Programación con Objective-C

Utilizando el entorno de desarrollo GNUstep
Segunda Edición

Germán A. Arias <germanandre@gmx.es>

Índice General

1 GNUstep

GNUstep es un entorno de desarrollo que originalmente nació con la intención de ofrecer una alternativa libre a las librerías del sistema operativo *NeXTstep*. En 1994 la empresa *NeXT* libera al publico las especificaciones *OpenStep*, abriendo con ello la posibilidad de crear una implementación libre de estas. Es entonces cuando **Paul Kunz** y otros en el *Stanford Linear Accelerator Center* deciden crear la librería *libobjcX*, software que sería la base de lo que actualmente se conoce como *GNUstep*. *GNU* debido al proyecto de la *Free Software Foundation* al cual pertenece *GNUstep*. Posteriormente la empresa *NeXT* sería comprada por *Apple*, la cual basaría su entorno de desarrollo *Cocoa* en las especificaciones *OpenStep*. De allí que *GNUstep* tenga compatibilidad con el sistema operativo *Mac OS X*.

GNUstep permite el desarrollo de aplicaciones para sistemas *GNU/Linux*, **BSD*, *GNU/Hurd*, *Solaris*, *Darwin*, *Mac OS X* y *Windows*. Asimismo es una *plataforma de desarrollo cruzada*. Lo que significa que utilizando el mismo código se pueden desarrollar aplicaciones que funcionen en diferentes sistemas operativos y arquitecturas.

Este libro pretende ser una introducción breve, pero sólida, al entorno de desarrollo de *GNUstep*. Tanto para novatos en la programación, como para aquellos que ya tengan experiencia. Se describen primero las especificaciones *OpenStep* para aquellos usuarios que quieran utilizar *GNUstep* con el estilo y tema por defecto. Se presentan asimismo los conceptos básicos del lenguaje C necesarios para aprender Objective-C. Así como los fundamentos de la programación orientada a objetos y del diseño de aplicaciones con *GNUstep*. Todo esto acompañado de varios ejemplos para que el lector vaya practicando a la par que avanza por los distintos temas. Al terminar de leer este libro, esperamos que el lector este capacitado para desarrollar aplicaciones con *GNUstep*, haciendo uso de las facilidades que este entorno proporciona y tomando en cuenta las consideraciones para portar estas a otras plataformas.

La mayoría de imágenes en este libro, muestran aplicaciones usando el tema y estilo que por defecto utiliza *GNUstep*. Esto no significa que el lector deba utilizar este mismo tema y estilo. Ya que los ejemplos en este libro, están pensados para que funcionen sin importar el estilo que decida utilizar el lector. Aun cuando se dejan para el último capítulo, las consideraciones a tomar en cuenta para que las aplicaciones soporten diferentes estilos.

La página oficial del proyecto *GNUstep*, en inglés, es:

`http://www.gnustep.org/`

Pero para los hispanohablantes es mejor consultar el siguiente enlace:

`http://gnustep.wordpress.com/`

Esta página contiene varios recursos, además de publicar regularmente noticias relacionadas a *GNUstep*.

GNUstep se pronuncia como se escribe, sin vocal entre la *g* y la *n*. Y su logo, creado por **Ayis Theseas Pyrros**, representa el *yin-yang* de la programación que nos permite ir unos pasos adelante. Aunque no hay una mascota oficial del proyecto, es común utilizar una orca en documentos y playeras utilizadas en eventos relacionados al proyecto.

Los frameworks de *GNUstep* se liberan bajo la licencia *LGPL*, lo que significa que se permite el desarrollo de programas propietarios. Véase:

`http://www.gnu.org/copyleft/lesser.html`

para mayor información sobre esta licencia.

2 Especificaciones OpenStep y configuración

En este capítulo se cubren algunos aspectos de la instalación de *GNUstep*. Se presenta también un resumen de las especificaciones *OpenStep*, las cuales es recomendable conocer independientemente del tema que se utilice y de la plataforma en que se utilice *GNUstep*. Por último, se presentan algunos aspectos importantes de la configuración de *GNUstep* que deben tenerse en cuenta antes de comenzar a desarrollar aplicaciones.

2.1 Instalando GNUstep

La instalación de *GNUstep* depende del sistema operativo y del compilador que se desee utilizar. Actualmente existen paquetes para una gran variedad de sistemas que facilitan la instalación, por lo que no cubriremos aquí este tema. En el siguiente enlace, en la sección de *Instalación de GNUstep*, puede consultarse una lista de paquetes disponibles. Y, por si se requiere, el procedimiento de instalación desde el código fuente.

`http://gnustep.wordpress.com/`

Para poder seguir los ejemplos presentados en este libro, deben instalarse también las aplicaciones *SystemPreferences* y *Gorm*. Debe instalarse también un editor de código, por ejemplo *Emacs* o *Vim*. O utilizarse uno más sencillo como *Gemas*, el cual esta construido con *GNUstep* y tiene soporte para archivos propios de este entorno.

Se necesitara también hacer uso de una *Terminal* o *Shell*. Nombres con los que comúnmente se conoce al *Interprete de comandos*. Casi todos los sistemas traen una *Terminal* instalada, que generalmente se encuentra en la sección de *Accesorios* del menú de *Aplicaciones*. Para el caso de *Windows*, el instalador se encarga de instalar este. Y se puede acceder a través del menú del sistema en Programas → GNUstep → Shell o mediante la búsqueda de aplicaciones con el nombre *Shell*. Para los usuarios que no tengan experiencia usando una *Terminal*, y especialmente para los usuarios de *Windows*, recomendamos leer el apéndice A. El cual contiene lo básico para manejar esta aplicación.

GNUstep ofrece una herramienta llamada *ProjectCenter*, la cual es un *entorno de desarrollo integrado* o *IDE* (*Integrated Development Environment*), que facilita la creación de programas. Sin embargo, en este libro, no se hará uso de dicha herramienta. Esto porque el *IDE* oculta parte del procedimiento para crear un programa, y creemos que un principiante en *GNUstep* debe tener una idea clara de dicho procedimiento. Sin embargo, una vez entendido este, el uso de *ProjectCenter* no ofrecerá ningún misterio.

2.2 Especificaciones OpenStep

La imagen 0-1 muestra el escritorio *WindowMaker* y la aplicación *Ink* (hecha con *GNUstep*) ejecutándose en este. El escritorio *WindowMaker* implementa las especificaciones *OpenStep*, y es por lo tanto el escritorio que mas se menciona cuando se habla de *GNUstep*. Sin embargo, como veremos en este capítulo, *GNUstep* puede utilizarse en otros escritorios.

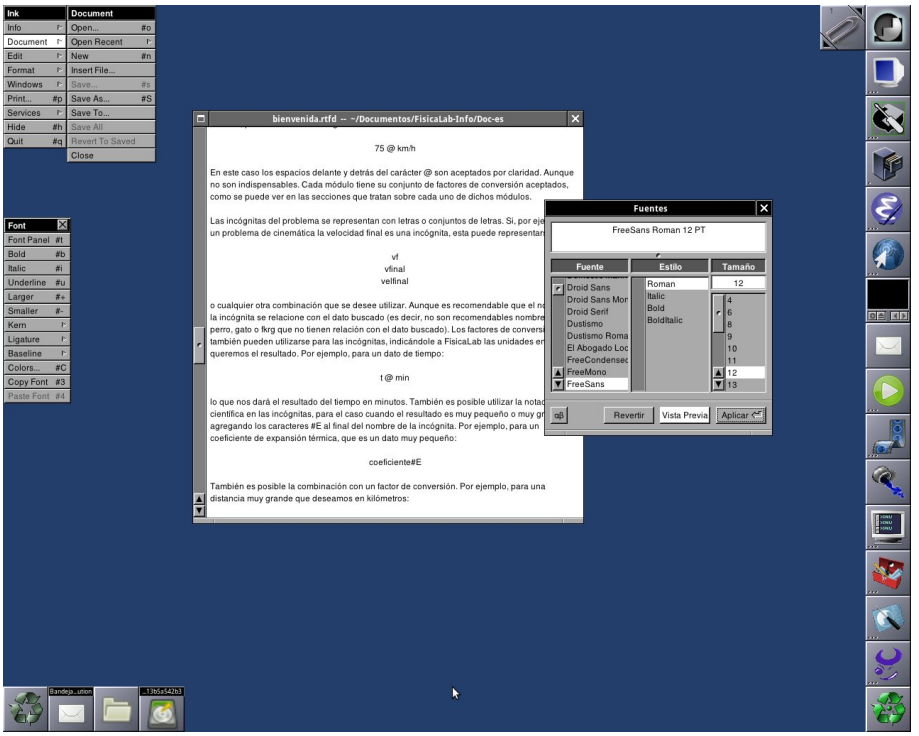

Imagen 0-1.

Las especificaciones *OpenStep* engloban una gran cantidad de características de un programa. En esta sección veremos parte de estas especificaciones, dejando otras para secciones posteriores. De acuerdo a estas especificaciones, todo programa debe desplegar un icono que lo identifica, llamado a veces *AppIcon*. El escritorio *WindowMaker* permite *anclar* estos iconos en el *Dock* (la columna vertical de iconos en la parte derecha de la imagen 0-1). Un icono de aplicación puede estar en uno de tres estados. En la imagen 0-2 se muestran tres iconos anclados en el *Dock*. El de arriba es el icono de una aplicación que se esta ejecutando. El de abajo es el de una aplicación que *no* se esta ejecutando, esto se indica con los puntos suspensivos de la parte inferior. Dando un doble clic sobre este icono la aplicación correspondiente se ejecutara. El icono de en medio es el de una aplicación que se encuentra oculta. Las aplicaciones hechas con *GNUstep* tienen en

el menú una opción llamada *Ocultar (Hide)*, la cual remueve del escritorio todas las ventanas desplegadas por la aplicación, excepto el icono de esta. Este estado se indica con un punto en la esquina inferior izquierda del icono. Dando un doble clic sobre el icono todas las ventanas de la aplicación se restauran a sus posiciones en el escritorio.

Imagen 0-2.

El menú vertical de la aplicación tiene en la parte superior una barra que muestra el nombre de la aplicación. Arrastrando la barra de título con un clic izquierdo del ratón es posible ubicar el menú en cualquier otro lugar del escritorio. Las opciones que tengan submenú, las que tienen una flecha a la derecha, despliegan el submenú en la parte derecha al dar clic sobre estas. El submenú queda desplegado aun después de haber seleccionado una opción en este, aunque puede ocultarse dando otro clic en la opción que lo desplegó. Como se ve en la imagen 0-3, los submenús se pueden desprender del menú principal arrastrando con el ratón la barra de título del respectivo submenú. Esto hará que la barra de título muestre un botón de cerrar en su extremo derecho. Esto permite organizar el menú de tal forma que podamos acceder rápidamente a las opciones mas utilizadas.

Si se da un clic en el menú principal para desplegar algún submenú que ha sido desprendido, este se desplegara en su forma *transitoria*. La forma *transitoria* requiere que el usuario mantenga el botón izquierdo presionado para poder navegar el submenú. Para seleccionar una opción el ratón se debe soltar sobre la opción deseada, o fuera del submenú si lo que se desea es cerrar este sin seleccionar ninguna opción. Al cerrar un submenú que ha sido desprendido, usando el botón de cerrar que aparece en su barra de título, este se desplegara nuevamente de forma normal en el menú principal. Sin embargo, siempre es posible utilizar la forma *transitoria* para desplegar un submenú, de tal forma que este se cierre después de seleccionar una opción. Esto, por supuesto, requiere dar un clic sobre la opción que abre el sumenú y, manteniendo el botón izquierdo presionado, seleccionar la opción deseada.

Las aplicaciones guardan la ubicación del menú principal y de los submenús desprendidos (si los hay), de tal forma que estos se despliegan en las mismas posiciones al volver a utilizarlas.

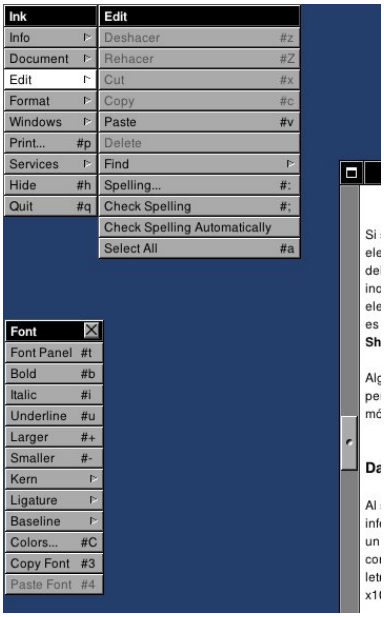

Imagen 0-3.

Otras especificaciones pueden verse en las ventanas, cuya barra de titulo tiene dos botones, uno en el extremo izquierdo y otro en el extremo derecho, imagen 0-4. El del extremo derecho, que muestra una cruz, sirve para cerrar la ventana. Mientras que el del extremo izquierdo, que muestra un pequeño cuadro, sirve para minimizar la ventana. Al dar un clic en este control se crea un icono que nos indica que la ventana ha sido minimizada (similar al *AppIcon*, pero llamado *Miniwindow*), y dando doble clic sobre este *Miniwindow* la ventana regresa a su posición. No hay un control para maximizar/restaurar la ventana, ya que esto es algo que no existe en las especificaciones *OpenStep*.

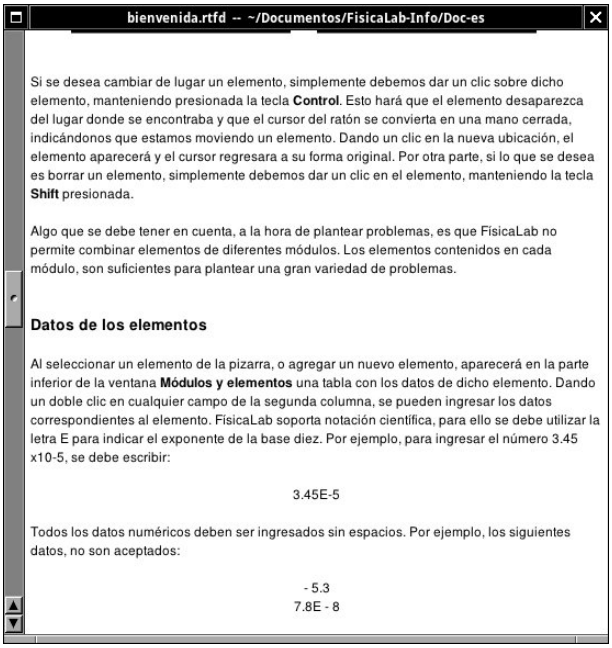

Imagen 0-4.

La imagen 0-5 muestra tres iconos de aplicación, a la izquierda, y una miniventana (*Miniwindow*) a la derecha. Cuando el icono de una aplicación no esta anclada al *Dock*, su icono se despliega en la parte inferior del escritorio. La miniventana representa una ventana minimizada y se distingue claramente por la barra de la parte superior que muestra el título de la ventana. Las aplicaciones no ancladas al *Dock* se pueden ejecutar desde el menú de *WindowMaker* que se despliega con un clic derecho sobre el fondo del escritorio. Este es un menú vertical acorde a las especificaciones *OpenStep*.

Imagen 0-5.

Para redimensionar la ventana, se utiliza la barra para dicho fin que se encuentra al pie de la misma y que esta dividida en tres partes, imagen 0-6. Arrastrando la parte central con el ratón, podemos modificar la altura de la

ventana. Mientras que las partes izquierda y derecha nos permiten modificar la altura y el ancho simultáneamente.

Imagen 0-6.

En la imagen 0-4 se ve también una barra de desplazamiento vertical. Arrastrando el *tirador* con el ratón puede recorrerse el documento mostrado. Si la tecla *Alt* se mantiene presionada mientras se arrastra el *tirador* el desplazamiento sera mas suave. Los botones con flechas en la parte inferior de la barra de desplazamiento también pueden utilizarse para desplazarse por el documento.

Por defecto, las aplicaciones de documentos usan una ventana por cada documento abierto. Los documentos que tienen cambios sin guardar muestran un botón cerrar diferente, en el cual sólo los extremos de la cruz son dibujados, imagen 0-7.

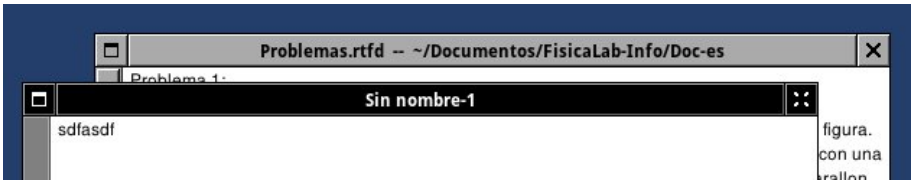

Imagen 0-7.

Dos de los componentes disponibles en *GNUstep*, *Pull-Down List* y *Pop-Up List* que se muestran en la imagen 0-8, se comportan como menús transitorios. Es decir, que requieren que el usuario mantenga el botón izquierdo del ratón presionado para poder navegar la lista. La opción deseada se selecciona soltando el ratón sobre ella. O fuera de la lista si no se desea seleccionar ninguna opción.

Imagen 0-8.

El color que por defecto tienen las aplicaciones hechas con *GNUstep*, es tan bien parte de las especificaciones *OpenStep*. Los tonos grises se eligieron con el objeto de no cansar la vista de los usuarios.

2.3 Configuración de GNUstep

Lo primero que se debe configurar, excepto para los usuarios de *Windows*, son las teclas modificadoras. No es estrictamente necesario establecer estas teclas, pero como desarrolladores nos serán muy útiles. Para ello ejecutamos la aplicación *SystemPreferences*, imagen 0-9. Y seleccionamos el icono *Modifier Keys*, imagen 0-9.

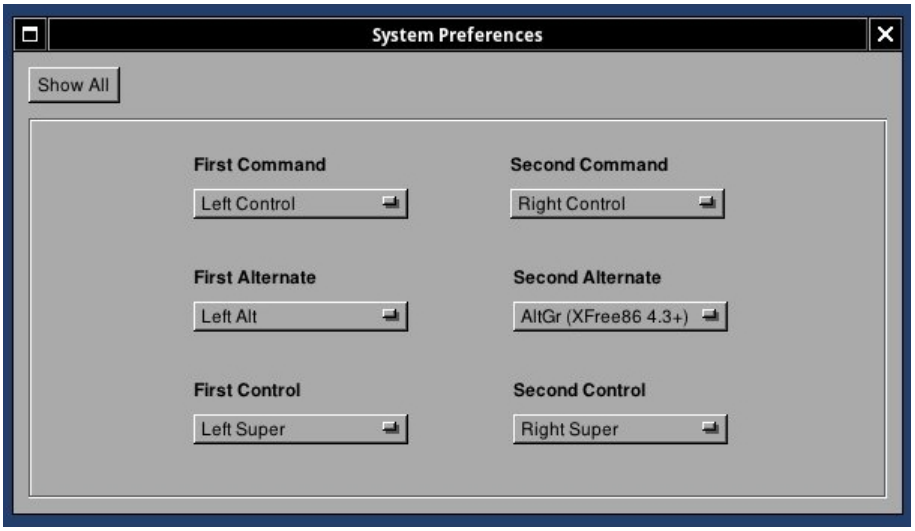

Imagen 0-9.

El primer par de teclas *First y Second Command* son las teclas que por defecto acceden a las opciones de los menús. Por ejemplo, en la imagen se ve que la opción *Quit* del menú tiene asignada la combinación de teclas *#q*, donde el carácter **#** representa la tecla *Command*. Para cerrar la aplicación, podemos entonces utilizar la combinación de teclas *Command* + *q*. El segundo par de teclas *First y Second Alternate*, y el tercero *First y Second Control* también pueden utilizarse para acceder al menú, aunque aquí nos serán útiles para realizar ciertas operaciones en la aplicación *Gorm* (que veremos mas adelante). Cuando *GNUstep* se utiliza en *Windows*, todas las teclas modificadoras se asignan a teclas como *Control*, *Alt* y *Shift*. Como se puede comprobar en los menús, donde no se observa el carácter **#**.

Cuando no se utiliza el escritorio *WindowMaker*, los iconos de aplicación y las miniventanas se irán acumulando uno sobre otro en la esquina inferior izquierda del escritorio. Para evitar esto, se puede instalar la herramienta *IconManager*, que permite ir colocando los iconos de una forma ordenada.

La imagen 0-10 muestra una aplicación *GNUstep* corriendo en el escritorio *Gnome*. Donde la posición y tamaõ de los iconos y miniventanas, ubicados en la parte inferior derecha, esta manejado por la herramienta *IconManager*.

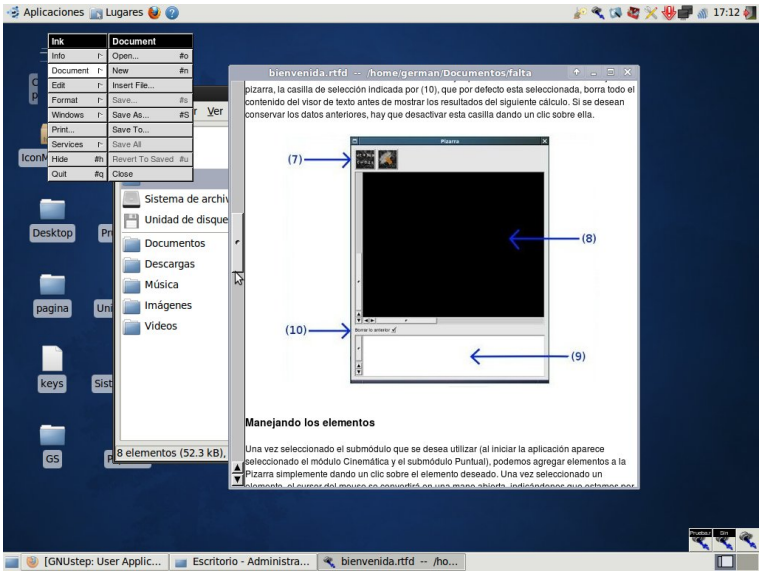

Imagen 0-10.

Otra opción es configurar *GNUstep* para no hacer uso de los iconos de aplicación y de las miniventanas (las ventanas se minimizan entonces en la barra de tareas). También es posible configurar el estilo del menú para que este empotrado en una ventana (siempre y cuando la aplicación que se vaya a utilizar este diseñada con soporte para este estilo). La imagen 0-11 muestra la aplicación *Gorm* ejecutándose en el escritorio *Gnome*, utilizando un menú empotrado en la ventana y el tema *Silver*.

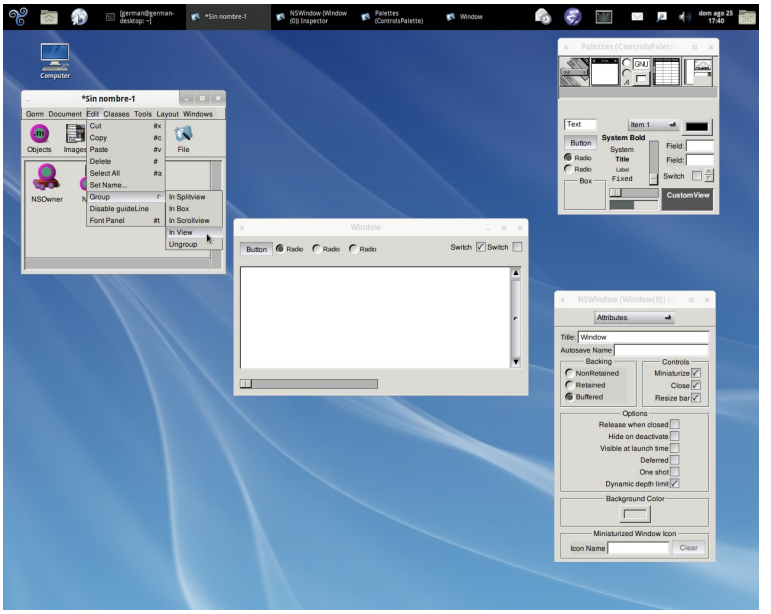

Imagen 0-11.

Esta configuración se puede realizar en la sección *Defaults* en la aplicación *SystemPreferences*. Las variables a configurar son:

- *GSAppOwnsMiniwindow* establecido a NO.
- *GSSuppressAppIcon* establecido a YES.
- *GSWorkspaceApplication* dejar vacío el campo.
- *NSMenuInterfaceStyle* establecido a *NSWindows95InterfaceStyle*.

Debe darse clic en el botón *Set* después de establecer el valor deseado para cada variable, de otra forma no se guardaran los valores en las preferencias.

La sección *Date & Time* de la aplicación *SystemPreferences* permite establecer la zona horaria. La sección *Fonts* permite establecer la fuente para diferentes componentes gráficos de *GNUstep*. Y la sección *Themes* permite establecer un tema diferente al establecido por defecto. El siguiente enlace contiene algunos temas para *GNUstep*:

`http://wiki.gnustep.org/index.php/Themes`

En el sistema operativo *Windows*, por defecto, las ventanas se minimizan en la barra de tareas y no se utilizan los iconos de aplicación. Y en el caso de que se active el tema *WinUXTheme*, que se puede activar en la instalación, la apariencia de las aplicaciones es como las nativas de *Windows*. Esto es, con el menú principal empotrado en una ventana (siempre y cuando las aplicaciones hayan sido diseñadas con soporte para este estilo).

El comportamiento de los componentes *Pull-Down List* y *Pop-Up List* (imagen 0-8), puede ser modificado por el tema utilizado. Este es el caso de los temas *Silver* y *WinUXTheme*, que no requieren que el usuario mantenga presionado el botón izquierdo para poder navegar las listas.

3 El lenguaje Objective-C

Antes que nada, ¿Que es un programa? Podríamos decir que un programa no es más que una serie de instrucciones que le indican a la computadora las acciones que queremos que realice. Evidentemente no nos podemos comunicar con la computadora como nos comunicamos con otra persona, para hacerlo necesitamos de un lenguaje especial, es decir de un lenguaje de programación. Como cualquier lenguaje humano, un lenguaje de programación tiene un conjunto de palabras aceptadas (llamadas palabras *reservadas*) y una sintaxis que nos dice como utilizar esas palabras.

Para comprender esto claramente, hagamos una analogía con el español. En el español, las palabras reservadas serian las que son aceptadas por la *Real Academia de la Lengua*, y la sintaxis es la forma correcta de utilizar esas palabras. Por ejemplo, sabemos que es correcto decir "Vamos a la montaña", pero es incorrecto decir "Montaña a la vamos". En ambos casos hicimos uso de las mismas palabras, pero solo una de las frases es correcta. De la misma forma, en un lenguaje de programación debemos conocer la forma correcta de utilizar las palabras, de otra manera la computadora no comprenderá nuestras instrucciones.

Así como existen diferentes lenguajes humanos (español, francés, inglés, alemán, etc.), también existen diferentes lenguajes de programación. *GNUstep* hace uso del lenguaje de programación Objetive-C. El cual es necesario conocer antes de intentar programar (lo que equivaldría a poder *hablar* en Objective-C). Objective-C esta basado en el lenguaje de programación C. Sólo que a diferencia de este, Objective-C ofrece la posibilidad de utilizar el paradigma de *programación orientada a objetos*, algo que veremos más adelante.

3.1 Palabras reservadas en Objective-C

En la tabla 1-1 se presentan las palabras *reservadas* del lenguaje Objective-C. Como se aprecia inmediatamente, a diferencia de un lenguaje humano, las palabras reservadas en Objective-C son pocas. Sin embargo, son suficientes para crear complejos programas. Hay que mencionar que aparte de las palabras reservadas, existen ciertas palabras para los *tipos de datos* y las *funciones*, las cuales veremos mas adelante.

auto	else	import	struct
break	enum	int	switch
case	export	long	typedef
char	extern	register	union

const	float	return	unsigned
continue	for	short	void
default	goto	signed	volatile
do	if	sizeof	while
double	include	static	

Tabla 1-1. Palabras reservadas en Objective-C.

Las palabras reservadas deben escribirse como aparecen en la tabla, es decir, con minúsculas. Si, por ejemplo, en lugar de *if* escribimos *IF*, la computadora no sera capaz de entender la palabra. Esto es algo que se debe tener en cuenta a la hora de escribir programas con Objective-C.

Como vemos, estas palabras están tomadas del inglés. Sin embargo, no podemos esperar que la computadora entienda inglés, o cualquier otro idioma humano. Los lenguajes de programación, como Objective-C, se llaman *lenguajes de alto nivel*, debido a que utilizan una sintaxis que sea fácil de entender para nosotros los humanos. Sin embargo, las computadoras solo entienden el *lenguaje de bajo nivel* o *lenguaje maquina*, que consiste en series de ceros y unos. Así, para que la computadora pueda entender nuestro programa, primero debemos *traducirlo* al lenguaje utilizado por la computadora. A este proceso de traducir un programa le llamaremos *compilación*, el cual es llevado a cabo por una herramienta llamada *compilador*. Este proceso de compilación produce lo que se conoce como un *binario* o *ejecutable*, el cual puede ser interpretado directamente por la computadora. En realidad, un compilador hace mucho mas que traducir un lenguaje de programación, pero por el momento nos bastara con esta definición.

En las próximas secciones, aprenderemos el uso básico del compilador. *GNUstep* puede usar tanto el compilador *GCC* como el compilador *CLANG*. Este último ofrece algunas características adicionales del lenguaje Objective-C. Sin embargo, en este manual nos limitaremos al lenguaje Objective-C heredado de *NeXT*.

3.2 Librerías

Al momento de programar necesitaremos a veces de ciertas funciones como, por ejemplo, la funciones *seno*, *coseno*, *logaritmo*, etc. Las cuales, por comodidad, ya se encuentran implementadas en las *librerías*. Una librería no es mas que un conjunto de funciones que podemos utilizar a la hora de programar. Para entender esto, utilicemos una analogía con una calculadora científica. Esta tiene varias teclas con distintas funciones, por ejemplo: *raíz cuadrada*, *raíz cúbica*, *logaritmo natural*, *seno*, *coseno*, etc. Así, podemos decir que las teclas de una calculadora científica constituyen un grupo de fun-

ciones matemáticas. De la misma forma, una librería constituye un grupo de funciones. La única diferencia es que en lugar de presionar una tecla, como en la calculadora, utilizamos una llamada para utilizar una de dichas funciones. Las librerías agrupan funciones que tienen cierta relación. Por ejemplo, la librería *mat.h* agrupa funciones matemáticas, la librería *stdio.h* agrupa funciones de manejo de archivos, así como de entrada y salida de datos, la librería *string.h* agrupa funciones de manejo de texto, etc. La forma, o sintaxis, para decirle al compilador que utilizaremos una cierta librería es:

```
#import <nombre de la librería>
```

Obsérvese que la línea comienza con un símbolo de numeral `#`, seguido de la palabra reservada *import*. Luego viene el nombre de la librería entre los signos `< >`. Por ejemplo, para indicar que haremos uso de la librería *math.h*, debemos escribir:

```
#import <math.h>
```

3.3 La función *main*

La función *main* es una función muy especial, ya que todo programa escrito en Objective-C debe tener una. La sintaxis de la función *main* es de la siguiente forma:

```
main (void)
{
  Aquí van las instrucciones de nuestro programa.
}
```

Obsérvese que después del nombre de la función *main*, va la palabra *void* encerrada entre paréntesis (el significado de esta palabra lo veremos mas adelante). En la siguiente linea hay una llave de apertura que abre el cuerpo de la función, el cual es cerrado por la llave de cierre en la última linea. Es entre estas dos llaves donde debemos escribir las ordenes que queremos lleve a cabo el programa.

3.4 Nuestro primer programa

Los programas en Objective-C se escriben en archivos con extensión *m*. Por ejemplo, nuestro primer programa podría estar contenido en un archivo llamado '`practica.m`'. Estos archivos se crean con editores de texto, y la extensión *m* es importante para que el compilador sepa que el programa esta escrito en Objective-C. Esto porque el compilador, *GCC* o *CLANG*,

maneja otros lenguajes aparte de Objective-C. De aquí en adelante, nos referiremos a lo escrito en estos archivos como el *código* (o el *código fuente*) del programa. Es recomendable que estos archivos, contengan únicamente caracteres validos en el idioma inglés. Por esta razón, no se utilizaran los signos ortográficos tilde y diéresis, la letra *ñ*, o los signos de apertura ¡ y ¿, en el código de estos. El tema de adaptar nuestros programas a otros idiomas diferentes al inglés, lo dejaremos para un capítulo posterior.

Bien, antes que nada, deberemos crear una carpeta para guardar nuestras practicas. Aquí, asumiremos que el nombre de esta carpeta es *practicas*. Hecho esto, abramos el editor de código de nuestra preferencia y escribamos el siguiente código en un archivo nuevo:

```
#import <stdio.h>

main (void)
{
  printf("Hola mundo \n");
}
```

Analicemos el código anterior. La primera línea le dice al compilador que nuestro programa hará uso de la librería *stdio.h*. Luego viene la función *main* con las instrucciones de nuestro programa. En este caso, únicamente la instrucción:

```
printf("Hola mundo \n");
```

La función `printf` nos permite escribir en la ventana de la terminal. Esta función pertenece a la librería *stdio.h*, y es por esto que hemos incluido dicha librería. La sintaxis de esta función es:

```
printf("Aquí va lo que queramos escribir en la terminal");
```

Obsérvese el punto y coma (`;`) al final de la línea. Los caracteres `\n` hacen que el cursor, después de escribir '`Hola mundo`', baje a la siguiente línea. Si no incluimos estos, el cursor se quedara al final de esta (hágase la prueba). Guardemos ahora el archivo como '`hola.m`' en nuestra carpeta *practicas*.

Ahora, abrimos una *Terminal* o *Shell*, y nos ubicamos en nuestra carpeta *practicas* con el comando:

```
cd practicas
```

Y llamamos al compilador *GCC* con el siguiente comando, para el compilador *CLANG* simplemente cámbiese `gcc` por `clang`:

```
gcc hola.m -o hola
```

Primero aparece el comando para llamar al compilador, seguido por el nombre de nuestro archivo '`hola.m`'. Luego aparece la opción '`-o`', que nos permite asignarle un nombre a nuestro binario o ejecutable. En este caso, el nombre '`hola`'. Si no se cometió algún error al escribir el código, veremos en la carpeta *practicas* un ejecutable llamada '`hola`'. Para ejecutar este programa simplemente escribimos el nombre del ejecutable en la terminal, precedido por `./`. Suponiendo que aun estamos dentro de la carpeta *practicas*:

```
./hola
```

Esto imprimirá el texto '`Hola mundo`' en la terminal. El punto y la diagonal al inicio, le indican a la terminal que el ejecutable se encuentra en la carpeta actual. De lo contrario, se buscara el ejecutable en las carpetas del sistema.

Con esto tenemos ya nuestro primer programa en Objective-C. Muy sencillo por cierto. Pero aun nos falta mucho por aprender, antes de poder escribir programas mas complejos.

3.5 Declaración de variables

La mayoría de los programas necesitan manejar datos, por lo que necesitaremos apartar espacio en la memoria de la computadora para almacenarlos. A esto le llamamos *declaración de variables*. Objective-C puede manejar datos de texto, numéricos y muchos otros. Para cada uno de estos tipos de datos existe una palabra reservada. La siguiente tabla muestra los datos numéricos para enteros y reales, así como para caracteres simples (un solo carácter).

Palabra reservada	**Tipo de dato que almacena**
int	Declara variables que almacenan números enteros. Su capacidad va del -32768 al 32767.
unsigned int	Declara variables que almacenan números enteros positivos.

float	Declara variables que almacenan números reales. Su capacidad va del 1.7014x10E38 al 2.9385x10E-39.
double	Declara variables que almacenan números reales. Su capacidad es mayor que la de *float*.
char	Declara variables que almacenan caracteres simples: `a`, `&`, `4`, etc.

Tabla 1-2. Algunos tipos de datos en Objective-C.

Los datos presentados en la tabla no son los únicos que existen, pero son los básicos para iniciarnos en la programación. La tabla también presenta la capacidad de las variables `int` y `float`. Es importante tener presente, cuando nuestro programa maneja números muy grandes o muy pequeños, que las computadoras tienen sus limitaciones para expresar números. Estas limitaciones dependen de la arquitectura de nuestra computadora, y los valores listados en la tabla son sólo ciertos para determinadas arquitecturas.

Para declarar variables necesitamos darles *identificadores*, es decir nombres. Sin embargo, estos identificadores deben cumplir con ciertas reglas, las cuales son:

- Un identificador se forma con una secuencia de letras (del alfabeto Inglés) y dígitos.
- El carácter subrayado `_` es también aceptado.
- Un identificador no puede contener espacios en blanco ni ningún otro carácter a parte de los mencionados anteriormente.
- El primer carácter de un identificador no puede ser un dígito.
- Se hace distinción entre mayúsculas y minúsculas. De esta forma, el identificador `cantidad` es distinto de los identificadores `Cantidad` y `CANTIDAD`.

Como ejemplo, los siguientes son nombres de identificadores válidos:

```
sumando1
_cantidad
masaCarro
```

Y los siguientes son inválidos:

```
1erCantidad
masa carro
%crecimiento
```

La declaración de variables es muy sencilla. Por ejemplo, para declarar dos variables, una de tipo `int` con el identificador de `numero` y otra de tipo `float` con el identificador de `velocidad`, escribiríamos el siguiente código:

```
int numero;
float velocidad;
```

Obsérvese el punto y coma al final de cada línea. Ahora para asignarles datos, por ejemplo 15 y 8.25, respectivamente, procedemos de la siguiente forma:

```
numero = 15;
velocidad = 8.25;
```

Nótese nuevamente el punto y coma al final de cada línea. Algo a tener en cuenta es que la variable a la cual se le va a asignar un dato, debe estar siempre a la izquierda del signo `=`, y el dato a asignar a la derecha. Es decir, que seria incorrecto escribir algo como:

```
15 = numero;
```

También es posible asignarle datos a las variables mediante expresiones. Por ejemplo, si queremos realizar un programa que sume dos enteros, podemos entonces declarar tres variables de tipo `int`, de la siguiente forma:

```
int sumando1, sumando2, resultado;
```

`sumando1` es la variable que almacenara el primer número a sumar, `sumando2` la que almacenara al segundo número y `resultado` la variable que almacenara la respectiva suma. Obsérvese que los identificadores están separados por una coma y que al final de la línea va el punto y coma. Ahora, para asignar la suma de `sumando1` y `sumando2` a `resultado`, procedemos de la siguiente forma:

```
resultado = sumando1 + sumando2;
```

Donde la expresión `sumando1 + sumando2` ha sido asignada a la variable resultado. Esto es similar a la forma en que estamos acostumbrados a realizar operaciones en álgebra.

También es posible darles valores a las variables al momento de declararlas (esto es a veces necesario). Por ejemplo, para que las variables `sumando1`, `sumando2` y `resultado` tengan un valor inicial igual a `0`, escribimos el siguiente código:

```
int sumando1 = 0, sumando2 = 0, resultado = 0;
```

Es recomendable que los nombres de las variables tengan relación con los datos que contienen, como los ejemplos aquí mostrados. Podríamos llamar a nuestras variables simplemente `a`, `b`, `c`, etc, pero estos nombres no nos dirían nada acerca de los datos que contienen. Esto además nos facilita la tarea de encontrar posibles errores o de hacerle modificaciones al código. Más aun si revisamos el código semanas o meses después de haberlo escrito. Lo recomendado en Objective-C, es que los nombres de las variables sean combinaciones de palabras, donde la segunda, tercera, cuarta, etc, palabras, comiencen con mayúscula. Por ejemplo, una variable que contenga la altura de una persona podría llamarse `alturaPersona`, donde la primera palabra esta en minúsculas y la segunda comienza con mayúscula. Otro ejemplo seria una variable que contenga el precio del maní japones, cuyo nombre podría ser `precioManiJapones`.

Por último debemos mencionar, aunque esto pueda parecer obvio, que los nombres de las variables deben ser únicos. Es decir que no podemos pretender utilizar el mismo nombre para distintas variables, ya que esto crearía una confusión.

3.6 Las funciones *printf()* y *scanf()*

Anteriormente hemos hecho uso de la función *printf()*. Sin embargo, no hemos dicho todo acerca de esta función. La sintaxis de esta es:

```
printf ("texto a imprimir", variable1,
        variable2 , variable3, ...);
```

Donde el texto entre comillas, llamado *cadena de control*, se imprime junto con las variables indicadas. Los puntos suspensivos al final, indican que se pueden incluir cuantas variables se deseen. Sin embargo, el tipo de las variables y el formato con que se imprimirán, deben especificarse en la *cadena de control*. Para especificar el formato, se utilizan las combinaciones de caracteres `%i` ó `%d` para variables tipo `int`, `%u` para variables tipo `unsigned int`, `%f` para variables tipo `float` y `double`, y `%c` para variables tipo `char`. Por ejemplo, supongamos que tenemos las siguientes variables con los datos indicados:

```
int edad = 35;
float altura = 1.78;
```

Los cuales podríamos imprimir de la siguiente forma:

```
printf("Su edad es de %d y su altura en metros es %f.",
       edad, altura);
```

Obsérvese que al final, las variables están ordenadas en el orden en que serán impresas, correspondiéndose con el tipo declarado en la cadena de control. Esto debe tenerse en cuanta, ya que si escribiéramos el código siguiente:

```
printf("Su edad es de %f y su altura en metros es %d.",
       edad, altura);
```

los tipos en la cadena de control no se corresponderían con las variables y tendríamos resultados indeseados. Algo como:

```
Su edad es de 0.000000 y su altura en metros es 0.000000.
```

Por defecto, los datos de tipo `float` y `double` (números reales) se escriben con 6 decimales. Sin embargo, podemos indicar la cantidad de decimales deseados utilizando *`%.nf`* donde *`n`* es la cantidad de decimales que deseamos. Por ejemplo, el siguiente código:

```
float velocidad = 78.5623;
printf("La velocidad es de %.2f metros/segundo.",
       velocidad);
```

Produce la siguiente salida:

```
La velocidad es de 78.56 metros/segundo.
```

Es posible también especificar la cantidad de espacios que queremos que ocupe el dato a imprimir. Para ello colocamos la cantidad de espacios deseados inmediatamente después del carácter *`%`*. Por ejemplo *`%mf`*, donde *`m`* es la cantidad de espacios. Si este espacio indicado es menor que el ocupado por el dato, *`m`* es ignorado y el dato se imprime con su longitud. Por ejemplo, el siguiente código:

```
int numero = 185463;
printf("%5d \n", numero);
```

```
printf("%6d \n", numero);
printf("%7d \n", numero);
printf("%8d \n", numero);
printf("%9d \n", numero);
```

Produce la salida:

```
185463
185463
 185463
  185463
   185463
```

Para las variables de tipo real, podemos especificar tanto la cantidad de espacios a ocupar, como la cantidad de decimales. Esto se haría con la combinación de caracteres *%m.nf*, donde *m* es la cantidad de espacios a ocupar y *n* la cantidad de decimales. ¡Hágase la prueba!.

Pasemos ahora a considerar la función *scanf()*. Esta función lee los datos ingresados a través del teclado y los almacena en las variables indicadas. Debe tenerse presente que una entrada de datos desde el teclado se produce cuando, después de escribir el dato, el usuario presiona la tecla ENTER. Mientras el usuario no presione esta tecla, no habrá un ingreso de datos. La sintaxis de esta función es:

```
scanf("Tipos de datos", &variable1,
      &variable2, &variable3, ...);
```

La cadena de texto entre comillas lista los formatos de los datos que se van a ingresar, los que se almacenaran en las variables indicadas. Esto es, el primer dato se almacena en `variable1`, el segundo dato en `variable2`, etc. Obsérvese el carácter *&* delante del identificador de cada variable. Como en el caso de la función anterior, los puntos suspensivos indican que se pueden leer cuantos datos se deseen. Por ejemplo, el siguiente código, le indica al usuario que ingrese su número de carné y su edad:

```
int carne, edad;
printf("Ingrese su numero de carne y su edad \n");
scanf("%d%d", &carne, &edad);
```

Obsérvese que como los dos datos esperados son de tipo `int`, los tipos de datos se han declarado como *%d%d*, un *%d* para cada dato. Noté además que no es necesario dejar espacios entre estos caracteres. Y puesto que los datos solamente son ingresados hasta que se presiona la tecla ENTER, solamente se puede ingresar un dato por línea y no es necesario usar los caracteres *\n*.

Tómese en cuenta que si le decimos al programa que lea datos de tipo `int`, como en el ejemplo anterior, pero le ingresamos datos de tipo `float`, se producirá un error que detendrá la ejecución del programa.

3.7 Nuestro segundo programa

Vamos a realizar ahora nuestro segundo programa, el cual consistirá simplemente en un programa que nos pide dos números, para luego mostrarnos la suma, resta, multiplicación y división de estos.

Este programa lo guardaremos en un archivo llamado '`matematicas.m`' en nuestra carpeta *practicas*. Y el código correspondiente se muestra a continuación:

```
#include <stdio.h>

main(void)
{
  float numero1, numero2, suma, resta, mult, div;

  printf("Ingrese dos numeros: \n");
  scanf("%f%f", &numero1, &numero2);

  suma = numero1 + numero2;
  resta = numero1 -- numero2;
  mult = numero1*numero2;
  div = numero1/numero2;

  printf("Suma: %.2f \n", suma);
  printf("Resta: %.2f \n", resta);
  printf("Multiplicacion: %.2f \n",  mult);
  printf("Division: %.2f \n", div);
}
```

Obsérvese que se han dejado algunas líneas de separación con el propósito de mejorar la presentación del código. Guardado el archivo abrimos una terminal y, una vez ubicados en nuestra carpeta *practicas*, ejecutamos el comando:

```
gcc matematicas.m -o matematicas
```

Para el compilador *CLANG* simplemente cámbiese `gcc` por `clang`. Si no hay ningún error en el programa, se creara el ejecutable '`matematicas`' en nuestra carpeta *practicas*. Ahora para ejecutarlo utilizamos el comando `./matematicas`. A continuación se muestra la ejecución del programa para los números 5.6 y 7.8:

```
Ingrese dos numeros:
5.6
7.8
Suma: 13.40
Resta: -2.20
Multiplicacion: 43.68
Division: 0.72
```

3.8 Operadores y sentencias

Objective-C hace uso de varios *operadores* que, en conjunto con las *sentencias*, nos permiten darle a un programa la capacidad de tomar decisiones. Por supuesto, esto nos brindara la posibilidad de crear programas mas complejos, como se vera en los ejemplos de las próximas secciones.

3.8.1 Operadores aritméticos

En nuestro último programa hicimos uso de los operadores aritméticos `+`, `-`, `*` y `/`. Para las operaciones suma, resta, multiplicación y división, respectivamente. Además de estos cuatro operadores, existe el operador `%` que se utiliza solamente con datos de tipo `int`. Este operador es llamado *resto*. Y, como su nombre lo indica, calcula el resto de la división de dos enteros. Por ejemplo, en las operaciones:

```
m = 21%7;
n = 22%7;
p = 24%7;
```

La variable m toma el valor de 0, n el valor de 1 y p el valor de 3.

En Objective-C, la precedencia de las operaciones aritméticas es la misma que utilizamos comúnmente. Por ejemplo, en la expresión:

```
r = 5 + 6*8;
```

Primero se lleva a cabo la multiplicación 6*8 y el resultado de está operación se suma a 5. Y así como en la matemática que aprendemos en la escuela, los paréntesis pueden utilizarse para crear operaciones aritméticas más complejas.

3.8.2 Operadores relacionales

Los operadores *relacionales* de Objective-C aparecen listados en la tabla 1-3. Como se observa, son los familiares operadores relacionales de la lógica, con algunas variaciones en su notación.

Operador	Significado
==	Igual a.
<	Menor que.
>	Mayor que.
<=	Menor o igual que.
>=	Mayor o igual que.
!=	No es igual a.

Tabla 1-3. Operadores relacionales en Objective-C.

Los operadores relacionales son útiles para crear *condiciones simples*. Ejemplos de condiciones simples son las siguientes:

```
m < n
p == q
r <= s
t != u
```

Donde `m`, `n`, `p`, `q`, `r`, `s`, `t` y `u` son variables. Como en la lógica, estas condiciones pueden ser verdaderas o falsas, y su uso lo veremos en las siguientes secciones.

3.8.3 Operadores lógicos

La tabla 1-4 presenta los operadores *lógicos* del lenguaje Objective-C, junto a la sintaxis correspondiente a cada uno.

Operador	Significado	Sintaxis
&&	Y (And).	(condición1) && (condición2) && ... && (condiciónN)
||	O (Or).	(condición1) || (condición2) || ... || (condiciónN)

!	Inverso o negativo (Not).	!condición

Tabla 1-4. Operadores lógicos de Objective-C.

Las condiciones *condición1*, *condición2*, etc, son condiciones simples. Y los puntos suspensivos significan que se pueden agregar cuantas condiciones simples se deseen. Estos operadores nos permiten crear *condiciones compuestas*, haciendo uso de las condiciones simples vistas en la sección anterior. Por ejemplo, para el operador `&&`, la siguiente condición compuesta:

```
(m > 5) && (m < 35)
```

es verdadera sólo si las dos condiciones simples son verdaderas. Es decir, si la variable *m* es mayor que 5 y a la vez menor que 35. En general, una condición compuesta que usa el operador `&&` solo es verdadera si cada una de las condiciones simples que la componen son verdaderas, de lo contrario es falsa. Por otro lado, una condición compuesta que hace uso del operador `||`, es verdadera si al menos una de las condiciones simples que la componen es verdadera, y sera falsa sólo en el caso de que todas las condiciones simples sean falsas. Por ejemplo, la condición compuesta:

```
(r >= 8) || (s > 12) || (t <= 4)
```

Sera verdadera en tres casos: cuando las tres condiciones simples sean verdaderas, cuando dos de estas sean verdaderas o cuando sólo una de estas sea verdadera. Y sera falsa sólo en el caso de que las tres condiciones simples sean falsas.

Por último, una condición que hace uso del operador `!` como, por ejemplo:

```
!(a == 31)
```

Es verdadera solamente cuando la condición simple que la compone es falsa. Es decir, cuando *a* no es igual a 31. Y es falsa cuando esta condición es verdadera. De ahí su nombre de *inverso* o *negativo*.

Estos tres operadores pueden combinarse en condiciones más complicadas, haciendo uso de paréntesis para indicar el orden en que deben realizarse las comprobaciones. Por ejemplo, consideremos la siguiente condición:

```
(a != 12) && ( (b <= 8) || !(c == 5) )
```

Obsérvese que la condición `(c == 5)` esta invertida o negada por el operador `!`. Esta condición compuesta sera verdadera en dos casos:

1. Cuando a no sea igual a 12 siendo b menor o igual que 8, sin importar el valor de c.
2. Cuando a no sea igual a 12 y c no sea igual a 5, sin importar el valor de b.

En todos los otros casos, esta condición compuesta sera falsa. Como veremos más adelante en este mismo capítulo, estas condiciones nos permiten darle a un programa la capacidad de tomar decisiones.

3.8.4 Sentencias condicionales

El lenguaje Objective-C provee dos *sentencias condicionales*, las cuales le permiten a un programa tomar decisiones. La primera de estas es la sentencia *if*, cuya sintaxis es:

```
if (condición)
  {
    Acciones a realizar si
    la condición se cumple
  }
else
  {
    Acciones a realizar si
    la condición no se cumple
  }
```

Donde *if* y *else* son palabras reservadas. La condición indicada puede ser una condición simple o compuesta y, si esta condición se cumple, el programa lleva a cabo las instrucciones encerradas entre el primer par de llaves, de lo contrario se llevan a cabo las instrucciones encerradas en el segundo par de llaves. En algunos casos, no necesitamos que el programa lleve a cabo instrucciones si la condición no se cumple, por lo que la sintaxis se reduce a:

```
if (condición)
  {
    Acciones a realizar si
    la condición se cumple
  }
```

Un ejemplo del uso de esta sentencia seria el siguiente código:

```
if (edad >= 18)
  {
    printf("Usted es mayor de edad");
  }
else
```

```
  {
    printf("Usted es menor de edad");
  }
```

Es posible anidar sentencias *if*. Esto es, colocar sentencias *if* dentro de otras sentencias *if*. Veremos un ejemplo de esto en nuestro tercer programa. Un arreglo útil cuando se desean hacer varias verificaciones lo muestra el siguiente código:

```
if (edad >= 100)
  {
    printf("Usted ha vivido mas de un siglo!");
  }
else if (edad >= 50)
  {
    printf("Usted es mayor de cincuenta");
  }
else if (edad >= 18)
  {
    printf("Usted es mayor de edad");
  }
else
  {
    printf("Usted es menor de edad");
  }
```

Además de la sentencia condicional *if*, Objective-C provee otra sentencia llamada *switch*, cuya sintaxis común es:

```
switch (variable)
{
case valor1:
  Instrucciones a realizar si
  variable es igual a valor1.
  break;
case valor2:
  Instrucciones a realizar si
  variable es igual a valor2.
  break;
case valor3:
  Instrucciones a realizar si
  variable es igual a valor3.
  break;
  ...
  ...
```

```
case valorN:
  Instrucciones a realizar si
  variable es igual a valorN.
  break;
default:
  Instrucciones a realizar si
  variable no es igual a ninguno
  de los valores anteriores.
}
```

Donde se hace uso de las palabras reservadas *switch*, *case*, *break* y *default*. Los puntos suspensivos indican que se pueden agregar cuantos valores se deseen para *variable*. El funcionamiento de esta sentencia es fácil de entender. Cuando *variable* es igual a *valor1*, se ejecutan las instrucciones correspondientes a este caso. Y lo mismo para cuando *variable* es igual a cualquier otro valor. Solamente cuando *variable* no es igual a ninguno de los valores indicados, es cuando se ejecutan las instrucciones de la opción *default*. Observe que la opción *default* no necesita la palabra reservada *break*. En realidad la opción *default* es opcional. Sin embargo, es recomendable que el programa tenga instrucciones que llevar a cabo, para cuando la variable no tenga ninguno de los valores esperados. El dato a evaluar, *variable*, debe ser un dato numérico.

En caso de que queramos que para un rango determinado de números se ejecuten las mismas instrucciones, podemos utilizar una forma modificada de la opción *case*, la cual es:

```
case valorA ... valorB:
  Instrucciones a realizar si variable
  es igual a valorA o a valorB, o esta
  entre ambos valores.
  break;
```

Obsérvense los tres puntos entre *valorA* y *valorB*, los cuales deben estar separados por un espacio de ambos valores. Un ejemplo de este caso sería:

```
case 11 ... 15:
  printf("El valor esta en el rango de 11 a 15");
  break;
```

O, si se desea que las mismas instrucciones se ejecuten para varios valores específicos, estas deben ir precedidas de los casos para los cuales deseamos su ejecución. Por ejemplo:

```
case 11:
case 49:
```

```
case 93:
  printf("El valor es 11, 49 ó 93");
  break;
```

Donde la instrucción se ejecuta cuando la variable vale 11, 49 ó 93.

3.8.5 Sentencias iterativas

Las *sentencias iterativas* permiten repetir la ejecución de un grupo de instrucciones un número determinado de veces, o hasta que cierta condición se cumpla.

La sentencia iterativa *for* repite la ejecución de un grupo de instrucciones un determinado número de veces, su sintaxis es:

```
for (valor inicial; condición; expresión de control)
  {
    Instrucciones a repetir
  }
```

Donde tanto el valor inicial, la condición y la expresión de control, hacen uso de una misma variable de control. Obsérvese la separación mediante punto y coma. El funcionamiento de esta sentencia, se entiende mejor con un ejemplo:

```
int x, mult;

for (x = 1; x <= 12; x = x + 1)
  {
    mult = 3*x;
    printf("3 por %d = %d", x, mult);
  }
```

Que produce la siguiente salida:

```
3 por 1 = 3
3 por 2 = 6
3 por 3 = 9
3 por 4 = 12
3 por 5 = 15
3 por 6 = 18
3 por 7 = 21
3 por 8 = 24
3 por 9 = 27
3 por 10 = 30
3 por 11 = 33
```

```
3 por 12 = 36
```

Como se ve, este código imprime la tabla de multiplicar para el número 3. En este ejemplo, la variable de control es `x`, y su valor inicial es 1. La condición indica que mientras `x` sea menor o igual a 12, las instrucciones se deben repetir. Y la expresión de control, indica la forma en que la variable de control se ira incrementando para llegar al valor final. En este ejemplo, la variable se incrementa en 1 en cada ciclo. Si, por ejemplo, la expresión de control hubiera sido `x = x + 2`, solamente se hubieran impreso los resultados para los números 1, 3, 5, 7, etc. En algunos casos, puede ser necesario hacer que la variable de control, vaya disminuyendo su valor en lugar de incrementarlo.

La otra sentencia iterativa es *while*. La cual repite un conjunto de instrucciones, mientras se cumpla una determinada condición. La sintaxis de esta sentencia es:

```
while (condición)
  {
    Instrucciones a repetir
  }
```

Un ejemplo de esta sentencia, que imprime la anterior tabla de multiplicación del 3, es:

```
int x = 1, mult;

while (x <= 12)
  {
    mult = 3*x;
    printf("3 por %d = %d", x, mult);
    x = x + 1;
  }
```

Obsérvese que a la variable de control `x`, se le asigna el valor de 1 fuera de la sentencia *while*. Y que la expresión de control `x = x + 1`, se encuentra dentro del cuerpo de la sentencia. Obsérvese, también, que si la variable de control hubiera tenido inicialmente un valor mayor de 10, las instrucciones dentro de la sentencia *while* nunca se hubieran ejecutado. Algunas veces, puede ser necesario que estas instrucciones se ejecuten por lo menos una vez. Una forma de asegurar esto, es utilizando una forma modificada de la sentencia *while*, cuya sintaxis es:

```
do
{
  Instrucciones a repetir
```

```
}
while (condición)
```

En este caso, la condición es evaluada al final de la sentencia, lo que asegura que las instrucciones se ejecuten al menos una vez. En los ejercicios de las siguientes secciones, se entenderá mejor la utilidad de estas sentencias.

Al hacer uso de estas sentencias iterativas, también llamadas *bucles*, debemos asegurarnos de que en algún momento se alcance el valor final, en el caso de la sentencia *for*, o de que la condición llegue a ser falsa, en el caso de la sentencia *while*. Ya que de no suceder esto, el programa seguirá repitiendo las instrucciones indefinidamente, y tendremos que detener su ejecución por algún medio externo al programa.

3.9 Nuestro tercer programa

El siguiente programa nos pide tres números enteros y, mediante sentencias *if* anidadas, determina el orden de estos.

```
#import <stdio.h>

main(void)
{
  int numero1, numero2, numero3, mayor, intermedio, menor;

  printf("Ingrese tres numeros enteros: \n");
  scanf("%d%d%d", &numero1, &numero2, &numero3);

  if (numero1 >= numero2)
    {
      mayor = numero1;
      intermedio = numero2;

      if (numero2 >= numero3)
        {
           menor = numero3;
        }
      else if (numero1 >= numero3)
        {
          intermedio = numero3;
          menor = numero2;
        }
      else
        {
          mayor = numero3;
          intermedio = numero1;
          menor = numero2;
```

```
        }
      }
    else
      {
        mayor = numero2;
        intermedio = numero1;

        if (numero1 >= numero3)
          {
            menor = numero3;
          }
        else if (numero2 >=numero3)
          {
            intermedio = numero3;
            menor = numero1;
          }
        else
          {
            mayor = numero3;
            intermedio = numero2;
            menor = numero1;
          }
      }

    printf("El mayor es: %d. \n", mayor);
    printf("El intermedio es: %d. \n", intermedio);
    printf("El menor es: %d. \n", menor);
}
```

¿Se entiende claramente este código? Cópiese y guárdese en un archivo llamado '`numeros.m`', en nuestra carpeta *practicas*. Y, una vez compilado, podemos ejecutarlo con el comando `./numeros`.

3.10 Nuestro cuarto programa

El siguiente programa realiza una tarea muy sencilla, pero muestra claramente el uso de las sentencias *switch* y *while*.

```
#import <stdio.h>

main(void)
{
  int eleccion = 0;
  float numero1, numero2, resultado;

  while (eleccion < 5)
```

```
    {
      printf("\n");
      printf("Elija una opcion para llevar a cabo: \n");
      printf("1. Sumar dos numeros. \n");
      printf("2. Restar dos numeros. \n");
      printf("3. Multiplicar dos numeros. \n");
      printf("4. Dividir dos numeros. \n");
      printf("5. Salir. \n");
      printf("\n");
      scanf("%d", &eleccion);

      switch (eleccion)
        {
        case 1:
          printf("Ingrese los dos numeros: \n");
          scanf("%f%f", &numero1, &numero2);
          resultado = numero1 + numero2;
          break;
        case 2:
          printf("Ingrese los dos numeros: \n");
          scanf("%f%f", &numero1, &numero2);
          resultado = numero1 - numero2;
          break;
        case 3:
          printf("Ingrese los dos numeros: \n");
          scanf("%f%f", &numero1, &numero2);
          resultado = numero1 * numero2;
          break;
        case 4:
          printf("Ingrese los dos numeros: \n");
          scanf("%f%f", &numero1, &numero2);
          resultado = numero1 / numero2;
          break;
        default:
          eleccion = 5;
        }

      if (eleccion != 5)
        {
          printf("El resultado es %.2f. \n", resultado);
        }
    }
}
```

Obsérvese que la sentencia *switch*, se encuentra dentro del cuerpo de la sentencia *while*. Obsérvese, también, que para salir del bucle *while*, el usuario

puede ingresar cualquier número diferente de 1, 2, 3 ó 4. Cópiese este código en un archivo llamado '`bucle.m`', en nuestra carpeta *practicas*. Y, una vez compilado, podemos ejecutarlo con el comando `./bucle`.

3.11 Funciones

Las *funciones* son un conjunto de instrucciones que realizan una tarea determinada, como las funciones de las librerías. Y que podemos llamar cuando queramos que dicha tarea se lleve a cabo. Las funciones nos permiten dividir un programa en pequeños módulos, que luego llamamos para que ejecuten sus tareas especificas. Esto hace mas fácil el diseño de un programa. Las funciones pueden recibir datos y devolver un dato como resultado de su tarea. Aunque, por supuesto, es posible que una función no reciba ningún dato, y que no devuelva ninguno. En general, la sintaxis de una función es de la siguiente forma:

```
tipo_devuelto nombre ( parámetros )
  {
    Instrucciones que lleva
    a cabo la función.
  }
```

Donde *tipo_devuelto* es el tipo del dato que la función devolverá o retornara. *nombre* es el nombre de la función y *parámetros* son los datos que la función necesita para su funcionamiento. Si la función no devolverá ningún dato, entonces no se especifica ningún tipo de dato. Sin embargo, si la función no necesita de ningún parámetro, los paréntesis deben colocarse encerrando la palabra *void*.

Un ejemplo de función que no necesita de ningún parámetro, y que no devuelve o retorna ningún dato, es una que simplemente imprime un mensaje, por ejemplo:

```
imprimirSaludo (void)
  {
    printf("Bienvenido al programa.");
  }
```

Donde *imprimirSaludo* es el nombre de la función. Un ejemplo de función que recibe un parámetro, pero que no retorna ningún dato, es el siguiente:

```
mayorDeEdad (int edad)
{
  if (edad >= 18)
    { printf("Usted es mayor de edad."); }
  else
```

```
    { printf("Usted es menor de edad."); }
}
```

Donde *mayorDeEdad* es el nombre de la función. Obsérvese que dentro de los paréntesis, va tanto el tipo de parámetro como el nombre de este. En este caso, *edad*. Se pueden especificar cuantos parámetros se deseen, simplemente separándoles con comas. Otro tipo de función, es aquella que no necesita de ningún parámetro, pero que si retorna un dato. Por ejemplo:

```
int imprimirOtroSaludo (void)
{
  printf("Bienvenido al programa.");
  return 0;
}
```

Donde el nombre de la función, *imprimirOtroSaludo*, esta precedido por el tipo de dato devuelto, *int*. Y donde se hace uso de la palabra reservada *return*, para retornar el valor numérico 0. A veces, se desea el retorno de algún valor, con el fin de saber si la tarea se realizo exitosamente. Para esta función de ejemplo, si el valor retornado es 0, significa que el mensaje se imprimió exitosamente. De lo contrario, algo salio mal.

El último tipo de función, es aquella que recibe parámetros y que a su vez retorna un dato. Por ejemplo, la siguiente función, recibe como parámetros los lados de un rectángulo, y retorna el área de este:

```
float calculoArea (float lado1, float lado2)
{
  float area;
  area = lado1*lado2;
  return area;
}
```

Donde el nombre de la función, *calculoArea*, esta precedido por el tipo de dato devuelto, *float*. Dentro del cuerpo de la función, se declara la variable *area*. Utilizada primero para almacenar el resultado del calculo, y retornarla posteriormente mediante la palabra reservada *return*.

¿Como podemos hacer uso de una función? Bueno, pues simplemente llamándola. Pero ¿Y como se llama a una función? Pues eso depende de si la función necesita parámetros o no, y de si retorna o no algún dato. Si una función no necesita parámetros y no retorna ningún dato, la llamada se realiza simplemente escribiendo su nombre seguido de un punto y coma. Por ejemplo, para hacer uso de la función *imprimirSaludo* vista anteriormente, se tendría:

```
imprimirSaludo;
```

Una función que no retorna ningún dato, pero que requiere un parámetro, necesita que el parámetro se le pase entre paréntesis (en el caso de varios parámetros, estos se separan con comas). Por ejemplo, la función *mayorDeEdad* vista anteriormente, puede usarse de la siguiente forma:

```
mayorDeEdad (edadCliente);
```

donde la variable *edadCliente* debe ser de tipo *int*, puesto que este fue el tipo de parámetro declarado en la función. Obsérvese que el nombre de la variable que se pasa como parámetro, no necesita ser el mismo que el nombre del parámetro declarado en la función (en este caso *edad*). En lugar de pasar una variable como parámetro, también puede pasarse directamente un valor. Por ejemplo:

```
mayorDeEdad (56);
```

Aunque esto no es muy útil. El uso de una función que no necesita de ningún parámetro, pero que retorna un valor, es como el de un valor cualquiera que se asigna a una variable. Por ejemplo, para la función *imprimirOtroSaludo*, tendríamos:

```
exito = imprimirOtroSaludo;
```

Donde la variable *exito*, almacena el valor retornado por la función *imprimirOtroSaludo*. Por supuesto, el tipo de esta variable debe ser *int*, ya que este fue el tipo declarado en la función para el valor retornado. Esta variable puede posteriormente verificarse, para saber si la función realizo su tarea con éxito. Si no deseamos realizar dicha verificación, podemos utilizar la función como una que no retorna ningún dato:

```
imprimirOtroSaludo;
```

En donde el valor retornado, no se asigna a alguna variable. Con lo visto hasta aquí, podemos adivinar como es el uso de una función que requiere parámetros y que devuelve un dato. Por ejemplo, para la función *calculoArea*, tendríamos:

```
resultado = calculoArea (base, altura);
```

Donde a la variable *resultado*, que debe ser de tipo *float*, se le asigna el dato retornado por la función. Y donde las variables *base* y *altura*, también de tipo *float*, se pasan como parámetros de la función. Obsérvese nuevamente

que estos parámetros, *base* y *altura*, no necesitan tener los mismos nombres que las variables declaradas como parámetros en la función (*lado1* y *lado2*).

Hasta aquí, hemos visto como crear y utilizar funciones. Sin embargo, para que nuestras funciones puedan ser utilizas, primero deben *declararse* al inicio del programa. Esto es muy sencillo, ya que para para hacerlo simplemente debemos indicar el tipo de dato devuelto, el nombre de la función y los parámetros que esta recibe. Básicamente, esto es copiar la primera linea de una función, y agregar un punto y coma al final. Por ejemplo, para declarar las cuatro funciones que hemos ejemplificado aquí, tendríamos que escribir el siguiente código:

```
imprimirSaludo (void);
mayorDeEdad (int edad);
int imprimirOtroSaludo (void);
float calculoArea (float lado1, float lado2);
```

Las funciones deben escribirse o *implementarse*, fuera del cuerpo de la función *main*. Y pueden ponerse antes o después de esta (lo recomendado es ponerlas antes). Además, las variables declaradas dentro de una función, solo existen para esa función. Es decir que las otras funciones no pueden *ver* dichas variables. Esto significa que dos, o mas funciones, pueden tener declaradas variables con el mismo nombre, puesto que una función dada no puede tener acceso a las variables de otras funciones.

Anteriormente dijimos que la función *main* es una función especial. Y es especial en el sentido de que no necesita ser llamada. Cuando un programa se pone en ejecución, esta función se llama automáticamente. Por esta razón, todo programa escrito en Objective-C, debe tener implementada esta función. De lo contrario, el programa simplemente no haría nada.

3.12 Nuestro quinto programa

El siguiente programa ejemplifica el uso de la función *calculoArea*, vista anteriormente:

```
#import <stdio.h>

float calculoArea (float lado1, float lado2);

float calculoArea (float lado1, float lado2)
{
  float area;
  area = lado1*lado2;
  return area;
}
```

```
main(void)
{
  float base, altura, resultado;

  printf("Ingrese la base y la altura: \n");
  scanf("%f%f", &base, &altura);
  resultado = calculoArea(base, altura);
  printf("El area es %.2f. \n", resultado);
}
```

Obsérvese la declaración de la función *calculoArea* al inicio del programa. El dato devuelto por esta función se almacena en la variable *resultado*, para posteriormente imprimir el área calculada. Cópiese este código en un archivo llamado '`area.m`', en nuestra carpeta *practicas*. Y, una vez compilado, podemos ejecutarlo con el comando `./area`.

El lector puede pensar que crear la función *calculoArea* no tiene mucho sentido, ya que el cálculo podría haberse hecho directamente dentro de la función *main*. El objetivo de este ejemplo, ha sido simplemente mostrar como crear y utilizar funciones. Sin embargo, cuanto mas grande es un programa, mas evidente se hace la utilidad de las funciones. Ya que facilitan el diseño de este.

3.13 Otros operadores

Existen algunos otros operadores de uso muy frecuente entre los programadores. Todos estos están conformados por dos caracteres. Estos operadores son: `+=`, `-=`, `*=`, `/=`, `--` y `++`. El comportamiento de los primeros cuatro, se ejemplifica a continuación con las variables x y y:

`x += y;`	equivale a	`x = x + y;`
`x -= y;`	equivale a	`x = x - y;`
`x *= y;`	equivale a	`x = x * y;`
`x /= y;`	equivale a	`x = x / y;`

Veamos ahora el comportamiento del operador `++`. Este puede ir antes o después de una variable, e incrementa el valor de dicha variable en 1. Si el operador va antes de la variable, el valor de esta se incrementa primero, y el valor resultante se utiliza en la expresión en que aparezca la variable. Por ejemplo:

```
x = 5;
y = 3 * (++x);
```

Después de la ejecución de estas dos lineas, el valor de la variable x es 6, y el valor de la variable y es 18. Por el contrario, si el operador va después de la variable, el valor de esta se incrementa después de ser utilizada en la expresión. Por ejemplo:

```
x = 5;
y = 3 * (x++);
```

Después de la ejecución de este código, el valor de la variable x es 6, y el valor de la variable y es 15. El comportamiento del operador `--` es similar. Solo que en lugar de incrementar el valor de la variable, lo disminuye en 1.

3.14 Comentarios

Es recomendable, cuando un programa llega a cierto grado de complejidad, introducir comentarios en el código de este. De forma que cuando lo revisemos semanas, meses o incluso años después, comprendamos rápidamente el funcionamiento del código. En Objective-C existen dos formas de agregar comentarios en un programa, y estos se pueden agregar en cualquier parte de este. La primera de ellas, utiliza los caracteres `//` para indicar que lo sigue hasta el final de la linea es un comentario. Por ejemplo:

```
// Aqui puede ir un comentario.
```

Para agregar comentarios mas grandes, podemos utilizar la segunda forma que hace uso de los pares de caracteres `/*` y `*/`. Donde todo aquello que vaya entre estos pares de caracteres, se considera un comentario. Por ejemplo:

```
/* Aqui puede ir un comentario
mucho mas largo que explique
el funcionamiento de cierta
parte del programa. */
```

No comentar un programa extenso es una mala idea, puesto que al querer modificar posteriormente el mismo, se perderá mucho tiempo tratando de entender el funcionamiento de este. A pesar de esto, algunos de los programas que veremos en próximos capítulos, no tendrán comentarios, ya que se irán explicando conforme los desarrollemos. Sin embargo, el lector puede ir introduciendo comentarios donde considere necesario.

4 Programación orientada a objetos

El capítulo anterior, fue una introducción al lenguaje de programación Objective-C. Es en este capítulo, donde comenzaremos nuestro estudio del entorno de desarrollo *GNUstep*. Sin embargo, antes deberemos aprender un poco acerca de lo que es la *programación orientada a objetos* (*POO*), y una nueva terminología. Es posible que al lector este capítulo le parezca muy teórico, incluso puede sentirse algo confundido. Sin embargo, no debe preocuparse por esto. Ya que cuando realicemos nuestro primer programa en *GNUstep*, en el siguiente capítulo, se vera como todo va encajando.

4.1 Clases y objetos

Discutamos antes un par de ideas. En el mundo real, existen muchos objetos los cuales clasificamos por las características que tienen en común. Por ejemplo, tomemos la palabra *carro*. Con ella nos referimos a cualquier tipo de carro, sea pequeño, grande, de color azul, rojo, etc. La palabra *carro* es entonces algo abstracto, que hace referencia a todo medio de transporte que tiene cuatro ruedas. De esta forma, decimos que *carro* es una *clase*. Y que cualquier carro en particular, es un *objeto* de dicha *clase*. Y este objeto, un carro cualquiera, tiene particularidades que lo diferencian de los demás. Como su color, su motor, sus asientos, etc. Pero, a pesar de estas diferencias, sigue perteneciendo a la clase *carro*.

En la programación orientada a objetos, los programas son diseñados mediante una combinación de módulos que interactúan entre si. Entendiendo por un *módulo*, a un conjunto de datos y patrones de comportamiento. Es a estos módulos a los que se les da el nombre de *objetos*. En el capítulo anterior, donde vimos el uso de las funciones, comenzamos a hacer uso de la noción de dividir un programa en módulos. Cada uno de los cuales efectuaba una tarea especifica. Podemos decir que un objeto, es un concepto más avanzado que el de las funciones vistas anteriormente. En esencia, un objeto es un conjunto de datos y de funciones sobre esos datos. Solo que ahora utilizamos una terminología diferente, a las funciones de un objeto les llamamos *métodos* o *selectores*, y a sus datos *variables de instancia*. Un objeto es entonces una unidad modular con la cual podemos interactuar a través de sus métodos. Los métodos de un objeto no sólo nos permiten interactuar con el, sino que también le proporcionan al objeto una forma particular de comportarse. La siguiente figura muestra la forma en que podemos visualizar a un objeto.

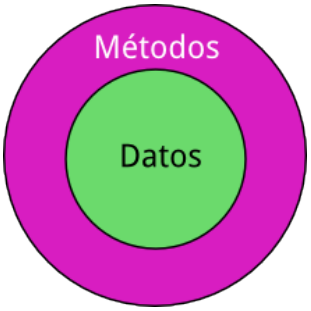

Figura 3-1. Visualización de un objeto.

Los objetos en programación, al igual que los objetos reales, pueden tener diferentes *estados*. Por ejemplo, un interruptor tiene dos estados, encendido y apagado. Los datos de un objeto, los cuales sólo se pueden acceder a través de sus métodos, son quienes determinan el *estado* en el cual se encuentra un objeto. Y dependiendo del estado en que se encuentre el objeto, así sera su comportamiento.

El concepto de objeto, se asemeja mas a la forma en que hacemos las cosas en el mundo real. Para construir un carro, por ejemplo, se ensambla una gran cantidad de objetos: motor, transmisión, carcasa, caja de cambios, etc. Cada uno de los cuales realiza un conjunto de tareas especificas. De esta forma, podemos visualizar un programa como un conjunto de objetos que interactúan entre si.

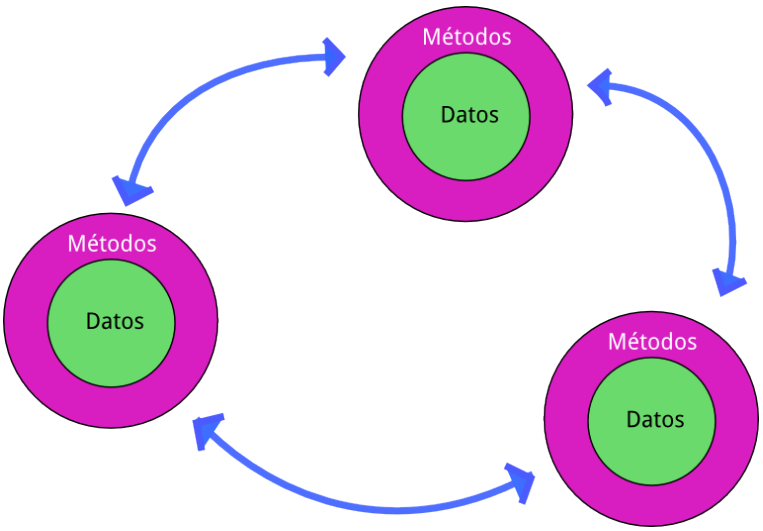

Figura 3-2. Objetos interactuando.

Decimos que un objeto tiene una *interfaz* y una *implementación*, de igual forma que un objeto real. Por ejemplo, un teléfono celular tiene una interfaz,

que es lo que nosotros vemos de él: su color, pantalla, teclas, etc. Y una implementación, que consiste en sus circuitos internos que implementan las funciones del teléfono. La *interfaz* de un objeto es, entonces, aquello que el resto del programa puede ver y con lo cual puede interactuar, mientras que su *implementación* es la parte interna del objeto que lo hace funcionar. La cual no esta disponible para el resto del programa.

Básicamente, podemos clasificar a los objetos en *visuales* y *no visuales*. Los objetos visuales de un programa, son aquellos que el usuario puede ver y con los cuales puede interactuar. Estamos muy familiarizados con los objetos visuales de un programa: ventanas, botones, casillas de selección, barras de desplazamiento, menús, listas desplegables, etc. Sin embargo, los objetos no visuales, aquellos que no tienen una representación visual, también juegan un papel importante en el funcionamiento de un programa. La siguiente imagen presenta un sencillo programa hecho en *GNUstep*. Como se puede apreciar, este esta compuesto por varios objetos visuales. Un objeto *ventana*, con el titulo '`Aplicación de Suma`'. Dos objetos *etiqueta*, con los textos '`+`' y '`=`'. Tres objetos *caja de texto*, dos para los sumandos y uno para el resultado de la suma. Y un objeto *botón*, con el titulo '`Sumar`', para llevar a cabo la suma. En conjunto, los objetos visuales también se conocen como *controles*.

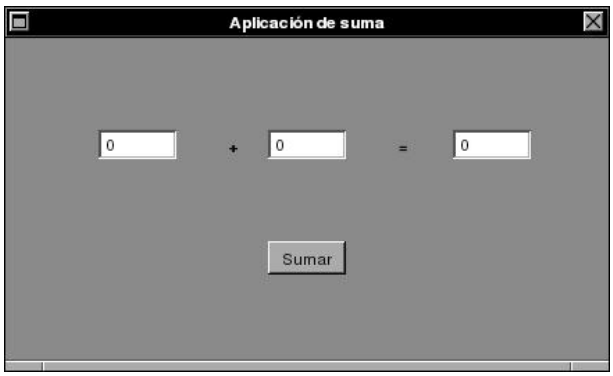

Imagen 3-1. Aplicación Suma.

Al inicio de esta sección, discutíamos que cada uno de los objetos que nos rodean, pertenecen a una cierta *clase*. Esto se cumple también en la programación orientada a objetos, donde cada objeto pertenece a una determinada clase. Básicamente, podemos decir que una *clase*, es la definición de un objeto. Así como '`Medio de transporte de cuatro ruedas`', es la definición de los objetos de la clase *carro*. Sin embargo, en programación, una *clase* es la definición de un objeto, en el sentido de que constituye el código que tienen en común todos los objetos de esa clase. Por ejemplo, todas las ventanas tienen una barra de titulo, bordes, un botón para cerrar, etc. Pero no solo esto, la clase también define el comportamiento de los objetos. Por ejemplo, el comportamiento de una ventana al mover sus bordes para modificar su dimensión.

Podemos decir entonces que una *clase*, es el plano para construir un objeto con sus aspectos y comportamiento básicos. Pero ¿cual es la razón de que hayan *clases*?. Pues antes que nada, facilitarnos las cosas a la hora de diseñar un programa. Si, por ejemplo, nuestro programa deberá tener una ventana con su barra de titulo y un botón para cerrarla, entonces podemos crear esta a partir de la clase *ventana*. Es decir, a partir del plano de construcción de una ventana. Para luego, de ser necesario, modificarla y adaptarla a nuestros propósitos. Esto, evidentemente, es mucho más sencillo que construir nuestra ventana desde cero.

4.2 Librerías Base y GUI, y herencia

En *GNUstep* las clases están agrupadas en dos grandes librerías, *Base* y *GUI*. A cada una de estas librerías que son un conjunto de clases, comúnmente se le conoce como *Framework*. La librería *Base* agrupa a todas las clases no visuales, y la librería *GUI* a todas las clases visuales. *GUI* proviene de *Graphical User Interface* (*Interfaz gráfica del usuario*). Todas estas clases, juntas, están organizadas en una jerarquía, ya que las clases tienen *herencia*. Para comprender esto, veamos la siguiente figura que ilustra la jerarquía de algunas de estas clases.

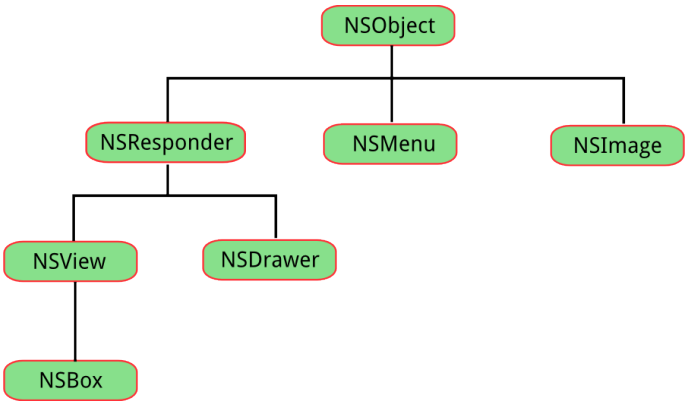

Figura 3-3. Jerarquía de algunas de las clases en *GNUstep*.

Como se observa, la clase `NSObject` se encuentra en el nivel más alto de la jerarquía. Mientras que las clases `NSResponder`, `NSMenu` y `NSImage`, derivan de la clase `NSObject`. A su vez, de la clase `NSResponder`, derivan las clases `NSView` y `NSDrawer`. Y por último, la clase `NSBox`, deriva de la clase `NSView`.

A las clases que derivan de una clase se les llama *subclases*, y a la clase de la cual deriva una clase se le llama *superclase*. Por ejemplo, la clase `NSResponder` tiene dos subclases (en realidad tiene mas, pero en la figura sólo se muestran dos), que son `NSView` y `NSDrawer`. Y la superclase de la clase `NSResponder`, es la clase `NSObject`. Obsérvese que la clase `NSObject` no tiene superclase, puesto que ella es la que se encuentra en lo más alto de

la jerarquía, por esto se le llama *clase raíz* (*root class*). Observe, también, que hay clases que no tienen subclases, como la clase `NSBox`.

Cuando se dice que una clase deriva de otra, lo que se quiere decir es que la clase adopta todos los métodos de la clase de la cual deriva. Por ejemplo, la clase `NSResponder`, al derivar de la clase `NSObject`, tiene todos los métodos de esta. Pero además, incorpora los métodos propios de su clase. Lo mismo sucede con las clases `NSMenu` y `NSImage`, que adoptan todos los métodos de la clase `NSObject`. Pero, a su vez, incorporan métodos propios de sus clases. Esto significa que las clases `NSResponder`, `NSMenu` y `NSImage`, tienen en común los métodos de la clase `NSObject`. A esto se le llama *herencia*, puesto que una clase hereda los métodos de la clase de la cual deriva. Es decir, de su *superclase*. Una clase en particular puede tener un método con el mismo nombre que uno presente en su *superclase*, o de otra clase más arriba en la jerarquía. A esto se le conoce como *redefinir* un método. Y más adelante veremos lo útil que puede ser en algunas situaciones.

El lector podrá preguntarse el porque de la herencia. Bueno, la herencia se da debido a que las clases se crean a partir de clases más básicas. Esto se hace así, porque es más fácil crear una clase a partir de otra, que crearla a partir de cero (bueno, en realidad hay otras razones para hacer esto, pero no las discutiremos aquí). Lo importante de conocer el concepto de herencia, es que en el sistema de documentación de *GNUstep*, solo aparecen documentados los métodos propios de cada clase, y no los que heredan. Sin embargo, es importante saber que un objeto creado a partir de una cierta clase, puede responder no sólo a los métodos de su clase, sino también a los métodos de su superclase. Y a los de la superclase de su superclase, y así sucesivamente hasta llegar a la *clase raíz*. Por ejemplo, un objeto creado a partir de la clase `NSBox` (figura 3-3), responde no sólo a sus propios métodos, sino también a los métodos de las clases `NSView`, `NSResponder` y `NSObject`. Esto puede parecer muy teórico, pero es necesario saberlo.

Así como para manejar números enteros o reales, debíamos declarar variables de tipo `int` o `float`, para manejar objetos debemos declarar variables de tipo *objeto*. Conocidas como *punteros a objetos* o simplemente *objetos*. Estas se declaran como las variables, con la diferencia de que el identificador de nuestro objeto, debe ir precedido por el carácter `*`. Por ejemplo, para declarar un objeto llamado *cuadro*, cuya clase sea `NSBox`, tendríamos:

```
NSBox *cuadro;
```

Y de forma similar para un objeto de cualquier otra clase. En algunas ocasiones, necesitaremos declarar objetos cuya clase desconocemos, en este caso el objeto se declara como de tipo `id`:

```
id nombre_objeto;
```

Obsérvese que en este caso no se utiliza el carácter `*`. Puede parecer extraño el hecho de que no conozcamos la clase de un objeto al momento de declararlo, pero más adelante veremos como se pueden dar estas situaciones.

4.3 Interfaz e implementación de una clase

GNUstep provee una gran cantidad de clases, a partir de las cuales podemos crear objetos. Sin embargo, es habitual el caso de tener que crear nuestras propias clases. Por lo que en esta sección, veremos como definir nuevas clases. A los objetos creados a partir de una clase, se les conoce como *instancias* de dicha clase. De allí el nombre *variables de instancia*, que se le da a los datos de un objeto.

En *GNUstep* las clases se conforman de dos archivos. Uno de ellos con extensión *h*, que contiene la *interfaz* de la clase, y el otro con extensión *m*, que contiene la *implementación* de esta. La estructura básica de un archivo de interfaz es la siguiente:

```
#import <librería necesaria>

@interface nombre_de_la_clase : superclase
{
  Aquí se declaran las variables de instancia.
}

Aquí se declaran los métodos o selectores

@end
```

Al inicio va la inclusión de la librería, o librerías, que necesite nuestra interfaz. Seguidamente va la declaración de la interfaz, que comienza con `@interface` y termina con `@end`. El nombre que le demos a la clase y la superclase de esta, separadas por dos puntos, se indican a continuación de `@interface`, en la misma linea. Luego se indican, entre llaves, las variables de instancia de la clase. Estas, son aquellas variables que deben estar disponibles para todos los métodos de la clase. Las variables de instancia, también se conocen como *atributos del objeto*, o simplemente *atributos*. Seguidamente, se declaran los métodos que estarán disponibles para el resto del programa. Es decir, los métodos o selectores a través de los cuales se podrá interactuar con los objetos creados a partir de esta clase. Como veremos más adelante, algunos métodos son únicamente para uso interno, y no deben declararse en esta parte de la interfaz.

Los métodos son similares a las funciones, pero con algunas diferencias a resaltar. Primero, los métodos pueden ser de dos tipos, los cuales son: *métodos de instancia* y *métodos de clase*. Más adelante veremos la diferencia entre estos tipos, ya que por el momento solo usaremos métodos de instancia.

Los métodos de instancia van precedidos por un signo menos (`-`), y los métodos de clase por un signo más (`+`). La segunda diferencia, es que el tipo del dato retornado por el método, debe ir indicado entre paréntesis. Y, en caso de que no se retorne ningún dato, la palabra *void* debe ir entre los paréntesis. La estructura básica de un método de instancia es (para un método de clase lo único que cambia es el signo):

```
- (tipo) nombreMetodo: (tipo)parametro0
            etiqueta1: (tipo)parametro1
            etiqueta2: ....
```

Donde los puntos suspensivos, indican que se pueden declarar cuantos parámetros se deseen. Como se puede observar, se indican tanto el tipo del dato retornado, como el de los parámetros. Cada parámetro esta precedido de una *etiqueta*, la cual se separa del parámetro por dos puntos. La etiqueta del primer parámetro, es el *nombre del método*. Y su *nombre completo*, es la unión del nombre del método y de los nombres de las etiquetas, incluyendo los dos puntos. Aunque las etiquetas son opcionales, es buena idea usarlas para indicar la finalidad del parámetro que le sigue. La declaración puede hacerse en una sola línea, siempre y cuando se deje un espacio de separación. Aquí se ha escrito de esta forma para mejorar la presentación. El siguiente, es un ejemplo de interfaz:

```
#import <AppKit/AppKit.h>

@interface Control : NSObject
{
  id titulo;
}
- (void) displayTime: (id)sender;
@end
```

La primera linea incluye el archivo '`AppKit/AppKit.h`', el cual nos permite usar todas las clases de la librería *GUI*. Luego viene la declaración de la interfaz de la clase, cuyo nombre es *Control* y cuya superclase es `NSObject`. Seguidamente vemos la declaración de una variable de instancia llamada `titulo`, de tipo `id`. Y por último, se declara el método de instancia `-displayTime:`, que no retorna ningún dato y que recibe un parámetro de tipo `id`. Obsérvese el punto y coma al final de la declaración de este método. La declaración de un método que no retorna ningún dato, y que no recibe ningún parámetro, sería de la siguiente forma:

```
- (void) display;
```

No se colocan dos puntos al final de la etiqueta *display*, por que no se pasa ningún parámetro. Veamos ahora como es la estructura básica de un archivo de implementación:

```
#import <librería necesaria>
#import "archivo de la interfaz"

@implementation nombre_de_la_clase

Aquí se implementan los métodos

@end
```

Primero se incluye la librería, o librerías, que necesite el código de la implementación. Luego se incluye el archivo de interfaz de la clase, entre comillas. Las comillas se utilizan únicamente cuando el archivo de la interfaz, se encuentra en la misma carpeta que el archivo de la implementación, de lo contrario deben utilizarse los caracteres `<>`. Seguidamente va la implementación del objeto entre `@implementation` y `@end`. Donde el nombre de la clase se coloca después de `@implementation`, en la misma linea. El siguiente, es el archivo de implementación de la interfaz del ejemplo anterior:

```
#import "Control.h"

@implementation Control

- (void) displayTime: (id)sender
{
  NSCalendarDate *date = [NSCalendarDate date];
  [date setCalendarFormat: @"%H : %M : %S"];
  [titulo setStringValue: [date description]];
}

@end
```

Esta clase solamente tiene un método, `-displayTime:`. No discutiremos aquí el código de este, ya que sólo se muestra como ejemplo.

4.4 Mensajes

La forma en que los objetos pueden comunicarse entre si, es mediante *mensajes*. La sintaxis de un mensaje que no requiere parámetros es:

```
[nombre_objeto nombre_método];
```

Donde *nombre_objeto* es el objeto al que se envía el mensaje, y *nombre_método* es el método que queremos ejecutar. Si el método necesita parámetros, estos deben ir precedidos por sus respectivas etiquetas (si hay), agregando los dos puntos de separación. La sintaxis general es:

```
[nombre_objeto nombre_método: parametro0
                    etiqueta1: parametro1
                    etiqueta2: parametro2
                    ......];
```

En esencia, un mensaje es la forma en que un objeto puede ejecutar métodos de otros objetos. Por supuesto, los métodos de ese otro objeto, deben estar declarados en su interfaz. Es decir, deben ser métodos accesibles desde el resto del programa. La llamada puede realizarse en una sola linea, siempre y cuando se deje un espacio de separación.

Los mensajes pueden retornar valores, por lo que estos pueden estar asignados a variables. Por ejemplo:

```
int x = [rectangulo ancho];
```

Donde se ejecuta el método `-ancho` del objeto `rectangulo`. Por supuesto, el tipo de la variable, debe ser el mismo que el del dato retornado por el mensaje. En este caso, hemos supuesto que este es de tipo `int`.

4.5 El paradigma Target-Action

El paradigma *target-action*, es una forma conveniente de hacer que un objeto visual, envié un mensaje a otro cada vez que el usuario interactué con el. Por ejemplo, supongamos un botń en una ventana, el cual debe ejecutar el método `-detener:` en el objeto `control`, cada vez que el usuario accione este. Llamando `boton` al identificador del botón, el siguiente par de mensajes aplicaría el paradigma target-action para estos objetos:

```
[boton setTarget: control];
[boton setAction: @selector(detener:)];
```

En este caso, el objeto `control` es el *target* (*objetivo*). Es decir, el objeto al que se le enviara el mensaje. El método a ejecutar, el *action*, es `-detener:`. Y el objeto que envía el mensaje, el *sender*, es el botón en la ventana. Obsérvese el uso de la función `@selector()`, utilizada para pasar el nombre del método a ejecutar. Esta función debe utilizarse siempre que se pase el nombre de un método como parámetro. Con este código, cada vez que el usuario accione el botón en la ventana, se enviara entonces el siguiente mensaje:

```
[control detener: boton];
```

Los métodos a utilizar con el paradigma target-action, deben ser de la forma:

```
- (void) nombreMetodo: (id)sender;
```

Es decir, métodos que no devuelvan objeto alguno, y que reciban como parámetro un objeto de tipo `id`. Lo normal, es que al parámetro recibido se le de el nombre de *sender*, ya que es el objeto que envía el mensaje. Este parámetro, es útil para obtener información del objeto que envía el mensaje.

Como se verá en el próximo capítulo, es posible aplicar este paradigma mediante la herramienta *Gorm*. Lo que, en general, nos evita utilizar los métodos `-setTarget:` y `-setAction:`, mostrados al inicio de esta sección.

4.6 El paradigma Outlet-Action

El paradigma *outlet-action*, es una forma de establecer conexiones entre los diferentes objetos de un programa, de tal forma que se puedan enviar mensajes entre ellos. Para un objeto en particular, un *outlet* es una referencia a otro objeto, que le permite enviar mensajes a ese otro objeto. Por esta razón, decimos que un outlet es una *conexión*. Como en el paradigma visto anteriormente, los *actions* son los métodos que un objeto pone a disposición del programa (los declarados en la interfaz de este). Por ejemplo, un objeto que tenga dos outlets y un action, podría representarse de la siguiente forma:

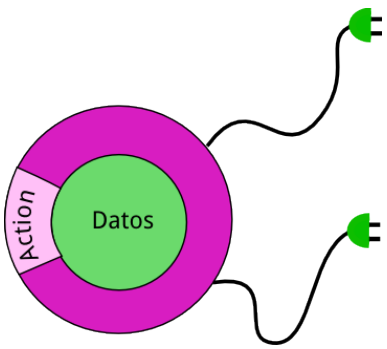

Figura 3-4. Objeto con dos outlets y un action.

Y la conexión con otros objetos podría representarse como:

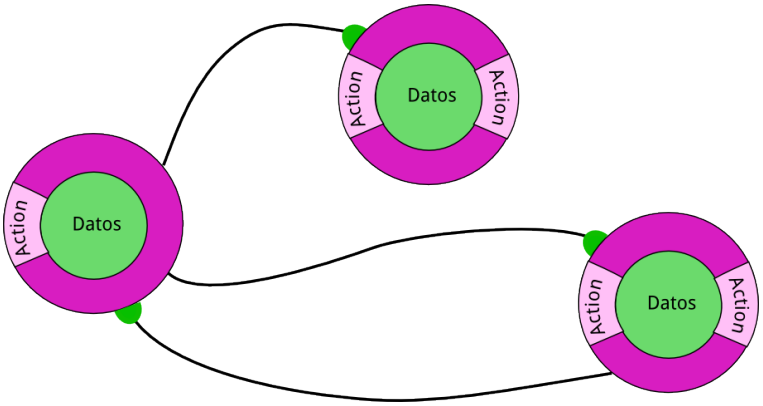

Figura 3-5. Conexión de outlets.

Los outlets se declaran junto a las variables de instancia, en la interfaz de un objeto. En esencia son variable de tipo `id`. Por ejemplo:

```
id outletOne;
```

Una vez realizada la conexión de este outlet, podemos utilizar esta variable para mandarle mensajes al objeto al cual este conectado. La conexiones entre outlets y objetos, se establecen gráficamente mediante la aplicación *Gorm*, lo cual veremos en el próximo capítulo. Es recomendable que los nombres de los outlets, hagan referencia al objeto al cual están conectados. Con las conexiones mostradas en la figura 3-5, el objeto a la izquierda puede enviar mensajes a los otros dos objetos. El objeto a la derecha puede enviar mensajes únicamente al objeto de la izquierda, y el objeto en la parte superior no puede enviar mensajes a ninguno de los otros objetos.

En el siguiente capítulo, crearemos nuestro primer programa en *GNUstep*. Este sera similar al mostrado en la imagen 3-1, un sencillo programa para sumar dos números. Para que este programa funcione, crearemos un objeto no visual que tendrá tres outlets y un action, conectados como muestra la siguiente figura:

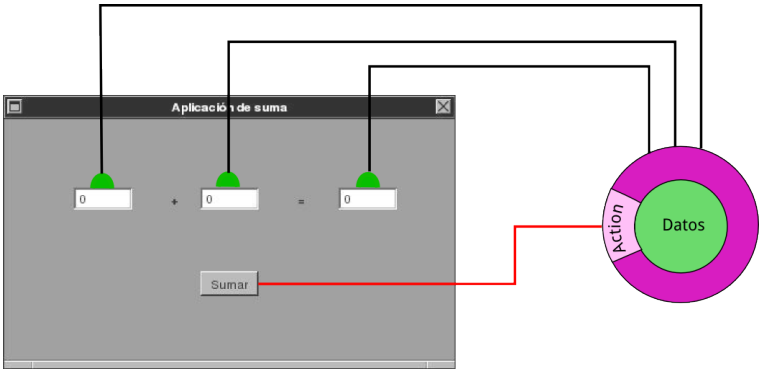

Figura 3-6. Conexión de un objeto y la interfaz del programa.

La linea roja, indica el uso del paradigma target-action, visto en la sección anterior. Al accionar el botón, se manda un mensaje al objeto no visual, para ejecutar el método que lleva a cabo la suma. Las tres lineas de color negro, son outlets del objeto no visual, que lo conectan con las tres cajas de texto en la interfaz. Estos, le permiten obtener los sumandos y desplegar el resultado de la suma.

5 Nuestra primera app con GNUstep

Se le llama *App* o *Aplicación* a un programa que despliega una interfaz gráfica. Esto es, programas que poseen un menú, ventanas, botones, etc. Por otro lado, los programas de linea de comandos, los cuales se ejecutan en una terminal, reciben el nombre de *Tool* o *Herramienta*. Con esto aclarado, vamos a crear nuestra primera app usando *GNUstep*. Para crear la interfaz gráfica de nuestra aplicación, utilizaremos la herramienta *Gorm*. Y utilizaremos un editor de código, el de nuestra preferencia, para crear el resto de archivos. Nos referiremos a este conjunto de archivos, como nuestro *proyecto*. Es probable que al lector le surjan varias preguntas conforme vayamos desarrollando esta primera app. Sin embargo, para mantener simple y claro el ejemplo, dejaremos para un capítulo posterior la explicación de todos los detalles.

La app que crearemos es la que se muestra en la figura 4-1, la misma que se discutió en el capítulo anterior. Una sencilla app para sumar dos números. Primero que nada, deberemos crear una carpeta para guardar los archivos que iremos creando en las siguientes secciones. En adelante, asumiremos que dicha carpeta se llama *Suma*. Es recomendable, especialmente para usuarios de *Windows*, que el nombre de la carpeta donde guardemos nuestro proyecto, así como la ruta donde se encuentre esta, no contengan espacios en blanco.

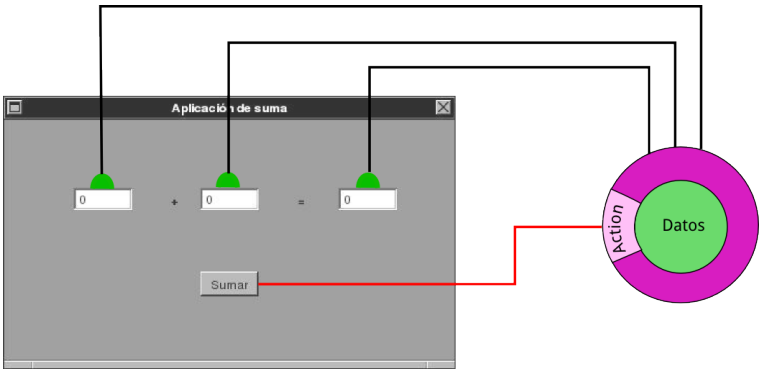

Figura 4-1. Esquema de nuestra primera app.

5.1 Los archivos de código

Vamos a escribir tres archivos de código. El primero de ellos, al que llamaremos '`SumaMain.m`', contendrá la función *main* que todo programa escrito en Objective-C debe tener. El código en este archivo es el siguiente:

```
#import <AppKit/AppKit.h>

int
main(int argc, const char *argv[])
```

```
{
  return NSApplicationMain (argc, argv);
}
```

Este código es prácticamente el mismo para cualquier app. Aquí se ejecuta la función `NSApplicationMain()`, la cual se encarga de iniciar la aplicación. No explicaremos aquí el papel de los parámetros *argc* y *argv* que recibe la función *main* y que son pasados a la función `NSApplicationMain()`. De cualquier forma, estos parámetros se utilizan muy raramente. Obsérvese que tal y como esta escrita la función *main*, debe retornar un número entero. De allí el uso de `return`, que devuelve el valor numérico retornado por la función `NSApplicationMain()`. En un capítulo posterior explicaremos la razón de retornar un número entero.

Como se muestra en la figura 4-1, la suma sera llevada a cabo por un objeto no visual, el único que aparece en el esquema. Ya que esta tarea es muy sencilla, haremos que la clase de este objeto, a la que llamaremos *SumaController*, sea una subclase de la clase raíz `NSObject`. Como se ve, este objeto debe tener tres outlets y un action (un método). Ya que en Objective-C es común que los archivos de interfaz e implementación de una clase, tengan el mismo nombre de la clase que representan, llamaremos a estos archivos '`SumaController.h`' y '`SumaController.m`', respectivamente. De esta forma, para el archivo de interfaz tenemos el siguiente código:

```
#import <Foundation/Foundation.h>

@interface SumaController : NSObject
{
  //Outlets
  id firstNumber;
  id secondNumber;
  id result;
}

//Action
- (void) makeSum: (id)sender;

@end
```

Los primeros dos outlets, estarán conectados a las cajas de texto donde el usuario ingresara los números a sumar. Mientras que el tercero, estará conectado a la caja de texto utilizada para desplegar el resultado de la suma, la cual sera llevada a cabo por el action `-makeSum:`. Ahora para el archivo de implementación, tenemos el siguiente código:

```
#import <AppKit/AppKit.h>
```

```
#import "SumaController.h"

@implementation SumaController

- (void) makeSum: (id)sender
{
  float a, b, c;

  a = [firstNumber floatValue];
  b = [secondNumber floatValue];

  c = a + b;

  [result setFloatValue: c];
}

@end
```

Dentro del método, primero se declaran tres variables de tipo `float`. Las primeras dos se utilizan para guardar los números ingresados por el usuario. Los cuales se obtienen usando el método `-floatValue`, el cual devuelve el valor numérico, en tipo `float`, del contenido de la caja de texto. La suma de estos números se almacena entonces en la tercera variable. Y por último, se utiliza el método `-setFloatValue:`, para mostrar el resultado en la correspondiente caja de texto. Por claridad, hemos llevado ha cabo la suma de esta forma. Sin embargo, es posible reducir todo esto a la siguiente forma:

```
- (void) makeSum: (id)sender
{
  [result setFloatValue:
   [firstNumber floatValue] + [secondNumber floatValue]];
}
```

El código dentro del método puede también escribirse en una sola linea.

Estos son los tres archivos de código que necesita nuestra app. En la siguiente sección, crearemos la interfaz gráfica y posteriormente el archivo de construcción.

5.2 La interfaz gráfica

Gorm es la herramienta de *GNUstep* para crear interfaces gráficas. No vamos a describir aquí todas las características de esta herramienta, ya que nos enfocaremos solamente en crear la interfaz de nuestra app. Al arrancar *Gorm* se muestra el menú principal y dos ventanas, una llamada *Palettes* y

otra llamada *Inspector*. Si *GNUstep* esta configurado para utilizar el menú en ventana, *Gorm* abrirá un documento nuevo, por lo que se mostrara una tercera ventana correspondiente a este nuevo documento.

Comencemos seleccionando en el menú Document → New Application, esto creara un nuevo documento, el cual ya contiene un menú principal y una ventana, como se muestra en la imagen 4-1. Si *Gorm* creo un nuevo documento vació al arrancar, cerremos dicho documento, ya que no lo utilizaremos.

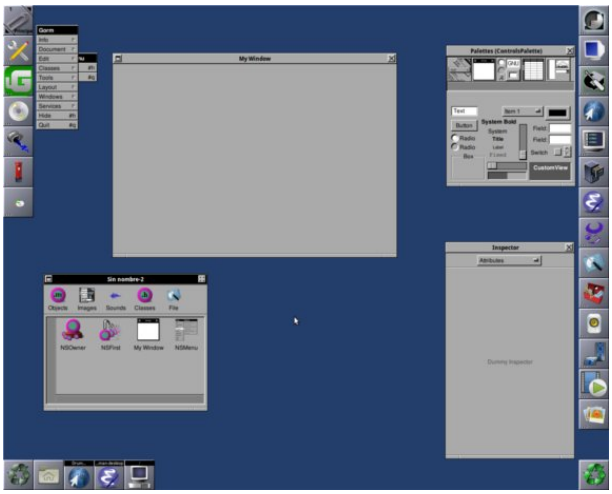

Imagen 4-1. *Gorm*.

La ventana *Palettes* contiene cinco paletas, entendiendo por paleta un conjunto de objetos visuales. Las cinco paletas son: *Menus*, *Windows*, *Controls*, *Containers* y *Data*. Estas paletas se seleccionan con los iconos de la parte superior de la ventana.

La ventana *Inspector* tiene una lista desplegable con cinco opciones: *Attributes*, que nos permite editar las propiedades del objeto seleccionado; *Connections*, para editar las conexiones del objeto; *Size*, para editar el tamaño y posición del objeto; *Help*, para agregar mensajes de ayuda (conocidos como *Tool Tips*), y *Custom Class* para modificar la clase del objeto.

La tercera ventana con el título *Sin nombre* es la ventana del documento, tiene cinco iconos en la parte superior. Los primera tres, *Objects*, *Images* y *Sound*, nos muestran los objetos, imágenes y sonidos de nuestra interfaz, respectivamente. El cuarto icono, *Classes*, nos permite trabajar con objetos no visuales que estén relacionados con la interfaz. Y el quinto, *File*, nos permite manejar la compatibilidad de la interfaz gráfica con versiones anteriores de *GNUstep*.

Para agregar controles a la ventana de nuestra app, simplemente debemos arrastrarlos con un clic izquierdo del ratón desde la paleta y soltarlos en la ventana. Seleccionemos entonces la paleta *Controls*, si no esta seleccionada

ya, y arrastremos tres cajas de texto, dos etiquetas (de las que tienen el texto *System*) y un botón. Los controles pueden reubicarse arrastrándolos con el ratón y su tamaño se puede modificar arrastrando con el ratón los nodos de color rojo. El texto que muestran puede editarse dando doble clic sobre ellos. Editemos entonces el texto de las etiquetas, en una de ellas escribamos `+` y en la otra `=`. Borremos el contenido de las cajas de texto y escribamos como título del botón *Sumar*. Redimensionemos la ventana y los controles, y reubiquemos estos de tal forma que el resultado final se vea como la imagen 4-2.

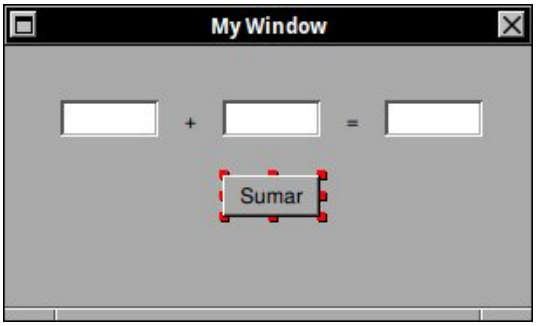

Imagen 4-2. Interfaz de nuestra app.

Téngase presente que al agregar textos en una interfaz gráfica, podemos utilizar todos los signos y caracteres del español.

Aun no hemos terminado nuestra interfaz gráfica pero es buena idea guardarla. Seleccionemos en el menú Document → Save, y guardemos el documento con el nombre '`suma`' en la carpeta de nuestro proyecto. El título de la ventana del documento '`gorm`' cambia ahora a *suma.gorm*. No confundir la *ventana del documento* con la *ventana de nuestra interfaz* (en la que hemos agregado los controles). El paso siguiente es hacer que nuestra interfaz sepa de la existencia de nuestra clase `SumaController`, de otra forma no podremos realizar las conexiones que se muestran en la figura 4-1. Seleccionemos entonces en la ventana del documento el icono *Classes*, esto hará que veamos toda la jerarquía de las clases disponibles en *GNUstep*. Seleccionemos ahora en el menú Classes → Load Class... y agreguemos nuestro archivo '`SumaController.h`'. Nuestra clase `SumaController` debe aparecer ahora como una subclase de la clase `NSObject`, imagen 4-3.

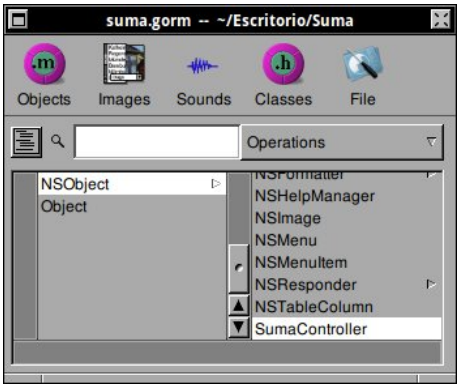

Imagen 4-3. Agregando nuestra clase.

Estando seleccionada nuestra clase, elijamos en el menú Classes → Instantiate. Esto automáticamente seleccionara la sección *Objects* en la ventana del documento. Y veremos un nuevo objeto en ella, llamado *SumaController*, imagen 4-4.

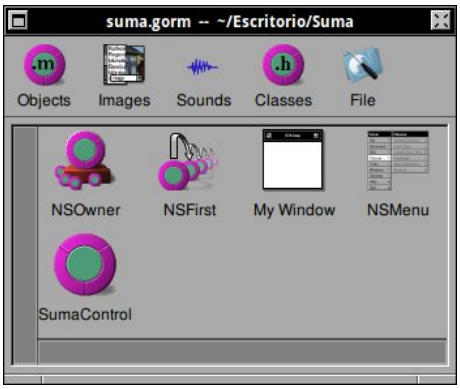

Imagen 4-4. El objeto SumaController.

En esta sección se muestran los objetos instanciados (creados) al momento de cargar o leer la interfaz gráfica. Hay tres objetos, el menú de nuestra aplicación, que es el objeto *NSMenu*; la ventana de nuestra aplicación, que corresponde al objeto *My Window* y el objeto creado a partir de nuestra clase *SumaController*. Los otros dos objetos, *NSOwner* y *NSFirst*, los veremos en un capítulo posterior. De esta forma, la interfaz gráfica de nuestra app se encargara de crear un objeto a partir de nuestra clase `SumaController`. Esto es bastante ventajoso. Por no mencionar que hasta el momento no hemos visto como crear un objeto a partir de una clase. Algo que veremos en el próximo capítulo.

El siguiente paso es crear las conexiones mostradas en la figura 4-1. Estas se crean arrastrando los objetos con el ratón, mientras se mantiene presion-

ada la tecla CONTROL de *GNUstep*. Esta tecla no necesariamente corresponde a la tecla *Ctrl* del teclado, ya que *GNUstep* permite configurar las teclas modificadoras como uno desee. La app *SystemPreferences*, en su sección *Modifier keys*, permite ver/establecer esta configuración. Las conexiones siempre se crean arrastrando el objeto fuente u origen (*Source*), hacia el objeto destino (*Target*). Para la conexión de outlets, el objeto que define los outlets es siempre el objeto *Source*. Mientras que para las conexiones *Target-Action*, el control visual (un botón, una casilla de selección, etc.), es siempre el objeto *Source*.

Procedamos entonces a conectar los tres outlets. Primero conectemos el outlet *firstNumber*, el cual conectaremos con la caja de texto a la izquierda. Para ello, manteniendo presionada la tecla CONTROL, realicemos la conexión arrastrando nuestro objeto *SumaController*, hacia dicha caja de texto. Luego en el *Inspector*, en la sección *Outlets*, seleccionemos el outlet *firstNumber*. Esto hará que un circulo con una *S*, de *Source*, se muestre al lado de nuestro objeto *SumaController*. Y otro con una *T*, de *Target*, se muestre al lado de la caja de texto, imagen 4-5. Por último, para crear la conexión, damos un clic en el botón *Connect* en la parte inferior del *Inspector*. Esto hará que la recién creada conexión, se muestra en la sección *Connections* de dicha ventana.

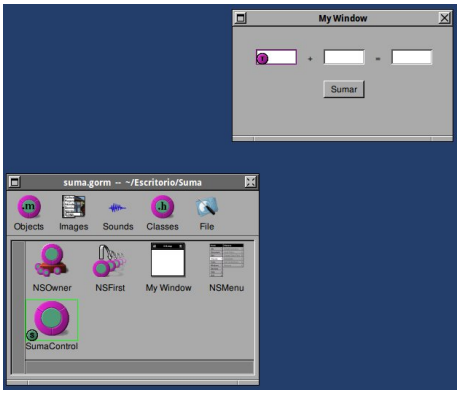

Imagen 4-5. Conexión del outlet *firstNumber*.

Realicemos de forma similar la conexión de los otros dos outlets. El outlet *secondNumber* a la caja de texto de en medio y el outlet *result* a la caja de texto a la derecha.

Ahora nos falta conectar el botón con nuestro objeto *SumaController*. Realizando la conexión del botón hacia nuestro objeto, seleccionemos en el *Inspector*, en la sección *Outlets*, la opción *target*. Esto desplegara los actions disponibles en nuestro objeto. Seleccionemos el action *makeSum:* (el único) y demos un clic en el botón *Connect* en la parte inferior del *Inspector*, imagen 4-6.

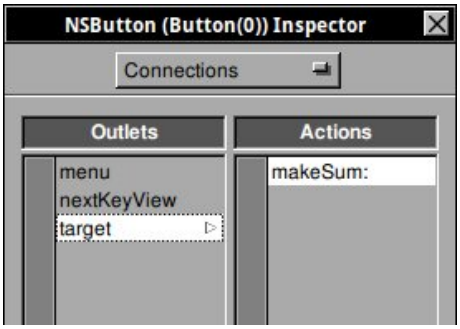

Imagen 4-6. Conexión del botón.

Hasta aquí es suficiente para que nuestra interfaz gráfica sea funcional. Sin embargo, vamos a modificarle algunos otros aspectos para mejorarla. Seleccionemos la ventana de nuestra interfaz, ya sea dando un clic en alguna parte de la ventana que no contenga algún control o seleccionando el icono *My Window* en la ventana del documento. Ahora en la lista desplegable en la parte superior del *Inspector*, seleccionemos la opción *Attributes*. Aquí podemos cambiar el título de nuestra ventana a algo mas descriptivo. Por ejemplo: *App para sumar*. Quitemos también la marca de la opción *Resize bar*, para que el tamaño de nuestra ventana no se pueda modificar. Seguidamente seleccionemos la caja de texto de la derecha, donde se mostrara el resultado. Y, en el *Inspector*, quitemos la marca de la opción *Editable*, para que el usuario no pueda editar el contenido de esta caja de texto.

Seguidamente estableceremos la forma en que el usuario, mediante el uso de la tecla TAB, puede moverse a través de los controles de nuestra interfaz. Queremos que el usuario pueda ir de la primera caja de texto (la de la izquierda) a la segunda caja de texto (en medio) y luego al botón *Sumar*. Para posteriormente poder retornar a la primera caja de texto. Hagamos entonces una conexión desde la primera caja de texto a la segunda, seleccionando en el *Inspector* el outlet *nextKeyView* y dando clic en el botón *Connect*. De forma similar realicemos la conexión de la segunda caja al botón *Sumar*, y de este a la primera caja.

Finalmente vamos a decirle a la ventana, que la primera caja de texto sea la que tenga el enfoque en cuanto la ventana de nuestra app este activa. Esto es razonable, ya que es allí donde debe ingresarse el primer dato. Para ello, hagamos una conexión desde el icono de nuestra ventana (el que aparece en la ventana de nuestro documento) hasta la primera caja de texto, seleccionando en el *Inspector* el outlet *initialFirstResponder* y dando clic en el botón *Connect*.

Con esto hemos terminado nuestra interfaz gráfica. Así que guardemos los cambios hechos a nuestro documento y cerremos *Gorm*. *Gorm* guarda la interfaz gráfica en una carpeta con extensión '`gorm`', la cual contiene tres archivos para recrear posteriormente la interfaz.

5.3 El archivo GNUmakefile

A diferencia de los programas que hemos realizado previamente, no trataremos de llamar al compilador para construir nuestra aplicación. Esto no es recomendable para compilar programas que hagan uso de los frameworks de *GNUstep*, ya que la cantidad de opciones que hay que pasarle al compilador complican la tarea (los programas anteriores no hacían uso de estos frameworks). En su lugar, escribiremos un archivo llamado '`GNUmakefile`'. El cual contendrá, entre otras cosas, una descripción de la estructura de nuestro proyecto. Como veremos en la siguiente sección, estos archivos son utilizados por la herramienta de construcción *gnustep-make* para compilar programas hechos con *GNUstep*. El contenido de este archivo para nuestra primera app es el siguiente:

```
include $(GNUSTEP_MAKEFILES)/common.make

# Nombre de la app
APP_NAME = Suma

# Archivo principal de interfaz grafica
Suma_MAIN_MODEL_FILE = suma.gorm

# Recursos de la app
Suma_RESOURCE_FILES = suma.gorm

# Archivos de interfaz de clases (Headers)
Suma_HEADER_FILES = SumaController.h

# Archivos de implementacion de clases
Suma_OBJC_FILES = SumaController.m

# Otros archivos
Suma_OBJC_FILES += SumaMain.m

# Makefiles
include $(GNUSTEP_MAKEFILES)/application.make
```

Las lineas que comienzan con el carácter `#`, son comentarios. La primera linea incluye un archivo que es necesario para la construcción de proyectos hechos con *GNUstep*, y debe ir al inicio de todo archivo '`GNUmakefile`'. Seguidamente, mediante la entrada `APP_NAME`, se define el nombre de nuestra app, en este caso *Suma*. Las siguientes entradas, donde se listan los archivos de nuestra app, deben comenzar con el nombre asignado a esta, tal y como se muestra. En la entrada `Suma_MAIN_MODEL_FILE`, se indica el *principal* archivo de interfaz gráfica de nuestra app, que en este caso es '`suma.gorm`'. La interfaz principal, es aquella que se debe cargar al momento de arrancar la

aplicación. Una app puede tener varios archivos de interfaz gráfica, pero sólo uno de ellos puede ser el principal. En la siguiente entrada, `Suma_RESOURCE_FILES`, se indican los recursos de nuestra app. Los recursos de una app, son los archivos que *no* contienen código. Por ejemplo, imágenes, sonidos, documentos, archivos de interfaz gráfica, etc. Es este caso, nuestra app sólo tiene un recurso, el archivo de interfaz gráfica. En las siguientes dos entradas se listan, respectivamente, los archivos de interfaz e implementación de las clases presentes en nuestro proyecto. En nuestro caso, se trata de los archivos '`SumaController.h`' y '`SumaController.m`', respectivamente. Finalmente, en la última entrada, se listan otros archivos de código que estén presentes en nuestro proyecto. En este caso, solamente queda el archivo '`SumaMain.m`'. Y la última linea, incluye un archivo que depende del tipo de proyecto que queramos construir. En este caso queremos construir una aplicación, por lo que se incluye el archivo '`application.make`'. Guardemos este archivo con el nombre '`GNUmakefile`' en la carpeta de nuestro proyecto. Y con esto estamos listos para compilar y probar nuestra primera app.

5.4 Compilando y probando nuestra app

Es momento de compilar nuestra app. Abramos una terminal y ubiquémonos en el directorio de nuestro proyecto, de ser necesario ejecutemos el *script* de *GNUstep* (ver apéndice B), y ejecutemos entonces la herramienta *gnustep-make* mediante el comando `make`. Esta leerá el archivo '`GNUmakefile`' y, en base a la información en el, compilara nuestra app.

```
german@german-desktop:$ make
Making all for app Suma...
 Creating Suma.app/....
 Compiling file SumaController.m ...
 Compiling file SumaMain.m ...
 Linking app Suma ...
 Creating Suma.app/Resources...
 Creating stamp file...
 Creating Suma.app/Resources/Info-gnustep.plist...
 Creating Suma.app/Resources/Suma.desktop...
 Copying resources into the app wrapper...
german@german-desktop:$
```

Si no hay errores en la compilación, podemos ejecutar nuestra app usando el comando `openapp`.

```
openapp ./Suma.app
```

O escribiendo la ruta del ejecutable:

```
./Suma.app/Suma
```

En caso de algún error, se mostrará el nombre del archivo y el número de linea donde se encontró el error. Dicho error debe corregirse antes de intentar compilar nuevamente el proyecto.

Efectuemos algunas sumas para probar nuestra app. Comprobemos como la tecla TAB nos permite movernos de un control a otro. Recuérdese que en *GNUstep*, cuando un botón esta seleccionado, la tecla SPACE puede utilizarse para ejecutar la acción asociada al mismo.

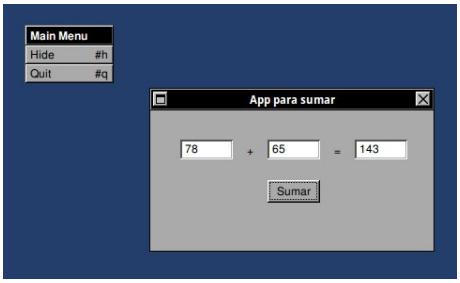

Imagen 4-7. Probando nuestra app.

Y con esto hemos terminado nuestra primera app hecha con *GNUstep*. Para mayor información sobre el uso de la herramienta *gnustep-make*, consúltese el apéndice B. Donde se abordan, entre otros temas, el procedimiento de instalación de un programa.

6 Clases y objetos en Objective-C

En este capítulo abordaremos varios temas importantes en Objective-C, como lo son el ciclo de vida de los objetos, la gestión de la memoria y el como redefinir métodos. Todo esto, nos permitirá realizar programas más complejos, ya que podremos crear objetos y manejarlos de forma eficiente. Sin embargo, primero veremos lo que son las *cadenas de texto*, y ampliaremos lo visto con anterioridad sobre los métodos y las variables de instancia.

6.1 Cadenas de texto (Strings)

Se conoce como *string*, o *cadena de texto*, a una serie de caracteres. Por ejemplo:

```
45rt5 5%fh  &/alkjkjaskj2764#"&%$7dk
```

Un espacio en blanco también se considera un carácter. Entre las clases no visuales de *GNUstep*, existe una que nos permite manejar *strings* o cadenas de texto, esta es la clase `NSString`. Hay varias formas de crear una cadena de texto mediante la clase `NSString`, pero por el momento nos ocuparemos de aquella que utiliza el constructor `@"..."`. Por ejemplo, el siguiente código:

```
NSString *nombre = @"Claudia";
```

Asigna la cadena de texto *Claudia*, al objeto *nombre* de tipo `NSString`. Los objetos `NSString` creados de esta forma, se conocen como *estáticos*. La razón de ello es que estos objetos son creados al momento de *compilar* el programa.

A veces el constructor `@"..."`, se utiliza directamente sin necesidad de asignar la cadena resultante a un objeto. Este procedimiento no crea objetos *estáticos*. Los ejemplos de las siguientes secciones, aclararan todo esto.

6.2 Variables de instancia

Objective-C permite declarar tres tipos de *ámbitos* para las variables de instancia: *Privadas*, *Protegidas* y *Publicas*. Las variables *privadas* solamente pueden ser utilizadas por la clase que las define. Las *protegidas* pueden ser utilizadas no solo por la clase que las define, sino también por las clases que deriven de esta. Mientras que las variables *publicas* pueden ser utilizadas por cualquier otra clase. Para declarar el ámbito de las variables, se utilizan las *directivas de compilador* `@private`, `@protected` y `public`, respectivamente. Cada directiva aplica para las variables listadas después de esta, hasta la siguiente directiva o hasta el final de la lista de variables. Sino se especifica el ámbito de una variable, esta se toma como *protegida*. En el siguiente código, la variable `age` es *privada*, la variable `identifier` es *protegida* y la

variable `owner` es *publica*. La variable `name`, al no tener asignado un ámbito, se toma como *protegida*:

```
@interface Entity : NSObject
{
  NSString *name;
@private
  int age;
@protected
  float identifier;
@public
  id owner;
}
```

6.3 Métodos

Ya hemos comentado en un capítulo anterior como se implementan los métodos. Y ya hemos escrito también un sencillo método. Sin embargo, para que este tema quede claro, veamos aquí unos cuantos ejemplos mas. Comencemos con un método que no recibe ningún parámetro y que no devuelve ningún dato. Este seria de la siguiente forma:

```
- (void) printName
{
  printf("Carlos.");
}
```

Obsérvese que la palabra `void`, entre paréntesis, es necesaria para indicar que el método no devuelve dato alguno. Recuérdese también que la convención en Objective-C, es que cuando un nombre o etiqueta esta formado por dos o mas palabras, todas ellas, exceptuando la primera, deben comenzar con una letra mayúscula.

Otro ejemplo seria el de un método que no recibe ningún parámetro, pero que devuelve un objeto cuya clase no esta determinada:

```
- (id) dataForController
{
  return data;
}
```

Obsérvese que en este caso la palabra `id`, entre paréntesis, indica que se devolverá un objeto cuya clase se desconoce. Noté también que el objeto es devuelto con `return`, de forma similar a lo que ocurre en las funciones.

Veamos ahora el caso de un método que recibe un parámetro de clase `NSString`, y devuelve un objeto de clase `NSBox`. No trate de entenderse el código dentro del método, solo obsérvese la estructura general.

```
- (NSBox *) boxWithTitle: (NSString *)titulo
{
  NSBox *box;

  box = [[NSBox alloc] initWithFrame:
                       NSMakeRect(0, 0, 100, 55)];
  [box setTitle: titulo];
  [box autorelease];

  return box;
}
```

En este ejemplo, el parámetro lleva el nombre de `titulo`. Obsérvese el uso del carácter `*`, al indicar la clase del objeto devuelto y la del parámetro recibido. Noté también los dos puntos después del nombre del método, que indican la recepción de un parámetro. Veamos, por último, un método que recibe dos parámetros de tipo `int` y que devuelve un objeto de tipo `NSWindow`. Nuevamente obsérvese solo la estructura general.

```
- (NSWindow *) makeWindowWithWidth: (int)width
                            height: (int)height
{
  NSWindow *w;

  w = [[NSWindow alloc] initWithContentRect:
                            NSMakeRect(0, 0, width, height)
                        styleMask: NSTitledWindowMask
                        backing: NSBackingStoreBuffered
                        defer: YES];

  return w;
}
```

Aquí el método usa la etiqueta `height`, seguida de dos puntos, para el segundo parámetro. Suponiendo que el outlet hacia el objeto que implementa este método se llama *object*, la utilización de este método seria de la siguiente forma:

```
NSWindow *window = [object makeWindowWithWidth: x
                                        height: y];
```

Donde `x` y `y`, son dos variables de tipo `int`, con los datos respectivos del ancho y el alto de la ventana a crear.

6.4 Métodos *accessor*

Ya hemos mencionado que los objetos tienen *atributos* o *variables de instancia*, que no son más que los datos que controlan el funcionamiento de estos. A los métodos para administrar estos datos se les conoce como métodos *accessor*, y se dividen en dos tipos. Los métodos *getter*, que se utilizan para obtener los atributos del objeto. Y los métodos *setter*, que se utilizan para establecer los atributos de este. Por ejemplo, para un objeto que tenga un atributo llamado *name*, cuya clase sea `NSString`, los métodos *getter* y *setter* serian de la siguiente forma:

```
- (NSString *) name; //Metodo getter.
- (void) setName: (NSString *)aName; //Metodo setter.
```

Aquí se muestran tal y como aparecerían en la interfaz de la clase. En Objective-C es común que el nombre del método *getter*, sea simplemente el nombre del atributo correspondiente. En este caso `-name`, y no `-getName` como se podría pensar. Para el nombre del método *setter*, se precede el nombre del atributo con el prefijo *set*, en este caso `-setName`. En una sección posterior se discutirá la implementación de estos métodos.

6.5 Métodos públicos y privados

Como ya hemos visto antes, los métodos públicos se declaran en la interfaz de la clase. Sin embargo, muchas veces es necesario tener métodos privados. Esto es, métodos que el objeto usa de forma interna y que no deben estar disponibles públicamente. Estos se declaran en el archivo de implementación, antes de la implementación de los métodos públicos. Debe agregarse un nombre o etiqueta, entre paréntesis, para identificarlos. Esta etiqueta, que debe ser única, es obligatoria, y se agrega después del nombre de la clase. Por ejemplo, es posible colocar todos los métodos privados bajo la etiqueta *Private*, o separarlos en diferentes categorías dependiendo del propósito de estos. La implementación de estos métodos, también se hace en una sección aparte, marcada con la misma etiqueta con la que fueron declarados (una sección para cada etiqueta utilizada). En el siguiente ejemplo, dos métodos privados han sido ubicados bajo la etiqueta *Private*:

```
// Declaracion de metodos privados.
@interface GemasEditorView (Private)

- (BOOL) isGSmarkupIndent: (NSString *)string;
- (BOOL) isGSmarkupBackIndent: (NSString *)string;
```

```
@end

// Implementacion de metodos privados.
@implementation GemasEditorView (Private)

- (BOOL) isGSmarkupIndent: (NSString *)string
{
  Implementacion del metodo.
}

- (BOOL) isGSmarkupBackIndent: (NSString *)string
{
  Implementacion del metodo.
}

@end

// Aqui comienza la implementacion de los metodos publicos.
@implementation GemasEditorView
  ...
  ...
```

O como en el siguiente ejemplo, donde hay dos categorías, *HandleView* y *DocumentData*:

```
/* Declaracion de metodos privados HandleView
   y DocumentData. */

@interface GemasEditorView (HandleView)

- (BOOL) removeContent;
- (void) setContent: (id)content;

@end

@interface GemasEditorView (DocumentData)

- (BOOL) setData: (NSArray *)data;
- (NSArray *) data;

@end

/* Implementacion de metodos privados HandleView
   y DocumentData. */
```

```
@implementation GemasEditorView (HandleView)

  Implementacion.

@end

@implementation GemasEditorView (DocumentData)

  Implementacion.

@end

// Aqui comienza la implementacion de los metodos publicos.

@implementation GemasEditorView
  ...
  ...
```

6.6 Ciclo de vida de los objetos y gestión de la memoria

Todos los objetos que intervienen en nuestros programas tienen un ciclo de vida. Básicamente podemos decir que son creados, utilizados y, por último, destruidos. Cuando un objeto es creado, se reserva una parte de la memoria **RAM** para almacenar al objeto. Mientras el objeto este en uso, esta parte de la memoria estará exclusivamente destinada al funcionamiento del objeto. Pero una vez que el objeto ya no sea útil, esta memoria debe ser liberada para poder ser utilizada por otros objetos o programas. De lo contrario, esta parte de la memoria no podrá volver a ser utilizada, y sera un desperdicio de recursos. Sólo podremos liberarla reiniciando la computadora.

En nuestra primera app, fue el archivo de la interfaz gráfica el que se encargo de crear los objetos de nuestra aplicación. Y ya que los archivos de interfaz se encargan de gestionar la memoria de los objetos que crean, no tuvimos que preocuparnos por esto. Sin embargo, es frecuente el tener que crear objetos con el fin de tener un mayor control sobre ellos. En situaciones como estas, nosotros como programadores, somos los responsables de manejar o administrar la memoria que estos utilizan. La manera en que se administra esta memoria, depende de la forma en que fue creado el objeto.

Existen dos formas de crear un objeto. Las cuales analizaremos a continuación, con sus respectivos procedimientos para administrar la memoria. No es necesario que el lector comprenda totalmente estos conceptos, sino solamente que tenga la idea. En los ejemplos de las secciones posteriores, todo quedara más claro.

6.6.1 Constructores convenientes (métodos factory)

Un *contructor conveniente* o método *factory* es un método de *clase* que devuelve un objeto (una instancia) de esa clase. Es una forma conveniente de crear un objeto, puesto que no tenemos que preocuparnos por la administración de la memoria que este utiliza. Sin embargo, un objeto creado mediante un constructor conveniente, solamente tiene una existencia temporal. Generalmente, el tiempo durante el cual se ejecuta el bloque de código donde este se creo. Por esto se dice que son objetos *autoliberados*. En algunos casos puede ser necesario *retener* un objeto creado a partir de un constructor conveniente. Es decir, hacer que el objeto exista durante un mayor tiempo. Un ejemplo de esto lo veremos más adelante. Los nombres de los métodos que corresponden a constructores convenientes, usualmente comienzan con el nombre de la clase (sin el prefijo `NS` o `GS`). Solamente las clases no visuales poseen constructores convenientes. Ya que no tiene sentido que un objeto visual exista por un breve tiempo. Un ejemplo de como crear un objeto mediante un constructor conveniente es el siguiente:

```
NSArray *lista = [NSArray arrayWithObjects: @"Uno",
                                            @"Dos",
                                            @"Tres",
                                            @"Cuatro",
                                            nil];
```

Un *array* es una lista de objetos. En este caso, hemos creado una lista que contiene las cadenas de texto *Uno*, *Dos*, *Tres* y *Cuatro* (`nil` solamente indica el final de la lista y no se incluye en esta). Este *array* creado de esta forma es autoliberado. Y no debemos preocuparnos por la administración de la memoria que este utiliza.

6.6.2 Asignación e inicio

Otra forma de crear un objeto es mediante la *asignación* e *inicio* de este. La *asignación* se refiere a asignarle un espacio en la memoria de la computadora. Generalmente se lleva a cabo mediante el método de *clase* `+alloc`. El *inicio* se refiere a preparar un objeto para que este sea útil. Dicho *inicio* se realiza mediante métodos de *instancia* que comienzan con el prefijo *init*. En general, las clases tienen más de un método de *inicio*, siendo el más sencillo el método `-init`. Por ejemplo, el siguiente código crea un objeto `NSMutableArray`:

```
NSMutableArray *lista = [[NSMutableArray alloc] init];
```

La clase `NSMutableArray` permite crear listas modificables, mientras que la clase `NSArray` crea listas no modificables. En este ejemplo se ha creado una lista modificable vacía. El método de clase `+new` puede utilizarse para realizar la *asignación* e *inicio*, mediante el método `-init`, en un sólo paso:

```
NSMutableArray *lista = [NSMutableArray new];
```

Por lo que este código equivale al mostrado previamente. Los objetos creados de esta forma, permanecerán en memoria hasta que nosotros les digamos que liberen el espacio en esta. Para hacerlo utilizamos el método de *instancia* `-release`. Por ejemplo, para liberar la lista creada anteriormente tendríamos:

```
[lista release];
```

Esto destruye al objeto `lista`. También es posible liberar objetos utilizando la función `RELEASE()` (en mayúsculas):

```
RELEASE(lista);
```

Aunque actualmente no se recomienda más el uso de esta función y sólo se muestra como referencia. Al diseñar un programa, es siempre recomendable que los objetos sean creados y liberados dentro de un mismo objeto. Esto es, si un objeto *A* crea un objeto *B*, dicho objeto *A* es también el responsable de liberar al objeto *B*. Esto se comprenderá mejor con los ejemplos de la siguiente sección.

6.7 Ejemplos

En esta sección vamos a crear diferentes *Tools* para ejemplificar lo visto anteriormente. Como ya hemos mencionado, una *Tool* es un programa de línea de comandos.

6.7.1 Hola mundo!

Nuestro primer ejemplo es bastante trivial, ya que simplemente imprime el texto '`Hola mundo!`'. Pero su objetivo es mostrarnos la estructura básica de una *Tool*. Supondremos que la carpeta de nuestro proyecto lleva por nombre *Hola*. El único archivo de código de este proyecto, al que llamaremos '`HolaMain.m`', contiene el siguiente código. Obsérvese que en la primera linea importamos la cabecera '`Foundation/Foundation.h`', que nos permite usar todas las clases del framework *Base*:

```
#import <Foundation/Foundation.h>

int
main(int argc, const char *argv[])
{
  NSLog(@"Hola mundo!");
```

```
  return 0;
}
```

Este archivo contiene la función *main* que todo programa en Objective-C debe tener. Aquí hacemos uso de la función `NSLog()`, la cual imprime en la terminal la cadena de texto que recibe como parámetro. Y seguidamente se retorna el valor numérico 0. En general, la función *main* se declara de tal forma que retorne un número entero, comúnmente el 0. De esta forma, si un programa retorna el valor 0 al momento de cerrarse, es indicio de que el programa se cerro correctamente. Cualquier otro valor devuelto indicaría que el programa se cerro abruptamente debido a un error. Ahora para el archivo de construcción, el cual siempre debe tener por nombre '`GNUmakefile`', tenemos, llamando a nuestra herramienta *Hola*:

```
include $(GNUSTEP_MAKEFILES)/common.make

# Nombre del tool
TOOL_NAME = Hola

# Otros archivos
Hola_OBJC_FILES += HolaMain.m

# Makefiles
include $(GNUSTEP_MAKEFILES)/tool.make
```

En la última linea se indica el tipo de proyecto que queremos construir, en este caso una herramienta ('`tool.make`'). Abramos una terminal y ubiquémonos en el directorio *Hola* de nuestro proyecto, de ser necesario ejecutemos el *script* de *GNUstep*, y entonces ejecutemos el comando `make`. Terminada la compilación, podemos ejecutar nuestra herramienta escribiendo la ruta del ejecutable. A diferencia de una app, una tool se compila en un subdirectorio llamado '`obj`'. Por lo tanto tenemos:

```
./obj/Hola
2013-05-21 23:30:58.524 Hola[2665] Hola mundo!
```

El punto al inicio indica que se trata de una ruta relativa al directorio en que nos encontramos, en este caso el directorio de nuestro proyecto. La función `NSLog()` imprime, junto con nuestra cadena de texto, la fecha, la hora y el nombre del programa que se esta ejecutando. De aquí en adelante no volveremos a utilizar la función `NSLog()`, debido a que imprime mas información de la deseada. Sin embargo, esta función se utiliza mucho durante el desarrollo de grandes programas, con el fin de imprimir el valor de variables que permitan comprobar que el programa funciona según lo esperado.

6.7.2 Días de nuestra era

Nuestra segunda herramienta simplemente imprimirá el número de días que han transcurrido en nuestra era (calendario *Gregoriano*), tomando en cuenta los años bisiestos y demás correcciones. Supondremos que la carpeta de nuestro proyecto lleva por nombre *Era*. Entonces, para el único archivo de código de este proyecto, al que llamaremos '`EraMain.m`', tenemos:

```
#import <stdio.h>
#import <Foundation/Foundation.h>

int
main(int argc, const char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];

  NSCalendarDate *date = [NSCalendarDate calendarDate];
  printf("Han transcurrido %d dias \n",
         [date dayOfCommonEra]);

  [pool release];
  return 0;
}
```

Primero se crea un objeto `NSAutoreleasePool` utilizando el método de clase `+new`. Estos objetos son los encargados de manejar la memoria de los objetos autoliberados y es el primer objeto que se debe crear en toda herramienta que haga uso de objetos, sean autoliberados o no. Muchos objetos utilizan, internamente, objetos autoliberados. Seguidamente creamos un objeto autoliberado de la clase `NSCalendarDate`, utilizando para ello el método *factory* `+calendarDate`. La clase `NSCalendarDate` nos permite manejar todo lo relativo a fechas. El método de instancia `-dayOfCommonEra`, utilizado a continuación, retorna como un número entero la cantidad de días transcurridos en la era actual. Este dato, junto al mensaje correspondiente, se imprime entonces con la función `printf()` (por eso se importa la cabecera '`stdio.h`'). Esta función ya se ha visto en un capítulo anterior. Y por último liberamos el objeto *pool*, el cual no es autoliberado. Por supuesto, no es necesario liberar al objeto *date*. Ahora el archivo de construcción, llamando a esta herramienta *Era*, es:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Era

Era_OBJC_FILES += EraMain.m
```

```
include $(GNUSTEP_MAKEFILES)/tool.make
```

Una vez compilada la herramienta la ejecutamos para ver el resultado:

```
./obj/Era
Han transcurrido 735010 dias
```

6.7.3 Creando una clase

En este ejemplo vamos a crear una sencilla clase con un método de clase y dos métodos de instancia. Por simplicidad vamos a escribir todo el código en un mismo archivo. Esto es, vamos a escribir la interfaz y la implementación de esta clase en el mismo archivo de la función *main*. Sin embargo, para herramientas mas complejas, se recomienda utilizar diferentes archivos. Llamaremos a esta herramienta *Clase* y el archivo de código, al que llamaremos '`ClaseMain.m`', es el siguiente:

```
#import <stdio.h>
#import <Foundation/Foundation.h>

/* Basado en un ejemplo de "Objective-C Language and
 * GNUstep Base Library Programming Manual" de Francis
 * Botto, Richard Frith-Macdonald, Nicola Pero y
 * Adrian Robert. Y en otro de "GNUstep Tutorial" de
 * Yen-Ju Chen.
 */


@interface MyClass : NSObject
{
}
+ (void) classString;
- (void) sayHello;
- (void) sayHelloTo: (NSString *)name;
@end

@implementation MyClass
+ (void) classString
{
  printf("Texto de la clase MyClass. \n");
}

- (void) sayHello
{
  printf("Hola Mundo. \n");
```

```
}

- (void) sayHelloTo: (NSString *)name
{
  printf("Hola %s. \n", [name cString]);
}
@end

int
main(int argc, const char *argv[])
{
  id pool = [[NSAutoreleasePool alloc] init];

  NSString *nombre = @"GNUstep!";

  [MyClass classString];

  MyClass *objeto = [[MyClass alloc] init];
  [objeto sayHello];
  [objeto sayHelloTo: nombre];

  [objeto release];
  [pool release];

  return 0;
}
```

Hemos llamado a nuestra clase *MyCLass*. Y como se ve, no tiene variables de instancia. En los tres métodos hacemos uso de la función `printf()`, para imprimir los mensajes correspondientes. En el método de instancia `-sayHelloTo:`, usamos el método de instancia `-cString` para convertir el dato *name*, un dato `NSString`, a una cadena de texto en lenguaje C. Esto porque la función `printf()` no puede manejar objetos `NSString`. Ya en la función *main*, primero creamos un objeto `NSString` estático, llamado *nombre*, y que contiene la cadena de texto *GNUstep!*. Seguidamente ejecutamos el método de clase `+classString`, de nuestra clase. Y a continuación creamos una instancia de esta, a la que llamamos *objeto*, para luego ejecutar sus dos métodos de instancia y posteriormente liberarlo. Por supuesto, no es necesario liberar el objeto estático *nombre*, ya que este fue creado en tiempo de compilación.

El archivo de construcción, llamando a nuestra herramienta *Clase*, es el siguiente:

```
include $(GNUSTEP_MAKEFILES)/common.make
```

```
TOOL_NAME = Clase

Clase_OBJC_FILES += ClaseMain.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

Una vez compilada la herramienta la ejecutamos para ver el resultado:

```
./obj/Clase
Texto de la clase MyClass.
Hola Mundo.
Hola GNUstep!.
```

6.8 Como se crea un objeto

En esta sección vamos a profundizar en la forma en que los objetos son creados y destruidos. Comencemos por mencionar que todos los objetos son creados mediante asignación e inicio. Incluso cuando utilizamos un constructor conveniente el objeto es creado de esta forma, lo que sucede es que el constructor se encarga de esto por nosotros.

Para comprender como se crea y destruye un objeto, comencemos utilizando una analogía con las clases que componen una casa. Para simplificar, supongamos que tenemos tres clases: *cimientos*, *paredes* y *techo*. Donde la clase *techo* es una subclase de la clase *paredes*, y esta es a su vez una subclase de la clase *cimientos*. Esto tiene sentido, puesto que para poner un techo primero se necesitan las paredes, y para poner estas se necesitan los cimientos. Supongamos que estas son las únicas clases en nuestro framework, y que deseamos crear una nueva clase llamada *antena*. Puesto que la antena debe ir colocada en el techo, lo mas lógico es que la clase *antena* sea una subclase de la clase *techo*. Imaginemos hora que creamos una instancia de nuestra clase *antena*. Surgen entonces las siguientes preguntas: ¿donde va a estar colocada nuestra antena? ¿no necesitamos un techo para colocarla? ¿y no necesitamos paredes para colocar ese techo? ¿y cimientos para esas paredes?. Se comprende entonces que al crear un objeto, este se debe encargar de crear todos los demás objetos que necesite para funcionar correctamente. En el caso de nuestro objeto antena, este debe crear un objeto (una instancia) de la clase *techo*. Este techo a su vez debe crear una instancia de la clase *paredes*. Y estas a su vez crear una instancia de la clase *cimientos*. De esta forma, nuestro objeto antena se asegura de construir toda la infraestructura necesaria para poder funcionar. Se comprenderá también que al momento de liberar a este objeto antena, este debe asegurarse de liberar el techo, las paredes y los cimientos que se crearon para su funcionamiento.

Veamos ahora como sucede esto en Objective-C y los mensajes involucrados. Cuando un objeto es creado, el método de clase `+alloc` se encarga de

reservar la memoria necesaria para el funcionamiento del objeto, tomando en cuenta los demás objetos que deberán ser creados para el funcionamiento de este. Hecho esto, se inicia el objeto con algún método para tal fin. Dicho método se encarga, entre otras cosas, de establecer valores para las variables de instancia del objeto. Y seguidamente, de ejecutar algún método de inicio de la *instancia* de la superclase. Este se encarga, a su vez, de establecer valores para las variables de la *instancia* de la superclase, y seguidamente de ejecutar el método de inicio de la *instancia* de *su* superclase. Y así sucesivamente hasta llegar a la clase raíz. Volviendo a la analogía, las variables de instancia corresponderían a datos como el color, el tamaño y la forma de los objetos (del techo, las paredes, etc.).

Una vez un objeto ya no es útil, este es liberado. Y esta acción ejecuta el método `-dealloc` de dicho objeto. Este método se encarga de liberar, de ser necesario, las variables de instancia del objeto, y seguidamente de ejecutar el método `-dealloc` de la *instancia* de su superclase. Este a su vez, después de liberar las variables de instancia (de ser necesario), ejecuta el método `-dealloc` de la *instancia* de *su* superclase, y así sucesivamente hasta llegar a la clase raíz. Volviendo a la analogía, al liberar nuestro objeto antena, este libera al objeto techo. Este techo a su vez se encarga de liberar al objeto paredes. Y este objeto paredes, finalmente, libera al objeto cimientos. De esta forma, todos los objetos creados por nuestro objeto antena son liberados al ser liberado este, y la memoria queda entonces disponible para otros objetos o programas.

Ahora contestemos la pregunta ¿de que sirve saber esto?. Como veremos mas adelante, es común toparse con la necesidad de redefinir o crear métodos de inicio. Y es necesario saber como llevar a cabo esto, para evitar que nuestro programa cometa un grave error. Todo esto lo pondremos en práctica mas adelante.

6.8.1 Los métodos de inicio

Anteriormente dijimos que los objetos pueden tener varios métodos de inicio. La razón de esto es ofrecerle al programador una mayor flexibilidad al momento de crear objetos. Estos métodos tienen una jerarquía, establecida en relación a la cantidad de parámetros que reciben. Estando en la parte más alta el método de inicio que más parámetros recibe, y en la parte más baja el que menos parámetros recibe o el que no recibe ningún parámetro. El método de inicio que esta en la parte más alta de la jerarquía es llamado *método designado de inicio*. En la sección anterior, vimos que los métodos de inicio se encargan de establecer valores para las variables de instancia del objeto, y de ejecutar el método de inicio de la instancia de la superclase. En realidad esto sólo lo hacen los métodos designados de inicio. El resto, se limitan a ejecutar el método de inicio que se encuentre arriba de ellos en la jerarquía. El siguiente esquema aclara esto:

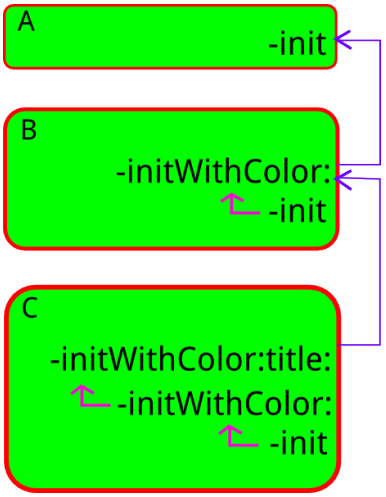

Figura 6-1. Jerarquía de los métodos de inicio.

En este esquema se muestran tres clases. La clase *C* es una subclase de la *B*, y esta a su vez es una subclase de la *A*. La clase *A* sólo tiene un método de inicio, `-init`, que no recibe ningún parámetro, y que es el método designado. La clase *B* tiene, además del método de la clase *A*, el método `-initWithColor:`, que recibe un parámetro y que es el método designado. La clase *C* tiene, además de los métodos de la clase *B*, el método `-initWithColor:title:`, el cual recibe dos parámetros y es el método designado.

Obsérvese que el método designado de inicio de una clase dada, ejecuta el método designado de su superclase. Mientras que el resto de métodos, ejecutan el método de inicio que se encuentre arriba de ellos en la jerarquía (en la misma clase). La razón de esto, es asegurar que la instancia de una clase inicie correctamente todas las variables de instancia. Es decir, que se le asignen valores a todas ellas. Por ejemplo, al iniciar un objeto de la clase *C* mediante `-initWithColor:`, este método ejecutara el método `-initWithColor:title:` pasándole como primer parámetro el color proveído, y como segundo parámetro un título por defecto. De esta forma, el método `-initWithColor:title:` iniciara el objeto con el color y el título proveídos (el título por defecto).

Esta es también la razón de que una subclase implemente nuevamente (redefina) los métodos de inicio de su superclase. De otra forma, a algunas variables de instancia no se les asignaría ningún dato. Por ejemplo, si la clase *C* no implementara el método `-initWithColor:`, al instanciar un objeto de esta clase mediante dicho método, este se ejecutaría en la superclase. Pero este método, en la superclase, no asignaría ningún título por defecto, y dicha variable de instancia quedaría sin ningún dato asignado. Recuérdese que un objeto no sólo responde a los métodos de su clase, sino también a los métodos *heredados*.

6.8.2 Implementando y redefiniendo métodos de inicio

Al crear una subclase y tener la necesidad de iniciar variables de instancia adicionales, lo común es redefinir el método designado de inicio de la superclase. Si la superclase no posee un método designado de inicio, deberemos buscar mas arriba en la jerarquía hasta encontrar uno. Ya sea que estemos redefiniendo el método designado de inicio o que estemos escribiendo un método designado especifico para nuestra clase, debemos asegurarnos siempre de llamar al método designado de inicio de la superclase (o al primero que aparezca en la jerarquía). Para realizar esto, Objective-C provee los receptores `self` y `super`, los cuales hacen referencia, respectivamente, a la instancia de la clase en cuestión y a la instancia de la superclase. Por ejemplo, la redefinición del método designado de inicio `-init` sería de la siguiente forma:

```
- (id) init
{
  self = [super init];

  // Aqui van nuestras modificaciones.

  return self;
}
```

En la primera linea se ejecuta el método designado de inicio de la superclase, utilizando para ello el receptor `super` que apunta a la instancia de esta. El objeto devuelto se asigna entonces al receptor `self`, que apunta a la instancia de la clase en cuestión. Y seguidamente se agregan las modificaciones al método de inicio, para finalmente retornar el objeto creado.

Algunas veces deberá verificarse si el método designado de la superclase pudo crear el objeto. Por ejemplo, cuando el objeto debe ser creado a partir de un archivo, puede darse el caso de que el archivo no se encuentre y que por lo tanto el objeto no pueda ser creado. El siguiente código es un ejemplo de esto:

```
- (id) initWithContentsOfFile: (NSString *)path
{
  self = [super initWithContentsOfFile: path];

  if (self)
    {
      // Aqui van nuestras modificaciones.
    }

  return self;
}
```

En este caso, si el objeto no puede ser creado no se ejecutaran las modificaciones. Ya que esto no tiene sentido si el objeto no pudo crearse. En caso de fallar la creación del objeto, `return` devolverá el receptor `self` con valor `nil`, el valor retornado por `-initWithContentsOfFile:` cuando no se puede leer el archivo.

Al redefinir un método de inicio que no sea el designado, se debe llamar al método de inicio que este arriba en la jerarquía (en la misma clase). Por ejemplo, la redefinición del método `-initWithColor:` en la clase *C* de la figura 6-1, podría ser de la siguiente forma:

```
- (id) initWithColor: (NSColor *)aColor
{
  self = [self initWithColor: aColor title: @"no-title"];

  // Modificaciones.

  return self;
}
```

Al crear una clase propia de nuestro programa, no es obligatorio redefinir todos los métodos de inicio de la superclase, únicamente el método designado. Siempre y cuando las instancias de dicha clase, se creen mediante el uso de este método. De lo contrario, como ya se menciono anteriormente, no se iniciaran correctamente los objetos. Sin embargo, en proyectos como librerías o frameworks, que serán usados por terceras personas, si debe tenerse especial cuidado en redefinir todos los métodos de inicio.

En el caso de que las variables de instancia que hayamos adicionado a nuestra clase sean objetos no autoliberados, debemos asegurarnos de liberar estas al momento de que sean liberadas las instancias de nuestra clase. Para ello, es necesario redefinir el método de instancia `-dealloc`. Por ejemplo, si hemos añadido una variable de instancia llamada *date*, la redefinición de dicho método sería de la siguiente forma:

```
- (void) dealloc
{
  [date release];
  [super dealloc];
}
```

Aquí primero se liberan las variables de instancia añadidas, en este caso *date*. Y luego se ejecuta el método `-dealloc` en la instancia de la superclase, utilizando para ello el receptor `super`. El método `-dealloc` únicamente debe usarse en la forma `[super dealloc]` al redefinir un método `-dealloc`. Y nunca para liberar un objeto en particular, para eso esta el método `-release`.

6.8.3 autorelease, retain y release

Ya hemos visto que los constructores convenientes devuelven objetos autoliberados. Sin embargo, en ocasiones, necesitamos que dichos objetos no sean autoliberados, sino que sean *retenidos* en la memoria. Esto puede realizarse ejecutando el método de instancia `-retain`. Por ejemplo, en el siguiente código, *retenemos* en la memoria el *array* creado por el constructor conveniente:

```
NSArray *lista;
lista = [NSArray arrayWithObjects: @"A", @"B", nil];
[lista retain];
```

Por supuesto, un objeto retenido en memoria de esta forma, debe posteriormente ser liberado mediante el método `-release`. Lo contrario también es posible. Es decir, un objeto creado mediante asignación e inicio (retenido en memoria), puede volverse un objeto autoliberado mediante el método de instancia `-autorelease`. Por ejemplo, para una hipotética clase *MyClass*, tendríamos:

```
MyClass *data = [MyClass new];
[data autorelease];
```

De esta forma, no debemos preocuparnos por liberar al objeto `data`.

Cuando un objeto le envía el mensaje `-retain` a otro objeto, le esta diciendo a ese objeto que permanezca en memoria por que necesita interactuar con el. Una vez que termine de interactuar con dicho objeto, le envía un mensaje `-release` para informarle de que *puede* destruirse y liberar la memoria. Ya que un objeto puede interactuar con mas de un objeto a la vez, es posible que reciba varios mensajes `-retain` o `-release` provenientes de diferentes objetos. Para administrar todos estos mensajes, los objetos tienen un contador interno, el cual tiene el valor de uno al momento de ser creado el mismo, no importando si es creado mediante asignación e inicio o mediante un método *factory*. Dicho contador incrementa su valor en uno con cada mensaje `-retain`, mientras que los mensajes `-release` lo disminuyen en uno. Y solo cuando dicho contador llega a tener un valor de cero, es cuando el objeto se destruye y libera la memoria que le había sido asignada. El siguiente código ejemplifica esto:

```
// Creamos el objeto, el contador vale 1.
MyClass *object = [MyClass new];

// Retenemos al objeto, el contador vale 2.
[object retain];
```

```
// Enviamos el mensaje -release, el contador vale 1.
[object release];

// Enviamos nuevamente el mensaje -release, el contador
// vale 0. Por lo que el objeto es destruido y se
// libera la memoria.
[object release];
```

Este ejemplo no tiene ningún uso practico. Su objetivo es mostrar como el contador aumenta y disminuye su valor con los mensajes `-retain` y `-release`, respectivamente. Sin embargo, en el ejemplo de la siguiente sección, veremos un caso típico en el cual un objeto debe retenerse en memoria.

La ejecución del método `-retain` retorna un objeto, el objeto que se esta reteniendo. Dependiendo del caso, puede ser necesario asignar este objeto devuelto a alguna variable.

6.9 Otro ejemplo

En este ejemplo crearemos una clase con dos variables de instancia. Llamaremos a la clase *Say* y las variables de instancia representaran la edad y el nombre de una persona. Llamando al archivo '`Say.m`', el código de esta herramienta es el siguiente (se han agregado varios comentarios):

```
#import <stdio.h>
#import <Foundation/Foundation.h>

@interface Say : NSObject
{
  // Variables de instancia
  NSUInteger age;
  NSString *name;
}
// Metodos Setter
- (void) setAge: (NSUInteger)aNumber;
- (void) setName:(NSString *)aName;

// Metodos Getter
- (NSUInteger) age;
- (NSString *) name;

// Inicio
- (id) init;

// Impresion de variables
- (void) printVars;
```

```
@end

@implementation Say
- (id)init
{
  self = [super init];

  if (self)
    {
      age = 0;
      name = @"Ninguno";
      //name = nil; //Establece 'name' a nada.
    }

  return self;
}

/* Metodo de liberacion. Este metodo no va en la interfaz
 * ya que nunca se llama directamente. Se ejecuta cuando la
 * instancia es liberada.
 */
- (void) dealloc
{
  [name release];
  [super dealloc];
}

// Metodos Setter
- (void) setAge: (NSUInteger)aNumber
{
  age = aNumber;
}

- (void) setName:(NSString *)aName
{
  /* El macro 'ASSIGN' libera primero el objeto en 'name'
   * y luego retiene una instancia del objeto 'aName' en el
   * puntero 'name'.
   */
  ASSIGN(name, aName);

  /* Es incorrecto hacer:

  name = [aName retain];
```

```
  */
}

// Metodos Getter
- (NSUInteger) age
{
  return age;
}

- (NSString *) name
{
  return name;
}

// Imprimir las variables de instancia
- (void) printVars
{
  printf("El nombre es %s y la edad %d \n",
         [name cString], age);
}
@end

int main (void)
{
   id pool = [NSAutoreleasePool new];

   Say *speaker;
   NSString *boy = @"Pancho";
   NSString *girl = @"Ana";

   speaker = [[Say alloc] init];

   [speaker printVars];
   [speaker setAge: 25];
   [speaker setName: boy];
   [speaker printVars];
   [speaker setAge: 28];
   [speaker setName: girl];
   [speaker printVars];

   [speaker release];
   [pool release];

   return 0;
}
```

Se redefine el método de inicio `-init`, con el fin de asignar valores iniciales a las variables de instancia al momento en que se cree una instancia de la clase. Esto es lo recomendado. Aquí le hemos asignado la cadena de texto *Ninguno* a la variable de instancia `name`. Aunque también pudo haberse asignado `nil`, como se muestra en el comentario. Obsérvese que se ha utilizado el tipo de dato `NSUInteger` para la edad, en lugar de `int`. `NSUInteger` es el tipo utilizado por *GNUstep* para representar enteros positivos, y es el que utilizaremos de aquí en adelante para representarlos.

En general, cuando las variables de instancia son objetos, es recomendable retener estas en la memoria. En este ejemplo no hemos retenido la cadena de texto `name`, ya que se trata de una cadena estática creada en tiempo de compilación. Sin embargo, otros objetos deben ser retenidos con `-retain`. Por otro lado, los nombres proveídos mediante el método *setter* `-setName`, si deberemos retenerlos en memoria.

En la redefinición del método `-dealloc` liberamos las variables de instancia (objetos) que estén retenidas. En este caso, únicamente tenemos a la variable `name`, ya que la variable `age` no es un objeto.

En el método *setter* `-setName`, se utiliza el *macro ASSIGN*, el cual libera primero el objeto en `name` y luego retiene una instancia del objeto `aName` en el puntero `name`. Este macro equivale (más o menos) al siguiente código:

```
if (name != nil)
  {
    [name release];
  }

if (aName != nil)
  {
    name = [aName retain];
  }
else
  {
    name = nil;
  }
```

Primero se libera el nombre anterior, si lo hay. Y luego se retiene el nuevo nombre en el puntero `name`. Si el nuevo nombre es `nil`, este valor es el asignado a `name`.

En el comentario se indica que escribir simplemente `name = [aName retain];` es incorrecto. ¿Por que? Porque de esta forma estamos reteniendo el nuevo nombre, pero sin liberar el nombre anterior, el cual se quedara en la memoria. Por esta razón se recomienda el uso del macro *ASSIGN*, que nos facilita esta tarea.

Llamando a nuestra herramienta *Say*, el archivo de construcción es el siguiente:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Say
Say_OBJC_FILES += Say.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

Una vez compilada la herramienta la ejecutamos para ver el resultado:

```
El nombre es Ninguno y la edad 0
El nombre es Pancho y la edad 25
El nombre es Ana y la edad 28
```

Modifíquese el programa para que el valor inicial de `name` sea `nil` (hay que compilar nuevamente el proyecto). Y ejecútese la herramienta para ver la diferencia.

`nil` se utiliza para indicar que un puntero no tiene asignado objeto alguno. En otras palabras, se utiliza para indicar que el puntero esta vacío.

6.10 Subclases

Ya que una subclase responde a todos los métodos de su superclase, es de esperarse que una instancia de esta pueda comportarse (y ser tratada) como una instancia de la superclase. Para entender esto veamos un ejemplo. Todos los objetos de clase `NSView` (o subclases de esta) responden al método `-window`, el cual retorna la ventana donde se encuentre el *view* en cuestión. De acuerdo a la documentación, el objeto retornado es de la clase `NSWindow`. De esta forma, el siguiente código obtiene el título de dicha ventana:

```
NSString *title = [[aView window] title];
```

Sin embargo, puede darse el caso de que el *view* se encuentre en un panel y de que queramos ejecutar un método de este. Los paneles son objetos de la clase `NSPanel`, la cual es una subclase de la clase `NSWindow`. Como ejemplo veamos el siguiente código:

```
BOOL works = [[aView window] worksWhenModal];
```

El método `-worksWhenModal` esta implementado en la clase `NSPanel`. Y como en este caso el *view* se encuentra en un panel, esperaríamos que este código funcionara correctamente. Salvo por el hecho de que el compilador trata al objeto retornado por el método `-window`, como a uno de clase `NSWindow`, y que por tanto no responde al método `-worksWhenModal`. Al compilar este código obtendríamos un *Warning*, advirtiéndonos de que

el objeto devuelto por `-window` no responde al método `-worksWhenModal`. Dependiendo del método que se trate de ejecutar, situaciones como esta podrían causar un error al momento de compilar el proyecto. Incluso en el caso de que este compile, podría suceder que el programa fallara al momento de ejecutar dicho método, cerrándose abruptamente.

Una forma de evitar esto, es reescribir el código de tal forma que quede claro cual es la clase del objeto. Por ejemplo:

```
NSPanel *aPanel = [aView window];
BOOL works = [aPanel worksWhenModal];
```

O especificando entre paréntesis la clase del objeto que devolverá la ejecución del método `-window`:

```
BOOL works = [(NSPanel *)[aView window] worksWhenModal];
```

6.11 La directiva `@class`

Consideremos el siguiente archivo de interfaz:

```
#import <AppKit/AppKit.h>

@interface MyClass : NSObject
{
  NSColor *color;
  NSView  *view;
}

//Setter
- (void) setView: (NSView *)aView;
- (void) setColor: (NSColor *)aColor;

//Getter
- (NSView *) view;
- (NSColor *) color;

@end
```

Aquí se importa la cabecera `AppKit/AppKit.h` que incluye todas las clases del framework *GUI* y algunas del framework *Base*. Sin embargo, especialmente en proyectos de librerías o frameworks, se recomienda importar únicamente las cabeceras de aquellas clases que son realmente necesarias. En este caso, la única clase realmente necesaria es la clase `NSObject`. Ya que la clase `MyClass` es una subclase de esta. Las otras clases, `NSView` y `NSColor`,

no se utilizan en absoluto. Aparecen simplemente para indicar la clase de los objetos involucrados en los métodos *setter* y *getter*. Sin embargo, si no se importan las cabeceras de estas clases, el compilador no podrá reconocer que `NSColor` y `NSView` son clases. Aquí es donde interviene la directiva `@class`, permitiéndonos indicarle al compilador que tanto `NSColor` como `NSView` son clases. Utilizando esta directiva, la interfaz quedaría de la siguiente forma:

```
#import <Foundation/NSObject.h>

@class NSColor, NSView;

@interface MyClass : NSObject
{
  NSColor *color;
  NSView *view;
}

//Setter
- (void) setView: (NSView *)aView;
- (void) setColor: (NSColor *)aColor;

//Getter
- (NSView *) view;
- (NSColor *) color;

@end
```

Aquí se importa sólo la cabecera de la clase `NSObject`. Y con la directiva `@class` se indica que `NSColor` y `NSView` son clases. También se pudo haber utilizado una directiva por cada clase, esto es:

```
@class NSColor;
@class NSView;
```

Por supuesto, las cabeceras de estas clases deben importarse en la implementación de esta clase. Ya que allí si se utilizaran instancias de estas.

Lectura Recomendada: El pequeño documento *Clases Básicas de la librería GNUstep Base* presenta y ejemplifica el uso de las clases `NSString`, `NSArray` y `NSDictionary`. También trata el tema de los archivos *Property List*, de los que haremos uso mas adelante. Este puede obtenerse en el enlace `http://gnustep.wordpress.com/`, en la sección *Documentos y Manuales*.

7 Otra app de ejemplo

En este capítulo se presenta un ejemplo a modo de integrar todo lo visto en los capítulos anteriores. Este se presenta inicialmente en una forma sencilla y luego se va modificando para hacerlo cada vez mas complejo.

7.1 Un cronómetro

Construyamos primero la interfaz gráfica del cronómetro. Abramos *Gorm* y seleccionemos en el menú Document → New Application. Ahora agreguemos una caja de texto a la ventana, estableciendo en sus propiedades una alineación centrada, que no sea editable y que no sea seleccionable. Asumiremos que la interfaz gráfica se guarda con el nombre *Chronometer*. Este cronómetro debe comenzar su funcionamiento al momento de ser lanzada la aplicación y en la caja de texto deberá mostrar el avance de los segundos. Un posible modelo para esta aplicación se presenta en la figura 7-1.

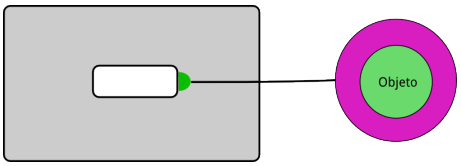

Figura 7-1. Modelo del cronómetro.

Así, necesitamos un objeto que controle el avance del tiempo y que comunique este avance a la caja de texto. Crearemos entonces una clase a la que llamaremos *Chronometer*, y que sera una subclase de la clase raíz `NSObject`. Esta clase tendrá un outlet para comunicarse con la caja de texto y una variable de instancia para llevar el conteo de los segundos, así como un método encargado de llevar a cabo dicho conteo. Por lo tanto, la interfaz de esta clase, a la que llamaremos '`Chronometer.h`', es la siguiente:

```
#import <AppKit/AppKit.h>

@interface Chronometer : NSObject
{
  NSUInteger seconds;

  //Outlets
  id textBox;
}
- (void) seconds: (NSTimer *)aTimer;
@end
```

En el archivo de implementación podríamos redefinir el método de inicio `-init` para asignar cero como valor inicial para la variable de instancia, y para

crear un objeto `NSTimer` encargado de contar los segundos. Sin embargo, en su lugar, haremos uso del método `-awakeFromNib`. Este método es ejecutado en todos aquellos objetos que son creados o instanciados por un archivo '`gorm`', al momento de que *GNUstep* ha terminado de recrear la interfaz gráfica a partir de dicho archivo. Es decir, se ejecuta en aquellos objetos que *Gorm* muestra en la sección *Objects* del documento '`gorm`'. Por lo tanto, este método solamente puede ser utilizado en objetos que son instanciados de esta forma. Y es bastante útil, ya que nos evita el tener que redefinir métodos de inicio. `-awakeFromNib` no debe declararse en la interfaz de la clase.

Ahora bien, los objetos `NSTimer` envían un mensaje a un objeto especificado después de que ha transcurrido un intervalo de tiempo indicado. Para crear este objeto utilizamos el constructor conveniente `+scheduledTimerWithTimeInterval: target: selector: userInfo: repeats:`. El primer parámetro es el intervalo de tiempo en segundos. El segundo y tercer parámetros son, respectivamente, el objeto al que se le enviara el mensaje y el método a ejecutar. El cuarto es un tercer objeto con el cual pueda ser necesario interactuar o que contenga información necesaria para el destinatario (puede ser `nil`). Y el último es un valor booleano, que indica si el mensaje se debe enviar de forma repetida o no. Como en este caso queremos que el mensaje se envié después de cada segundo, pasamos como parámetro `YES`. Este constructor conveniente devuelve un objeto `ŃSTimer` ya en marcha (autoliberado). Sin embargo, ya que no necesitamos interactuar con el, no lo asignamos a puntero alguno. De esta forma, el código del archivo de implementación, al que llamaremos '`Chronometer.m`', es el siguiente:

```
#import "Chronometer.h"

@implementation Chronometer

- (void) awakeFromNib
{
  seconds = 0;

  [NSTimer scheduledTimerWithTimeInterval: 1
           target: self
           selector: @selector(seconds:)
           userInfo: nil
           repeats: YES];
}

- (void) seconds: (NSTimer *)aTimer
{
  [textBox setIntValue: ++seconds];
```

```
}
@end
```

El método `-seconds:`, el cual es ejecutado por el objeto `NSTimer`, debe recibir como parámetro un objeto `NSTimer` tal y como se muestra. El código en este método es fácil de entender y no necesita explicación.

Creados estos archivos, procedemos a cargar y crear una instancia de nuestra clase *Chronometer* en el archivo de interfaz de nuestra app. Y de conectar el outlet a la caja de texto. Hecho esto, creamos el archivo de construcción. Llamando *AppMain.m* al archivo que contiene la función `main` (y que no mostramos aquí), tenemos:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = Chronometer

Chronometer_MAIN_MODEL_FILE = Chronometer.gorm

Chronometer_RESOURCE_FILES = Chronometer.gorm

Chronometer_HEADER_FILES = Chronometer.h

Chronometer_OBJC_FILES = Chronometer.m

Chronometer_OBJC_FILES += AppMain.m

include $(GNUSTEP_MAKEFILES)/application.make
```

Después de compilar nuestro proyecto, ejecutemos la app para verificar que esta funciona según lo esperado.

7.1.1 Agregando un botón Detener/Reanudar

Procedamos ahora a modificar esta app para agregarle un botón que nos permita detener o reanudar la marcha del cronómetro. Para ello utilizaremos un botón tipo *Toggle*, que a diferencia de los botones normales puede tener dos estados, mostrando un título diferente para cada uno de estos. Los estados se representan por `NSOffState` y `NSOnState`. Y de acuerdo a las recomendaciones para el diseño de una interfaz gráfica, dichos botones no deben ejecutar método alguno. En su lugar, el programa debe verificar el estado del botón y actuar acorde a este. Agreguemos entonces un botón a nuestra interfaz gráfica, colocándolo debajo de la caja de texto. Ahora en el *Inspector* pongámosle como título *Reanudar*, este será el texto mostrado en el estado `NSOffState`. Y como título alternativo pongámosle *Detener*, el texto que se mostrara en el estado `NSOnState`. Siempre de acuerdo a las recomendaciones de diseño, el título de un botón de dos estados debe indicar

el efecto que tendrá cuando el usuario de un clic sobre el y no el efecto que tiene actualmente. Ahora seleccionemos como tipo (*type*) del botón *Toggle*.

Debemos ahora agregar un outlet a nuestra clase *Chronometer*, el cual estará conectado al botón. De tal forma que nuestra clase pueda conocer el estado en que se encuentra este. Seleccionemos entonces nuestra clase *Chronometer* en la sección *Classes* de nuestro documento 'gorm'. Y añadamos, en el *Inspector*, un outlet con el nombre *button*. Hecho esto, regresemos a la sección *Objects* y realicemos la conexión entre este nuevo outlet y el botón. Y guardemos los cambios a nuestro documento.

Debemos, asimismo, agregar dicho outlet al archivo de interfaz de nuestra clase:

```
//Outlets
id textBox;
id button;
```

En la implementación de nuestra clase, haremos que el botón este inicialmente en el estado `NSOnState`, de tal forma que el cronómetro este en marcha al ser lanzada la app. Esto, por supuesto, lo llevamos a cabo en el método `-awakeFromNib`. Y en el método `-seconds` verificamos primero el estado del botón, y solamente si este se encuentra en el estado `NSOnState` incrementamos el contador de segundos.

```
#import "Chronometer.h"

@implementation Chronometer

- (void) awakeFromNib
{
  seconds = 0;
  [button setState: NSOnState];

  [NSTimer scheduledTimerWithTimeInterval: 1
           target: self
           selector: @selector(seconds:)
           userInfo: nil
           repeats: YES];
}

- (void) seconds: (NSTimer *)aTimer
{
  if ([button state] == NSOnState)
    {
      [textBox setIntValue: ++seconds];
    }
```

```
}
@end
```

Compilemos nuevamente el proyecto y comprobemos que funcione según lo esperado.

7.1.2 Agregando una segundera

Hagámosle una última modificación a esta app, agregándole una aguja que muestre el avance de los segundos. Por supuesto, dicha aguja debemos dibujarla nosotros y para ello crearemos una subclase de la clase `NSView`. El modelo que utilizaremos se muestra en la figura 7-2, donde se ve que hemos agregado un nuevo outlet a nuestra clase *Chronometer* para controlar el avance de la aguja.

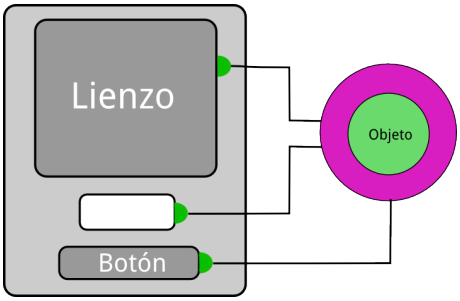

Figura 7-2. Nuevo modelo del cronómetro.

La clase `NSView` es la clase básica para dibujar en un área determinada, que se conoce como *lienzo* (*frame*). Pero antes de escribir nuestra subclase, familiaricémonos con los tipos de datos básicos para realizar dibujos: `NSPoint`, `NSSize` y `NSRect`. Estos no son objetos, sino que se trata de datos conocidos como *estructuras*. Comencemos con los puntos, `NSPoint`, que se crean mediante la función `NSMakePoint(x,y)`, donde x y y son las coordenadas del punto. Por ejemplo:

```
NSPoint punto = NSMakePoint(125, 236);
```

Las coordenadas de un punto llevan los nombres de x y y. Y podemos acceder a estos datos de la siguiente forma:

```
CGFloat x = punto.x;
CGFloat y = punto.y;
```

`CGFloat` es el tipo de *GNUstep* correspondiente a `float` o `double` (dependiendo de la arquitectura de la computadora). Esta notación también

puede utilizarse para cambiar las coordenadas de un punto. Por ejemplo, para cambiar la coordenada x del punto a 435:

```
punto.x = 435;
```

Por otro lado, `NSSize` representa las dimensiones de un rectángulo. Y se crea mediante la función `NSMakeSize(ancho,alto)`. Por ejemplo:

```
NSSize s = NSMakeSize(200, 100);
```

El ancho y el alto llevan los nombres de *width* y *height*, respectivamente. Por lo tanto, los datos se pueden acceder o modificarse de la siguiente forma:

```
CGFloat ancho = s.width;
CGFloat alto = s.height;

s.width = 300;
```

Por último tenemos el tipo `NSRect` que representa un rectángulo. La función correspondiente de creación es `NSMakeRect(x, y, ancho, alto)`. Donde *x, y* es la coordenada de la esquina inferior izquierda del rectángulo. Esta estructura esta conformada por un dato `NSPoint`, llamado *origin*, y por un dato `NSSize` llamado *size*. El siguiente ejemplo muestra como podemos acceder o cambiar los datos de una estructura `NSRect`:

```
NSRect rectangulo = NSMakeRect(50, 60, 200, 150);

// Obtenemos las coordenadas del origen.
CGFloat x = rectangulo.origin.x;
CGFloat y = rectangulo.origin.y;

// Obtenemos el ancho.
CGFloat ancho = rectangulo.size.width;

// Y el alto.
CGFloat alto = rectangulo.size.height;

// Cambiamos la coordenada x del origen.
rectangulo.origin.x = 90;

// Cambiamo el alto.
rectangulo.size.height = 175;

// Obtenemos el origen del rectangulo.
NSPoint point = rectangulo.origin;
```

```
// Obtenemos el tamaño del rectangulo.
NSSize s = rectangulo.size;
```

Pasemos entonces a escribir nuestra nueva clase, a la cual llamaremos *Clock*. Los archivos de interfaz e implementación serán entonces, respectivamente, '`Clock.h`' y '`Clock.m`'. Nuestra clase tendrá dos variables de instancia de tipo `NSPoint`. Que representaran el origen de la aguja, en el centro del lienzo, y la punta de la misma. Esta clase tendrá también un método de instancia encargado de hacer girar la aguja el ángulo correspondiente a un segundo. De esta forma, la interfaz de nuestra clase queda de la siguiente forma:

```
#import <AppKit/AppKit.h>

@interface Clock : NSView
{
  NSPoint origin, end;
}
- (void) move: (NSUInteger)seconds;
@end
```

Ahora en el archivo de implementación, aparte del método `-move`, debemos implementar el método `-awakeFromNib`, donde obtendremos el punto central del lienzo. Y el método `-drawRect:`, el cual se encarga de dibujar el contenido del lienzo.

```
#import <math.h>
#import "Clock.h"

/* Basado en un ejemplo de "GNUstep Tutorial" de
 * Yen-Ju Chen.
 */

@implementation Clock

- (void) awakeFromNib
{
 NSRect frame = [self frame];

 origin = NSMakePoint(frame.size.width/2,
                      frame.size.height/2);
 end = NSMakePoint(0, 0.45*frame.size.height);
}
```

```
- (void) drawRect: (NSRect)frame
{
    [[NSColor redColor] set];
    NSBezierPath *hand = [NSBezierPath bezierPath];

    [hand moveToPoint: origin];
    [hand relativeLineToPoint: end];
    [hand stroke];
}

- (void) move: (NSUInteger)seconds
{
  CGFloat length = 0.45*[self frame].size.height;

  end.x = length*sin(M_PI*seconds/30);
  end.y = length*cos(M_PI*seconds/30);

  [self setNeedsDisplay: YES];
}

@end
```

Estudiemos el método `-awakeFromNib`. Primero obtenemos el lienzo para dibujar, mediante el método de instancia `frame`. Seguidamente creamos el punto *origin* de tal forma que sus coordenadas estén en el centro del lienzo, y a continuación se crean las coordenadas iniciales del punto *end*. Las coordenadas del punto *end* las tomamos relativas al centro del lienzo (más adelante veremos por que). Como se ve, hemos hecho que la punta de la aguja este inicialmente en la posición de las doce. Y le hemos dado una longitud de 0.45 el alto del lienzo, de tal forma que la aguja no toque los bordes del lienzo (en *Gorm* haremos que el lienzo sea cuadrado).

Ahora el método `-drawRect:` es el encargado de dibujar en el lienzo. Y siempre debe implementarse en las clases que deriven de `NSView`, de lo contrario no se dibujara nada. Primero establecemos el color para dibujar utilizando la clase `NSColor` para obtener el color rojo mediante el método de clase `+redColor`. Y seguidamente lo establecemos como el color para dibujar mediante el método de instancia `-set`. Después creamos un objeto `NSBezierPath`, mediante el constructor conveniente `+bezierPath`. Estos objetos nos permiten crear rutas mediante puntos, lineas o curvas. Para luego dibujar o rellenar dicha ruta, con el color que se encuentre establecido. Luego mediante el método `-moveToPoint:` agregamos el punto *origin*. Este método agrega un punto tomando sus coordenadas relativas al lienzo. El cual tiene su origen en el extremo inferior izquierdo, con el eje **X** horizontal y positivo hacia la derecha, y el eje **Y** vertical y positivo hacia arriba. Por lo tanto, el punto *origin* se agrega en el centro del lienzo. Seguidamente agregamos

una linea hacia el punto *end*, mediante el método `-relativeLineToPoint:`. Este método agrega una linea entre el último punto en la ruta (nuestro punto *origin*) y el punto que se pasa como parámetro (nuestro punto *end*). Tomando las coordenadas de este último, relativas al primero. Lo hacemos de esta forma, porque así el calculo de las nuevas coordenadas del punto *end*, cuando avance la aguja, será más sencillo. Y por último dibujamos la ruta mediante el método `-stroke`. Este método hace uso del color que este establecido.

Por último veamos el método `-move:`, que recibe como parámetro el número de segundos transcurridos. Primero calculamos la longitud de la aguja, tal y como se hizo en el método `-awakeFromNib`. Seguidamente se calculan y establecen las nuevas coordenadas del punto *end*. Donde se hace uso de las funciones `sin(angulo)` y `cos(angulo)` que calculan, respectivamente, el seno y el coseno del ángulo, el cual debe estar en radianes. `M_PI` es una constante con el valor numérico de *pi*. Tomando en cuanta que el avance de un segundo equivale a una rotación de $360/60 = 6$ grados, multiplicamos el número de segundos por seis para obtener el ángulo de la aguja con respecto a la vertical. Luego convertimos dicho ángulo a radianes, de donde obtenemos *M_PI*seconds/30*. Y por último, redibujamos el contenido mediante el método `-setNeedsDisplay:`, pasándole como parámetro `YES`. Esta es la forma correcta de actualizar el contenido del lienzo, ya que el método `-drawRect:` nunca debe llamarse directamente.

Ahora debemos modificar nuestra clase `Chronometer`, agregándole un outlet para conectar con nuestra clase `Clock` y modificando el método `-seconds:`. Estos cambios son pequeños. Para la interfaz tenemos, llamando `clock` al outlet:

```
//Outlets
id textBox;
id button;
id clock;
```

Y en la implementación primero debemos agregar la interfaz de nuestra clase `Clock`, para poder interactuar con esta. Y luego modificar el método `-seconds:`:

```
#import "Clock.h"
#import "Chronometer.h"

@implementation Chronometer

- (void) awakeFromNib
{
  seconds = 0;
  [button setState: NSOnState];
```

```
  [NSTimer scheduledTimerWithTimeInterval: 1
           target: self
           selector: @selector(seconds:)
           userInfo: nil
           repeats: YES];
}

- (void) seconds: (NSTimer *)aTimer
{
  if ([button state] == NSOnState)
    {
      [textBox setIntValue: ++seconds];
      [clock move: seconds];
    }
}
@end
```

Donde esta claro por que no se pasa como parámetro `++seconds` al método `-move:`. A continuación debemos modificar el archivo '`gorm`' de nuestra interfaz gráfica, para incluir el reloj. Cambiemos el tamaño de la ventana y movamos los controles a la parte inferior, para crear un espacio donde agregaremos un elemento *CustomView*. Ahora hagamos que el alto y el ancho de este elemento sean iguales (un cuadrado).

A continuación, en la sección *Classes* de nuestro documento, agreguemos nuestra clase `Clock`. Luego seleccionemos el elemento *CustomView* y, en el *Inspector*, asignémosle la clase `Clock`. Esto hará que este elemento sea una instancia de dicha clase. Por último, agreguemos el outlet *clock* a nuestra clase `Chronometer`, y conectémoslo a nuestro elemento *CustomView*. Guardados los cambios tenemos casi todo listo para probar nuestra app. Sólo nos falta agregar los archivos de la clase `Clock` al archivo de construcción. Este debe quedar de la siguiente forma:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = Chronometer

Chronometer_MAIN_MODEL_FILE = Chronometer.gorm

Chronometer_RESOURCE_FILES = Chronometer.gorm

Chronometer_HEADER_FILES = \
Clock.h \
Chronometer.h
```

```
Chronometer_OBJC_FILES = \
Clock.m \
Chronometer.m

Chronometer_OBJC_FILES += AppMain.m

include $(GNUSTEP_MAKEFILES)/application.make
```

Obsérvese el uso de la contradiagonal para agregar mas de un archivo en las entradas correspondientes. Los archivos también pueden agregarse en la misma linea dejando un espacio entre los nombres. Pero el uso de la contradiagonal mejora la presentación y facilita la lectura cuando se trata de varios archivos.

Compilemos nuevamente nuestro proyecto y probemos su funcionamiento, imagen 7-1.

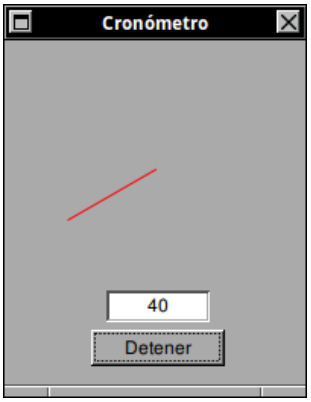

Imagen 7-1. Chronometer app.

En la imagen 7-2 se muestra la misma app con una imagen añadida como fondo del reloj. La imagen se agrega primero al documento seleccionando en el menú Document → Load Image. Seguidamente se agrega a la interfaz un elemento imagen desde la paleta *Data Palette*. Y en el inspector, en el campo con el título *Icon*, se escribe el nombre de la imagen sin extensión. O puede arrastrarse la imagen desde la sección *Images* del documento, para soltarla sobre el elemento imagen. Luego seleccionando en el menú Layout → Send To Back, se envía la imagen al fondo para que no tape la aguja del reloj. Las imágenes añadidas a documentos de interfaz se guardan en el archivo '`gorm`', por lo que no deben agregarse al archivo '`ĠNUmakefile`'.

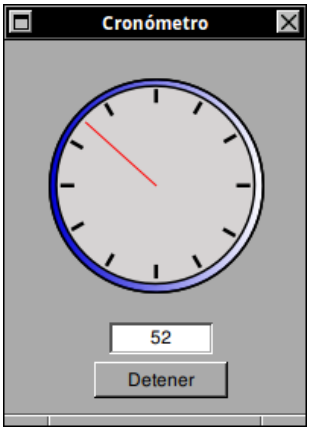

Imagen 7-2. Chronometer app mejorada.

8 Conceptos de diseño

En este capítulo veremos dos conceptos centrales en *GNUstep* que simplifican el diseño de una app. Nos referimos a los métodos *delegate* y a las *notificaciones*. También abordaremos el tema de los eventos del ratón y del teclado en *GNUstep*. Así como la forma de *cargar* otros archivos '`gorm`' en una app, lo que simplifica el desarrollo de la interfaz gráfica.

8.1 Delegate

delegate es un mecanismo para implementar ciertos métodos opcionales los cuales nos permiten personalizar el comportamiento de un objeto. No todos las clases tienen métodos *delegate*, solamente algunas de las clases visuales ofrecen esta posibilidad. Entre las clases que proveen métodos *delegate* están: `NSText`, `NSTextView`, `NSTableView`, `NSOutlineView`, `NSBrowser`, `NSWindow`, `NSApplication`. Estos métodos pueden tener diferentes propósitos, por ejemplo: obtener información a ser desplegada, como en los objetos `NSTableView`; informar que una determinada acción esta por ocurrir, o pedir aprobación para llevar a cabo dicha acción, o también informar que dicha acción ya ha ocurrido. Los métodos *delegate* que provee una clase en particular, se deben implementar en algún objeto que haga de *delegado* (*delegate*). Y deben agregarse tanto en la interfaz como en la implementación de la clase de dicho objeto.

Como hemos dicho, estos métodos son opcionales. Si *GNUstep* encuentra que están implementados en un objeto delegado, los ejecuta. De lo contrario no pasa nada. Puede surgir entonces la pregunta ¿Como sabe *GNUstep* cuando están presentes estos métodos?. Objective-C es un lenguaje muy dinámico, y nos permite preguntarle a un objeto si responde o no a un método. El siguiente código muestra un ejemplo de esto:

```
if ([delegate respondsToSelector: @selector(information:)])
  {
    newInformation = [delegate information: self];
  }
```

En nuestra primera app que suma dos números, es evidente que las cajas de texto (`NSTextField`) no deben permitir la inclusión de letras, sino únicamente la de números. Esto podemos controlarlo utilizando el método *delegate* `-controlTextDidChange:`, que se ejecuta cada vez que el contenido de la caja de texto cambia. Para hacerlo, primero debemos indicarle a las dos cajas de texto donde el usuario ingresa los números, que la instancia de nuestra clase *SumaController* será el *delegado*. Abramos nuestra interfaz gráfica en *Gorm* y conectemos las cajas de texto con nuestro objeto *SumaController*, seleccionando el outlet *delegate* en el *Inspector*. Hecho esto, agregamos el método *delegate* a los archivos de interfaz e implementación de nuestra clase *SumaController*. El código para este método es el siguiente:

```
- (void) controlTextDidChange: (NSNotification*)aNotification
{
  NSTextField *field;
  NSString *string;
  NSCharacterSet *allow, *chars;

  field = [aNotification object];
  string = [field stringValue];

  allow = [NSCharacterSet characterSetWithCharactersInString:
           @".0123456789"];
  chars = [NSCharacterSet characterSetWithCharactersInString:
           string];

  if (![allow isSupersetOfSet: chars])
    {
      [field setStringValue: @""];
    }
}
```

`[aNotification object]` devuelve el objeto que ejecuta el método delegado. Es decir, la caja de texto en la cual esta escribiendo el usuario. El conjunto de caracteres *allow* contiene los dígitos decimales (0 ... 9) y el punto decimal, mientras que *chars* contiene los caracteres en la caja de texto. Luego se verifica si *allow* es un superconjunto de *chars*. Es decir, si todos los caracteres en *chars* están presentes en *allow*. Si no es este el caso, se establece una cadena de texto vacía como el contenido de la caja de texto.

Un objeto *delegate* también puede establecerse mediante el método `-setDelegate:`, que recibe como parámetro al objeto que hará de *delegado*. Por ejemplo, lo hecho en *Gorm*, podría también realizarse en un método `-awakeFromNib` en nuestra clase *SumaController*:

```
- (void) awakeFromNib
{
  [firstNumber setDelegate: self];
  [secondNumber setDelegate: self];
}
```

8.2 Cadena de eventos (Responder chain)

Se conoce como *cadena de eventos* o *responder chain* al camino que siguen los eventos a través de los objetos de una aplicación. Nos ocuparemos aquí de los eventos generados con el ratón y el teclado, aunque existen otros. Toda la información relativa al evento ocurrido es almacenada en un objeto `NSEvent`.

Y es este objeto el que lleva la información del evento a los distintos objetos de la aplicación, mientras se recorre la cadena de eventos. Sin embargo, veamos antes algunos conceptos relacionados.

Los objetos `NSWindow`, que corresponden a las ventanas, utilizan varios objetos `NSView` para construir la ventana. De todos estos, nosotros solamente tenemos acceso al *view* conocido como *content view*. Es decir, al *view* que dibuja el contenido de la ventana (el rectángulo entre la barra de titulo y los bordes de la ventana). Es en este *view* donde se ubican los componentes de nuestra ventana, botones, campos de texto, imágenes, etc. Se dice entonces que estos componentes son *subviews* del *content view*. Y que el *content view* es el *superview* de cada uno de estos componentes. Para entender mejor esto, supongamos una ventana con dos componentes, como se muestra en la imagen 8-1. Donde el objeto `NSButton` esta contenido dentro de un objeto `NSBox`.

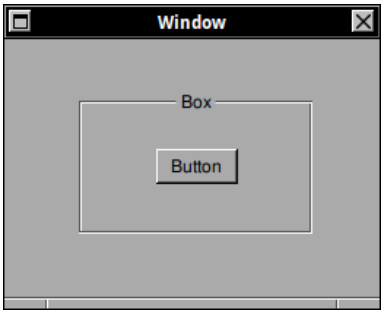

Imagen 8-1. Superview y subview.

En este caso, el objeto `NSBox` es un *subview* del *content view*. Y el objeto `NSButton` es un *subview* del objeto `NSBox`. Y a la inversa, el objeto `NSBox` es el *superview* del objeto `NSButton`, mientras el *content view* es el *superview* del objeto `NSBox`.

Ahora bien, en toda aplicación la ventana activa se conoce como *key window*. Si damos un clic con el ratón sobre alguna otra ventana, de la misma aplicación o de otra, dicha ventana se convierte entonces en la key window, y la ventana anterior se desactiva. Si las ventanas están superpuestas, una encima de otra, y damos un clic en la ventana de atrás, entonces ésta se convierte en la key window y pasa a primer plano. Además, el componente en la key window que recibe el enfoque (un botón, un campo de texto, etc.) se conoce como *first responder*. Este es el objeto que recibirá los eventos generados con el teclado.

La ventana key window siempre se distingue de las demás por tener una barra de título diferente. Asimismo, el componente que es el first responder se distingue fácilmente de los otros. La imagen 8-2 muestra como se ven algunos componentes cuando son el first responder (en el tema por defecto). La mayoría de ellos son resaltados mediante un borde de linea punteada.

Pero para la caja de texto, el *punto de inserción* es lo que indica que esta es el first responder.

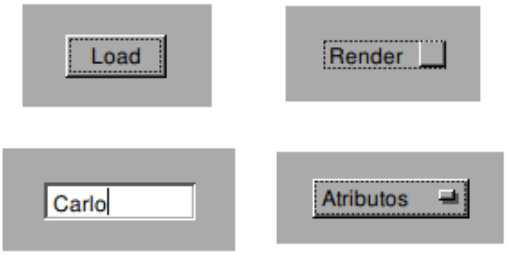

Imagen 8-2. First responder.

Con estos conceptos, ya estamos en capacidad de entender la cadena de eventos. Pasemos, entonces, a considerar la cadena de eventos generados con el ratón.

8.2.1 Eventos del ratón

Supongamos el evento de dar un clic con el botón izquierdo del ratón. El camino que sigue dicho evento es el siguiente:

1. El objeto `NSApplication` que controla nuestra aplicación, es el primero en recibir la información del evento ocurrido. Esta información, la recibe como un objeto `NSEvent`.
2. El objeto `NSApplication` ejecuta el método `-mouseDown:` de la ventana donde ocurrió el evento. Y le manda como parámetro el objeto `NSEvent`.
3. La ventana ejecuta su método `-hitTest:` y le envía como parámetro la coordenada, relativa a la ventana, donde ocurrió el evento. Dicho método retorna el objeto que se encuentra en esa coordenada, si lo hay. Por ejemplo, un botón.
4. Si la ventana donde ha ocurrido el evento es actualmente la *key window*, entonces el objeto `NSEvent` es enviado al objeto devuelto por el método `-hitTest:`. Si no es actualmente la *key window*, entonces la ventana se convierte en esta. Seguidamente, se ejecuta el método `-acceptsFirstMouse:` del objeto devuelto por `-hitTest:`. Si este método devuelve `YES`, es decir, si acepta el primer clic dado en la ventana, el objeto `NSEvent` es enviado a dicho objeto. Si devuelve `NO`, termina la cadena de eventos. Este último caso se observa en ventanas donde hay que dar primero un clic para activarlas, y luego un segundo clic para ejecutar el componente deseado.
5. Si el objeto que recibe el clic no puede procesar el evento, entonces este es enviado al objeto que este establecido como el *next responder*. Que por defecto es el *superview* del objeto. Si a su vez dicho objeto no puede procesar el evento, este es enviado al *next responder* correspondiente.

Y así sucesivamente hasta que se encuentre un objeto que responda al evento o hasta que se llegue al *content view*. Si al llegar al *content view* este no responde al evento, el objeto `NSEvent` es regresado al objeto `NSWindow` que lo envió y allí termina la cadena de eventos.

Por otro lado, el evento de soltar el botón del ratón, siempre es enviado al objeto donde ocurrió el clic del ratón. Independientemente de la posición del cursor.

El *next responder* se establece con el método `-setNextResponder:`. Y no debe confundirse con el objeto *next key view*, el cual se establece con el método `-setNextKeyView:`. Este último, fue el que utilizamos en nuestra primera app, para permitirle al usuario moverse en nuestra interfaz mediante la tecla TAB. Pasemos ahora a considerar la cadena de eventos para un evento generado con el teclado.

8.2.2 Eventos del teclado

Suponiendo que se presiona una sola tecla, y no una combinación de teclas como `#q`, la cadena de eventos es la siguiente:

1. El objeto `NSApplication` que controla nuestra aplicación, es el primero en recibir la información del evento ocurrido.
2. El objeto `NSApplication` ejecuta el método `-keyDown:` de la ventana que sea la *key window* en ese momento.
3. El método `-keyDown:` envía entonces el objeto `NSEvent` al objeto que sea el *first responder* en la ventana.
4. Si el objeto que es el *first responder* no responde al evento, puede que este inactivo en ese momento, el objeto `NSEvent` es enviado al objeto que este establecido como el *next responder*. Este proceso se sigue hasta encontrar un objeto que responda al evento. Y en el caso de que ningún objeto pueda responder a este, el objeto `NSEvent` es retornado a la ventana, la cual emitirá un *beep* indicando que ningún objeto respondió.

Si el evento que ocurre tiene una tecla modificadora, como `#q`, el evento es enviado entonces al menú de la aplicación. Si alguna opción del menú tiene asignada dicha combinación de teclas, entonces se ejecuta dicha opción. De lo contrario, el evento se procesa como cualquier otro evento del teclado.

8.3 Archivos Gorm

Hasta ahora hemos visto apps en donde solamente hay un archivo '`gorm`'. Sin embargo, puede darse el caso de que decidamos colocar algunas partes de la interfaz gráfica en otros archivos. Ya sea porque su uso no es frecuente y no deseamos cargarlo en la memoria desde el inicio. O porque forma parte de un *bundle* opcional y por lo tanto su interfaz gráfica solo estará disponible si dicho *bundle* esta instalado. En esta sección veremos como cargar una interfaz gráfica desde un archivo '`gorm`' que no sea el principal. Y como establecer conexiones con los elementos de dicha interfaz. Aquí nos

limitaremos al caso cuando el archivo forma parte de la misma app y no de un *bundle*.

En este caso, el archivo '`gorm`' se carga con el método de clase `+loadNibNamed:owner:`, de la clase `NSBundle`. El primer parámetro es el nombre del archivo '`gorm`', sin extensión. Mientras que el segundo es el objeto que hará de *propietario* de la interfaz gráfica. Por ejemplo, para un archivo llamado '`PrefsPanel.gorm`':

```
[NSBundle loadNibNamed: @"PrefsPanel" owner: self];
```

Donde el propietario es el mismo objeto (`self`) que carga la interfaz. El objeto que hace de propietario se representa en *Gorm* por el objeto *NSOwner*, imagen 8-3. Para el caso del archivo '`gorm`' principal, el propietario es el objeto `NSApplication` que controla la app.

Imagen 8-3. Objeto propietario.

Para poder utilizar este objeto, primero debemos importar la cabecera de la clase del objeto que hará de propietario, y luego asignar esta clase al objeto `NSOwner`. Hecho esto, podemos conectar los outlets de ese objeto a la interfaz gráfica, así como realizar conexiones *target-action*. Es decir, conectar los controles de la interfaz con los métodos del objeto propietario.

Por defecto, los objetos `NSWindow` están configurados para no liberar la memoria cuando la ventana sea cerrada. Es decir, que la ventana sigue existiendo en memoria aunque no se vea en la pantalla. De esta forma, si la ventana se requiere nuevamente, no será necesario crearla otra vez, bastara con mostrarla en la pantalla. Teniendo en cuenta esto, deberemos ser precavidos con el método que cargue el archivo '`gorm`'. Para no crear nuevamente una ventana que no es visible pero que ya existe en memoria. O incluso, para no duplicar una ventana que ya esta presente en la pantalla. Para ello, debe contarse con un outlet que conecte con la ventana (o ventanas) del archivo '`gorm`'. Suponiendo una sola ventana, el outlet hacia esta debe valer inicialmente `nil`. Esto puede hacerse en el método `-awakeFromNib` del propietario, o en algún método de inicio. De esta forma, el método que cargue la nueva interfaz puede ser de la siguiente forma. Donde hemos supuesto que este método muestra el panel de preferencias y que el outlet hacia la ventana se llama `panel`:

```
- (void) showPrefsPanel: (id)sender
{
   if (panel == nil)
     {
       [NSBundle loadNibNamed: @"PrefsPanel" owner: self];
       [panel makeKeyAndOrderFront: self];
     }
   else
     {
       [panel makeKeyAndOrderFront: self];
     }
}
```

Aquí primero se verifica si ya existe la ventana. Si no es así, se carga entonces la interfaz gráfica. Y mediante el método `-makeKeyAndOrderFront:`, se hace que la ventana sea la *key window* y que se muestre por encima del resto de ventanas. Por otro lado, si la ventana ya existe, simplemente se ejecuta su método `-makeKeyAndOrderFront:`.

Si la ventana esta configurada para liberarse al momento de ser cerrada (opción *Release when closed* en el *Inspector* de *Gorm*), deberemos asegurarnos de restablecer el outlet `panel` al valor `nil`, al momento de que la ventana sea cerrada. De lo contrario, la verificación `panel == nil` fallara, debido a que `panel` no tendrá ningún valor asignado. Esto puede hacerse haciendo que el objeto propietario sea el *delegado* de la ventana, he implementando el método delegado `-windowWillClose:`. Por ejemplo:

```
- (void) windowWillMove: (NSNotification *)aNotification
{
  if (panel == [aNotification object])
    {
      panel = nil;
    }
  else if (...)
    {
      ....
      ....
    }
}
```

Aquí hemos supuesto que el objeto propietario es el delgado de dos ventanas. Por lo que debemos verificar cual de estas es la que va a cerrarse. Y asignar el valor `nil` a `panel`, solamente cuando la ventana de preferencias sea la que este por cerrarse. Los objetos `NSNotification` los estudiaremos

en una sección posterior, aquí solamente es necesario saber que el método `-object`, devuelve la ventana que esta por cerrarse.

Para el caso en que la ventana es liberada al momento de cerrarse, nuestro método `-showPrefsPanel:` puede quedar igual al que se ha mostrado previamente. Así, para el caso en que la ventana este oculta detrás de otra, este método se encargara de llevarla al frente.

Otra razón para utilizar otros archivos '`gorm`', es simplificar el diseño de una interfaz gráfica. Supongamos una interfaz como la mostrada en la imagen 8-4. Sería engorroso, sino imposible, colocar los elementos dentro del *ScrollView* que, a su vez, esta dentro de un *TabView*.

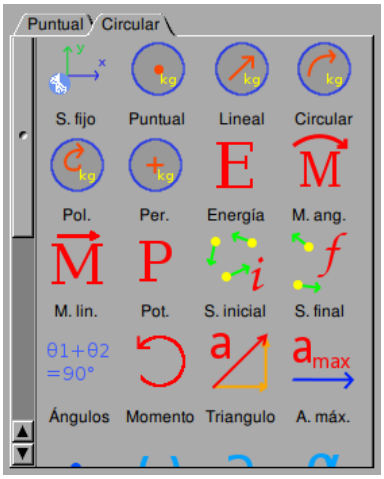

Imagen 8-4. Un *ScrollView* dentro de un *TabView*.

Una forma conveniente de hacerlo, es colocar todos los elementos en un archivo '`gorm`' diferente y posteriormente *empotrarlos* en el *TabView*. La imagen 8-5 muestra un archivo '`gorm`' consistente en una ventana donde se han colocado los elementos que se mostraran en el *TabView*. La ventana debe estar configurada para no ser visible al momento de cargar el archivo '`gorm`', mediante la opción *Visible at launch time* en el *Inspector*.

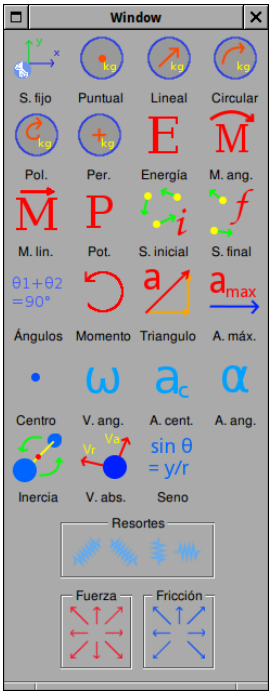

Imagen 8-5. Ventana con los elementos a mostrar.

Llamando *tabviewTwo* al outlet que conecta con el *TabView* donde se mostrarán los elementos, *elements* al archivo '`gorm`' de la imagen 8-4 y *elementsWindow* al outlet que conecta con la ventana de este archivo, el código para empotrar estos elementos en el *TabView* sería como el siguiente:

```
NSScrollView *scrollView;

// Se carga el archivo elements.gorm.
[NSBundle loadNibNamed: @"elements" owner: self];

// Se crea un scrollview con el tamaño del tabview.
scrollView = [[NSScrollView alloc] initWithFrame:
              NSMakeRect(0, 0, 270, 320)];

// Se agrega una barra de desplazamiento vertical.
[scrollView setHasVerticalScroller: YES];

/* Se establece como contenido del scrollview el
 * contenido de la ventana en el archivo elements.gorm.
 */
[scrollView setDocumentView: [elementsWindow contentView]];
```

```
// Se agrega el scrollview al tabview.
[tabviewTwo addSubview: scrollView];

// Se libera la ventana elements y el scrollview.
[elementsWindow release];
[scrollView release];
```

Este código podría ir en un método `-awakeFromNib`. Tanto el `scrollView` como la ventana del archivo *elements* se liberan al final. Ya que el contenido de la ventana es *retenido* por el `scrollView`, y este a su vez es *retenido* por el *TabView*.

En este ejemplo, el *TabView* tiene una dimensión fija, por lo que no nos preocupamos del ajuste automático de tamaño. Sin embargo, si el *TabView* fuera redimensionable, deberíamos establecer las opciones de ajuste automático. Por ejemplo, si el *TabView* puede cambiar tanto de ancho como de alto, debemos establecer el ajuste de tamaño tanto para el alto como para el ancho del `scrollView`. El código en este caso podría quedar de la siguiente forma, solo se muestran las lineas relevantes:

```
// Ajuste automatico del scrollview.
[scrollView setAutoresizingMask:
                NSViewHeightSizable | NSViewWidthSizable];

// Se agrega el scrollview al tabview.
[tabviewTwo addSubview: scrollView];
```

El operador `|` se utiliza para establecer varias opciones, en este caso dos: `NSViewHeightSizable` y `NSViewWidthSizable`. No es necesario establecer este ajuste para el contenido de la ventana del archivo *elements*, ya que por defecto el *Content View* de una ventana tiene ajuste automático de ancho y alto.

8.4 First Responder

Como ya hemos visto, se conoce como *first responder* al control de la interfaz gráfica que tiene el foco, y que por lo tanto recibe los eventos del teclado. Este esta representado en los archivos '`gorm`' mediante el objeto *NSFirst*, imagen 8-6. Y nos permite ejecutar métodos en cualquiera que sea el *first responder* en un momento dado.

Imagen 8-6. Objeto *NSFirst*.

Es habitual que varios ítems del menú estén conectados al objeto *NSFirst*. En este caso, *GNUstep* comprueba si el objeto que sea el *first responder*, responde o no a los métodos en los ítems del menú, habilitando o inhabilitando estos dependiendo de si los métodos están presentes o no en el *first responder*. Por ejemplo, los ítems *Cortar*, *Copiar* y *Pegar* del menú de una app de documentos, están conectados al objeto *NSFirst*. En la imagen 8-7, el menú de la izquierda muestra el caso cuando no hay ningún documento abierto, por lo que los tres ítems están inhabilitados. El siguiente menú, muestra el caso cuando un documento tiene el foco y el *pasteboard* tiene contenido, por lo que la opción *Pegar* esta habilitada. Y el caso mostrado en el tercer menú, difiere del anterior en que se ha hecho una selección en el documento, por lo que los ítems *Cortar* y *Copiar* están habilitados.

Imagen 8-7. Ítems conectados a *NSFirst*.

Si el método que queramos ejecutar no aparece en el objeto *NSFirst*, lo podemos agregar a este seleccionando la clase *FirstResponder* en la sección *Classes* de nuestro documento '`gorm`'. Y añadiendo el método deseado en la pestaña *Actions* del *Inspector*.

Imagen 8-8. Clase *FirstResponder*.

Obsérvese que *FirstResponder* no es realmente una clase. Es sólo una forma conveniente de representar a cualquier objeto que sea el *first responder* en un momento dado.

8.5 Notificaciones

Como ya hemos visto, los métodos delegados son muy útiles para personalizar el comportamiento de nuestros objetos. Las *notificaciones* son similares a estos métodos, pero con la diferencia de que podemos agregarlos a cualquier objeto y para cualquier acción o evento ocurrido. Como su nombre lo indica, son mensajes que notifican a un objeto de algo que ha ocurrido en otro objeto. Una notificación se puede crear mediante la clase `NSNotification`, y posteriormente enviarse mediante la clase `NSNotificationCenter`. O puede crearse y enviarse directamente con esta última clase. Esta es la forma que trataremos aquí.

Toda notificación debe tener asignado un nombre y un objeto, generalmente el objeto que manda la notificación. La notificación debe enviarse al *centro de notificaciones*, que toda app o tool tiene por defecto. Por ejemplo, para enviar una notificación llamada *ColorDidChange* que informa sobre un cambio en el color, tendríamos:

```
[[NSNotification defaultCenter]
  postNotificationName: @"ColorDidChange"
                object: self];
```

Obsérvese que la notificación simplemente se envía al centro de notificaciones, sin indicar a que objeto debe enviarse esta (el destinatario). Esto es así, porque son los otros objetos los que deben indicarle al centro de notificaciones que notificaciones son las que quieren recibir. También puede agregarse un objeto `NSDictionary` con información útil para el destinatario, esto se hace de la siguiente forma:

```
[[NSNotification defaultCenter]
  postNotificationName: @"ColorDidChange"
                object: self;
              userInfo: aDictionary];
```

Si un determinado objeto debe recibir la notificación *ColorDidChange*, debemos entonces *agregar* dicho objeto al centro de notificaciones como *observador*, indicando el nombre de la notificación y el método a ejecutar cuando esta se reciba. Esto puede hacerse en el método `-awakeFromNib` o en un método de inicio. Por ejemplo, para ejecutar el método `-colorChanged:` al recibir la notificación anterior, tendríamos:

```
[[NSNotificationCenter defaultCenter]
```

```
        addObserver: self
           selector: @selector(colorChanged:)
               name: @"ColorDidChange"
             object: nil];
```

El último parámetro indica el objeto del cual queremos recibir las notificaciones. En este ejemplo, hemos pasado como parámetro `nil`, indicando con esto que queremos recibir todas las notificaciones *ColorDidChange*, sin importar de que objetos provengan. Podemos agregar un objeto para recibir cuantas notificaciones queramos. Pero debemos asegurarnos de *removerlo* del centro de notificaciones, al momento de que sea liberado de la memoria. Esto se hace en el método `-dealloc`. Por ejemplo:

```
- (void) dealloc
{
  [[NSNotificationCenter defaultCenter]
                                removeObserver: self];
  [super dealloc];
}
```

No importa cuantas veces hayamos agregado dicho objeto al centro de notificaciones, solamente debemos removerlo una vez. Si no removemos un observador del centro de notificaciones, se producirá un error la próxima vez que este trate de enviarle una notificación a un objeto que ya no existe.

El método que ejecuta una notificación debe recibir como parámetro un objeto `NSNotification`, y debe estar declarado en la interfaz de la clase. Los métodos de instancia `-object` y `-userInfo` del objeto `NSNotification` devuelven, respectivamente, el objeto que envía la notificación y el objeto `NSDictionary` con la información proporcionada (o `nil` si no existe tal objeto). Por ejemplo, para nuestro método `-colorChanged:`:

```
- (void) colorChanged: (NSNotification *)aNotification
{
  id obj = [aNotification object];
  NSDictionary *info = [aNotification userInfo];

  ...
  ...
  ...
}
```

El framework *GUI* envía diferentes notificaciones sobre algunos de los cambios que han ocurrido en una app. Se pueden entonces agregar observadores al centro de notificaciones para ejecutar determinados métodos cuando estos cambios ocurran. Algunas de

estas notificaciones están asociadas con métodos *delegate* (vistos anteriormente), mientras que otras solamente se pueden recibir si se agrega un observador al centro de notificaciones. Ejemplos de estas últimas son las notificaciones `NSViewBoundsDidChangeNotification` y `NSViewFrameDidChangeNotification` que informan, respectivamente, de cambios en el marco (*bounds*) y en el lienzo (*frame*) de un objeto `NSView`.

A la pregunta: ¿Para que usar notificaciones si podemos utilizar el paradigma target-outlet?, contestamos que dicho paradigma solamente puede utilizarse en los objetos instanciados por archivos de interfaz gráfica (las conexiones realizadas en *Gorm*). Y aunque es posible hacer que todos los objetos de una app sean instanciados desde archivos '`gorm`', esto puede resultar impráctico en algunos casos. Imaginemos por ejemplo una app de documentos, donde la interfaz gráfica asociada a un documento esta guardada en un archivo '`gorm`'. Ya que el usuario puede abrir cuanto documento desee, resultaría impráctico tener un outlet para cada uno de estos documentos. Primero porque no sabemos de antemano cuantos outlets serán necesarios, y segundo porque deberíamos tener un archivo '`gorm`' para cada uno de los documentos y así poder realizar las conexiones. Sin embargo, todo esto no es necesario, ya que las notificaciones nos simplifican el diseño.

Las notificaciones enviadas al centro de notificaciones, no se envían inmediatamente a los objetos que las requieren. En su lugar, van a una *cola de notificaciones* donde esperan su turno para ser enviadas. Una vez que una notificación es enviada y el método correspondiente se termina de ejecutar en el observador, se procede a enviar la siguiente notificación en la cola. Es por esta razón que se recomienda que los métodos a ejecutar en los observadores no sean muy extensos, para no retrasar el envió de las siguientes notificaciones en la cola.

9 Una app de documentos

En este capítulo vamos a crear una sencilla app de documentos, consistente en un editor gráfico capaz de dibujar lineas, trazos a mano alzada, rectángulos y círculos. Una vez terminada, esta app se vera como la mostrada en la imagen 9-1. Se abordaran también los conceptos básicos para implementar las opciones de *Deshacer/Rehacer* en una app de documentos. Llamaremos a esta app *PowerPaint*.

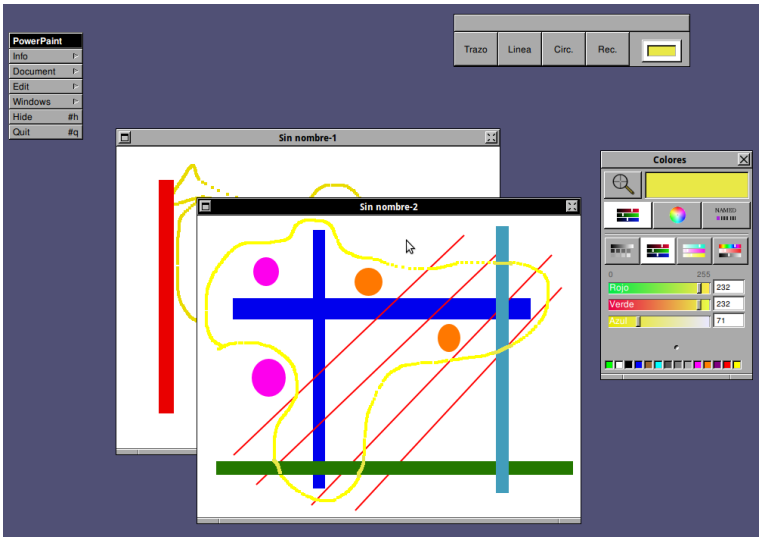

Imagen 9-1. Nuestra app de documentos *PowerPaint*.

Aunque esta app es bastante sencilla, se hará uso de clases y métodos que no se habían presentado anteriormente. Y a pesar de que se describe detenidamente el funcionamiento del código presentado, se recomienda consultar la documentación de *GNUstep* si se presentan dudas respecto al uso de algunos métodos o clases.

9.1 Los archivos de código

Básicamente necesitamos tres archivos de código, con sus respectivas interfaces. El primero de ellos, será el encargado de gestionar el panel de herramientas, y de permitirle a los documentos conocer que herramienta y que color están seleccionados. Ya que en Objective-C es habitual agregar a los nombres de los archivos de código un prefijo de dos o mas letras que haga referencia a la app, llamaremos en este caso '`PPController.h`' y '`PPController.m`' a los archivos de esta clase. La cual será una subclase de la clase raíz `NSObject`. Esta clase tendrá una variable de instancia, de tipo `NSInteger`, que representará la herramienta seleccionada en el panel. Y un

outlet que conectara con la paleta de colores. Veamos entonces el código del archivo de implementación:

```
#import "PPController.h"

@implementation PPController

- (void) awakeFromNib
{
  /* Por defecto se selecciona la
     herramienta de trazos. */
  tool = 0;
}

/* Se retorna el outlet que conecta con
   la paleta de colores. */
- (id) colorWell
{
  return colorWell;
}

/* Metodos Accessor para establecer/obtener la
   herramienta seleccionada por el usuario. */
- (void) setTool: (id)sender
{
  tool = [sender tag];
}

- (NSInteger) tool
{
  return tool;
}

@end
```

Como se ve, se establece la herramienta 0 (trazo a mano alzada) en el método `-awakeFromNib`. El método `-colorWell` retorna el outlet a la paleta de colores, de tal forma que los documentos puedan obtener el color establecido en esta. Cada botón de herramienta en el panel, estará conectado al método `-setTool` (paradigma *target-action*). El método `-tag` retorna el número que identifica al botón en cuestión, el cual se asigna a la variable de instancia `tool`. Como se vera en la siguiente sección, la instancia de esta clase sera designada como el *delegado* del objeto `NSApplication` que controla la app.

La interfaz de esta clase es la siguiente:

```
#import <AppKit/AppKit.h>

@interface PPController : NSObject
{
  NSInteger tool;

  // Outlet
  id colorWell;
}

- (void) awakeFromNib;
- (id) colorWell;
- (void) setTool: (id)sender;
- (NSInteger) tool;

@end
```

Nuestro editor gráfico debe manejar, por supuesto, imágenes. Por simplicidad, le daremos soporte únicamente para archivos *JPG*. Toda app de documentos debe tener una subclase de la clase `NSDocument`, la que, entre otras cosas, se encargará de decirle a la app como leer y guardar los archivos soportados. Si una app va a soportar diferentes tipos de archivos puede ser necesario, dependiendo de la complejidad, tener una subclase `NSDocument` para cada tipo soportado. En otros casos, una sola subclase puede ser suficiente para manejar todos los tipos de archivos. Para nuestra app *PowerPaint*, crearemos una subclase llamada `PPDocumentClass`, que tendrá una variable de instancia para almacenar el archivo leído por el usuario (`rep`), y un outlet hacia el objeto `NSView` que se encargará de mostrar la imagen en la ventana (`view`). La implementación de esta clase es la siguiente:

```
#import <AppKit/AppKit.h>
#import "PPDocumentView.h"
#import "PPDocumentClass.h"

@implementation PPDocumentClass

- (id) init
{
  self = [super init];

  if (self)
    {
      rep = nil;
    }
```

```
  return self;
}

- (NSString *) windowNibName
{
  return @"Document";
}

- (BOOL) readFromFile: (NSString *)fileName
               ofType: (NSString *)fileType
{
  NSData *data = [NSData dataWithContentsOfFile: fileName];
  rep = [[NSBitmapImageRep imageRepWithData: data] retain];

  if (rep != nil)
    {
      return YES;
    }
  else
    {
      return NO;
    }
}

- (BOOL) writeToFile: (NSString *)fileName
              ofType: (NSString *)fileType
{
  NSData *file = [[view currentContent]
                   representationUsingType: NSJPEGFileType
                                properties: nil];

  return [file writeToFile: fileName atomically: YES];
}

- (void) windowControllerDidLoadNib:
            (NSWindowController *)windowController
{
  [view setFile: [rep autorelease]];
}

@end
```

La clase que se importa al inicio, `PPDocumentView`, y que veremos mas adelante, es la encargada de mostrar el archivo en la interfaz gráfica y de manejar los eventos del usuario. Todos los métodos en esta clase, son re-

definiciones de métodos en la clase `NSDocument`. Y es la maquinaria interna de *GNUstep* la que se encarga de ejecutarlos en el momento adecuado.

En la redefinición del método `-init` se establece `nil` como valor inicial para la variable de instancia `rep`. Esta es de clase `NSBitmapImageRep`, es decir, una imagen en mapa de bits. El método `-windowNibName`, retorna el nombre del archivo '`gorm`' que representa el documento de nuestra app. En este caso *Document*.

El siguiente método, `-readFromFile:ofType:`, es el encargado de leer los archivos de nuestra app, y se ejecuta al abrir un archivo con el panel *Open*. Este no es el único método para tal fin, pero es el mas sencillo y por lo tanto será el que utilizaremos aquí. Este método debe retornar un valor booleano (`YES` o `NO`) dependiendo de si la lectura del archivo fue exitosa o no. El parámetro `fileName` es la ruta del archivo seleccionado por el usuario en el panel *Open*. El segundo parámetro es el tipo de archivo, que no utilizamos aquí puesto que nuestra app solo maneja un tipo de archivo. Primero se lee el contenido del archivo mediante la clase `NSData`. Y posteriormente se crea un objeto `NSBitmapImageRep` con esta información, objeto *retenido* que se asigna a la variable `rep`. Se hace de esta forma ya que cuando el método `-readFromFile:ofType:` se ejecuta, la interfaz gráfica todavía no se ha cargado. Por lo que no es posible mostrar la imagen en la ventana del documento en este momento. Si la imagen se creo correctamente, se retorna `YES`, de lo contrario se retorna `NO`.

El siguiente método, `-writeToFile:ofType:`, es el encargado de guardar el documento, y se ejecuta al guardar un documento con el panel *Guardar*. Nuevamente, este no es el único método para tal fin, pero es el que utilizaremos aquí. Este método debe retornar un valor booleano (`YES` o `NO`) dependiendo de si la escritura del archivo fue exitosa o no. El parámetro `fileName` es la ruta seleccionada por el usuario en el panel *Guardar*, incluyendo el nombre dado al documento. Aquí primero se crea un objeto `NSData` con el contenido del documento, utilizando una codificación *JPG* (`NSJPEGFileType`). El método `-currentContent` devuelve el contenido del documento como un objeto `NSBitmapImageRep`. Seguidamente se guarda el documento en el disco con el método `-writeToFile:atomically:`. El segundo parámetro, en caso de ser `YES`, escribe primero el documento en un archivo temporal y luego lo renombra colocándolo en la ruta indicada. Por otro lado, si se pasa `NO`, el documento se guarda directamente en la ruta elegida. Este método retorna un valor booleano dependiendo de si el archivo se guardo correctamente o no. Y este valor retornado es el mismo que se retorna para el método `-writeToFile:ofType:`.

Finalmente, el método `-windowControllerDidLoadNib:`, que se ejecuta una vez que la interfaz gráfica ha sido cargada, se encarga de pasar la imagen a la ventana del documento. El objeto `rep` se pasa como un objeto autoliberado, puesto que no se necesita mas en esta clase, y además será retenido por el objeto encargado de la representación gráfica. Por último, la interfaz de esta clase es la siguiente:

```
#import <AppKit/NSDocument.h>

@interface PPDocumentClass : NSDocument
{
  NSBitmapImageRep *rep;

  // Outlet
  id view;
}

@end
```

Finalmente necesitamos una subclase de la clase `NSView`, que será la encargada de la representación gráfica de los documentos. Es en esta clase donde se implementarán las herramientas de dibujo, así como las capacidades *Deshacer/Rehacer* de nuestra app. Por defecto, en una app de documentos, todo documento cuenta con un objeto `NSUndoManager`. El cual se encarga de administrar dos *pilas* de eventos, los eventos *undo* (los eventos que el usuario puede *deshacer*) y los eventos *redo* (los que el usuario puede *rehacer*). Existes dos formas de registrar eventos *undo/redo*, aquí veremos la mas sencilla de estas. Veamos primero la interfaz de esta clase, a la que llamaremos `PPDocumentView`:

```
#import <AppKit/NSView.h>

@class PPDocumentClass;

@interface PPDocumentView : NSView
{
  BOOL selection;
  NSPoint locA, locB;
  NSRect selectedRect;
  NSBitmapImageRep *lastFrame;

  // Outlet
  PPDocumentClass *document;
}

- (void) setFile: (NSBitmapImageRep *)image;
- (void) setContent: (NSBitmapImageRep *)data;
- (NSBitmapImageRep *) currentContent;

@end
```

Se tienen cinco variables de instancia. Una de tipo booleano, `selection`, para saber cuando el usuario esta dibujando algo en el frame. Dos de tipo `NSPoint`, `locA` y `locB`, que se utilizarán para dibujar lineas o trazos a mano alzada. `selectedRect`, de tipo `NSRect`, para cuando el usuario dibuje rectángulos y círculos. Y `lastFrame`, un objeto `NSBitmapImageRep`, que representa el contenido del documento. Se tiene también un outlet hacia la instancia de nuestra clase `PPDocumentClass`.

El primer método de instancia, `-setFile:`, es el utilizado previamente para establecer el documento abierto por el usuario. El segundo, `-setContent:`, es utilizado por los eventos *Deshacer/Rehacer*. Y el tercero, `-currentContent`, que retorna el contenido del documento, también es utilizado por los eventos *Deshacer/Rehacer* y por el método encargado de guardar el documento, como se vio previamente.

Veamos ahora el código de la implementación, el cual presentamos por partes para comentarlo detenidamente. Los método que son redefiniciones de métodos en la clase `NSView`, se indican con un comentario. Esto para distinguirlos de los métodos propios de nuestra clase.

```
#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>
#import "PPController.h"
#import "PPDocumentClass.h"
#import "PPDocumentView.h"

@implementation PPDocumentView

- (void) awakeFromNib
{
  lastFrame = nil;
  selection = NO;
}

// Redefinicion
- (BOOL) acceptsFirstResponder
{
  return YES;
}

// Redefinicion
- (BOOL) acceptsFirstMouse: (NSEvent *)theEvent
{
  return YES;
}
```

En el método `-awakeFromNib` se establecen valores iniciales para algunas de las variables de instancia. Aquellas que es estricto que tengan un valor inicial, como veremos mas adelante. En el método `-acceptsFirstResponder` se retorna `YES`, para permitir que el *view* pueda ser el *first responder*. De otra forma, no podrá responder a las opciones del menú que estén conectadas al objeto `NSFirst`. El siguiente método, `-acceptsFirstMouse:`, retorna `YES`, lo que le permite al *view* responder al primer clic dado con el ratón. Lo que es útil después de que el usuario a interactuado con la paleta de colores.

Veamos ahora los siguientes tres métodos que son los métodos públicos que aparecen en la interfaz:

```
- (void) setFile: (NSBitmapImageRep *)image
{
  ASSIGN(lastFrame, image);
  [self setNeedsDisplay: YES];
}

- (void) setContent: (NSBitmapImageRep *)data
{
  ASSIGN(lastFrame, data);

  [[document undoManager] registerUndoWithTarget: self
                      selector: @selector(setContent:)
                        object: [self currentContent]];
  [self setNeedsDisplay: YES];
}

- (NSBitmapImageRep *) currentContent
{
  NSBitmapImageRep *bitmap;

  bitmap = [self bitmapImageRepForCachingDisplayInRect:
                       [self frame]];

  return bitmap;
}
```

El método `-setFile:` asigna la imagen que se pasa como parámetro (el documento seleccionado por el usuario) a la variable de instancia `lastFrame`, la cual representa el contenido del *view*. Y luego actualiza el contenido de este mediante `-setNeedsDisplay:`. El siguiente método, `-setContent:`, es el método que se ejecutará cada vez que el usuario seleccione *Deshacer* o *Rehacer* en el menú. Y lo analizaremos en detalle mas adelante. El tercer método, `-currentContent`, retorna el contenido del *view* como un objeto `NSBitmapImageRep`. Para ello se utiliza el método `-`

`bitmapImageRepForCachingDisplayInRect:`, pasándole como parámetro el lienzo del *view*, puesto que queremos capturar todo el contenido de este.

Veamos ahora los métodos `-mouseDown:`, `-mouseDragged:` y `-mouseUp:`, que serán los encargados de obtener los puntos `locA` y `locB` entre los que esta dibujando el usuario. Y de actualizar el contenido del *view* acorde a los cambios introducidos. Estos métodos se ejecutan, respectivamente, cuando el usuario presiona el botón izquierdo del ratón, arrastra el ratón manteniendo presionado este botón y cuando finalmente lo suelta.

```
// Redefinicion.
- (void) mouseDown: (NSEvent *)theEvent
{
  NSBitmapImageRep *bitmap = [self currentContent];

  [[document undoManager] registerUndoWithTarget: self
                          selector: @selector(setContent:)
                          object: bitmap];
  ASSIGN(lastFrame, bitmap);

  locA = [self convertPoint: [theEvent locationInWindow]
                  fromView: nil];
  selectedRect.origin = locA;
  selection = YES;

  switch ([[NSApp delegate] tool])
    {
    case 0:
      [[document undoManager] setActionName: @"Trazo"];
      break;
    case 1:
      [[document undoManager] setActionName: @"Linea"];
      break;
    case 2:
      [[document undoManager] setActionName: @"Circulo"];
      break;
    case 3:
      [[document undoManager] setActionName: @"Rectangulo"];
      break;
    }
}

// Redefinicion
- (void) mouseDragged: (NSEvent *)theEvent
{
  locB = [self convertPoint: [theEvent locationInWindow]
```

```
                    fromView: nil];
    NSSize size = NSMakeSize(locB.x - locA.x,
                             locB.y - locA.y);
    selectedRect.size = size;

    if ([[NSApp delegate] tool] == 0)
      {
        [self setNeedsDisplayInRect: NSMakeRect(locB.x - 2,
                                                locB.y - 2,
                                                4, 4)];
      }
    else
      {
        [self setNeedsDisplay: YES];
      }
  }

  // Redefinicion
  - (void) mouseUp: (NSEvent*)theEvent
  {
    ASSIGN(lastFrame, [self currentContent]);
    selection = NO;
  }
```

En el método `-mouseDown:`, antes de que el usuario agregue un cambio, obtenemos el contenido actual del documento mediante el método `-currentContent`. Seguidamente se registra un evento *undo* en el objeto `NSUndoManager` del documento. Esto se realiza mediante el método `-registerUndoWithTarget:selector:object:`. El primer parámetro de este método es el objeto al que se le enviará el mensaje cuando el usuario seleccione *Deshacer* en el menú. En este caso se trata de nuestro mismo objeto `PPDocumentView`. El segundo parámetro es el método a ejecutar, en este caso `-setContent:`. Y el último, es el objeto que se le pasará como parámetro a dicho método, en este caso el contenido del documento. De esta forma, el usuario podrá revertir el documento al estado en que se encontraba antes de que lo modificara.

Seguidamente se asigna el estado del documento a la variable `lastFrame`. Esto porque se debe preservar el contenido del mismo mientras el usuario termina de dibujar la linea, rectángulo o circulo. Luego se almacena, en la variable `locA`, la coordenada del punto donde el usuario presiono el botón. Para esto, primero se convierte dicha coordenada al sistema del *view*, ya que el objeto `theEvent` provee este dato en relación al sistema de la ventana. Se asigna también este punto como el origen de la variable `selectedRect`. Seguidamente se establece la variable `selection` a `YES`, la que nos permitirá saber posteriormente, en el método `-drawRect:`, si el usuario esta dibujando

o no. A continuación, mediante una sentencia `switch`, se determina la herramienta seleccionada y se establece el nombre del evento *undo* acorde a esta. Esto no es estrictamente necesario, pero mejora la experiencia del usuario final, al mostrar un texto descriptivo en el menú sobre las acciones que se pueden deshacer.

Veamos ahora el método `-mouseDragged:`, que se ejecuta cada vez que el usuario mueve el ratón con el botón izquierdo presionado. Primero se guarda en `locB` la ubicación del puntero, convirtiendo la coordenada como se menciono previamente. Luego se crea un objeto `NSSize` con las dimensiones del rectángulo entre los puntos `locA` y `locB`. Y seguidamente se asigna este tamaño a la variable `selectedRect`. Luego se verifica la herramienta seleccionada. Si es la de trazo, simplemente se actualiza (se redibuja) la pequeña porción del frame donde se dibujará la nueva sección del trazo. De otra forma se borraran las partes anteriores. Por otra parte, si la herramienta no es la de trazo, se actualiza todo el frame.

Y en el método `-mouseUp:`, que se ejecuta cuando el usuario termina de dibujar (cuando suelta el botón del ratón), se guarda el nuevo contenido del documento en la variable `lastFrame`, con el objeto de preservar el dibujo añadido por el usuario. Y seguidamente se establece la variable `selection` a `NO`.

Veamos por último el método `-drawRect:`, el encargado de dibujar el contenido del documento, y el método `-dealloc`.

```
// Redefinicion
- (void) drawRect: (NSRect)rect
{
  if (lastFrame != nil)
    {
      [lastFrame drawInRect: [self frame]];
    }
  else
    {
      [[NSColor whiteColor] set];
      [[NSBezierPath bezierPathWithRect: rect] fill];
    }

  if (selection)
    {
      NSBezierPath *path;
      [[[[NSApp delegate] colorWell] color] set];

      switch ([[NSApp delegate] tool])
        {
        case 0:
          {
```

```
                path = [NSBezierPath bezierPathWithRect:
                          NSMakeRect(locB.x - 2, locB.y - 2,
                                     4, 4)];
                [path fill];
              }
              break;
            case 1:
              {
                path = [NSBezierPath bezierPath];
                [path moveToPoint: locA];
                [path lineToPoint: locB];
                [path setLineWidth: 2];
                [path stroke];
              }
              break;
            case 2:
              {
                path = [NSBezierPath bezierPathWithOvalInRect:
                                              selectedRect];
                [path fill];
              }
              break;
            case 3:
              {
                path = [NSBezierPath bezierPathWithRect:
                                              selectedRect];
                [path fill];
              }
              break;
            }
        }
    }

    // Redefinicion
    - (void) dealloc
    {
      [lastFrame release];
      [super dealloc];
    }

    @end
```

El método `-drawRect:`, se ejecuta cada vez que se envían los mensajes `-setNeedsDisplay:` y `-setNeedsDisplayInRect:`. Si `lastFrame` no es igual a `nil`, es decir si hay contenido previo en el documento, se dibuja este. Para ello se utiliza el método `-drawInRect:`, pasándole como parámetro

el frame del *view*. Aquí no se utiliza el parámetro `rect`, porque este no necesariamente es el frame del *view*. Recuérdese que cuando se utiliza la herramienta de trazos, solamente se actualiza una pequeña porción del *view*. Por otro lado, si `lastFrame` es igual a `nil`, se dibuja un fondo blanco en el *view*. Aquí se utiliza el parámetro `rect`, porque esto solo ocurre cuando se abre un documento nuevo, y en este caso `rect` es el frame del *view*. Seguidamente, si el usuario esta dibujando algo (si `selection` es igual a `YES`), se traza el correspondiente dibujo.

En el caso de la herramienta de trazos, se dibuja un pequeño rectángulo de 4x4 píxeles. Aquí no se utiliza el parámetro `rect` pasado por `-setNeedsDisplayInRect:` (la pequeña porción a actualizar). Ya que cuando se utiliza este método, o similares a este, puede ocurrir que *GNUstep* modifique ligeramente la región a actualizar. Esto en vistas a mejorar el rendimiento. Sin embargo, esto haría que se dibujaran trazos de diferente grosor. Y no es esto lo que deseamos.

Finalmente, en el método `-dealloc`, se libera cualquier posible contenido de la variable `lastFrame`.

Por último, vamos a crear una subclase de la clase `NSPanel`, para asignarla al panel de herramientas. La interfaz de esta clase, a la que llamamos `PPToolsPanel`, es la siguiente:

```
#import <AppKit/NSPanel.h>

@interface PPToolsPanel : NSPanel
{
}
@end
```

Y la implementación:

```
#import "PPToolsPanel.h"

@implementation PPToolsPanel

- (BOOL) canBecomeKeyWindow
{
  return NO;
}

@end
```

Aquí simplemente se redefine el método `-canBecomeKeyWindow` para retornar `NO`. La razón de esto, es que en esta app el panel de herramientas no necesita recibir eventos del teclado. Por lo que no hay razón para que esta ventana se convierta en la *key window*.

Y con esto tenemos los archivos de código necesarios para nuestra app. Es momento de crear la interfaz gráfica.

9.2 La interfaz gráfica

Por defecto, *GNUstep* implementa el diseño *SDI* (*Simple Document Interface*) para las apps de documentos, lo que significa que cada documento se muestra en una ventana diferente. Esto se puede apreciar en la imagen 9-1. De esta forma, deberemos crear dos archivos de interfaz gráfica. El primero de ellos, que será la interfaz principal, contendrá el menú y el panel de herramientas. Y el segundo, será la ventana correspondiente a un documento de nuestra app.

Comencemos con el archivo de la interfaz principal. Abriendo *Gorm* seleccionamos en el menú Document → New Module → New Empty. O, si se creo un nuevo documento al abrir *Gorm*, simplemente eliminemos la ventana de este. Seguidamente agreguemos los ítems *Info*, *Document*, *Edit* y *Windows* al menú de la app. Arrastrándolos desde la paleta *Menus*, imagen 9-2.

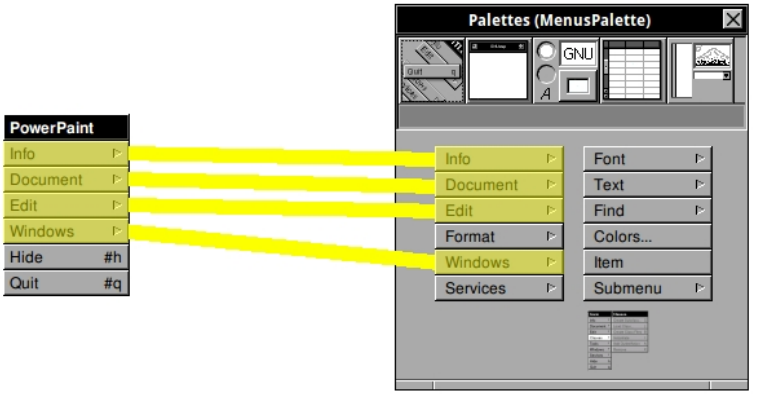

Imagen 9-2. Menú de la app *PowerPaint*.

En los submenús *Edit* y *Document* deben removerse aquellos ítems que no implementaremos en esta app. Estos deben quedar como se muestran en la imagen 9-3. El ítem *Close*, en el menú *Document*, esta conectado al método `-close:` en el objeto `NSFirst`. Sin embargo, debe cambiarse al método `-performClose:` del mismo objeto. Si se desea, pueden ponerse todos los títulos en español, excepto los de *Undo*/*Redo*/*Clear List* que son manejados por *GNUstep*, el cual se encargará de traducirlos. Siempre de acuerdo a las recomendaciones para una interfaz gráfica, los ítems que abren paneles deben terminar con puntos suspensivos.

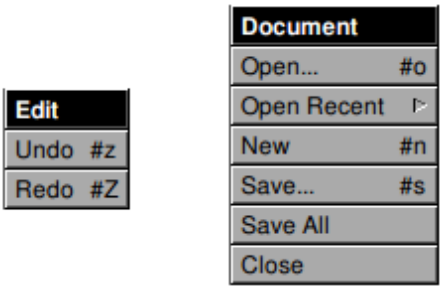

Imagen 9-3. Menús *Edit* y *Document*.

Estas opciones en los menús *Edit* y *Document* están conectadas al objeto `NSFirst`. Para que estas funcionen según lo esperado, nosotros simplemente debemos preocuparnos por proveer métodos para leer y guardar los documentos, y de registrar los eventos *undo/redo*.

La opción *Revert To Saved*, requiere que se implemente el método `-revertDocumentToSaved:` en nuestra clase `PPDocumentClass`. Y las opciones *Cut*, *Copy*, *Paste* y *Select All*, requieren la implementación de los métodos `-cut:`, `-copy:`, `-paste:` y `-selectAll:`, respectivamente, en nuestra clase `PPDocumentView`. Sin embargo, por simplicidad, omitiremos estas opciones aquí.

En caso de implementar las opciones *Cut/Copy/Paste*, es recomendable implementar, en nuestra clase `PPDocumentView`, el método `-validateMenuItem:`. Este método recibe como parámetro un objeto `NSMenuItem` y retorna un valor booleano. Es ejecutado cada vez que *GNUstep* actualiza el menú, para saber si un determinado ítem debe o no estar habilitado en el menú. Funciona de la siguiente forma:

- El menú busca el objeto que implementa el método que el ítem ejecuta. Si no se encuentra ninguno, entonces se deshabilita el ítem.
- Si se encuentra un objeto que implemente el método, entonces se busca la implementación del método `-validateMenuItem:` en el mismo objeto. Si no se encuentra, se habilita el ítem, puesto que el método que este ejecuta se ha encontrado.
- Si se encuentra implementado el método `-validateMenuItem:`, entonces el estado del ítem es determinado por el valor devuelto por este método.

En nuestra app, este método debería manejar las opciones *Cut/Copy/Paste*. Estas opciones podrían diferenciarse entre si agregando un *tag*, en el *Inspector*, a cada uno de los ítems. De esta forma, se tendría algo como:

```
- (BOOL) validateMenuItem: (NSMenuItem *)anItem
```

```
{
  switch ([anItem tag])
    {
    case 1:
      {
        // Instrucciones.
      }
      break;
    case 2:
      {
        // Instrucciones.
      }
      break;
    case 3:
      {
        // Instrucciones.
      }
      break;
    }
}
```

Obsérvese que para implementar las opciones *Cut/Copy* debería, primero, implementarse una herramienta de selección.

Regresando a nuestra interfaz gráfica, agreguemos ahora un panel al documento *gorm*, desde la paleta *Windows*. Deben agregarse cuatro botones y una paleta de colores a este panel. Cada botón corresponde a una herramienta de dibujo, imagen 9-4. Lo ideal es agregar iconos en vez de texto en los botones, pero lo dejaremos así por el momento. Siempre de acuerdo a las recomendaciones para una interfaz gráfica, los paneles no deben tener botones de miniaturizar. Y como para este caso no tienen sentido un botón cerrar y la barra para redimensionar, los eliminamos también. El panel debe estar configurado para ser visible al momento de leer el archivo '`gorm`'.

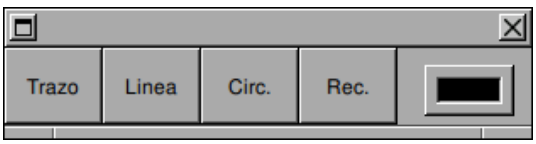

Imagen 9-4. Panel de herramientas.

Para poder identificar a cada herramienta asignaremos un *tag*, en el *Inspector*, a cada uno de los botones en el panel. El valor cero será para la herramienta de trazos, uno para la de lineas, dos para la de círculos y tres para la de rectángulos. Hecho esto, carguemos los archivos de interfaz de nuestras clases `PPToolsPanel` y `PPController` y creemos una instancia de esta última. Acto seguido, conectemos los cuatro botones del panel con

el método `-setTool:` de nuestra clase `PPController`. Luego conectemos el outlet *colorWell*, de esta clase, a la paleta de colores en el panel de herramientas. Y finalmente conectemos el objeto `NSOwner` (el objeto `NSApplication` que controla nuestra app), a la instancia de nuestra clase `PPController` mediante el outlet *delegate*.

Por último, asígnese la clase `PPToolsPanel` como la clase para el panel de herramientas. Y con esto tenemos la interfaz principal de nuestra app, la que debemos guardar con el nombre de '`PowerPaint`'.

Vamos a crear ahora la interfaz para los documentos de nuestra app. Nuevamente abramos *Gorm* y seleccionemos en el menú Document → New Module → New Empty. O, si se creo un nuevo documento al abrir *Gorm*, simplemente eliminemos el menú de este. Ahora agreguemos un elemento *CustomView* a la ventana, y modifiquemos su tamaño de tal forma que ocupe toda el área de la ventana. Seguidamente configuremos el ajuste de tamaño automático para este elemento. En el *Inspector*, en la sección *Size*, establezcamos que tanto el alto como el ancho se ajusten automáticamente, imagen 9-5.

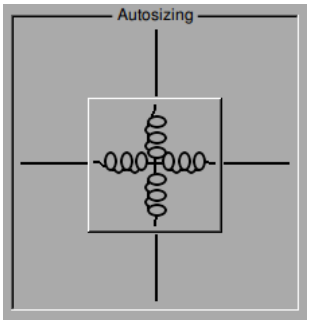

Imagen 9-5. Ajuste automático.

Carguemos entonces los archivos de interfaz de nuestras clases `PPDocumentClass` y `PPDocumentView`. Y hagamos que el objeto `NSOwner` sea una instancia de la clase `PPDocumentClass`. Y que el objeto *CustomView* sea una instancia de la clase `PPDocumentView`. Hecho esto, conectemos el outlet *view* del objeto `NSOwner` (nuestra clase `PPDocumentClass`) al objeto *CustomView*. Hagamos también una conexión entre el objeto `NSOwner` y la ventana, mediante el outlet *_window*, imagen 9-6. Este en un outlet privado de la clase `NSDocument`, y le permite a esta clase saber cual es la ventana del documento.

Imagen 9-6. Outlet *_window*.

Ahora debe realizarse una conexión entre el *CustomView* y el objeto `NSOwner` mediante el outlet *document*. Y con esto tenemos finalizada la interfaz del documento, la que debemos guardar con el nombre '`Document`'. En la siguiente sección crearemos el archivo *GNUmakefile* y otro archivo indispensable para el funcionamiento de una app de documentos.

9.3 GNUmakefile e Info.plist

Por supuesto, necesitamos un archivo con la función *main*, al que llamaremos '`PPMain.m`' y cuyo contenido es el habitual:

```
#import <AppKit/AppKit.h>

int
main(int argc, const char *argv[])
{
  return NSApplicationMain (argc, argv);
}
```

Veamos ahora el archivo '`GNUmakefile`', el cual no necesita mayor explicación:

```
include $(GNUSTEP_MAKEFILES)/common.make

# Application
APP_NAME = PowerPaint
PowerPaint_MAIN_MODEL_FILE = PowerPaint.gorm

# Resource files
PowerPaint_RESOURCE_FILES = \
PowerPaint.gorm \
Document.gorm

# Header files
PowerPaint_HEADER_FILES = \
PPController.h \
```

```
PPDocumentClass.h \
PPDocumentView.h \
PPToolsPanel.h

# Class files
PowerPaint_OBJC_FILES = \
PPController.m \
PPDocumentClass.m \
PPDocumentView.m \
PPToolsPanel.m

# Other sources
PowerPaint_OBJC_FILES += \
PPMain.m

# Makefiles
-include GNUmakefile.preamble
include $(GNUSTEP_MAKEFILES)/aggregate.make
include $(GNUSTEP_MAKEFILES)/application.make
-include GNUmakefile.postamble
```

Por último, necesitamos un archivo al que llamaremos '`PowerPaintInfo.plist`'. Este es un archivo *property list*, de allí la extensión *plist*, con información que la app puede acceder cuando esta sea requerida. El contenido de este archivo es el siguiente:

```
{
  ApplicationName = PowerPaint;
  ApplicationDescription = "My first document app!";
  ApplicationRelease = "0.1";
  Copyright = "Copyright (C) 2013";
  CopyrightDescription = "Released under the GPLv3";
  NSTypes = (
    {
      NSName = "jpg";
      NSHumanReadableName = "Imagen JPG";
      NSIcon = "FileIcon_jpg.tiff";
      NSUnixExtensions = ( jpg );
      NSDOSExtensions = ( jpg );
      NSRole = Editor;
      NSDocumentClass = PPDocumentClass;
    }
  );
}
```

Los nombres de las *claves* en este diccionario, son los valores que por defecto se buscarán en este archivo. Los primeros cinco datos, son información general sobre la app que aparecerá en el panel de información de esta (opción del menú Info → Info Panel...). Obsérvese que se utilizan comillas dobles en aquellos datos que contienen espacios.

Aquí el dato importante para nuestra app es `NSType`, el cual contiene un *array* de diccionarios. Uno para cada tipo de archivo soportado. Como nuestra app solo soporta archivos *JPG*, entonces aparece solamente un diccionario. Pasemos entonces a analizar la información de este.

El dato `NSName` es un texto para identificar, de forma interna, el tipo de archivo. Este es el parámetro `fileType` que se le pasa a los métodos `-readFromFile:ofType:` y `-writeToFile:ofType:` vistos anteriormente, cuando el usuario abre o guarda un archivo. El siguiente dato, `NSHumanReadableName`, es el texto que se mostrará en la lista desplegable (con los tipos soportados) del panel *Guardar*. Como nuestra app solo soporta un tipo de archivo, este es el único ítem que aparecerá en dicha lista desplegable. El dato `NSIcon`, es el icono que representa al tipo de archivo en cuestión. Aquí se muestra como ejemplo, puesto que no se agregará ninguna imagen. Este icono es el utilizado en los paneles *Abrir*/*Guardar* para representar a estos archivos. El siguiente dato, `NSUnixExtensions`, es un *array* con las extensiones utilizadas por este tipo de archivo en los sistemas tipo *Unix*. Aquí solamente se ha agregado una entrada, aunque también puede agregarse `JPG` para el caso cuando la extensión este en mayúsculas. El dato `NSDOSExtensions` representa lo mismo que la entrada anterior, solo que para sistemas *Windows*. Estas dos entradas le permiten a los paneles *Abrir* y *Guardar* saber que tipos de archivos deben mostrar.

La entrada `NSRole`, indica si la app es un editor de este tipo de archivo (`Editor`), o si solamente es un visor (`Viewer`). En este caso, nuestra app es un `Editor`. La siguiente entrada, `NSDocumentClass`, indica que clase es la que controla el tipo de archivo en cuestión. De esta forma, el objeto `NSApplication` que controla nuestra app, sabe que clase es la que debe instanciar cuando el usuario abre o crea un determinado tipo de archivo. Obsérvese que es el *nombre de la clase*, no el nombre del archivo de la clase. Que aunque en general es el mismo, puede que en algunos casos sea diferente.

Como se ve, este archivo es muy importante para que funcione toda la maquinaria de una app de documentos. Este archivo no debe agregarse en el '`GNUmakefile`', ya que la herramienta *gnustep-make* lo busca siempre que se compila un proyecto.

Por último, solo nos queda compilar y probar nuestra app, imagen 9-7. Compruébese el funcionamiento de las herramientas, de la paleta de colores y de las distintas opciones del menú.

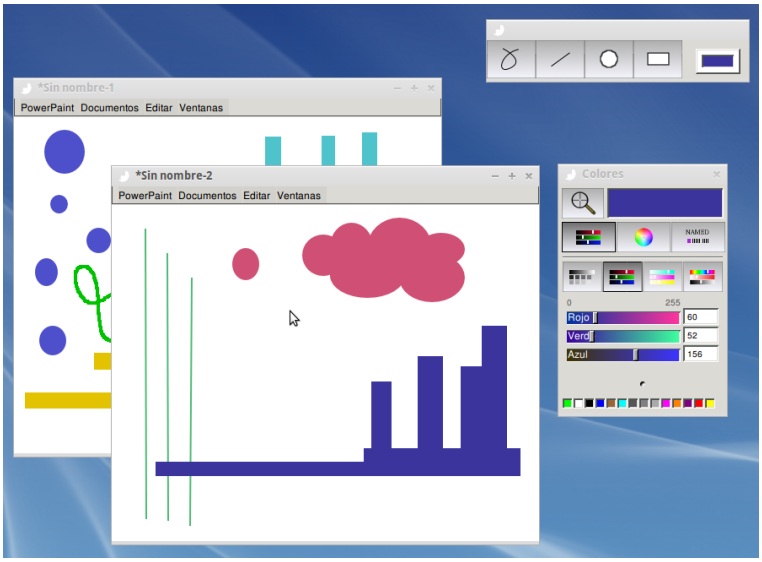

Imagen 9-7. Probando *PowerPaint*.

Como se ve en la imagen se han agregado iconos, de 30x30 píxeles, a los botones del panel de herramientas. Y se han traducido todos los textos en el menú.

Para el caso de que se utilice el estilo de menú empotrado en la ventana, como se muestra en la imagen, es necesario realizar todavía un pequeño cambio para que funcione sin problemas. Sin embargo, dejaremos esto para el último capítulo, donde se aborda detenidamente lo referente al estilo del menú.

10 Proyectos en GNUstep

En este capítulo veremos como hacer que una app este disponible en varios idiomas. Así como la forma en que podemos organizar nuestros proyectos, para el caso cuando estos contienen muchos archivos y se vuelve necesario establecer un orden en el mismo. Por último, veremos algunas opciones adicionales que podemos agregar a los archivos '`Info.plist`'.

10.1 Internacionalización

La *internacionalización* se refiere a cuando una aplicación esta disponible en más de un idioma. Esto se logra agregando carpetas para los distintos idiomas disponibles e indicando esto en el archivo *GNUmakefile*. Estas carpetas deben llevar por nombre el nombre del idioma correspondiente, el nombre estandarizado, y la extensión '`lproj`'. Y deben contener el mismo número de archivos con los mismos nombres. Con la diferencia de que en cada carpeta, el contenido de estos debe estar en el respectivo idioma. A continuación se muestran los nombres de estas carpetas para algunos idiomas:

```
English.lproj
Spanish.lproj
German.lproj
French.lproj
```

Ahora bien, la traducción a otros idiomas se hace en dos partes. La primera parte se refiere a los textos presentes en el código de nuestros archivos, en el caso de que hayan y deban ser traducidos. Estas cadenas de texto deben escribirse dentro de la función `_()`. Por ejemplo, en el código para mostrar un panel de alerta:

```
NSRunAlertPanel(_(@"Error"),
                _(@"An error occurred when copy the file."),
                _(@"OK"), nil, nil);
```

Al ejecutarse este código, las tres cadenas de texto serán traducidas al respectivo lenguaje. En el caso de no existir una traducción para estas, se dejaran tal y como están. Esta traducción se realiza utilizando un archivo llamado *Localizable.strings*, uno para cada idioma en las carpetas de idiomas. A la hora de arrancar la aplicación, se busca el archivo *Localizable.strings* correspondiente al idioma, y se traducen todas aquellas cadenas de texto presentes en el código que tengan aplicada la función `_()`. *GNUstep* provee una herramienta llamada *make_strings*, que facilita la construcción de dichos archivos. Esta herramienta se ejecuta en la carpeta de nuestro proyecto (en una terminal), y recibe como parámetros los lenguajes a añadir en la app y los archivos en los que deben buscarse las cadenas de texto. Por ejemplo:

```
make_strings -L "English Spanish French" *.m
```

El parámetro `L` le indica a la herramienta los lenguajes de la app, los cuales se pasan a continuación entre comillas dobles. Seguidamente se indican los archivos en los que deben buscarse las cadenas de texto. En este ejemplo, todos los archivos con extensión '`m`'. El carácter `*`, es un *comodín* para representar cualquier secuencia de caracteres. Esta herramienta agrega también varios comentarios que indican en que archivo, y en que número de linea, se utiliza cada una de las cadenas de texto. Con lo que no nos perderemos a la hora de traducir las mismas en aplicaciones que contengan una buena cantidad de estas. El siguiente es parte de un archivo *Localizable.strings* generado con esta herramienta, y para la traducción del inglés al español. Las líneas que comienzan con `/*` y terminan con `*/`, son sólo comentarios:

```
/***
Spanish.lproj/Localizable.strings
updated by make_strings 2010-10-21 19:00:31 -0600
add comments above this one
***/


/*** Keys found in multiple places ***/

/* File: Board.m:753 */
/* Flag: untranslated */
/* File: Board.m:777 */
/* Flag: untranslated */
/* File: Board.m:783 */
/* Flag: untranslated */
"Game Over" = "Fin del juego";

/* File: main.m:57 */
/* Flag: untranslated */
/* File: main.m:60 */
/* Flag: untranslated */
"Info" = "Informaci\U00F3n";
```

El *flag untranslated* debe cambiarse a *translated* para aquellas cadenas de texto que ya estén traducidas. De otra forma, *make_strings* eliminara las traducciones al regenerar el archivo. Regeneración que debe hacerse cada vez que se agreguen nuevas cadenas de texto en el código. El lenguaje de las cadenas de texto en el código debe estar a la izquierda del carácter `=` y la correspondiente traducción a la derecha. Los caracteres que no pertenezcan

al inglés deben ingresarse mediante su código correspondiente. En el ejemplo anterior, no se ha escrito *ó* en *Información*. En su lugar se ha agregado el correspondiente código *U00F3*, precedido por el carácter \. El editor de código *Gemas* escribe directamente estos códigos cuando se trata de ingresar un carácter que no pertenece al inglés, lo que evita tener que buscar el correspondiente código en un panel de caracteres.

Como se infiere de lo anterior, las cadenas de texto en el código deben contener únicamente caracteres del idioma inglés, o preferiblemente estar en inglés. Y luego realizar la corrección o traducción correspondiente en el archivo *Localizable.strings*.

La segunda parte de la traducción, consiste en traducir los archivos de interfaz gráfica. Estos archivos deben estar presentes en todas las carpetas de idiomas, traducidos al idioma correspondiente. Abriendo cada uno de estos en *Gorm* pueden traducirse los textos que contengan, así como corregir el tamaño de los componentes para que las traducciones se muestren correctamente. O incluso redistribuir estos para una mejor presentación. Para facilitar esta tarea, *Gorm* ofrece la posibilidad de generar un archivo de cadenas de texto con la opción del menú Document → Translate → Export Strings, para luego traducir el archivo resultante de la misma forma en que se hace con los archivos *Localizable.strings*. Y, una vez traducido, cargarlo mediante la opción del menú Document → Translate → Load Strings, lo que traducirá los textos de la interfaz gráfica. Esta opción facilita bastante la traducción, aunque si hay textos ocultos (porque la opción que los muestra no esta seleccionada en ese momento), estos no estarán presentes en el archivo de cadenas de texto. Sin embargo, es posible agregarlos a este con un editor, en el caso de que no se desee traducirlos directamente en el archivo '`gorm`'. Recuérdese que en *Gorm* podemos introducir directamente los signos y caracteres del español sin ningún problema.

Si una app se ejecuta en un idioma que no este disponible, *GNUstep* la ejecutara en inglés. Si este idioma tampoco esta disponible, no se mostrara ninguna interfaz gráfica. Esta es la razón por la que se recomienda que el idioma inglés este siempre disponible. Sin embargo, las apps que no apliquen la *internacionalización*, siempre se ejecutaran tal y como están.

10.2 Archivos GNUmakefile

Hasta ahora hemos colocado todos los archivos de nuestras apps o tools dentro de una carpeta, sin establecer orden alguno. Sin embargo, cuantos mas archivos contenga nuestro proyecto, mas necesario se vuelve el ordenar estos en subcarpetas, para facilitar el mantenimiento del mismo. Esto puede hacerse en la forma deseada, siempre y cuando reflejemos en nuestro archivo *GNUmakefile* los cambios correspondientes. Para archivos de recursos, como imágenes o sonidos, es posible agregar toda la carpeta sin necesidad de enumerar todo su contenido. El siguiente archivo *GNUmakefile* es un ejemplo de esto, donde además se ejemplifican otras opciones que no se habían mostrado anteriormente:

```
include $(GNUSTEP_MAKEFILES)/common.make

# Las siguientes dos lineas son utilizas para el nombre
# del paquete de codigo o instalador.
VERSION = 0.1
PACKAGE_NAME = powerpaint

# Nombre de la app.
APP_NAME = PowerPaint

# Nombre del icono de la app.
PowerPaint_APPLICATION_ICON = paint

# Archivo principal de interfaz grafica.
PowerPaint_MAIN_MODEL_FILE = PowerPaint.gorm

# Lenguages de nuestra app.
PowerPaint_LANGUAGES = \
English \
Spanish \
French

# Recursos localizables de nuestra app.
PowerPaint_LOCALIZED_RESOURCE_FILES = \
PowerPaint.gorm \
Document.gorm \
Localizable.strings

# Recursos de nuestra app.
PowerPaint_RESOURCE_FILES = \
Resources/paint.tiff \
Resources/paint.ico \
Resources/Images

# Archivos de interfaz de nuestros objetos.
PowerPaint_HEADER_FILES = \
Headers/PowerPaintController.h \
Headers/PowerPaintDocument.h \
Headers/PaintView.h

# Archivos de implementacion de nuestros objetos.
PowerPaint_OBJC_FILES = \
Core/PowerPaintController.m \
Core/PowerPaintDocument.m \
Core/PaintView.m
```

```
# Archivo con la funcion main.
PowerPaint_OBJC_FILES += \
PowerPaint_main.m

# Makefiles.

# Archivo con flags para enlazar o utilizar programas
# externos a GNUstep.
-include GNUmakefile.preamble

include $(GNUSTEP_MAKEFILES)/application.make

# Instrucciones a utilizar despues de la instalacion.
# Raramente utilizado.
-include GNUmakefile.postamble
```

Como se indica en el comentario, las entradas `VERSION` y `PACKAGE_NAME` son utilizadas para crear los paquetes de código o instaladores de *Windows*. Para este ejemplo los nombres de estos serían, respectivamente: *powerpaint-0.1.tar.gz* y *powerpaint-0.1-setup.exe*. Luego se indica el nombre de la imagen que hace de icono de la aplicación, sin extensión. Dicha imagen debe ser de 48x48 píxeles. La entrada de lenguajes le indica a la herramienta de construcción que carpetas de idiomas deben incluirse en la app. Estas carpetas deben ser subcarpetas de la carpeta principal del proyecto. Es decir, no pueden reubicarse dentro de esta. Como se ve, los recursos de la app han sido ubicados en una subcarpeta llamada *Resources*. Hay dos archivos para el icono de la app, uno en formato *TIFF*, para sistemas tipo *Unix*, y otro en formato *ICO*, para sistemas *Windows*. El resto de imágenes han sido ubicadas dentro de una subcarpeta llamada *Images*. Y solamente se ha agregado la carpeta, evitando enumerar todas estas. Por último, los archivos de interfaz se han colocado en la subcarpeta *Headers*, mientras que los archivos de implementación se han ubicado en la subcarpeta *Core*.

El guión antes de `include`, para los archivos `GNUmakefile.preamble` y `GNUmakefile.postamble`, le indican a la herramienta de construcción que se continué con la compilación en caso de que estos archivos no estén presentes.

10.3 Archivos Info.plist

Este archivo puede llamarse '`NombreAppInfo.plist`' o '`NombreApp-Info.plist`'. Y debe estar ubicado directamente en la carpeta del proyecto. Como hemos visto, en este archivo se agregan los datos que serán mostrados en el panel de información de la app. Información como: nombre de la app, descripción de esta, licencia, etc. Otros datos que pueden agregarse, aparte de los vistos en el ejemplo del

capítulo 9, son los nombres de los autores, la página de internet y alguna información adicional sobre la versión de la app:

```
Authors = (
   "Germ\U00E1n Arias"
);
NSBuildVersion = "March 2013";
URL = www.ejemplo.com;
```

La entrada *Authors* es un *array*, de allí el uso de los paréntesis. Para el caso de mas de un autor, estos deben separarse por comas. Por ejemplo:

```
Authors = (
   "Germ\U00E1n Arias",
   Fulano
);
```

Obsérvese que al igual que en los archivos de cadenas de texto, los caracteres que no pertenecen al inglés deben escribirse mediante su código correspondiente.

Por supuesto, estos archivos también pueden utilizarse en apps que no sean de documentos.

11 Apariencia y portabilidad

En este capítulo consideraremos primero los aspectos a tener en cuenta si queremos que nuestras apps soporten el estilo de menú empotrado en la ventana (`NSWindows95InterfaceStyle`). Así como aquellas relativas a la *portabilidad* de nuestros programas. Por ejemplo, si desarrollamos nuestro programa en un sistema *GNU/Linux* y queremos que este se ejecute en un sistema *Windows* y/o *Mac OS X*.

11.1 Apariencia

Por defecto *GNUstep* utiliza el menú vertical estilo *NextStep* (`NSNextStepInterfaceStyle`). Pero podría ser que quisiéramos darle soporte a nuestra app para utilizar un menú empotrado en la ventana (`NSWindows95InterfaceStyle`). Consideremos primero el caso de una app que no es de documentos. En este caso, el principal requisito es que la app provea una ventana para colocar el menú. Para el caso en que nuestra app no muestra una ventana al momento en que es lanzada, deberemos asegurarnos de hacer visible alguna ventana para que el menú pueda ser insertado en ella. Si nuestra app muestra mas de una ventana, deberemos elegir una de estas para colocar el menú. Aunque lo recomendable es que una app tenga solamente una ventana principal y que el resto sean paneles (`NSPanel`). Por defecto, los paneles no pueden albergar un menú.

Cuando se utiliza el estilo `NSWindows95InterfaceStyle`, *GNUstep* agrega el menú en aquellas ventanas cuyos métodos `-canBecomeMainWindow` devuelven `YES`. Ya que el menú debe estar presente solamente en una ventana, deberemos asegurarnos que el resto de ventanas, en caso de haberlas, retornen `NO` en sus métodos `-canBecomeMainWindow`. Para realizar esto, debemos crear una subclase de la clase `NSWindow`, para redefinir el método `-canBecomeMainWindow` de tal forma que retorne `NO`. Y hacer que el resto de ventanas sean una instancia de esta clase. La interfaz de dicha clase, que aquí llamamos *OtherWindow*, seria:

```
#include <AppKit/NSWindow.h>

@interface OtherWindow : NSWindow
{
}
@end
```

Y su correspondiente implementación:

```
#include "OtherWindow.h"

@implementation OtherWindow
```

```
- (BOOL) canBecomeMainWindow
{
  return NO;
}

@end
```

Luego debemos agregar esta clase en nuestro archivo de interfaz gráfica. Y hacer que las ventanas que no deben mostrar el menú, sean una instancia de esta clase. Esto es, cambiar la clase de dichas ventanas en el *Inspector*. Por supuesto, debemos agregar los archivos de la clase *OtherWindow* (o como queramos llamarla) a nuestro archivo *GNUmakefile*.

Por defecto, cuando se utiliza el estilo de menú `NSWindows95InterfaceStyle`, *GNUstep* cierra la app si el usuario cierra la última ventana o panel desplegado en la pantalla. O, en el caso de que haya mas de una ventana en pantalla, *GNUstep* cierra la app si el usuario cierra la única ventana cuyo método `-canBecomeMainWindow` retorna `YES`. Es decir, si cierra la ventana que contiene el menú. Para los otros estilos de menú, `NSNextStepInterfaceStyle` o `NSMacintoshInterfaceStyle`, *GNUstep* no cerrará la app así el usuario haya cerrado todas las ventanas. Ya que, en general, puede volver a abrir dichas ventanas desde alguna opción del menú. Y en caso de que el menú no sea visible en pantalla, puede hacerlo visible dando un doble clic sobre el *AppIcon* de esta.

Sin embargo, cuando se utiliza el estilo de menú `NSWindows95InterfaceStyle`, deben considerarse varios escenarios. Consideremos primero el caso de una app que tiene solamente una ventana. Aquí el comportamiento por defecto es conveniente, ya que la app se cierra cuando el usuario cierra la ventana. Y esto es, por supuesto, lo esperado. Ahora consideremos el caso cuando la app tiene dos ventanas, o una ventana y un panel, ambas con un botón cerrar. Si ambas ventanas están desplegadas en la pantalla y el usuario cierra la que no tiene el menú, la app no se cerrará, ya que el método `-canBecomeMainWindow` de la otra ventana retorna `YES`. Pero si el usuario cierra la ventana con el menú, la app se cerrará, ya que el método `-canBecomeMainWindow` de la otra ventana (panel) retorna `NO`. Hasta aquí todo bien. Sin embargo, debe considerarse el caso cuando una de las ventanas este minimizada. En este caso, no importando que ventana sea la que este minimizada, *GNUstep* cerrará la app si el usuario cierra la ventana en pantalla. Ya que dicha ventana es la *última* ventana en pantalla. No obstante, si deseamos que la app se cierre únicamente cuando la ventana con el menú sea cerrada, deberemos implementar un par de métodos delegados. El método `-windowWillClose:` de la clase `NSWindow` y el método `-applicationShouldTerminateAfterLastWindowClosed:` de la

clase `NSApplication`. El código sería el siguiente, donde *mainWindow* es un outlet hacia la ventana que contiene el menú:

```
- (void) windowWillClose: (NSNotification *)aNotification
{
  // Se establece el outlet a nil al cerrar la ventana.
  mainWindow = nil;
}

- (BOOL) applicationShouldTerminateAfterLastWindowClosed:
                                                    (id)sender
{
  if (NSInterfaceStyleForKey(@"NSMenuInterfaceStyle", nil)
      == NSWindows95InterfaceStyle)
    {
      // Cerramos la app solo si la ventana con el menu
      // ha sido cerrada.
      if (mainWindow == nil)
        {
          return YES;
        }
      else
        {
          return NO;
        }
    }
   else
    {
      // Con un menu independiente no cerramos la app.
      return NO;
    }
}
```

Aquí se utiliza la función `NSInterfaceStyleForKey()`, para conocer el estilo del menú establecido en las preferencias del usuario (*NSMenuInterfaceStyle*). El método delegado `-windowWillClose:`, se ejecuta cuando la ventana con el menú esta por cerrarse. Aquí se establece el outlet *mainWindow*, que conecta con dicha ventana, a `nil`. De esta forma, podremos evaluar posteriormente si esta ventana ha sido cerrada. El otro método, `-applicationShouldTerminateAfterLastWindowClosed:`, es ejecutado por *GNUstep* cuando el usuario cierra la última ventana o panel desplegado en la pantalla. O cuando se cierra la única ventana en pantalla cuyo método `-canBecomeMainWindow` retorna `YES`. En este método, si el estilo es `NSWindows95InterfaceStyle`, verificamos si la ventana con el menú ha sido cerrada. Si este es el caso, se retorna `YES` para cerrar la

app. De lo contrario se retorna `NO`. Por otro lado, si el estilo de menú no es `NSWindows95InterfaceStyle`, se retorna `NO`.

Para el caso de una app de documentos, las consideraciones son ligeramente diferentes. Como ya hemos visto, cuando se trata de apps de documentos, *GNUstep* implementa el diseño *SDI* (*Simple Document Interface*). Lo que significa que cada documento posee su propia ventana, como se muestra en la imagen 10-1. Aunque es posible redefinir algunas clases para obtener un diseño *MDI* (*Multiple Document Interface*), donde todos los documentos estén contenidos en una misma ventana, este caso se corresponde con lo visto en los párrafos anteriores. Por lo que nos enfocaremos aquí en el diseño *SDI*.

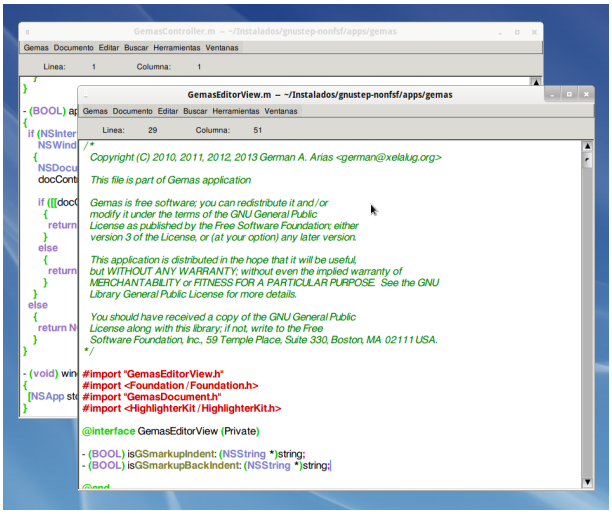

Imagen 10-1. Simple Document Interface.

Como se ve en la imagen, cada ventana (documento) tiene su propio menú. Por defecto, toda app de documentos crea un documento nuevo al momento de iniciar. Esto es conveniente, ya que de lo contrario no abra una ventana donde colocar el menú y el usuario no tendrá forma de interactuar con la app. Sin embargo, es posible que se desee este comportamiento únicamente cuando se utilice el estilo de menú empotrado en la ventana. Esto se puede lograr fácilmente con el método delegado `-applicationShouldOpenUntitledFile:` de la clase `NSApplication`, el cual se ejecuta al iniciar la app. La implementación de este método sería de la siguiente forma:

```
- (BOOL) applicationShouldOpenUntitledFile:
                              (NSApplication *)sender
{
  if (NSInterfaceStyleForKey(@"NSMenuInterfaceStyle", nil)
      == NSWindows95InterfaceStyle)
```

```
    {
      return YES;
    }
  else
    {
      return NO;
    }
}
```

Si el estilo es `NSWindows95InterfaceStyle` se retorna `YES`, de lo contrario se retorna `NO`. Ya que si el menú no debe ir empotrado en una ventana, no es necesario crear un documento nuevo. El tipo de documento creado, será el primero que aparezca en la lista de tipos del archivo '`Info.plist`'.

El siguiente paso es asegurar que la app se cierre únicamente cuando el usuario cierre la ventana del último documento. Para ello, debemos verificar primero si hay documentos minimizados en la barra de tareas. De lo contrario, *GNUstep* cerrara la app si el usuario cierra la última ventana en pantalla, no importando si hay documentos minimizados. Utilizando el método delegado `-applicationShouldTerminateAfterLastWindowClosed:`, la implementación sería de la siguiente forma:

```
- (BOOL) applicationShouldTerminateAfterLastWindowClosed:
                                                    (id)sender
{
  if (NSInterfaceStyleForKey(N̈SMenuInterfaceStyle", nil)
      == NSWindows95InterfaceStyle)
    {
      NSDocumentController *docController =
        [NSDocumentController sharedDocumentController];

      // Si hay documentos abiertos, no cerramos la app.
      if ([[docController documents] count] > 0)
        {
          return NO;
        }
      else
        {
          return YES;
        }
    }
  else
    {
      return NO;
    }
}
```

La verificación se realiza únicamente si el estilo de menú es `NSWindows95InterfaceStyle`. Para los otros estilos se retorna `NO`. Primero se obtiene el objeto `NSDocumentController` encargado de administrar los documentos. Aquí asumimos que se esta utilizando el objeto controlador de documentos que por defecto provee *GNUstep*. Seguidamente, mediante el método `-documents`, obtenemos el *array* que contiene los documentos abiertos. Y luego, mediante `-count`, se obtiene la cantidad de casillas en dicho *array* (la cantidad de documentos abiertos). Si la cantidad es mayor que cero, es decir, si hay documentos abiertos pero minimizados, se retorna `NO`. De lo contrario se retorna `YES`.

GNUstep reorganiza el menú antes de colocarlo en la ventana. Los ítems simples (los que no despliegan submenús), mas los ítems *Información* y *Servicios*, se agrupan en un nuevo ítem con el nombre de la app, el cual se convierte en el primer ítem del menú. La imagen 10-2, muestra esta reorganización para la app *Gorm*. La razón de esto es simplemente mejorar la presentación del menú para este estilo.

Imagen 10-2. Reorganización del menú.

Y con esto hemos cubierto las consideraciones a tomar en cuenta si se desea que una app tenga soporte para el estilo de menú `NSWindows95InterfaceStyle`.

11.2 Portabilidad

La principal consideración a tomar en cuenta es en lo referente al sistema *Mac OS X*. Aunque *GNUstep* puede instalarse en este sistema operativo, y de esta forma ejecutar nuestras apps usando *GNUstep* en *Mac OS X*, no hay paquetes actualizados de *GNUstep* para este sistema. Por lo que la instalación debe realizarse desde el código fuente. Esto, por supuesto, complica la instalación de nuestras apps. De esta forma, lo común es que las apps se ejecuten utilizando los frameworks de *Cocoa*. El principal problema con esto, es que *Cocoa* no puede manejar los archivos '`gorm`', los cuales son exclusivos de *GNUstep*. Una solución es abrir los archivos de interfaz en

Gorm y guardarlos con formato '`nib`' en la misma carpeta y con el mismo nombre. *Gorm* tiene un soporte limitado para este formato, por lo que no debe extrañar que no pueda realizarse la conversión para algunos archivos '`gorm`'. Los archivos '`nib`' son los utilizados por *Cocoa* para almacenar la interfaz gráfica de una app. *Apple* a publicado un nuevo formato para estos archivos, llamado '`xib`'. Y se planea que *Gorm* pueda tener soporte para este en un futuro cercano. Algunos desarrolladores incluso utilizan las herramientas propias de *Cocoa*, para recrear sus archivos de interfaz en formato '`nib`' o '`xib`'. De esta forma, si la app se ejecuta utilizando *GNUstep*, los archivos utilizados serán los de extensión '`gorm`'. Mientras que si se utiliza *Cocoa*, se utilizaran los de extensión '`nib`' o '`xib`'.

La otra alternativa es utilizar el framework *Renaissance*, con el cual es posible almacenar la interfaz gráfica en un archivo *XML*. De esta forma, no habrá necesidad de duplicar los archivos de interfaz gráfica (excepto el del menú principal). No cubriremos en este libro el uso de este framework, pero es bueno saber de su existencia. Para más información sobre este, puede consultarse su página oficial:

`http://www.gnustep.it/Renaissance/`

También se debe tomar en cuenta que *GNUstep* ofrece algunos métodos y clases que no existen en *Cocoa*. Por ejemplo, el método `-stringByTrimmingSpaces` de la clase `NSString`, no esta disponible en *Cocoa*. Por lo que si se desea que nuestras apps se ejecuten con *Cocoa*, debe evitarse el uso de dichos métodos y clases. Las funciones `_()` y `ASSIGN()` que hemos visto anteriormente, así como muchas otras, también son propias de *GNUstep*. Sin embargo, estas funciones están disponibles cuando se utiliza el framework *Renaissance* (no así los métodos y clases adicionales). Si se desean utilizar estas funciones en una app que no utilice el framework *Renaissance*, podemos agregarlas fácilmente copiando el archivo '`Source/GNUstep.h`', del framework *Renaissance*, a nuestro proyecto, e incluyendo esta cabecera en los archivos donde se utilicen dichas funciones.

En el caso de que nuestra app utilice alguna otra librería externa a *GNUstep*, deberemos asegurarnos de que esta funcione en las otras plataformas a las cuales queramos portar nuestra app. De nada servirá que *GNUstep* funcione en otras plataformas, si la librería externa que utilizamos no lo hace.

Por último, debe considerarse lo referente a la compilación y empaquetado de nuestra app. Obtener un binario para una plataforma en especifico, requiere compilar la app en dicha plataforma. Para *Windows* esto es trivial, ya que la app se puede compilar utilizando la herramienta *gnustep-make* y luego crear un instalador con *NSIS*. Tal y como se describe en el documento *Empaquetando para Windows*. Para el sistema *Mac OS X*, la app también puede compilarse utilizando la herramienta *gnustep-make*. Aunque primero deberemos instalar dicha herramienta y el compilador para Objective-C, el

cual se instala al instalar la herramienta de desarrollo *XCode*. *XCode* es el *IDE* de *Cocoa*, el equivalente a *ProjectCenter* de *GNUstep*.

Actualmente, como parte del proyecto *GNUstep*, esta en desarrollo un framework llamado *xcode*, el cual permite convertir un proyecto *GNUstep* a un proyecto *XCode*. De esta forma, nuestra app puede ser compilada utilizando dicha herramienta, la que además facilita la creación de un instalador. Este framework se debe obtener desde el repositorio *SVN*, ya que aun no se ha liberado ninguna versión estable. De cualquier forma, siempre es posible crear un proyecto en *XCode* agregando los archivos de nuestro proyecto *GNUstep* y estableciendo las propiedades correspondientes. Aunque esto puede resultar tedioso para proyectos con muchos archivos.

Aunque no cubrimos en detalle estas alternativas y consideraciones para portar nuestras apps, es importante conocerlas para decidir de antemano el diseño de estas. Por ejemplo, si se utilizaran archivos '`gorm`' y '`nib`'/'`xib`' o si se hará uso del framework *Renaissance*.

Apéndice A Introducción al Shell

El *Shell*, *Interprete de Comandos* o *Terminal* es un programa que nos permite interactuar con el sistema a través de *comandos*. Típicamente es una ventana como la mostrada en la imagen A-1. En *Windows* la diferencia es que no hay una barra de menú, pero con un clic derecho sobre la barra de título se pueden acceder diferentes opciones. Entre estas *Copiar* y *Pegar*.

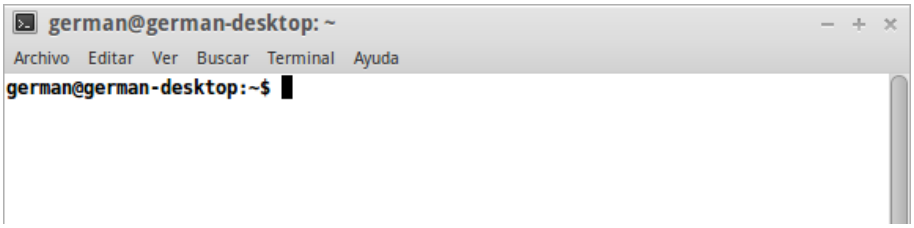

Imagen A-1. Ventana típica de una *Terminal*.

La linea de entrada de comandos se llama *prompt* y termina con el carácter *$* o **#**, dependiendo de si la terminal la esta utilizando un usuario normal o un *superusuario*. Los comandos se escriben inmediatamente después del *prompt* y son *interpretados* por la terminal al presionar la tecla ENTER. Generalmente, antes del carácter *$/#*, la terminal muestra la *ruta* en la que estamos ubicados. El carácter `~`, indica la ruta de nuestra carpeta personal. Algunos comandos básicos, junto a su descripción, se muestran a continuación:

- `pwd`
  Nos muestra la ruta en la que estamos ubicados.
- `ls`
  Nos muestra los archivos en la ubicación actual.
- `cd <nombre>`
  Nos permite ingresar a la carpeta cuyo nombre/ruta se pase como parámetro.
- `cd ..`
  Sale de la carpeta actual y nos ubica un nivel arriba en el árbol de carpetas.
- `cd`
  Nos ubica en nuestra carpeta de usuario.
- `mkdir <nombre>`
  Crea una carpeta, en la ubicación actual, con el nombre que se pase como parámetro.
- `rm <nombre>`
  Borra el archivo, en la ubicación actual, cuyo nombre se pase como parámetro.

- `rm -rf <nombre>`
  Borra la carpeta, en la ubicación actual, cuyo nombre se pase como parámetro, incluyendo todo su contenido.

La mayoría de los comandos aceptan diferentes opciones, como el caso de la opción `-rf` que se muestra en el último ejemplo. Para obtener mas información sobre un comando, puede utilizarse el comando `man`, pasándole como parámetro el nombre del comando deseado. Por ejemplo:

```
man ls
```

Nos desplegara la página de información del comando `ls`. La tecla `q` nos permite cerrar dicha página de información y regresar a la terminal.

Apéndice B La herramienta gnustep-make

La herramienta de construcción *gnustep-make* es la encargada de compilar un proyecto a partir de la información contenida en un archivo '`GNUmakefile`'. La razón de utilizar esta herramienta y no la *GNU make*, es evitar la necesidad de escribir complicados archivos '`makefile`', que son los utilizados por esta última herramienta. En su lugar, solamente debemos escribir archivos '`GNUmakefile`', cuya estructura es relativamente sencilla.

Esta herramienta se ejecuta con el comando `make` y soporta diferentes opciones. Las mas comunes se listan a continuación:

- `make`
  Compila el proyecto en la carpeta actual. Si se encuentra un error, se mostrará el nombre del archivo y el número de linea donde se encontró el error. Dicho error debe corregirse antes de intentar compilar nuevamente el proyecto. A veces se muestran también *advertencias* (*warnings*) sobre posibles errores. En estos casos, queda a criterio del programador determinar si se trata de errores o no.
- `make clean`
  Elimina lo creado por la compilación (lo inverso a compilar).
- `make distclean`
  Similar al anterior, pero también elimina lo creado por el `script` de configuración.
- `make install`
  Instala la App/Tool en el sistema. Generalmente se requieren permisos de *superusuario* para poder copiar los archivos a las carpetas del sistema. La obtención de dichos permisos puede variar de un sistema a otro. En el caso de *Windows* se requiere que el usuario tenga permisos de *administrador* para instalar programas.
- `make uninstall`
  Desinstala la App/Tool del sistema. Generalmente se requieren permisos especiales para llevar a cabo esta acción.
- `make dist`
  Crea un paquete *tar.gz* con todos los archivos del proyecto, con el nombre y la versión del programa que estén indicados en el archivo '`GNUmakefile`'. Ver capítulo 10.
- `make nsis`
  Crea un instalador para *Windows*, con el nombre y la versión del programa que estén indicados en el archivo '`GNUmakefile`'. Únicamente para proyectos de apps. Véase el documento *Empaquetando para Windows* para mayor información.

Cuando la herramienta *gnustep-make* este instalada en una ruta no prevista por el sistema, se producirá un error al tratar de ejecutarla. En estos

casos, será necesario ejecutar primero el `script` de *GNUstep*, algo como (obsérvese el espacio después del punto):

```
. /usr/GNUstep/System/Library/Makefiles/GNUstep.sh
```

O la ruta acorde a la instalación. Este `script` deberá ejecutarse también cuando se utilice la herramienta como *superusuario*.

Apéndice C La herramienta defaults

La herramienta *defaults* es un programa que permite leer o modificar las preferencias del usuario. La preferencias se almacenan en *dominios*, siendo `NSGlobalDomain` el dominio *global*. La opción `read`, pasándole como parámetro el nombre del dominio, imprime todas las *llaves*, y sus respectivos valores, definidos en el dominio indicado. Por ejemplo:

```
german@german-desktop:~$ defaults read NSGlobalDomain
NSGlobalDomain GSSecondCommandKey Control_R
NSGlobalDomain GSFirstAlternateKey Alt_L
NSGlobalDomain GSFirstControlKey Super_L
NSGlobalDomain 'Local Time Zone' America/Guatemala
NSGlobalDomain GSSecondControlKey Super_R
NSGlobalDomain GSSecondAlternateKey ISO_Level3_Shift
NSGlobalDomain GSFirstCommandKey Control_L
```

Aquí se muestra la configuración de las teclas modificadoras (establecidas en *SystemPreferences*. Cada app tiene su propio dominio, cuyo nombre es el mismo que el de la app. En este dominio pueden establecerse opciones especificas para cada app. O incluso establecer valores diferentes para las *llaves* establecidas en el dominio global. Por ejemplo, para leer las preferencias de la app *Gemas*:

```
german@german-desktop:~$ defaults read Gemas
Gemas NSDefaultOpenDirectory /home/german
Gemas HKFontSize 16
Gemas HKFont 'Courier 10 Pitch-Bold'
```

La opción `write` permite escribir un valor en la *llave* y el dominio indicados. Un ejemplo de esto es cuando se modifica el lenguaje de *GNUstep*:

```
defaults write NSGlobalDomain NSLanguages "(Spanish)"
```

Se utilizan paréntesis porque los lenguajes se almacenan en un *array*, en orden de preferencia. Esto es, se utiliza el primer lenguaje, de los listados, que este disponible en la app o servicio en cuestión.

Para borrar una *llave*, en el dominio indicado, o para borrar un dominio completo, se utiliza la opción `delete`. Por ejemplo, para borrar el tamaño de letra en *Gemas*:

```
defaults delete Gemas HKFontSize
```

La opción `help` brinda más información sobre esta herramienta.

En general, no es necesario utilizar esta herramienta (aparte de para establecer el idioma). Sin embargo, cuando se trabaja con el panel de preferencias de una app, esta nos permite comprobar si el panel esta guardando las preferencias correctamente. Asimismo, nos permite borrar el dominio de la app en cuestión, lo que permite probar nuevamente el panel de preferencias.

Apéndice D GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or

to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not

add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such

new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year* *your name*.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ‘‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with. . . Texts.” line with this:

> with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Índice

N

O

P

Q

R

S

T

V

W