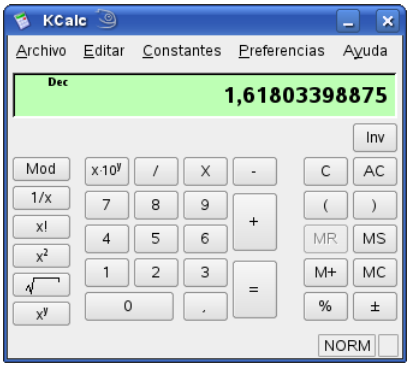
Los principales escritorios de GNU/Linux han creado sus propias calculadoras con entorno gráfico: KCalc y gcalctool.

6.3 gcalctool

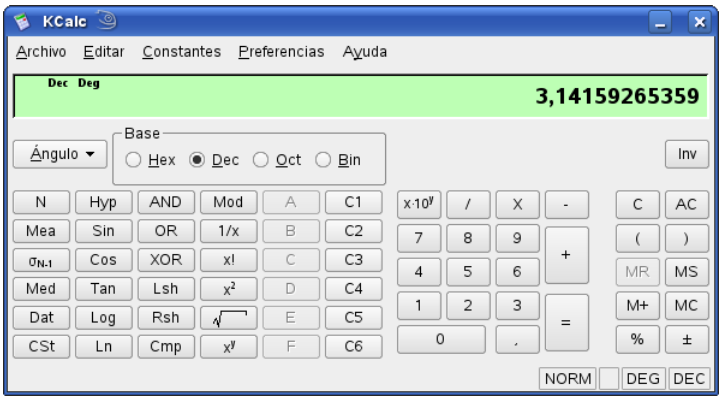
La calculadora del proyecto GNOME es gcalctool. Admite modos básico, avanzado financiero y científico (en 6.1 se puede ver este último).

6.3.1 KCalc

KCalc es la calculadora del proyecto KDE. En las figuras 6.2*a* y *b* se muestra su sencillo e intutivo aspecto.



a) Aspecto básico.



b) Aspecto avanzado.

Figura 6.2: Calculadora científica KCalc.

7

Representación gráfica de datos

7.1 GNUplot

Habitualmente empleamos la herramienta GNU Plot para representar gráficamente los resultados obtenidos del análisis empírico de los algoritmos. Esta herramienta es muy completa, y merece la pena estudiarla por sí misma; para más información, basta escribir la orden **help** tras ejecutar `gnuplot`; también podemos consultar la información disponible en el sistema GNU Info.

El siguiente fichero recoge una sesión de GNU Plot cuyo objetivo es mostrar las gráficas correspondientes a los datos almacenados en una serie de ficheros `.tmp`. Las órdenes que contiene pueden ser introducidas desde dentro del programa; no obstante, resulta muy cómodo y útil agruparlas en un *guión* y pasar éste como parámetro desde el exterior al ejecutar `gnuplot`. Para automatizar todo el proceso, solemos hacer esto dentro de un *makefile* que se encarga de compilar el programa, ejecutarlo para obtener los ficheros `.tmp` y obtener las gráficas.

```
1  # Título de cada eje.
2
3  set xlabel "n"
4  set ylabel "t(n) (tiempo en segundos)"
5
6  # Estilo de presentación (puntos interpolados linealmente).
7
8  set data style linespoints
9
10 # Representación gráfica.
```

```
11
12  set terminal x11 1
13  plot "prueba-1.tmp" title "Fibonacci recursivo trivial"
14
15  set terminal x11 2
16  plot "prueba-2.tmp" title "Fibonacci dinámico"
17
18  set terminal x11 3
19  plot "prueba-1.tmp" title "Fibonacci recursivo trivial",\
20       "prueba-2.tmp" title "Fibonacci dinámico"
21
22  # Pausa (hasta que se pulse [Intro]).
23
24  pause -1 "[Intro] para terminar\n"
```

Fibonacci/graficas.plot

El formato de los ficheros de datos es muy sencillo. Son ficheros de texto donde cada línea contiene la abscisa y la ordenada de un punto, como en el siguiente fragmento de ejemplo:

```
41      10.22
42      16.53
43      26.75
44      43.28
45      70.02
```

La opción **terminal** controla el dispositivo gráfico a través del que se generarán las gráficas. El dispositivo especial x11 representa los datos en una ventana gráfica. Como se observa, es posible crear varias ventanas gráficas numerándolas. También se puede representar más de una gráfica simultáneamente en cada ventana, lo que resulta muy útil a efectos de comparación.

Dado que puede ser interesante obtener gráficas en formatos apropiados para

su impresión e inclusión en otros documentos (como, éste), resulta útil preparar un guión específico.

```
# Codificación ISO Latin-1 y terminal EPS.

set encoding iso_8859_1
set terminal postscript eps solid

# Título de cada eje.

set xlabel "n"
set ylabel "t(n) (tiempo en segundos)"

# Estilo de presentación (puntos interpolados linealmente).

set data style linespoints

# Creación de los ficheros EPS.

set output "prueba-1.eps"
plot "prueba-1.tmp" title "Fibonacci recursivo trivial"

set output "prueba-2.eps"
plot "prueba-2.tmp" title "Fibonacci dinámico"

set output "pruebas-1-y-2.eps"
plot "prueba-1.tmp" title "Fibonacci recursivo trivial",\
    "prueba-2.tmp" title "Fibonacci dinámico"
```

Fibonacci/graficas–eps.plot

Este guión es muy parecido al anterior. Las líneas clave son las que permiten cambiar la codificación de los caracteres, el dispositivo a través del que se generan los gráficos y el destino de la propia salida a la que se envían éstos.

Gracias a la opción encoding es posible obtener correctamente las representaciones gráficas correspondientes a los caracteres de códigos distintos del ASCII (ISO

646). Empleamos el código Latin 1 (ISO 8859-1).

En este caso, obtendremos código PostScript y, más concretamente, EPS. Esto se consigue cambiando la opción **terminal** que en nuestro sistema, por omisión, es x11. Por último, con la opción **output** se redirige la salida estándar al fichero indicado. Por omisión, los datos se envían a la salida estándar (salvo en el caso de dispositivos especiales como x11).

7.2 Obtención de otros formatos gráficos

Para obtener las gráficas en formato PNG basta cambiar la opción **terminal**:

set terminal png

No existe, por el momento, una opción PDF por lo que resulta necesario emplear una herramienta externa si deseamos convertir los ficheros EPS a PDF. Una herramienta de línea de órdenes muy apropiada para esta tarea es `epstopdf` que, como sólo puede trabajar con un fichero a la vez, puede emplearse en un bucle:

```
> for i in *.eps; do epstopdf "$i"; done
```

Este bucle es prácticamente equivalente a:

```
> ls *.eps | xargs -I '{}' epstopdf '{}'
```

o, de manera, más robusta:

```
> find *.eps -print0 | xargs -0 -I '{}' epstopdf '{}'
```

Por último, si disponemos de una gran cantidad de ficheros EPS posiblemente no sólo en el directorio actual, sino también en subdirectorios, podemos realizar una búsqueda recursiva convirtiéndolos conforme los vayamos encontrando:

```
> find -name '*.eps' -type f -exec epstopdf '{}' \;
```

Existen herramientas capaces de convertir SVG a distintos formatos. Una herramienta sencilla de línea de órdenes es `rsvg-convert`. Por ejemplo:

```
> rsvg-convert -w 32 -a grafico.svg -o grafico.png
```

Las opciones `-w` y `-h` controlan respectivamente la anchura y altura en píxeles, respectivamente. La opción `-a` indica que se mantenga la relación de aspecto. La opción `-f` controla el formato (al menos, permite la conversión a PNG, PDF y PS) que, por omisión, es PNG.

7.3 Otras herramientas

Además de GNUplot podemos destacar otras dos herramientas:

kmplot KmPlot dibuja funciones matemáticas. Forma parte del proyecto KDE-EDU (perteneciente a KDE). Permite exportar las gráficas a BMP, PNG y SVG.

kpl es otro programa perteneciente al proyecto KDE que permite la representación gráfica de funciones y conjuntos de datos en dos y tres dimensiones.

8

Trabajo colaborativo

8.1 Introducción

Hoy en día no entenderíamos la informática sin las redes de ordenadores. Estas redes permiten, entre otras cosas, acceder a información remota, sincronizar sistemas y, como veremos en este capítulo, trabajar de manera colaborativa dentro de un entorno distribuido.

Cuando un equipo de trabajo tiene que desarrollar un trabajo a realizar con ordenadores de manera colaborativa (es decir, aportando cada miembro una parte al proyecto) hay que organizar el flujo de trabajo. En casos sencillos, como trabajos con fases secuenciales bien delimitadas o aquellos en los se pueden definir interfaces muy claras entre los trabajos de cada integrante del equipo, no es necesario un soporte para la colaboración.

Sin embargo, la mayoría de trabajos en equipo (desarrollar un sistema informático, realizar un documento entre varios autores, etc) se benefician enormemente de tener un sistema de apoyo al trabajo colaborativo que permita integrar las aportaciones de cada uno de los miembros fácilmente.

8.2 Sistemas de control de versiones

Se entiende como *sistemas de control de versiones* sistema de control de versiones aquel software que nos facilita la evolución de un sistema que se desarrolla (normalmente) entre varios autores, guardando versiones intermedias y registros de todas las actividades relacionadas.

En el mundo del software libre se han usado estos sistemas desde hace muchos años, pues la naturaleza del desarrollo del software libre es distribuida. Los

sistemas de control de versiones se pueden clasificar en dos tipos principales: centralizados y distribuidos

En el primer grupo, los centralizados, existe un servidor que aglutina toda la información que maneja el sistema. Tiene la ventaja de que es fácil de administrar y muy fácil de usar. Sin embargo tiene el problema de que el servidor es un punto débil (en caso de caída o si un usuario no puede conectar con él se complica la colaboración). Los sistemas de control de versiones libres más utilizados son *Control Version System* (CVS) y su sucesor *Subversion* (SVN).

Por otro lado están los sistemas distribuidos. En ellos no existe un único repositorio central de la información, sino que este se encuentra distribuido dinámicamente en los diferentes equipos que trabajan con él. Tiene la ventaja de que es muy potente y flexible, permitiendo flujos de trabajos que se adapten perfectamente a todo tipo de situaciones. Por el contrario su administración y uso se complica bastante. Dentro de ellos destacamos *Git*, *Bazaar*, *Mercurial* o *SVK* (todos ellos libres).

En este documento explicaremos el funcionamiento básico de SVN. Este sistema está entre los más usados por varias razones, aunque probablemente la principal sea que apareció como el sucesor natural de CVS, superándolo en algunos aspectos pero manteniendo su facilidad de uso y permitiendo una adaptación rápida si se conoce este último. Entre los proyectos libres que lo usan podemos destacar Apache Software Foundation, KDE, GNOME, FreeBSD, GCC, Python, Django, Ruby, Mono o SourceForge.net.

SVN fue desarrollado en 2000 por CollabNet, empresa que liberó la herramienta y su documentación (incluido el magnífico libro libre "Version Control with Subversion" COLLINS-SUSSMAN ET AL. [2008]). Como sistema centralizado, SVN es mucho más sencillo tanto de usar como cliente como de administrar, por lo que muchas forjas de software lo ofrecen como sistema de control de versiones para todos sus proyectos. Por último, otro aspecto a destacar es que están apareciendo herramientas relacionadas que le dan más potencia como interfaces gráficas para clientes, sistemas webs para consulta de estado, etc.

8.3 Trabajo colaborativo con Subversion

Antes de comenzar a explicar una sesión de trabajo con SVN vamos a realizar algunas aclaraciones y explicar algunos conceptos necesarios.

8.3.1 Conceptos en trabajo colaborativo centralizado

Cuando usamos un sistema de trabajo colaborativo centralizado como SVN tenemos que tener siempre en cuenta que existe una única "última versión" centralizada de los datos accesible por los usuarios. Esa versión es la que mantiene el servidor.

Pero, evidentemente, no es cómodo desarrollar software sobre un servidor. Es más, puede ser que el software que desarrolle sea incompatible con el servidor por las versiones de bibliotecas que se usen, por su sistema operativo o, incluso su arquitectura. Así que el primer paso para trabajar cómodamente será descargar la última versión en un equipo concreto. Para descargar la "última versión" de un proyecto con SVN necesito saber su URL y, si necesita autenticación, un par usuario/clave. La orden para descargar es tan sencilla como:

```
svn checkout <URL> --username <USR>
```

Esto nos pedirá la clave para el usuario indicado (si hemos indicado la opción larga *username*) y descargará el contenido al que apunte la URL en el directorio de trabajo local. Ese contenido descargado se denomina *copia local*.

Si el uso que hacemos de dicha versión no implica modificación alguna del código fuente no hay problema. Podría ser el caso de probar si se le ha añadido alguna funcionalidad concreta al sistema o ver si el programa pasa una serie de casos de prueba (para certificarlo por ejemplo). En ese caso puedo usar la copia que tengo de la "última versión" y, cuando termine de usarla, eliminarla.

Ahora bien, el modelo se complica si lo que quiero es modificar el código fuente para añadir funcionalidades, corregir errores, añadir documentación, reorganizar determinadas estructuras, etc. En los siguientes apartados veremos algunas situaciones que se pueden dar al trabajar con SVN.

8.3.2 Flujo de trabajo simple

En caso de que tengamos una copia local y estemos trabajando con ella tendremos que actualizarla periódicamente la copia central con el contenido local. Para ello simplemente será necesario usar la siguiente orden:

```
svn commit -m ''<Mensaje>''
```

En importante que pongamos en cada aportación un mensaje descriptivo. De este modo, al ver el historial de cambios podremos seguir la evolución del sistema más fácilmente. Además, si necesitamos buscar versiones anteriores del sistema podremos identificarlas de manera sencilla.

Un aspecto importante es que si estamos en el directorio raíz descargado el envío se efectuará por defecto sobre todos los cambios en el árbol. Pero si estamos en un subdirectorio sólo afectará a la porción del árbol que salga de él.

En caso de que todo haya ido correctamente nos irán apareciendo en pantalla los nombres de los ficheros que se van enviando al servidor y finalmente nos dirá el número de revisión correspondiente a nuestra aportación. Este número es un entero creciente a partir de uno que identifica de manera inequívoca cada aportación.

También puede darse el caso de que deseemos añadir o eliminar ficheros. Para ello basta con usar los comandos *add* o *delete* de SVN. Por ejemplo, con las siguientes órdenes añadimos al control de SVN todos los ficheros con extensión *tex* y *directorio1*, y eliminamos de él los ficheros cuyo nombre empiece por *temp*.

```
svn add *.tex directorio1
svn delete temp*
```

El contenido del repositorio dependerá del proyecto en concreto, así como la decisión de qué ficheros controlar y cuales no (por ejemplo, los ficheros de respaldo creados por editores de texto no suelen incluirse). Pero lo normal es que se use al menos el primer nivel de directorios para separar versiones, creándose normalmente una rama *trunk* para el desarrollo y una *stable* para congelar versiones.

La creación inicial del repositorio dependerá del servidor SVN que usemos en concreto y de la configuración del servidor. Lo normal es que el administrador del sistema tenga establecido algún protocolo concreto para ello.

El modo de trabajo de SVN para una única persona que hemos visto proporciona muchas ventajas sobre el trabajo local:

- Se genera una copia de seguridad en el servidor siempre que se haga un envío.
- Cada envío queda registrado, lo que permite el seguimiento del desarrollo del sistema.
- En caso de que necesitemos en cualquier momento revertir algunos cambios hecho o simplemente recuperar una versión de un fichero en una fecha dada lo podemos hacer fácilmente (ya veremos cómo más adelante).

- Si alguien quiere unirse al desarrollo del proyecto tenemos ya la infraestructura creada).

8.3.3 Flujo de trabajo colaborativo con conflictos

Una vez somos capaces de trabajar solos con SVN pasamos a explotar su principal ventaja: el trabajo colaborativo. Para ello usaremos varios ejemplos de flujo de trabajo entre dos usuarios, *Tiopepe* y *Tiomateo*.

En este primer ejemplo no se producen conflictos:

1. Tiopepe y Tiomateo bajan la última versión disponible en el servidor.
2. Tiopepe hace algunos cambios en un fichero *fich1*.
3. Tiomateo hace cambios en otro fichero *fich2*.
4. Tiopepe envía sus cambios al servidor.
5. Tiomateo envía sus cambios al servidor.

En el momento en que Tiopepe envía sus cambios al servidor no sabe que Tiomateo tiene una copia sobre la que está trabajando, así que no recibe ningún aviso en ningún momento. Igualmente, cuando Tiomateo mando subir sus cambios al servidor no recibe ninguna notificación (pero verá que entre el número de la revisión que bajó y la que ha subido hay otra intermedia).

Por lo tanto en este caso tanto ningún usuario ha tenido problemas al hacer sus aportaciones. Pero si nos fijamos bien ambos tienen una copia desactualizada de la "última versión". Para comprobar el estado de su copia local los usuarios pueden invocar:

```
svn status
U Makefile
A README
```

Esta orden devuelve un listado con un fichero de la copia local por línea. Cada fichero va antecedido de su estado, que puede ser:

A fichero añadido.

D fichero marcado para borrar en la siguiente versión.

M fichero modificado.

? fichero que no se encuentra bajo el control de SVN.

C fichero con cambios locales que crean conflicto.

Si un usuario quiere actualizar su copia local sólo tiene que escribir:

```
svn update
```

Aparecerá una línea por cada fichero modificado antecedido de su estado, que puede ser:

A fichero añadido.

D fichero borrado.

U fichero modificado.

G fichero con cambios locales con conflictivos.

C fichero con cambios locales que crean conflicto.

En el caso de ficheros actualizados, añadidos o borrados la actualización es sencilla: se cambia el fichero por la versión más reciente, se crea con su contenido o se elimina respectivamente. Ahora bien, ¿qué significa que un fichero tenga cambios no conflictivos? Esto se da, por ejemplo, con el siguiente flujo de trabajo:

1. Tiopepe y Tiomateo bajan la última versión disponible en el servidor.
2. Tiopepe hace algunos cambios en la parte final del fichero *fich1*.
3. Tiomateo hace cambios en la parte inicial del fichero *fich1*.
4. Tiopepe envía sus cambios al servidor.
5. Tiomateo envía sus cambios al servidor.

En este caso SVN vería ambos cambios y generaría un error indicando que es necesario actualizar el fichero *fich1*. El usuario tiomateo debería ejecutar por lo tanto:

```
svn update
G fich1
```

Así SVN vería que los cambios realizados localmente y los de la "última versión" del servidor son compatibles y los fundiría en la versión local, de modo que ahora el usuario puede enviar los cambios al servidor finalmente.

Para evaluar la compatibilidad de los dos conjuntos de cambios y realizar ese fundido, SVN utiliza un algoritmo que intenta recopilar los cambios realizados por uno y otro lado y ver si el fichero admite los dos según una serie de heurísticas. Pero claro, puede ser que dicha heurística falle o, simplemente, los cambios afecten a un mismo elemento de un mismo fichero.

En ese caso, al realizar una actualización de la copia local, el sistema nos indicará que hay un conflicto y por cada fichero *fich1* que presente conflictos creará tres ficheros nuevos:

fich1.mine versión que se ha intentado enviar al repositorio.

fich1.rX donde *X* es la versión que se descargó del servidor y sobre la que se ha trabajado.

fich1.rY donde *Y* es la última versión en el servidor que entra en conflicto con el fichero local.

Ahora es el usuario quien debe decidir cómo resolver el conflicto antes de volver a realizar un envío. Una vez hayan desaparecido los tres ficheros SVN permitirá subir el contenido de *fich1*. Las formas más comunes de solucionar el conflicto son:

- Editando manualmente el fichero *fich1* (probablemente tras consultar los otros), borrando el resto y ejecutando *svn resolved fich1*.
- Sobrescribiendo el archivo con uno de los nuevos creados y ejecutando *svn resolved fich1*.
- Usando *svn revert fich1* para volver el fichero en cuestión al estado de la copia local que se realizó inicialmente (revirtiendo los cambios realizados).

Por último, cabe decir que existen normalmente abreviaturas para los comentado más usados de SVN (por ejemplo *checkout* es *co*), así como comandos más refinados (por ejemplo *svn diff*, que muestra diferencias entre ficheros al estilo *diff*). Para más información consulte el manual.

8.4 Herramientas auxiliares

Lo comentado anteriormente es el uso de SVN desde una terminal. Existen inteerfaces gráficas para diversos sistemas operativos, así como programas auxiliares para el servidor que facilitan el uso del sistema.

En el primer grupo podemos destacar *KDESVN* del proyecto KDE, *TortoiseSVN* para Windows, *Syncro SVN* (multiplataforma al estar realizado en Java) o el *plug-in* de Eclipse *Subclipse*.

En el segundo creemos que la mejor alternativa es instalar, cuando sea posible un sistema que permita ver gráficamente las diferencias entre dos versiones de un mismo fichero, como *Trac*, *Redmine* (este último puede verse en la captura 8.1).

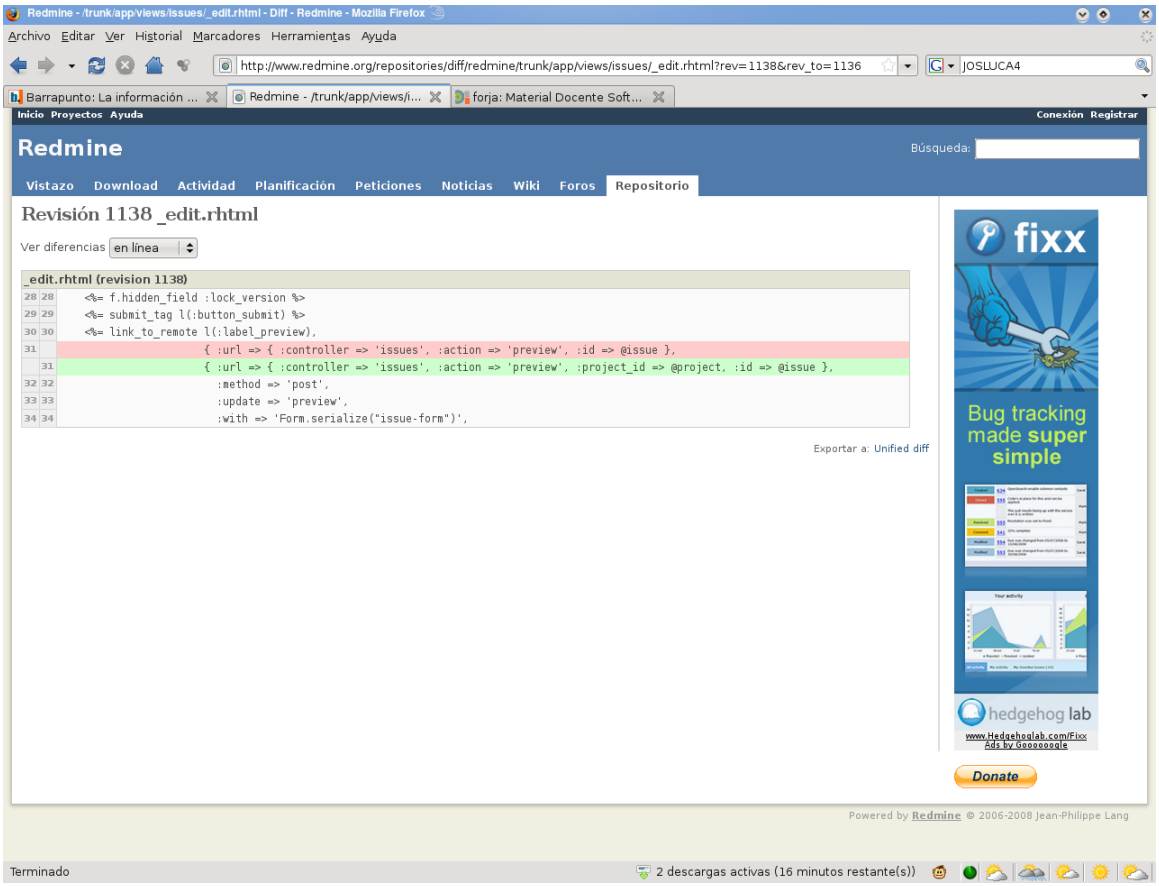


Figura 8.1: Captura de Redmine corriendo en redmine.org

9 Generación de documentación

Cuando se desarrolla un proyecto la generación de documentación relacionada con él una de las labores más importantes si deseamos que el proyecto tenga una implantación considerable.

Entre la documentación que debemos realizar sobre un proyecto se encuentra:

9.1 Documentación de ingeniería

Como producto de ingeniería que es el sistema, debemos desarrollar todos los documentos que se indiquen en la metodología que usemos. Por supuesto, no es intención de este documento hacer un repaso exhaustivo de las diferentes metodologías existentes. Por ello simplemente comentaremos el soporte que ofrecen algunas herramientas libres para su implementación.

Existen herramientas libres para dar soporte a la realización de diagramas, como *Kivio*, *Inkscape*, *Dia*, *Umbrello*, *Graphviz* o *ArgoUML*. En la figura 9.3 se puede ver un diagrama UML realizado con *dia*.

También existen herramientas para la generación de documentación de calidad profesional, como Latex (con su interfaz gráfica *Lyx*) o *Docbook*.

9.2 Documentación técnica incorporada en el código fuente

Una posibilidad muy usada últimamente para facilitar la incorporación de nuevos desarrolladores a un proyecto es incorporar documentación en el código fuente del sistema. Para ello tenemos un excelente sistema libre, *Doxygen*.

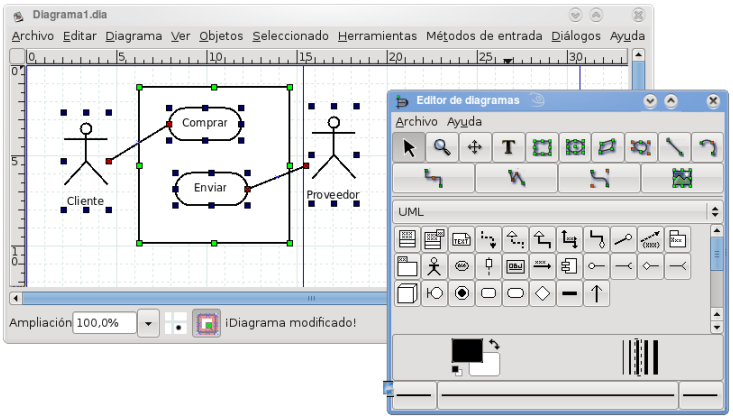


Figura 9.1: Realización de un diagrama UML con Dia

9.2.1 Doxygen

Doxygen es un sistema que genera automáticamente documentación a partir del código fuente de un sistema. Doxygen se puede integrar actualmente con una gran cantidad de lenguajes de programación, como C, C++, C#, Java, Python o PHP. Además es relativamente sencillo de extender para que soporte otros lenguajes. Su web oficial es VAN HEESCH [2009].

Existen dos modos de operación de Doxygen:

- Doxygen puede recorrer el árbol con el código fuente de un programa cualquiera y generar cierta información sobre él. Por supuesto, esa información será limitada y principalmente relativa a la organización del sistema en ficheros, clases, etc.
- En caso de que los programadores hayan incluido comentarios para Doxygen, éste los incorporará a la documentación que genera para darle mayor semántica.

La salida de Doxygen puede solicitarse en los siguientes formatos:

- HTML
- Latex
- RTF

- Postscript / PDF
- Unix man pages
- Windows help compressed HTML (CHM)
- XML

La configuración de Doxygen se almacena en un fichero de texto que puede crearse “a mano” o bien mediante algunas de las interfaces gráficas que existen para GNU/Linux, Mac o Windows.

Fichero de configuración

Para crear un fichero patrón de Doxygen podemos invocar la orden *doxygen -g*. En dicho fichero (llamado *Doxyfile*) veremos parejas de *ETIQUETA=VALOR*, algunas de las etiquetas generales más importantes son:

PROJECT_NAME = nombre del proyecto.

INPUT = nombre del fichero o directorio a documentar.

FILE_PATTERNS = patrones de ficheros a analizar (por ejemplo *.cpp).

Es importante revisar el fichero para encontrar opciones que nos puedan interesar, por ejemplo, para que un fichero PDF incluya hiperenlaces basta con activa la siguiente opción: *PDF_HYPERLINKS = YES*.

Comentarios para Doxygen

El caso más habitual es incluir comentarios Doxygen justo antes del elemento a comentar (una clase, una función, etc).

Para que Doxygen reconozca que tiene que procesar un comentario se pueden incluir estos de diferentes formas:

- `/**`
- `///`
- `/*!`

De este modo, al llegar a él, Doxygen elimina los asteriscos, interpreta las líneas en blanco como separadores de párrafos e interpreta los comandos que le indiquemos con el símbolo arroba (@). Por ejemplo, si incluimos **@brief Línea** usará dicha línea como descripción breve y el resto del comentario como descripción detallada.

Ejecución de Doxygen

Para ejecutar Doxygen basta con invocar *doxygen* en el directorio donde tengamos el fichero de configuración. El sistema generará la documentación automáticamente según las opciones activadas dentro el directorio de salida que se indicara en el fichero de configuración.

Por defecto Doxygen genera información en formato HTML dentro del directorio *html* y documentación en formato LaTeX dentro del directorio *latex*. Para acceder a la primera basta con abrir el fichero *index.html* con un navegador, mientras que para la segunda será necesario compilar los fuentes con la orden *make* y abrir el fichero PDF resultante con un lector de dicho formato.

En la figura 9.2 se puede ver una web generada automáticamente con Doxygen que recopila información sobre todos los miembros de todas las clases de la biblioteca *libgann*.

Generación de gráficos con Doxygen

Si queremos que Doyxgen, además de los enlaces entre clases, fichero, etc genere también información en formato gráfico (diagramas de clase, etc) tenemos que instalar el paquete *graphviz* y activar la opción *HAVE_DOT = YES*.

Después según las opciones que hayamos activado (CLASS_DIAGRAMS, GRAPHICAL_HIERARCHY, COLLABORATION_GRAPH, etc.) Doxygen incorporará gráficos generados automáticamente a su salida. En la figura 9.3 se puede observar información gráfica generada por Doxygen para una clase de la biblioteca *libgann*.

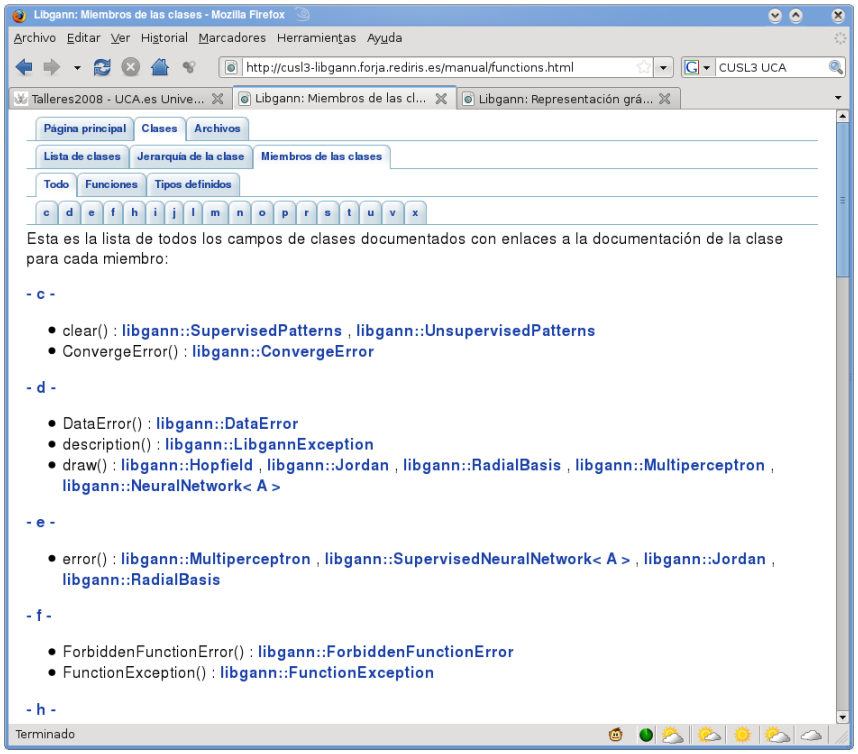


Figura 9.2: Navegación por la información generada por Doxygen

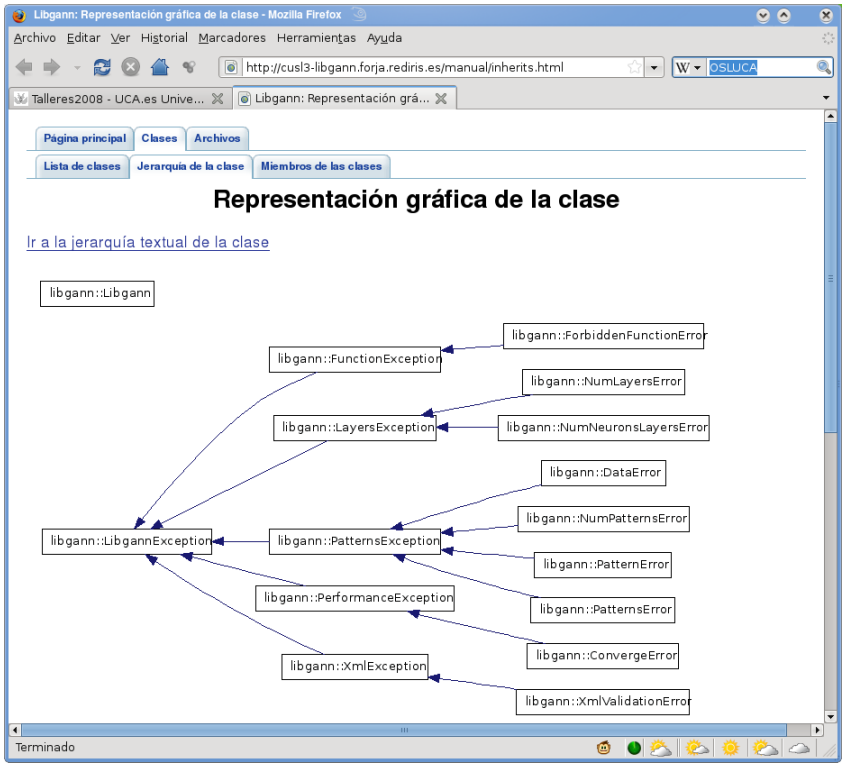


Figura 9.3: Información sobre una clase generada por Doxygen

10 Visibilidad del trabajo

Uno de los objetivos de liberar un software es maximizar su difusión. Esto permitirá aumentar la comunidad de desarrolladores y usuarios, beneficiándonos de las aportaciones que estos hagan (corrección de errores, creación de documentación, etc). Por ello es importante, en esta época de globalización, hacer un uso adecuado de las herramientas disponibles.

10.1 Forja

El recurso principal es la forja. Una forja es un sistema web que permite alojar proyectos de desarrollo de software y ofrece acceso a algunas herramientas adicionales para su gestión desde la misma web.

Las herramientas que podemos encontrar comúnmente en las forjas son:

- Sistemas de control de versiones.
- Listas de correo.
- Foros.
- Servicios de portal web: publicación de archivos para descarga, búsquedas, noticias, sindicación RSS, etc.
- Sistemas de gestión de fallos tipo *bugzilla*.

Otros incluyen herramientas adicionales como:

- Wikis.

- Sistemas para planificación de proyecto y asignación de tareas.
- Navegación web por el árbol de ficheros del código fuente. En algunos casos también comparación entre diferentes versiones de un mismo fichero con resaltado de diferencias.
- Notificaciones de actividad vía email
- Interfaz con grupos de noticias o IRC.

Algunas de las forjas más importantes a nivel mundial son:

- SourceForge.net
- BerliOS
- GNU Savannah
- Google code
- Gna.org

Dentro del mundo hispano está tomando fuerza la Forja de RedIris (Red nacional española de educación e investigación). Se puede ver una captura en la figura 10.1.

10.2 Listas de correo

Las listas de correo son el medio de comunicación preferido en las gran mayoría de proyectos libres de gran entidad. Tienen las siguientes ventajas, la mayoría derivadas del hecho de que están basadas en el envío de mensajes de correo electrónico:

- Necesitan pocos recursos para su funcionamiento, a diferencia de otros servicios que corren en web. Esto no quita para que haya sistemas web para la interacción con listas de correo que se puedan instalar opcionalmente.
- Necesitan pocos recursos para usarse, tanto en el equipo (basta con que tenga un gestor de correo) como en la conexión a la red (pues los mensajes son texto sencillo).

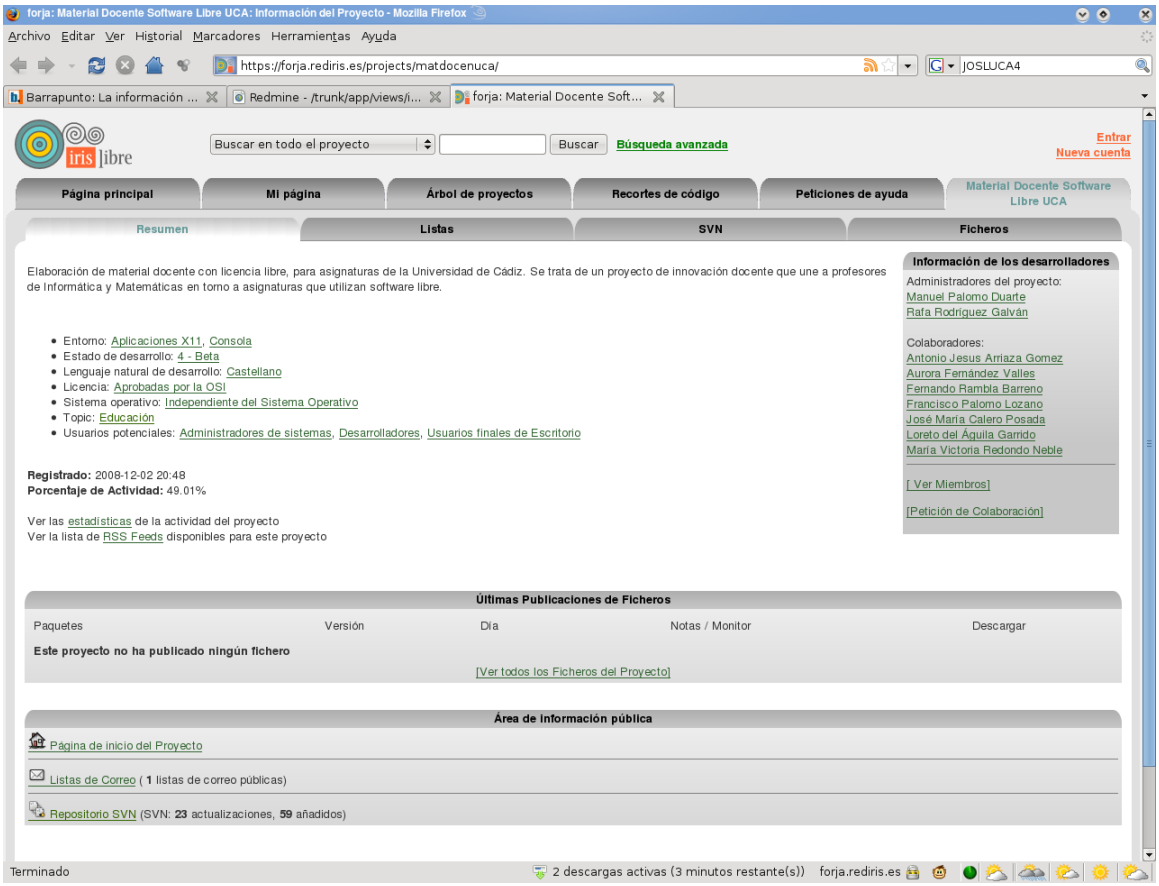


Figura 10.1: Captura de la interfaz web de la Forja de REDIris

- Permiten un alto grado de personalización en su uso: cada usuario puede usar el gestor de correo que desee, con sus opciones, clasificación por emisor, tema, etiquetas en el asunto, etc.
- Ocupan poco tamaño en disco por mensaje.
- Facilidad para trabajo *off-line*: se pueden descargar los mensajes a un portátil, leerlos y contestarlos sin conexión y cuando vuelva a disponerse de conexión, enviarlos.
- Todos los correos quedan registrados en orden de llegada, lo que facilita su seguimiento.

Es importante remarcar que existen unas normas muy estrictas a la hora de participar en determinadas listas de correo. Por lo general todas las listas permiten la lectura de los mensajes por parte de cualquier usuario. Es más, casi todas las listas suelen tener un histórico de mensajes disponibles desde web.

Sin embargo, para enviar mensajes, muchas listas requieren una aprobación previa. E, independientemente de que requieran o no aprobación, en la práctica totalidad de listas existen normas propias, y los usuarios suelen ignorar si no condenar mensajes de usuarios noveles que no las respeten.

Como curiosidad, existen determinados estudios que usan la actividad en las listas de correos de un proyecto como medida de su actividad complementaria a los envíos de contribuciones de código, y existen herramientas que lo soportan como Israel Herráiz y Barahona [2004].

11 Planificación

En la informática, como en cualquier otro campo, tener un plan de trabajo es un apoyo importante a la hora de desarrollar un proyecto. Por supuesto, dicho plan normalmente sufrirá modificaciones. Cada una de estas modificaciones suele implicar un reajuste, un punto de reflexión sobre cómo se está realizando la construcción de nuestro sistema.

En este capítulo usaremos el programa libre *Planner* para ilustrar los ejemplo. Hay muchos otros sistemas libres de similares características: *GanttProject*, *TaskJuggler* o, quizás el más completo *eGroupWare* (que funciona con una interfaz web).

11.1 Fases del proyecto

En la mayoría de proyectos de ingeniería se pueden usar medidas relativamente objetivas para seguir la evolución de un proyecto: metros de altura al hacer una presa, kilómetros de carretera, etc.

Sin embargo en los proyectos de desarrollo de software esto no suele ser tarea sencilla. La principales aproximaciones usan como medidas el número de líneas de código, pero no es una medida consistente: una línea de código correcta y no es igual que una incorrecta. Esta segunda hará que aparezcan errores en el programa, y que obligará a perder tiempo en detectarla y cambiarla por otra correcta. Además desconocemos cuántas líneas de código harán falta para completar el sistema y, por supuesto desconocemos su dificultad.

Es por ello que la mayoría de herramientas que existen para planificación y seguimiento de proyectos no ofrecen más que la posibilidad de dividir un proyecto informático en fases divididas en subfases y hacerlas dependientes unas de otras.

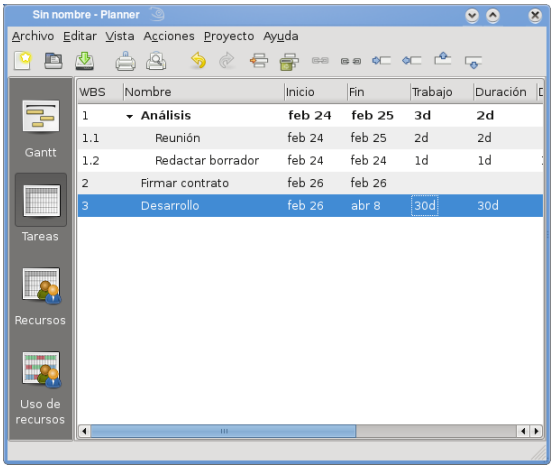


Figura 11.1: Especificación de tareas en Planner.

Normalmente una fase se puede dividir en varias subfases. Mientras no se indique lo contrario, las subfases comenzarán todas a la vez lo antes posible y la fase superior terminará al terminar la última de ellas. A su vez estas pueden subdividirse las veces que se deseen hasta que se establezcan tareas con una duración concreta. Las dependencias entre fases pueden ser de cuatro tipos:

De fin a principio es la más normal: la fase B no comienza hasta que no termine la A. En informática podría servir para modelar “La codificación no comienza hasta que no termine el diseño del sistema”.

De fin a fin se suele usar en situaciones en la que una actividad depende de la terminación de otra para terminarse, pero no para empezarse. Por ejemplo, puedo empezar a programar los controles de una interfaz gráfica antes de que esta esté lista, pero hasta que no esté puesto el último control no podrá terminar de programarlos.

De principio a principio hace que dos actividades comiencen a la vez. En informática es raro que se den, pues suele implicar una dependencia entre ellas a lo largo de todo su desarrollo. Un ejemplo del sector servicios sería “Si un horno hace palmeras de chocolate, el comienzo de la labor de poner chocolate a las palmeras depende del comienzo de la elaboración de las palmeras (en concreto de que salga la primera”.

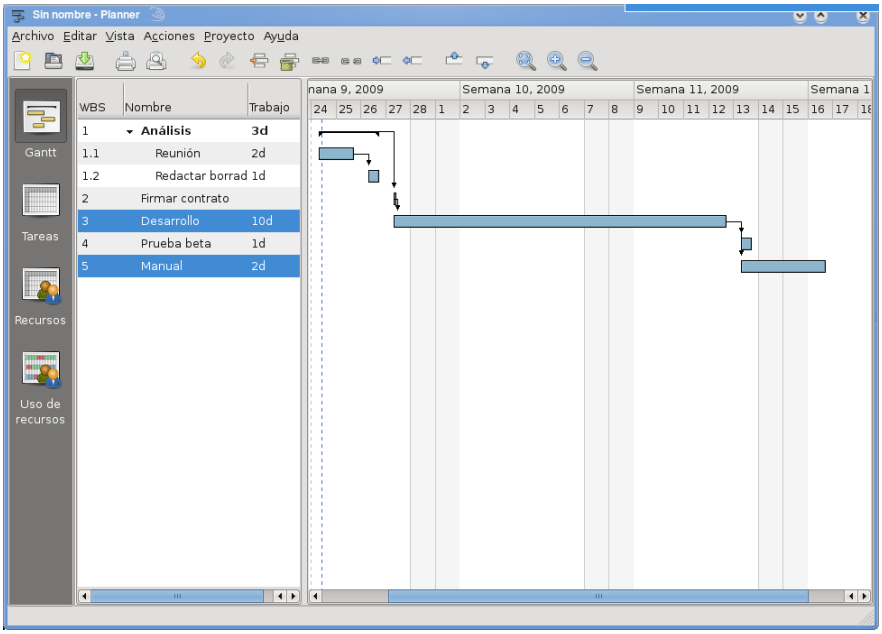


Figura 11.2: Diagrama de Gantt.

De principio a fin es una situación extraña por lo general. Un ejemplo (no informático) sería si quisiéramos pintar una pared. Tenemos que mover la mezcla antes de empezar a pintar. Pero ¿cuándo paramos de moverla?. Está claro que cuando empecemos a pintar (si no se pondría dura).

Para ver las relaciones mejor se puede usar un diagrama de Gantt como el de la figura 11.2.

A estas relaciones se pueden desplazar en ambos sentidos. Por ejemplo una actividad puede empezar un día después o antes de que acabe otra (siendo una relación de fin a principio).

Las tareas por lo general tiene una duración fija (establecida en horas o jornadas/días de ocho horas). Sin embargo hay excepciones: las tareas continuas y los hitos. Las tareas continuas son tareas que no se paran al acabar una jornada. Por ejemplo, si un barco está cruzando el atlántico lo cruza de manera constante, no a jornadas de ocho horas en días laborales. Por otro lado los hitos son tareas de duración cero que sirven para marcar un punto en el proyecto, por ejemplo, la

entrega de la primera versión al cliente o la recepción del pago que hace de señal para comenzar una fase. Está claro que en hacer ambas cosas se tarda un tiempo insignificante, pero tiene una importancia para la organización del proyecto.

11.2 Esfuerzo

Es importante remarcar la diferencia entre duración y esfuerzo. El duración es el tiempo que se tarda en hacer algo, midiendo el día y hora en la que se acabó menos el día y hora de comienzo. Ahora bien, no se tarda lo mismo en hacer algo sólo que con un ayudante. Por lo tanto, si ponemos dos personas a hacer una tarea los sistemas de planificación de proyectos dividirán por dos la duración pero mantendrán el esfuerzo constante (y por consiguiente el coste). Igualmente, si asigno una persona a media jornada tardará el doble aunque me seguirá costando lo mismo.

Por supuesto hay excepciones a esta regla: a veces algunas tareas tardan un tiempo fijo haya más o menos gente implicada. Por ejemplo, en al hacer un desplazamiento en coche por más gente que meta no va a ir más rápido (es una tarea de duración fija). Incluso a veces es lo contrario: en las reuniones cuanta más gente haya más suelen tardar.

11.3 Recursos

Una vez establecida las fases (subdivididas en tareas) y el esfuerzo queda asignar los recursos. Por lo general, estos suelen ser de dos tipo:

Tipo esfuerzo (o trabajo) son recurso que tienen una capacidad limitada de trabajo y se pagan por hora. El ejemplo más claro son las personas: sólo pueden estar en un sitio a la vez, y cobran por hora de trabajo (quizás agrupadas en jornadas o meses). También pasa con las máquinas: sólo pueden estar en un sitio y consumen gasolina por unidad de tiempo de trabajo. Se asignan al proyecto por dedicación (tiempo completo, media jornada, etc) y hay un tope máximo. Por ejemplo, normalmente las personas suelen agruparse por capacidad y se dice “hay una capacidad de programación del 300 %"para decir que tenemos tres programadores a tiempo completo o dos a tiempo completo y dos a media jornada.

Tipo recurso (o material) son inagotables y tienen un coste por unidad. Por ejemplo: el metro de cable de red, una silla, etc. Se asignan por gasto que se hace de ellas y no establecen capacidades máximas no conflictos de uso.

11.4 Seguimiento del proyecto

Una de las principales ventajas de usar una herramienta de este tipo para gestionar un proyecto es que permite actualizar la planificación en un momento dado para actualizarla al estado real del proyecto.

Estas actualizaciones permiten ver si el proyecto van en fecha y presupuesto o no. Y, en caso de que vaya tarde a qué es debido y qué medidas se podrían tomar para paliarlo. Estas medidas suelen ser añadir más recursos, o acordar funcionalidades o calidades.

Bibliografía

ABURRUZAGA GARCÍA, GERARDO. *Make. un programa para controlar la recompilación*. 2008. http://www.uca.es/softwarelibre/publicaciones. 28

COLLINS-SUSSMAN, BEN; FITZPATRICK, BRIAN W. y PILATO, C. MICHAEL. *Version Control with Subversion*. O' Reilly, segunda edición. 2008. ISBN 978-0-596-51033-6. 46

ISO, INTERNATIONAL STANDARDS ORGANIZATION. *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*. International Standard ISO/IEC 10 646–1. International Organization for Standardization. 2000. 12

ISRAEL HERRÁIZ, GREGORIO ROBLES y BARAHONA, JESÚS M. GONZÁLEZ. *CVSAnalY: una herramienta libre para el análisis de repositorio*. En *Actas de las IVJornadas Andaluzas de Software Libre*, páginas 35–39. ADALA, Asociación para la Difusión y el Avance del Soofttware Libre en Andalucía, Algeciras, España. 2004. ISBN 84-88783-71-X. 64

UNICODE, THE UNICODE CONSORTIUM. *The Unicode Standard, version 5.0*. Addison-Wesley. 2006. 12

UNICODE, THE UNICODE CONSORTIUM. *Code charts for symbols and punctuation*. 2008*a*. http://www.unicode.org/charts/symbols.html. 12

UNICODE, THE UNICODE CONSORTIUM. *The unicode character code charts by script*. 2008*b*. http://www.unicode.org/charts. 12

VAN HEESCH, DIMITRI. *Doxygen home page*. 2009. http://www.doxygen.org. 56