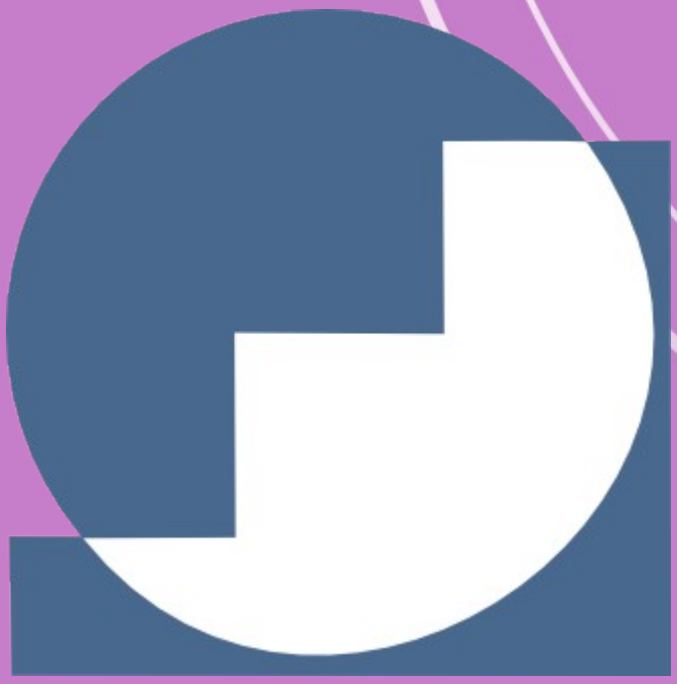
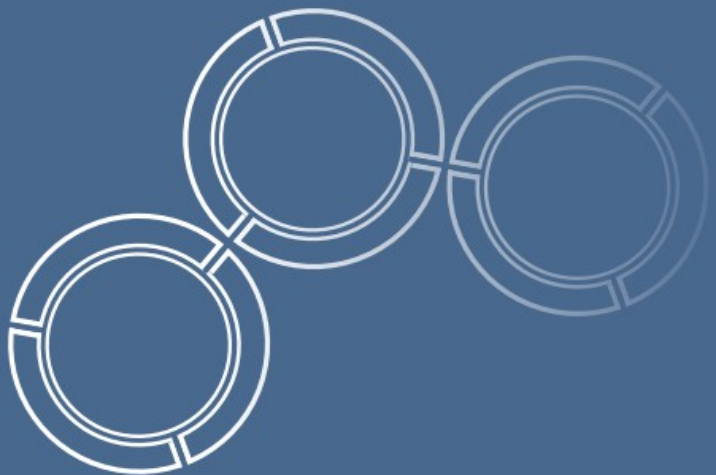


Introducción al entorno de desarrollo GNUstep

Germán A. Arias

GNUstep



Licencia de este documento

Tabla de Contenidos

INTRODUCCIÓN

El objetivo de este manual es dar a conocer al lector el entorno de desarrollo **GNUstep**, así como el lenguaje de programación **Objective-C** del cual hace uso. Este manual no solo esta dirigido a aquellos lectores que ya tienen conocimientos de programación, sino también a aquellos que por primera vez tienen contacto con ésta.

El entorno de desarrollo **GNUstep** es uno de los tantos proyectos de la **Free Software Foundation**, y su pagina oficial es www.gnustep.org, página que se encuentra en ingles, pero puedes consultar la página en español wiki.gnustep.org/index.php/User:Espectador. Como podrá comprobar el lector a lo largo de este manual, las aplicaciones desarrolladas con GNUstep tienen una apariencia y comportamiento diferentes a las que probablemente este acostumbrado a usar. Esto se debe a que GNUstep implementa las especificaciones OpenStep, las cuales fueron desarrolladas por **NeXT Software** (ahora **Apple Computer**) y **Sun Microsystem**.

GNUstep esta principalmente desarrollado para ser usado en sistemas libres basados en UNIX, como las distribuciones GNU/Linux y FreeBSD. Pero esto no significa que no pueda ser utilizado en otros sistemas operativos (Windows, MacOS X, Solaris, etc.). Asimismo, GNUstep 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 manual comienza con la presentación del lenguaje **Objective-C**, para posteriormente pasar al entorno de desarrollo **GNUstep**. El manual no es solamente teórico, sino que contiene varios ejemplos para que el lector vaya practicando a la par que avanza por el. Como es de esperarse, este manual no pretende abarcar todos los aspectos de GNUstep. Sin embargo, al terminar de leerlo, el lector estará capacitado para desarrollar aplicaciones en GNUstep, y adaptar su funcionamiento a diferentes sistemas operativos.

Capítulo 0

Instalación de GNUstep y las especificaciones OpenStep

Este pequeño capítulo presenta una forma sencilla de instalar GNUstep en sistemas GNU/Linux. Así como un breve vistazo a las especificaciones OpenStep, las cuales son implementadas por GNUstep.

0.1 Instalando GNUstep

Antes de instalar GNUstep en un sistema GNU/Linux, se requiere instalar un conjunto de paquetes los cuales dependen de la distribución que se este utilizando. En la sección **Donwload** de la página oficial de GNUstep, se encuentra información sobre los paquetes requeridos en distintas distribuciones. Una vez instalados los paquetes requeridos, se procede a la instalación de los paquetes que conforman GNUstep, y de las herramientas de desarrollo. Para poder realizar los ejercicios presentados en este manual, se necesita instalar los siguientes paquetes: **gnustep-startup-x.x.x.tar.gz**, **gworkspace-x.x.x.tar.gz**, **ProjectCenter-x.x.x.tar.tar**, **gorm.x.x.x.tar.tar**, **SystemPreferences-x.x.x.tar.gz** (donde x.x.x es la versión del paquete). Estos paquetes instalan todo lo necesario para comenzar a trabajar con GNUstep. El primer paquete que se debe instalar es el **gnustep-startup**. Para ello, primero se descomprime el archivo y luego, en una terminal, nos ubicamos en la carpeta creada y escribimos

./InstallGNUstep

Y elegimos la opción para instalar GNUstep en nuestra carpeta personal. Esto se encargara de realizar la instalación en nuestra carpeta personal, creando para ello una carpeta llamada GNUstep. Hecho esto, procedemos a descomprimir el resto de los paquetes y a instalarlos. El orden en que se instalen estos paquetes no es importante. Sin embargo, antes de proceder con la instalación de estos paquetes, debemos configurar el entorno de GNUstep. Para ello escribimos lo siguiente en la terminal

```
. /home/nombre_usuario/GNUstep/System/Library/Makefiles/GNUstep.sh
```

O la ruta donde se encuentre el archivo GNUstep.sh. Observese el punto al inicio y el espacio que le sigue. Y que, **nombre_usuario** es el nombre de la carpeta personal.

Procedamos, ahora, con la instalación del paquete **gorm**. Para ello, en una terminal, nos ubicamos en la carpeta **gorm** creada y escribimos

```
make
make install
```

El paquete **ProjectCenter** se instala de igual forma que el paquete **gorm**. Para la instalación del paquete **gworkspace**, nos ubicamos en la carpeta creada y escribimos

```
./configure
make install
```

Y, por último, para la instalación del paquete **SystemPreferences**, nos ubicamos en su carpeta y escribimos

```
make install
```

Instalados los paquetes. Necesitamos copiar el archivo **openapp**, que se encuentra en la ruta /home/nombre_usuario/GNUstep/System/Tools (o la ruta donde se haya instalado GNUStep), a la ruta /usr/bin (o la ruta donde se encuentren los comando utilizados por el shell). Para hacer esto, se necesitan permisos de root. Hecho esto, podemos ya crear una entrada para GNUstep en el menú de nuestro sistema. Utilizando algún editor de menús, como Alacarte, creamos una entrada para la aplicación GWorkspace utilizando el comando

```
openapp /home/nombre_usuario/GNUstep/System/Applications/GWorkspace.app
```

Y con esto podemos ya iniciar la aplicación GWorkspace, la cual sera nuestra área de trabajo en GNUstep.

La imagen A presenta el aspecto de la aplicación GWorkspace. En la esquina superior izquierda se encuentra el menú de la aplicación, en la esquina inferior se encuentra el icono de la aplicación, en la parte derecha se encuentra el **Dock** y en el centro aparece el **File viewer** de GWorkspace. El **Dock** es un lugar para ubicar aplicaciones y tener fácil acceso a ellas, y es aquí donde ubicaremos las aplicaciones System Preferences, Project Center y GORM. Para hacer esto, primero ubicamos en el File viewer las aplicaciones GORM y System Preferences (los archivos Gorm.app y SystemPreferences.app) en la ruta /home/nombre_usuario/GNUstep/System/Applications. Y, seleccionando el archivo Gorm.app, damos un clic izquierdo sobre el icono de la aplicación (ver imagen B) y, manteniendolo presionado, lo arrastramos al **Dock** y lo colocamos entre los dos iconos que se encuentran ya en el **Dock**, hasta que se habrá un lugar para el nuevo icono donde lo colocaremos soltando el botón izquierdo del mouse. Se procede de igual forma para ubicar las aplicaciones System Preferences y Project Center (el archivo ProjectCenter.app se encuentra en la ruta /home/nombre_usuario/GNUstep/Local/Applications). Por último, en **Preferences...** en la opción **Info** del menú de GWorkspace, podemos establecer alguna imagen para usar como fondo de la aplicación (En realidad, GWorkspace no es indispensable, además presenta dificultades con **compiz**. Así que en caso de que no se desee instalarlo, deberemos crear en nuestro menú, entradas para acceder directamente a Gorm, Project Center y System Preferences).

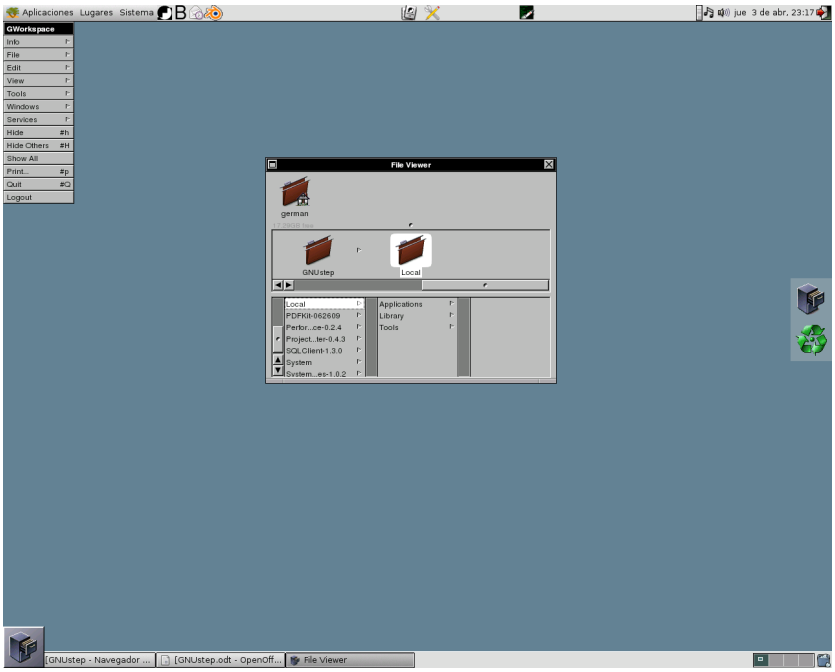


Imagen A. El escritorio GWorkspace.

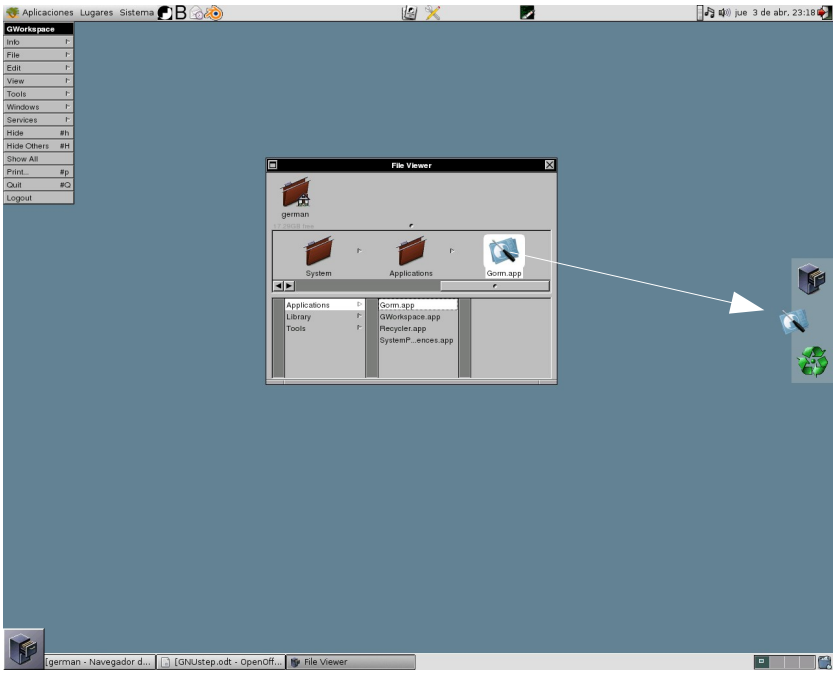


Imagen B. Arrastrando el icono de GORM.

Y con esto tenemos todo listo para comenzar a programar en GNUstep. Este entorno de desarrollo lo comenzaremos a utilizar en el capítulo 4, ya que antes debemos familiarizarnos con el lenguaje Objective-C.

0.2 Especificaciones OpenStep

Las especificaciones OpenStep engloban una gran cantidad de características de un programa, y no es objetivo de este manual el tratar estas especificaciones. Pero el lector podrá encontrar en Internet bastante información acerca de estas. Entre estas especificaciones, aquellas que se notan inmediatamente se refieren al comportamiento de un programa, así como a la apariencia de este. GWorkspace es un ejemplo de una aplicación que implementa estas especificaciones. En la imagen A, podemos notar algunas de estas especificaciones. La primera, y más obvia, es que el menú no se encuentra integrado a ninguna ventana, sino que es independiente, pudiendo localizarse en cualquier parte de la pantalla. La segunda, es que la aplicación tiene un icono localizado en la esquina inferior izquierda (llamado AppIcon), y que sirve para indicar que la aplicación esta ejecutándose. Si se da un clic en la opción **Hide** del menú de GWorkspace, la aplicación se oculta, y podemos hacerla nuevamente visible dando doble clic sobre el AppIcon. Otras diferencias las podemos ver en la ventana del **File viewer**, cuya barra de titulo tiene un control con un cuadro a la izquierda, que 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. En el lado derecho de la barra de titulo se encuentra el control para cerrar la ventana, similar a los que el lector esta acostumbrado a usar. No hay un control para maximizar/restaurar la ventana, ya que esto es algo que no existe en las especificaciones OpenStep. Además, para redimensionar la ventana, se utiliza la barra para dicho fin que se encuentra al pie de la ventana y que esta dividida en tres partes, como lo muestra la imagen C. Dando un clic izquierdo y sostenido sobre la parte central y arrastrando el mouse, podemos modificar la altura de la ventana, mientras que las partes izquierda y derecha nos permiten modificar la altura y el ancho simultáneamente.

Imagen C. Barra para redimensionar la ventana.

Otra diferencia, la encontrara el lector en los menús emergentes. Estos menús aparecen al dar un clic derecho y, manteniendo presionado el mismo, se ubica el cursor sobre la opción deseada para entonces soltar el mouse. Si no se desea hacer nada, simplemente se coloca el cursor fuera del menú emergente y se suelta el botón derecho. Esto es algo completamente diferente a los menús emergentes de uso común, donde se requiere un clic derecho para hacer visible el menú y luego un clic izquierdo para seleccionar la opción o abrir un submenú y así hasta seleccionar la opción deseada. En el capítulo 8 se vera como adaptar algunas de estas características a distintos sistemas operativos, así como modificar la apariencia de las aplicaciones. Por ahora, comencemos nuestro estudio del lenguaje Objective-C en el siguiente capítulo.

Nota: La apariencia de las aplicaciones de este manual no se corresponde con la apariencia de las aplicaciones de GNUstep cuando se usa **gnustep-startup-0.19.3** (al menos en los sistemas GNU/Linux). Esto debido a que a partir de esta versión la variable *GSX11HandlesWindowDecoration*, que controla la apariencia de las aplicaciones, viene por defecto establecida a YES, cuando anteriormente venia establecida a NO (ver capítulo 8).

Capítulo 1

El lenguaje de programación Objective-C

Antes que nada, ¿Que es un programa? Podríamos decir que un programa no es más que una serie de instrucciones que le indica 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**).

1.1 Palabras reservadas de Objective-C

En la tabla 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 más 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. 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 ingles. Sin embargo, no podemos esperar que la computadora entienda inglés, o cualquier otro idioma humano. Los lenguajes de programación, como el Objective-C, se llaman **lenguajes de alto nivel**, debido a que utilizan palabras reservadas que sean fáciles de entender para nosotros los humanos. Sin embargo, las computadoras solo entienden lenguajes de **bajo nivel** o **lenguajes maquina**, que consisten en series de pulsos eléctricos que, por comodidad, representamos con ceros y unos. Así, para que la computadora pueda entender nuestro programa, primero debemos traducirlo a un lenguaje que entienda la computadora. A este proceso de traducir un programa le llamaremos **compilación**, el cual es llevado a cabo por una herramienta llamada **compilador**. En realidad, un compilador hace mucho más que traducir un lenguaje de programación, pero por el momento nos bastara con esta definición. **GNUstep** hace uso del compilador **GCC**, del cual aprenderemos su funcionamiento básico en las siguientes secciones.

1.2 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 librería. Una librería no es más que un conjunto de funciones que podemos utilizar a la hora de programar. Para entender esto, utilicemos una analogía. Una calculadora científica 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 funciones 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

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

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

```
#include <math.h>
```

1.3 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 va el código de nuestro programa
}
```

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

1.4 Nuestro primer programa

Los programas escritos en Objective-C se crean en archivos con extension **m**. Por ejemplo, nuestro primer programa podría estar contenido en un archivo llamado **practica1.m**, estos archivos se crean con editores de texto (más adelante utilizaremos el entorno de GNUstep). La extension **m** es importante para que el compilador GCC sepa que el programa esta escrito en Objective-C. Esto porque el compilador GCC maneja otros lenguajes además del Objective-C. Bien, es momento de crear nuestro primer programa en Objective-C.

Ante que nada, crea una carpeta para guardar tus practicas. Para este manual yo he creado la carpeta **practicas** en mi directorio personal. Una vez hecho esto, abrimos algún editor de texto de nuestra preferencia, en mi caso usare **gedit**. Y copia el siguiente código en un archivo nuevo.

```
#include <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**, y dentro de ella el código de nuestro programa, el cual consiste únicamente de una línea, el código

```
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");
```

Observese el punto y coma (;) al final de la línea. En nuestro ejemplo, los caracteres **\n**, hacen que el cursor, después de escribir "Hola mundo", baje a la siguiente línea. Si no incluimos estos caracteres el cursor se quedara al final de la línea (haz la prueba). Guarda ahora el archivo como **hola.m**, o con el nombre que desees pero siempre con la extension **m**.

Abre ahora una terminal y colocate en la carpeta donde guardaste el archivo, en mi caso escribo

```
cd /home/german/practicas
```

Ahora llamamos al compilador GCC con el siguiente comando

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

Primero aparece el comando para llamar al compilador[1], **gcc**, luego esta el nombre de nuestro archivo, **hola.m**, luego viene el parámetro **-o**, que nos permite asignarle un nombre a nuestra aplicación, en este caso es el nombre que aparece al final, **hola**. Si no cometiste algún error al escribir el código, veras en tu carpeta una aplicación llamada **hola**. Para ejecutar esta aplicación escribe en la terminal (suponiendo que aun estas dentro de la carpeta de tus practica)

```
./hola
```

Y veras imprimirse el texto “Hola mundo” en la terminal. Bien, con esto ya has creado tu primera aplicación, muy sencilla por cierto, en Objective-C. Es momento de conocer más a fondo este lenguaje.

1.5 Declaración de variables

La mayoría de los programas necesitan manejar datos, por lo que necesitaremos apartar espacios en la memoria de la computadora para almacenarlos, a esto le llamamos **declaración de variables**. Objective-C puede manejar tipos de datos numéricos enteros y reales. Para cada uno de estos tipos de datos existe una palabra reservada, como lo muestra la tabla 2.

Palabra reservada	*Tipo de dato que almacena*
int	Declara variables que almacenan números enteros. Su capacidad va del -32768 al 32767.
float	Declara variables que pueden almacenar números reales. Su capacidad va del 1.7014×10^{38} al 2.9385×10^{-39}.

Tabla 2. Tipos básicos de datos en Objective-C.

Los datos representados en la tabla no son los únicos que existes, 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.

Para declarar variables necesitamos darles **identificadores**, es decir nombres. Sin embargo, estos

1 Verifica en tu distribución cual es el comando para llamar al compilador GCC, en la mayoría de las distribuciones el comando es simplemente gcc.

identificadores deben cumplir con ciertas reglas, las cuales son:

1. Un identificador se forma con una secuencia de letras y dígitos.
2. El carácter subrayado _ es también aceptado.
3. Un identificador no puede contener espacios en blanco ni ningún otro carácter a parte de los mencionados anteriormente.
4. El primer carácter de un identificador no puede ser un dígito.
5. 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 validos

```
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 una de tipo **float** con el identificador de **velocidad**, escribiríamos el código

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

Observese 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 ;
```

Notese nuevamente el punto y coma al final de cada linea. 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

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

también es posible asignarle datos a las variables mediante expresiones. Por ejemplo, si queremos realizar un programa que sume dos enteros y nos entregue el resultado, podemos entonces declarar tres variables de tipo **int**, de la siguiente forma

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

En este ejemplo, he llamado **sumando1** a la variable que almacenara el primer número a sumar, **sumando2** a la que almacenara al segundo número y **resultado** a la variable que almacenara la suma. Observese que los identificadores están separados por comas 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 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 creado. Lo recomendado en Objective-C, es que los nombres de las variables sean combinaciones de palabras, donde la segunda, tercera, cuarta, etc, palabras, empiezan 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.

1.6 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 función 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 todas las variables que se deseen. Sin embargo, el tipo de las variables y el formato con que se imprimirán, debe especificarse dentro del texto entre comillas. Para especificar el formato, se utilizan las combinaciones de caracteres **%d** para variables tipo **int** y **%f** para variables tipo **float**. Por ejemplo, supongamos que tenemos los siguientes variables con los datos indicados

```
edad = 35 ;
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”) ;
```

Observese 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** (números reales) se escriben con 6 decimales. Sin embargo, podemos indicar la cantidad de decimales deseados utilizando la cantidad de caracteres **%.nf**, donde **n** es la cantidad de decimales que deseamos. Por ejemplo, el siguiente código

```
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 utilizamos la combinación de caracteres **%md** para las variables de tipo **int**, y **%mf** para las variables de tipo **float**, donde **m** es la cantidad de espacios a ocupar. 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
```

Donde lo sombreado, indica la cantidad de espacios asignados. Para el caso de las variables tipo float, podemos especificar tanto la cantidad de espacios a ocupar, como la cantidad de decimales. Esto se hace con la combinación de caracteres **%m.nf**, donde **m** es la cantidad de espacios a ocupar y **n** la cantidad de decimales, !Haz 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,...) ;
```

Entre las comillas se colocan los formatos de los datos que se van a ingresar, y luego van las variables donde se almacenaran dichos datos. Los datos son almacenados en las variables, en el orden en que aparecen estas, es decir, que el primer dato se almacena en variable1, el segundo dato en variable2, etc. Observese el carácter **&** delante del identificador de cada variable. Además, los puntos suspensivos indican que se pueden leer la cantidad de datos deseados. Por ejemplo, el siguiente código, le indica al usuario que ingrese su número de carnet y su edad

```
int carnet, edad ;

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

Observese 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, note 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 además, que si le decimos al programa que espere datos de tipo **int**, como en el ejemplo anterior, y le ingresamos un dato de tipo **float**, se producirá un error que detendrá la ejecución del programa.

1.7 Nuestro segundo programa

Vamos a realizar ahora nuestro segundo programa, el cual consistirá simplemente en un programa que nos pide dos números, y luego nos presenta la suma, resta, multiplicación y division de dichos números.

En tu carpeta de practicas, crea un archivo llamado **matematicas.m** (o con el nombre que desees, pero sin olvidar la extension **m**). Y escribe el siguiente código en el

```
#include <stdio.h>

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

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

suma = numero1 + numero2 ;
resta = numero1 – numero2 ;
mult = numero1*numero2 ;
div = numero1/numero2 ;
```

```
printf("La suma de los números es: %.2f \n", suma) ;
printf("La resta de los números es: %.2f \n", resta) ;
printf("La multiplicación de los números es: %.2f \n", mult) ;
printf("La división de los números es: %.2f \n", div) ;
}
```

Observa que he dejado algunas líneas en blanco, estas son sólo para darle claridad al programa. Guarda el archivo, abre una terminal y, una vez ubicado en la carpeta del archivo, escribe el comando

```
gcc matematicas.h -o matematicas
```

Si no hay ningún error en el programa, se creara el ejecutable **matematicas** en tu carpeta de practicas. Para ejecutarlo escribe el comando **./matematicas**. A continuación se muestra la ejecución del programa para los números 5.6 y 7.8

```
Ingrese dos números:
5.6
7.8
La suma de los números es: 13.40
La resta de los números es: -2.20
La multiplicación de los números es: 43.68
La división de los números es: 0.72
```

Capitulo 2

Operadores, sentencias y funciones

En este capitulo, profundizaremos más en las características del lenguaje Objective-C, veremos los operadores que utiliza este lenguaje, y como dotar a un programa de la capacidad de decision en base a ciertas condiciones. Realizaremos, también, nuestro primer programa al estilo GNUstep, aunque no sera hasta el capítulo 4 en que haremos uso de las librerías y herramientas del entorno de desarrollo GNUstep.

2.1 Operadores aritméticos

En nuestro último programa, hicimos uso de los operadores aritméticos **+**, **-**, ***** y **/** para las operaciones suma, resta, multiplicación y division, respectivamente. Además de estos cuatro operadores, existe el operador **%** que se utiliza solamente con datos de tipo **int**, este operador es llamado **resto**. Este operador, como su nombre lo indica, calcula el resto de la division 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.

La precedencia de estas 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 esta 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

2.2 Operadores relacionales

Los operadores relacionales de Objective-C aparecen en la tabla 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 3. Operadores relacionales.

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.

2.3 Operadores lógicos

La tabla 4 presenta los operadores lógicos del lenguaje Objective-C, también aparece indica la sintaxis de cada uno.

Operador	*Significado*	*Sintaxis*
&&	Y (And)	(condicion1) **&&** (condicion2) **&&** ... **&&** (condicionN)
||	O (Or)	(condicion1) || (condicion2) || ... || (condicionN)
!	Inverso o negativo (Not)	**!**condicion

Tabla 4. Operadores lógicos de Objective-C.

En la sintaxis, las condiciones **condicion1**, **condicion2**, etc, son **condiciones simples**, y los puntos suspensivos significan que se pueden utilizar 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 en que todas las condiciones simples sean falsa. 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 las condiciones sean verdaderas y cuando sólo una de las condiciones sea verdadera. Y sera falsa sólo en el caso en que las tres condiciones 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. Por ejemplo, consideremos la siguiente condición

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

Esta condición compuesta sera verdadera en tres casos (observese que la condición (c == 5) esta invertida o negada por el operador **!**)

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

En todos los otros casos esta condición compuesta sera falsa.

Como veremos más adelante en este capítulo, estas condiciones nos permiten darle a un programa la capacidad de tomar decisiones.

2.4 Sentencias condicionales

El lenguaje Objective-C provee la sentencia condicional **if** útil para hacer que un programa pueda tomar decisiones en base a ciertas condiciones. La sintaxis de la sentencia **if** 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, en este caso 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 la sentencia **if**, es decir, que es posible colocar sentencias **if** dentro de otras sentencias **if**.

Ademas de la sentencia **if** condicional, 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 reservada **switch**, **case**, **break** y **default**. Los puntos suspensivos indican que se pueden agregar cuantos valores para **variable** se deseen. El funcionamiento de esta sentencia es fácil de entender, cuando **variable** es igual a **valor1**, se ejecutan las instrucciones correspondientes, y los mismo para cuando **variable** es igual a los otros valores. 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 para realizar cuando la variable no tenga ninguno de los valores esperados.

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 ;
```

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

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

2.5 Sentencia iterativas

Las sentencias iterativas permiten repetir la ejecución 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. Observese además que estas tres expresiones, **valor inicial**, **condición** y **expresión de control**, van separados por 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 del 3. **Valor inicial**, indica el valor inicial para la variable de control, en este caso **x**. **condicion** 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 va a ir incrementando para llegar al valor final (en algunos caso puede ser necesario hacer que la variable vaya disminuyendo su valor). Observese que si la expresión de control hubiera sido x = x + 2, solamente se hubieran impreso los resultados para los números 1, 3, 5, 7, etc.

Por otra parte, la sentencia iterativa **while** 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 condición, que imprime la anterior tabla de multiplicación del 3, es

```
int x = 1, mult ;

while (x <= 10)
{
mult = 3*x ;
printf("3 por %d = %d", x, mult) ;
x = x + 1 ;
}
```

Observese 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. Observese, 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. Por último, hay que mencionar que a la hora de utilizar estas sentencias iterativas (también llamadas **bucles**), debe estarse seguro de que se alcanza el valor final (en el caso de la sentencia **if**) o de que la condición llega 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.

2.6 Nuestro tercer programa

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

```
#include <stdio.h>
```

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

printf("Ingrese tres números 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 numero mayor es: %d. \n", mayor) ;
```

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

}
```

¿Se entiende claramente este código? Observese que las llaves pueden colocarse donde se deseen, siempre y cuando su significado este claro. Copia este código en un archivo nuevo con extension **m**, compilalo y ejecutalo.

2.7 Nuestro cuarto programa

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

```
#include <stdio.h>

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

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

        switch (eleccion)
        {
        case 1:
                printf("Ingrese los dos números: \n") ;
                scanf("%f%f", &numero1, &numero2) ;
```

```
                resultado = numero1 + numero2 ;
                printf("El resultado es %.2f. \n", resultado) ;
                break ;
        case 2:
                printf("Ingrese los dos números: \n") ;
                scanf("%f%f", &numero1, &numero2) ;
                resultado = numero1 - numero2 ;
                printf("El resultado es %.2f. \n", resultado) ;
                break ;
        case 3:
                printf("Ingrese los dos números: \n") ;
                scanf("%f%f", &numero1, &numero2) ;
                resultado = numero1 * numero2 ;
                printf("El resultado es %.2f. \n", resultado) ;
                break ;
        case 4:
                printf("Ingrese los dos números: \n") ;
                scanf("%f%f", &numero1, &numero2) ;
                resultado = numero1 / numero2 ;
                printf("El resultado es %.2f. \n", resultado) ;
                break ;
        defaul:
                eleccion = 5 ;
        }
}
}
```

Observese que la sentencia **switch** se encuentra dentro del cuerpo de la sentencia **while**. Observese, también, que para salir del bucle **while** la variable **eleccion** puede tener un valor igual o mayor a 5.

2.8 Funciones

Las funciones son un conjunto de instrucciones que realizan una tarea determinada (como las funciones de las librerías) y que podemos llamar cuando queremos que dicha tarea se lleve a cabo. Las funciones nos permiten dividir un programa en pequeños sub programas que luego llamamos para que ejecuten sus tareas especificas, esto hace más fácil el diseño de un programa. Las funciones pueden recibir datos que nosotros les enviemos y que sean necesarios para su funcionamiento, asimismo, pueden 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. La sintaxis de una función es de la siguiente forma

```
tipo_del_valor_ devuelto    nombre_de_la_función ( parámetros )
{
Instrucciones que lleva a cabo la función.
}
```

Donde **tipo_del_valor_devuelto** es el tipo del dato que la función devolverá o retornara. **nombre_de_la_funcion** 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. Si 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. Observese que dentro de los paréntesis va tanto el tipo de parámetro como el nombre del parámetro, en este caso **x**. Se pueden especificar cuantos parámetros se deseen, simplemente separándoles con comas. Otro tipo de función, es aquella que no necesita de ningún parámetro, pero 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 se hace uso de la palabra reservada **return** para retornar el valor 0. A veces se desea el retorno de algún valor para saber si la tarea se realizo exitosamente. Es decir, para esta función de ejemplo, que si la función retorna el valor 0, entonces el mensaje se imprimió exitosamente, de lo contrario algo salio mal.

El último tipo de función, es aquella que necesita parámetros y que a su vez retorna un dato. Por ejemplo, la siguiente función recibe como parámetros los lados de 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**, para almacenar el resultado del calculo, y luego el contenido de la variable **area** es retornado haciendo uso de 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 ninguna dato, para utilizar la función simplemente se escribe su nombre seguido de un punto y coma. Por ejemplo, para hacer uso de la función **imprimirSaludo**, vista anteriormente, se tiene

```
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, esto se separan con comas). Por ejemplo, la función **mayorDeEdad** puede usarse de la siguiente forma

```
mayorDeEdad (edadCliente) ;
```

donde la variable **edadCliente** contiene un dato entero (puesto que el tipo del parámetro en la función fue declarado como tipo **int**). Observese 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 de **imprimirOtroSaludo**, tendríamos

```
exito = imprimirOtroSaludo ;
```

Donde la variable **exito** almacena el valor retornado por la función **imprimirOtroSaludo** (el tipo de la variable **exito** debe ser **int**, puesto que este fue el tipo declarado en la función para el valor retornado). El valor de esta variable, puede posteriormente verificarse para saber si la función realizo su tarea con éxito. Si no deseamos realizar la verificación, simplemente utilizamos la función como una que no retorna ningún dato

```
imprimirOtroSaludo ;
```

En donde el valor retornado no es asignado a ninguna 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** se le asigna el dato retornado por la función (la variable **resultado** debe ser de tipo **float**), y donde las variables **base** y **altura** (que deben ser de tipo **float** también, puesto que así se declaro en la función) se pasan como parámetros. Nuevamente, las variables **base** y **altura** no necesitan tener el mismo nombre que las variables declaradas como parámetros en la función (**lado1** y **lado2**). Aunque el valor retornado por la función **calculoArea** no necesita ser asignado a alguna variable (como en el caso de la función **imprimirOtroSaludo**, visto anteriormente) esto no tiene sentido, puesto que si le pasamos los lados de un rectángulo a la función **calculoArea**, es por que deseamos conocer el área.

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, para hacerlo, simplemente debemos copiar la primera linea de nuestras funciones seguidas de un punto y coma. Por ejemplo, para declarar las cuatro funciones que hemos ejemplificado aquí, tendríamos que escribir

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

Ya hemos visto como crear funciones y como utilizarlas. Pero ¿donde deben crearse? Las funciones, y

esto no se debe olvidar, se crean *fuera* del cuerpo de la función **main**, es decir, fuera de las llaves de la función **main**. Y podemos ponerlas antes o después de esta (lo recomendado es ponerlas antes). Otro punto a tener en cuenta, es que las variables declaradas dentro de una función, solo existen para esa función, es decir que para las otras funciones dichas variables no existen. Esto significa que dos, o más 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. Ya que, a diferencia de las funciones que nosotros creamos, no necesita ser llamada. Cuando ejecutamos un programa la función **main** es automáticamente llamada. Por esta razón, todo programa escrito en Objective-C debe tener una función **main**. De lo contrario, al ejecutar un programa y no encontrar este a la función **main**, el programa simplemente no haría nada.

2.9 Matrices

Las matrices son un tipo de conjunto de datos que a menudo resultan útiles. Las matrices en Objective-C son similares a las matrices que vemos en matemática, y agrupan un conjunto de datos del mismo tipo bajo un mismo identificador o nombre. Y, al igual que las matrices vistan en matemática, estas pueden ser de una dimension, dos dimensiones, tres dimensiones, etc. Por ejemplo la forma de declarar una matriz de una dimension es

```
tipo identificador[tamaño de la matriz] ;
```

Y para declarar una matriz llamada **datos** que albergue 10 números enteros, tendríamos

```
int datos[10] ;
```

una matriz de más dimensiones se declara de forma similar

```
tipo identificador[dimension1] [dimensio2] [dimension3] ... [dimensionN] ;
```

Donde los puntos suspensivos indican que se pueden declarar cuantas dimensiones se deseen. Por ejemplo, una matriz llamada **vectores** que alberga las componentes reales **X** y **Y** de 5 vectores, podría ser de la siguiente forma

```
float vectores[5] [2] ;
```

Esta matriz correspondería a una matriz del tipo

a_{00} a_{10}
a_{01} a_{11}
a_{02} a_{12}
a_{03} a_{13}
a_{04} a_{14}

Observese que Objective-C, numera las filas y las columnas comenzando desde cero. En la matriz anterior, le asignamos 5 reglones o filas a la matriz, y estas se numeran como 0, 1, 2, 3 y 4 respectivamente.

La asignación de datos a una matriz, es como la asignación a las variables normales, con la adición de que se tiene que especificar la casilla donde se desea almacenar el dato. Por ejemplo, para asignar el valor de 3 a la casilla $\mathbf{a}_{13}$, tendríamos

```
vectores[1] [3] = 3 ;
```

Para poder utilizar los datos de una matriz, simplemente se especifica la casilla del dato que se desea. Los datos de una matriz pueden utilizarse en expresiones matemáticas como las variables vistas anteriormente. Por ejemplo, para utilizar en una expresión el dato almacenado en la casilla $\mathbf{a}_{13}$ de la matriz anterior, podríamos tener

```
fuerza = 2 + 3*vectores[1] [3] ;
```

Es posible asignarles valores a las casillas de una matriz al momento de declararla. Los siguientes ejemplos muestran como hacer esto

```
int datos[3] = {5, 3, 8} ;
float estaturas[4] = {1.2, 6.7, 9.4, 3} ;
int vectores[3] [2] = { {3, 5} {8, 6} {7, 1} } ;
int edades[5] = {3, 2, 5} ;
int telefonos[80] = {0} ;
float mediciones[10] [20] = {0} ;
```

En el primer ejemplo, a las casillas de la matriz se le asignan los valores 5, 3 y 8 respectivamente. De forma similar, en el segundo ejemplo, los valores 1.2, 6.7, 9.4 y 3 se asignan respectivamente a las casillas de la matriz. En el tercer ejemplo, la matriz queda de la forma

3	5
8	6
7	1

En el cuarto ejemplo, a las primeras tres casillas se les asignan los valores 3, 2 y 5 respectivamente, y a las casillas restantes se les asigna el valor de 0. Y en el quinto y sexto ejemplos, a todas las casillas se les asigna el valor de 0.

2.10 Quinto programa

El siguiente programa muestra el uso de la función **calculoArea** vista anteriormente

```
#include <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 del rectángulo: \n") ;
scanf("%f%f", &base, &altura) ;
resultado = calculoArea(base, altura) ;
printf("El área es %.2f. \n", resultado) ;
}
```

Observese la declaración de la función **calculoArea** al inicio del programa. Cuando la función **main** llama a la función **calculoArea**, la función **main** deja de ejecutarse y comienza la ejecución de la función **calculoArea**, una vez esta termina sus instrucciones, la función **main** reanuda su ejecución en la instrucción que esta a continuación de la llamada a la función **calculoArea** (en este ejemplo la instrucción: printf("El área es %.2f. \n", resultado) ;).

El lector puede pensar que el crear la función **calculoArea** no tiene mucho sentido, ya que el calculo podría haberse hecho directamente dentro de la función **main**. El objetivo de este programa ha sido

simplemente mostrar como crear y utilizar funciones. Sin embargo, cuanto mayor es un programa, se comprende mejor la utilidad de estas. Ademas, como veremos en los próximos capítulos, las funciones nos ayudan a que el diseño de un programa sea más fácil.

2.11 Sexto programa

En la sección 2.9 cuando hablamos de las matrices, supusimos que de antemano sabíamos el tamaño de la matriz que necesitaríamos. Puesto que a la hora de su declaración, forzosamente debíamos darle un tamaño. Es decir que, por ejemplo, si pretendemos hacer un programa que almacene un cierto número de datos, y queremos utilizar una matriz para almacenarlos, deberemos conocer antes de cuantos datos se trata, o no podremos declarar la matriz. ¿No sera posible hacer un programa que nos pregunte primero la cantidad de datos a almacenar, y que posteriormente nos pida ingresar los datos? La respuesta es **Si**, haciendo uso de funciones.

El siguiente programa realiza dicha tarea

```
#include <stdio.h>

crearMatriz(int longitud) ;

crearMatriz(int longitud)
{
int vector[longitud] ;
int x ;

for(x = 0; x <= (longitud - 1); x = x + 1)
{
scanf("%d.", &vector[x]) ;
}

}


main(void)
{
int cantidad ;
printf("Ingrese la cantidad de datos a almacenar: \n") ;
scanf("%d", &cantidad) ;
```

```
printf("Ingrese la lista de datos: \n") ;
crearMatriz(cantidad) ;
printf("Los datos han sido almacenados. \n") ;
}
```

Observese la declaración de la función **crearMatriz** al inicio del programa. Esta función recibe un parámetro de tipo **int**, el cual utiliza para declarar la matriz. Una vez declara la matriz, una sentencia **for** se encarga de almacenar los datos en la matriz. Observese que la condición de esta sentencia es

x <= (longitud - 1)

Donde se resta 1 a la variable **longitud**, ya que los renglones en una matriz se cuentan comenzando por 0. Una vez almacenados los datos finaliza la función **crearMatriz**, y entonces la función **main** imprime un mensaje.

Es la función **main** quien se encarga de pedir al usuario del programa la cantidad de datos a almacenar y luego pasar este dato a la función **crearMatriz**, quien es la que declara la **matriz**. Puesto que el tamaño de la matriz solo se conoce hasta que se ejecuta el programa, a la hora de compilar este archivo el compilador emitirá un aviso. Sin embargo, si copiaste el código correctamente no debe haber ningún problema al ejecutarlo.

2.12 Más operadores

Existen algunos otros operadores que a veces utilizan los programadores. Estos operadores, no son esenciales, puesto que todo lo que se puede realizar con ellos, se puede realizar con los operadores vistos anteriormente. Sin embargo, los programadores los utilizan frecuentemente, debido a que ahorran algunas lineas de código, y por lo tanto es necesario conocerlos. Todos estos operadores están formados por dos caracteres. Estos operadores son: **+=**, **-=**, ***=**, **/=**, **--** y **++**. El comportamiento de los primeros cuatro operadores, se muestra 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 operador 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) ;
```

Al final 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 ejecutarse 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, la disminuye en 1.

2.13 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 de este. En Objective-C existen dos formas de agregar comentarios en un programa, y estos se pueden agregar en cualquier parte del programa. La primera de ellas utiliza los caracteres **//** para indicar que lo sigue hasta el final de la linea es un comentario. Por ejemplo,

```
// Aquí puede ir un comentario.
```

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

```
/* Aquí puede ir un comentario
mucho más 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. Sin embargo, los

programas que veremos en próximos capítulos, no tendrán comentarios, ya que se van explicando conforme los creamos. Sin embargo, conforme los vayas realizando por tu cuenta, ve introduciendo comentarios donde consideres necesario.

Capítulo 3

Programación orientada a objetos (POO)

Los dos capítulos anteriores, fueron una introducción al lenguaje de programación Objective-C. Es en este capítulo donde comenzamos nuestro estudio del entorno de desarrollo GNUstep. Sin embargo, antes debemos aprender un poco acerca de lo que es la programación orientada a objetos (POO), y una nueva terminología. Es posible que al lector este capitulo le parezcan muy teórico, incluso puede sentirse algo confundido. Sin embargo, el lector no debe preocuparse por esto, en cuanto realice su primer programa en GNUstep en el capitulo 4, vera como todo va encajando.

3.1 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, como, por ejemplo: *Piedra*, *Carro*, *Teléfono*, etc. 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 a cualquier carro en particular le llamamos *Objeto*. Y este *Objeto* (un carro cualquiera) tiene particularidades que lo diferencian de los demás. Como, por ejemplo, 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 interactuan entre si. Entendiendo por un modulo a un conjunto de datos y patrones de

comportamiento. Es a estos modules a los que se les llaman *Objetos*. En el capitulo anterior, donde vimos el uso de las funciones, comenzamos a hacer uso de la noción de dividir un programa en módulos. Es decir, en subprogramas, 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 Objecto es un subprograma conformado por 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* 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 solo 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

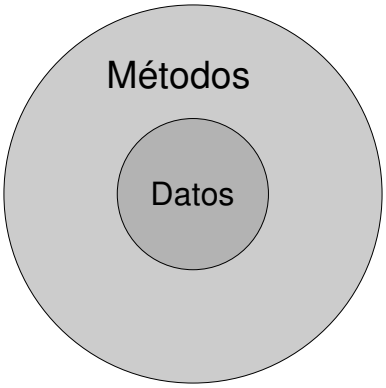


Figura 1. Visualización de un Objeto.

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 solo pueden ser accedidos 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 un Objeto, así sera su comportamiento.

El concepto de Objetos, se asemeja más 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, chasis, 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 interactuan entre si

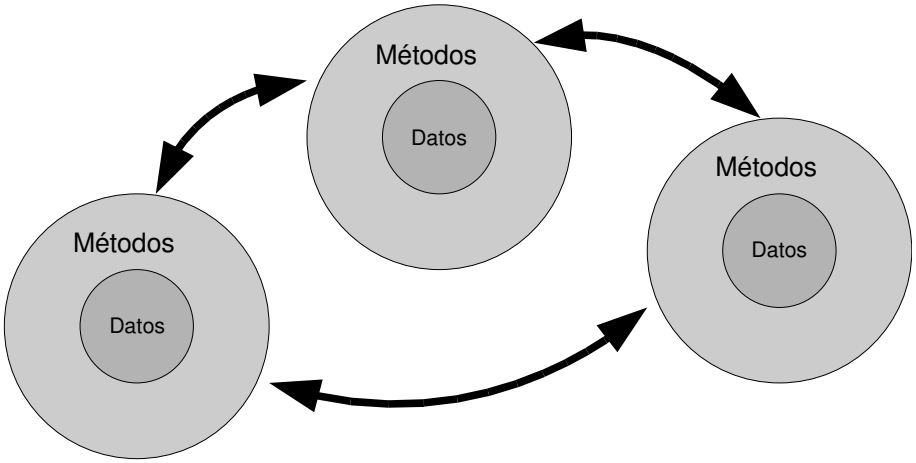


Figura 2. Objetos interactuando.

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 los programas (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 *titulo*, con los títulos “+” e “=”. Tres Objetos *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. Todos estos Objetos son visuales, puesto que el usuario del programa los puede ver e interactuar con ellos.

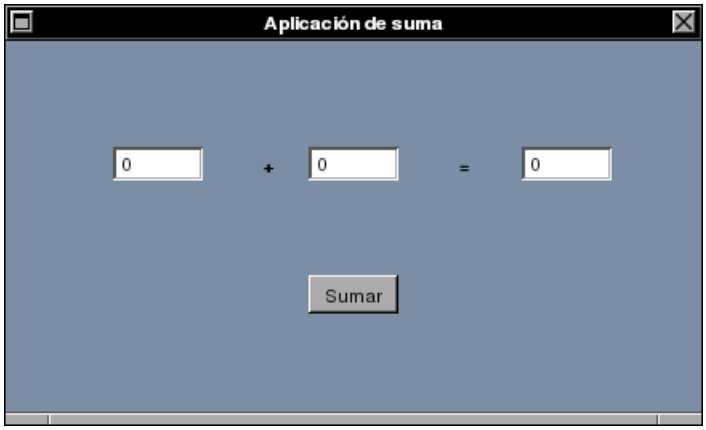


Imagen 1. Programa Suma.

Al inicio del capitulo, 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 (un *botón*, una *ventana*, un *titulo*, etc.) pertenece a una cierta 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 incluye el comportamiento de los objetos. Por ejemplo, el comportamiento de una ventana al mover sus bordes para cambiar su dimensión.

Podemos decir entonces que una *clase* es el plano para construir un Objeto con sus aspectos y comportamientos 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 nuestra ventana a partir de la clase *ventana* (es decir, a partir del plano de construcción de una ventana) y luego modificarla para adaptarla a nuestros propósitos. Evidentemente, esto es mucho más sencillo que construir nuestra ventana desde cero.

3.2 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 les 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.

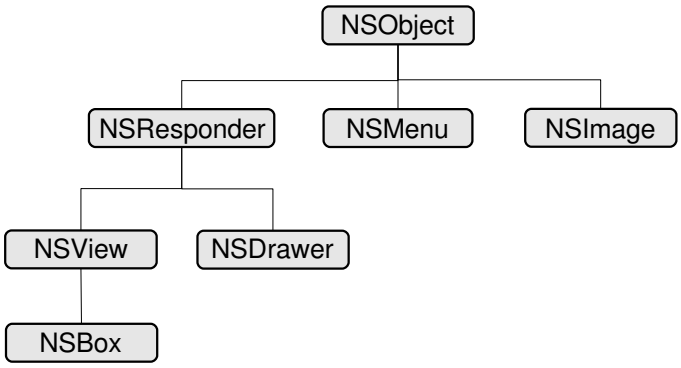


Figura 3. Jerarquía de algunas de las clases en GNUstep.

Como se observa, la clase NSObjetc 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.

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.

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 más, pero en la figura solo se muestran dos), que son NSView y NSDrawer. Y la *superclase*, de la clase NSResponder, es la clase NSObject. Observese 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.

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 porque es más fácil crear una clase a partir de otra,

que crear una clase 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 solo a los métodos de su clase, sino también a los métodos de la *superclase* de su clase. 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), responde no solo 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 (más conocidas como *punteros a Objetos* o simplemente *Objetos*). Los Objetos, o punteros a Objetos, se declaran como las variables, con la diferencia de que el identificador (el nombre) 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 aun no sabemos cual sera, en este caso el Objeto se declara como de tipo **id**

```
id nombre_objeto ;
```

Observese 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.

3.3 Clases y Objetos en GNUstep

Para desarrollar nuestros programas en GNUStep, haremos uso de las aplicaciones[2] **ProjectCenter** y **GORM**. **ProjectCenter** es el IDE[3] de GNUstep, y es una aplicación que nos permite crear y administrar proyectos. Y **GORM**[4] es una aplicación que nos permite crear fácilmente la interfaz gráfica de nuestros proyectos. En otras palabras, GORM nos permite manejar fácilmente los Objetos visuales que formaran parte de nuestra aplicación. De esta forma, nosotros, como programadores, nos ocuparemos principalmente de los Objetos no visuales. Los cuales, a su vez, pueden hacer uso de otros

2 En general, llamaremos **aplicación** a un programa que posee una interfaz gráfica. Asimismo, llamaremos **proyecto** al conjunto de archivos necesarios para crear un programa o aplicación.
3 IDE proviene de Integrated Development Environment (Entorno de Desarrollo Integrado).
4 GORM proviene de Graphical Object Relationship Modeller (Modelador Relacional de Objetos Gráficos).

Objetos.

En GNUstep los Objetos están conformadas por dos archivos. Uno de ellos con extensión **h**, que contiene la interfaz de la clase del objeto, y el otro con extensión **m**, que contiene la implementación de dicha clase. Cuando trabajemos con GORM, este se encargara de crear ambos archivos. Sin embargo, es importante conocer la estructura que tienen estos archivos, para comprender y modelar el funcionamiento del programa. Veamos, entonces, como es la estructura básica de una interfaz

```
#include <librería necesaria>

@interface nombre_del_objeto : clase_del_objeto
{
Aquí se declaran las variables de instancia.
}

Aquí se declaran los métodos

@end
```

Al inicio va la inclusión de la librería (o librerías) que necesite nuestro Objeto. Seguidamente va la declaración de la interfaz, que comienza con **@interface** y termina con **@end**. Después de **@interface**, va el nombre de nuestro Objeto seguido de dos puntos y luego la clase a la que pertenece nuestro Objeto. Luego viene, entre llaves, la declaración de las variables de instancia que estarán a disposición de todos los métodos del Objeto (cuando no es necesario que una variable este disponible para todos los métodos, se declara dentro del método que la usa). Las variables declaradas en esta parte se conocen como *Atributos del objeto* o, simplemente, *Atributos*. Después de las llaves, se declaran los métodos que estarán disponibles para el resto del programa. Es decir, los métodos a través de los cuales podrá interactuar el Objeto con el resto del programa (como veremos más adelante, algunos métodos son únicamente para uso interno del Objeto).

Los métodos son similares a las funciones, con algunas diferencias. 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 entre paréntesis y, en caso de que no 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. El primer *tipo*, es el tipo del dato retornado, los siguientes son los tipos del primer, segundo, etc, parámetros

respectivamente, cuyos nombres son *parametro0*, *parametro1*, etc. Observese que cada parámetro esta precedido de una etiqueta y de dos puntos (la etiqueta del primer parámetro es el nombre del método). El **nombre completo** de un método es la unión del nombre del método y de los nombres de las etiquetas. Aunque las etiquetas son opcionales, es buena idea usarlas para indicar la finalidad del parámetro que le sigue. El siguiente, es un ejemplo de interfaz

```
/* All Rights reserved */

#include <AppKit/AppKit.h>

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

Aquí, la primera linea es un comentario incluido automáticamente por GORM. La siguiente línea 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 del Objeto, cuyo nombre es **Control** y cuya clase es NSObject. Luego vemos la declaración de una variable de instancia llamada **titulo** de tipo **id**. Por último, se declara el método de instancia **mostrartiempo**, que no retorna ningún dato y que recibe un parámetro de tipo **id**. Observese el punto y coma al final de la declaración.

Veamos ahora, como es la estructura básica de un archivo de implementación.

```
#include <librería necesaria>
#include “archivo de la interfaz”

@implementation nombre_del_objeto

Aquí se implementan los métodos

@end
```

Primero se incluye la librería que necesite nuestro programa, luego el archivo de la interfaz del Objeto entre comillas (se utilizan comillas puesto que el archivo de la interfaz se encuentra en la misma carpeta que el archivo de la implementación). Seguidamente va la implementación del Objeto entre **@implementation** y **@end**. Donde el nombre del Objeto va después de **@implementation**. El siguiente, es el archivo de implementación de la interfaz del ejemplo anterior.

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "Control.h"

@implementation Control

- (void) mostrartiempo: (id)sender
{
  /* insert your code here */
NSCalendarDate *date = [NSCalendarDate date];
[date setCalendarFormat: @"%H : %M : %S"];
[titulo setStringValue: [date description]];
}

@end
```

Este Objeto solamente tiene un método, el método **mostrartiempo**. Y contiene dos comentarios, incluidos automáticamente por GORM (no discutiremos aquí el funcionamiento del método de esta clase, ya que sólo se muestra como ejemplo).

3.4 Mensajes

La forma en que los Objetos pueden comunicarse entre si es mediante **mensajes**. La sintaxis de un mensaje es

```
[nombre_objeto  nombre_método] ;
```

Donde *nombre_objeto* es el nombre del Objeto que recibe el mensaje, y *nombre_método* es el nombre del método que queremos ejecutar. Si el método necesita parámetros, estos deben ir separados por dos puntos y por sus respectivas etiquetas (si las tienen). Su forma 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 por resto del programa.

3.5 Outlets y Actions

Los *Outlets* y los *Actions* son dos conceptos importantes para comprender la forma en que GNUStep maneja la interacción de los Objetos. Para un Objeto en particular, los Outlets son rutas que le permiten enviar mensajes a otros Objetos. En otras palabras, son conexiones que permiten al Objeto hacer uso de los métodos de otros Objetos. Por el contrario, los Actions, son rutas que permiten a otros Objetos mandar mensajes a nuestro Objeto en particular. Es decir, son conexiones que permiten a otros Objetos hacer uso de los métodos de nuestro Objeto en particular.

Cuando hagamos uso de la aplicación GORM, veremos que a un Objeto en particular debemos asignarle el número de Outlets y Actions que necesite, antes de realizar sus conexiones. Para ello debemos darle un nombre a cada Outlet y a cada Action. Por ejemplo, un Objeto A, que tenga dos Outlets y un Action, podría representarse de la siguiente forma.

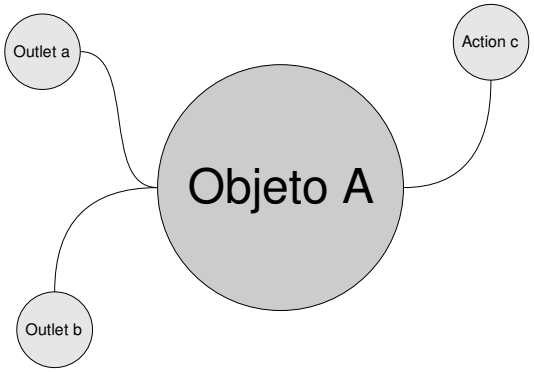


Figura 4. Objeto A con dos Outlets (a y b) y un Action (c).

Y la conexión con otros Objetos podría representarse como

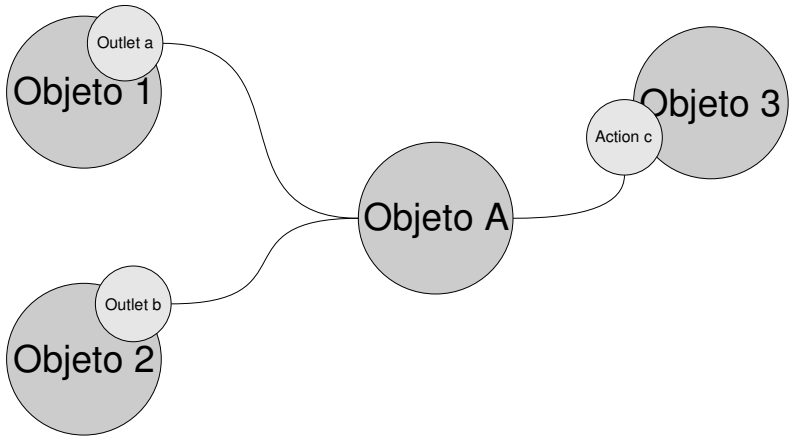


Figura 4. Conexión de Outlets y Actions.

Observe que los identificadores, o nombres, de los Outlets y Actions, no necesitan tener el mismo nombre que los Objetos con los cuales conectan. Pueden tener cualquier nombre, aunque es recomendable que el nombre diga algo sobre el objetivo de la conexión. En la figura 4, los Outlets *a* y *b* le permiten al Objeto A, enviar mensajes a los Objetos 1 y 2. Y el Action *c*, le permite recibir mensajes del Objeto 3. Observese, además, que lo que para el Objeto A es una conexión Outlet, para los Objetos 1 y 2 es una conexión Action, y también, que lo que para el Objeto A es una conexión Action, para el Objeto 3 es una conexión Outlet. Por lo que los Outlets y Actions son relativos.

En el siguiente capitulo, crearemos nuestro primer programa en GNUstep. Este sera similar al mostrado en la imagen 1. Para que este programa funcione, crearemos un Objeto no visual que tendrá tres Outlets y un Action conectados como muestra la siguiente figura

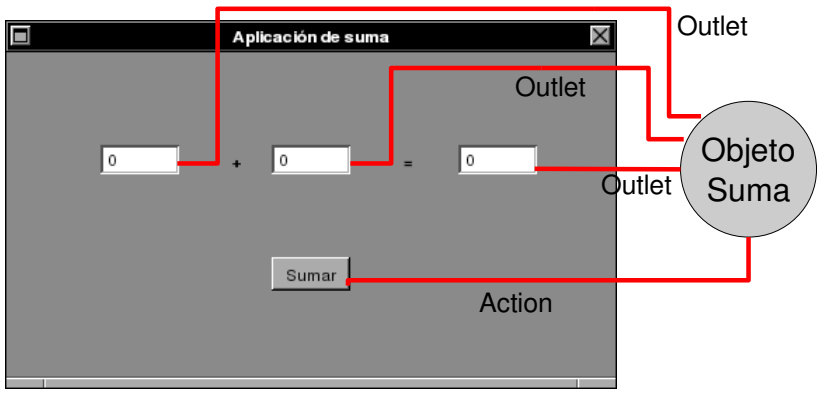


Figura 5. Conexión del Objeto Suma y la interfaz del programa.

Dos de los Outlets son para obtener los sumandos ingresados por el usuario[5], y el tercero para enviar el resultado. El Action esta conectado al botón **Sumar** que, al recibir un clic, manda un mensaje al Objeto *Suma* para que este lleve a cabo la operación.

Para finalizar, recordemos que en la sección anterior, dijimos que la forma de un mensaje era, para un método que no requiere parámetros,

[nombre_objeto nombre_método] ;

y par uno que requiere parámetros

[nombre_objeto *nombre_método*: parametro0 *etiqueta1*: parametro1 *etiqueta2*: parametro2 .....]

Podemos decir ahora que en lugar de *nombre_objeto* se puede colocar el nombre del Outlet que conecta con el Objeto al cual queremos enviar el mensaje. Es decir, el nombre de la conexión con el Objeto. De esta forma, no necesitamos saber el nombre del Objeto para enviarle un mensaje.

5 Aunque una conexión Outlet solo puede enviar mensajes, se pueden recibir datos como resultado de enviar un mensaje. Esto se da si el mensaje consiste en ejecutar un método que retorna un valor.

Capítulo 4

Nuestro primer programa con GNUstep

Comenzamos aquí el uso de las aplicaciones Project Center y GORM. Sin embargo, no describiremos detalladamente el uso de estas aplicaciones. El lector con un poco de curiosidad, puede descubrir por si mismo la función de cada una de las opciones del menú. En este capitulo, el procedimiento de crear nuestra primera aplicación se describe paso a paso, para evitar que el lector que nunca antes haya utilizado estas aplicaciones se pierda.

4.1 Creando la interfaz gráfica

Nuestra primera aplicación sera muy sencilla, ya que se tratara de una aplicación que simplemente suma dos números (similar a la de la imagen 1). Primero que nada abramos GWorkspace, desde donde podemos llamar fácilmente las aplicaciones de GNUstep.

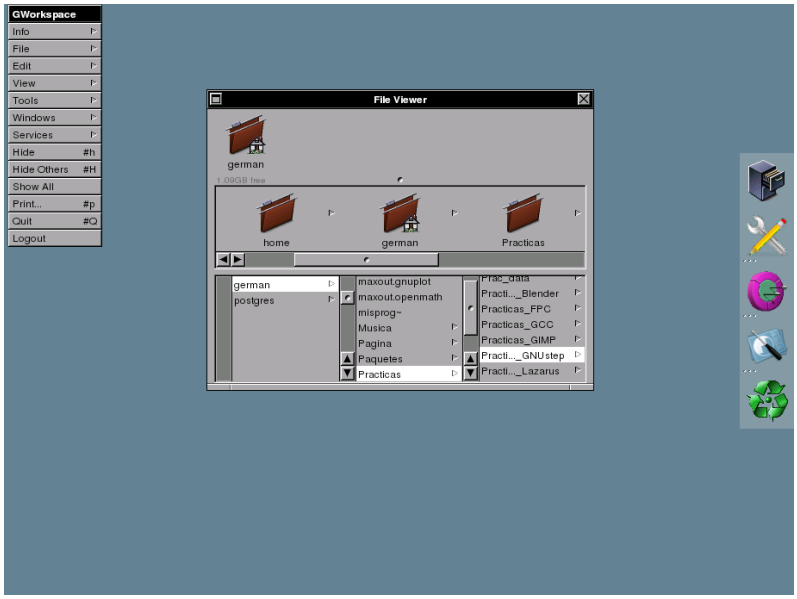


Imagen 2. GWorkspace.

Una vez iniciado GWorkspace, iniciamos Project Center. Y en la opción **Project**, del menú de Project Center, seleccionamos la opción **New...**

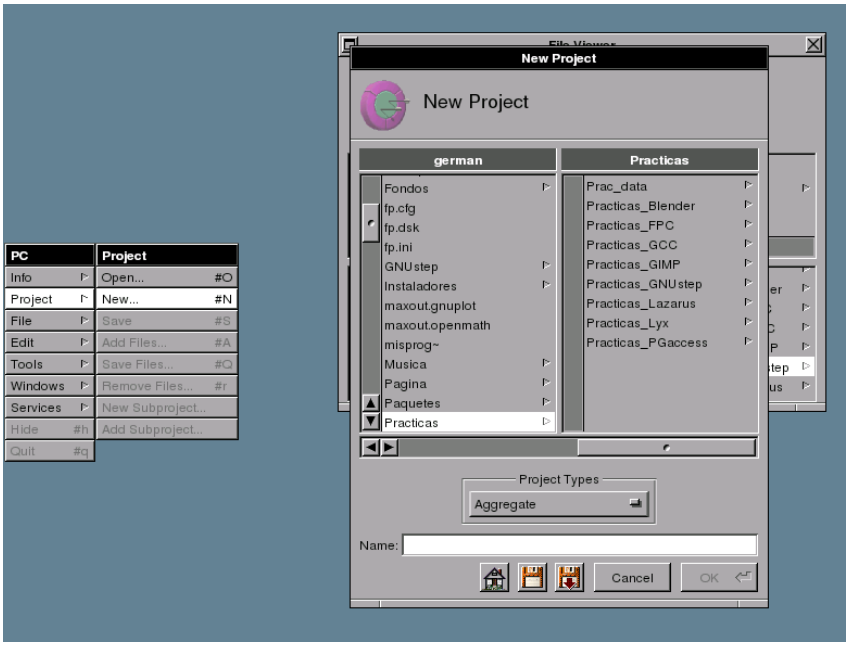


Imagen 3. Project Center.

Esto abre la ventana **New Project**, donde debemos elegir un proyecto de tipo **Application** (imagen 4), y seleccionar la carpeta donde deseemos guardar el proyecto. No es necesario crear una carpeta

especial para el proyecto ya que Project Center la crea por nosotros. Escribamos un nombre para dicha carpeta y luego demos un clic en **OK**. Esto creara, además de la carpeta para nuestro proyecto y otros archivos, el archivo **PC.project** el cual es el archivo de nuestro proyecto.

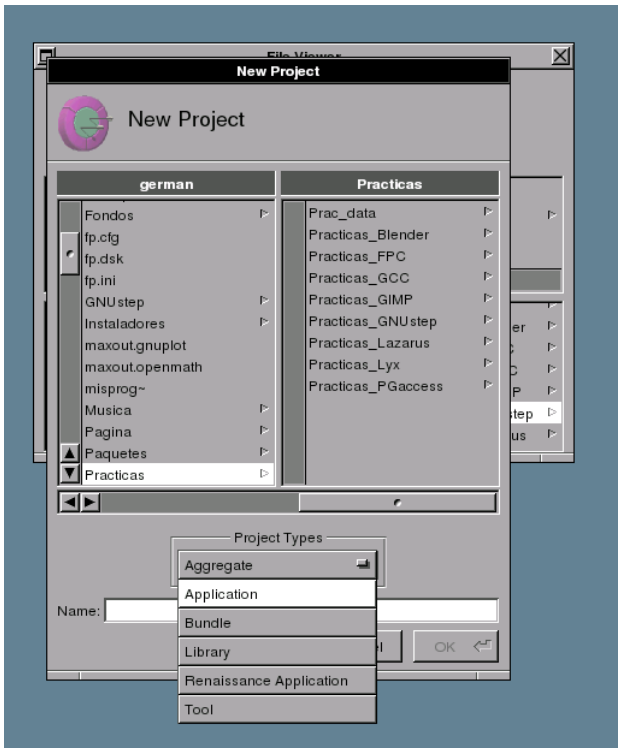


Imagen 4. Seleccionando el tipo **Application** para el proyecto.

Una ventana con el titulo **suma** y la ruta del proyecto, nos presentara la organización **virtual** de los archivos de nuestro proyecto. Es una organización virtual debido a que las carpetas Classes, Headers, etc, realmente no existen. Project Center organiza los archivos de nuestro proyecto en estas carpetas virtuales, para facilitarnos el movimiento a través de nuestro proyecto (el archivo PC.project no aparece en esta organización virtual).

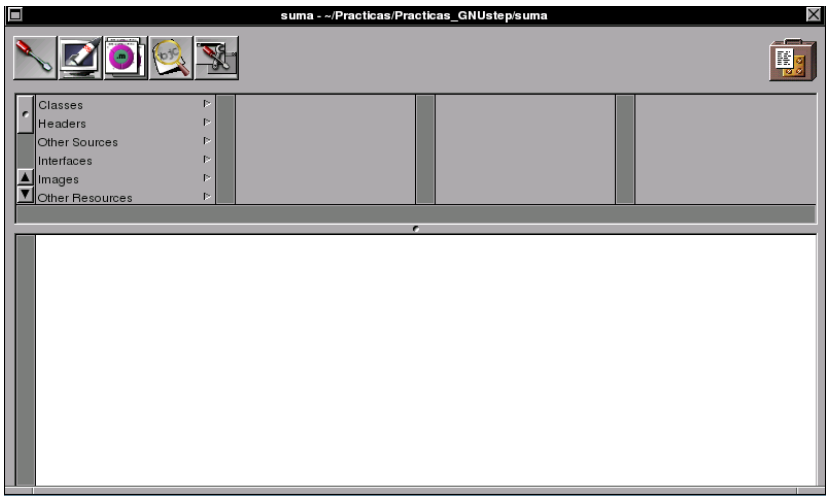


Imagen 5. Archivos del proyecto **suma**.

En la carpeta virtual **Interfaces**, Project Center crea un archivo llamado *suma.gorm*, el cual es la base de la interfaz gráfica de nuestra aplicación. Haciendo doble clic sobre este archivo, se inicia la aplicación GORM, con lo cual podemos comenzar a modelar la interfaz de nuestra aplicación. GORM abre tres ventanas al ser iniciado:

1. La ventana **main panel** con el titulo *suma.gorm*. Esta ventana tiene cinco pestañas. Las primeras tres nos muestran los Objetos, Imágenes y Sonidos de nuestra aplicación, respectivamente. La cuarta pestaña nos permite ver la jerarquía de las clases y elegir una para un nuevo Objeto. Y la quinta nos permite establecer algunas de las características del archivo *suma.gorm*.

Imagen 6. **main panel** de GORM.

2. La ventana **Palettes**, que contiene cinco paletas. Entendiendo por *paleta* un conjunto de Objetos visuales a los que comúnmente se les llama componentes o controles. Las cinco paletas son: *Menus Palette*, *Windows Palette*, *Controls Palette*, *Containers Palette* y *Data Palette*. Estas paletas se seleccionan con los iconos de la parte superior de la ventana.

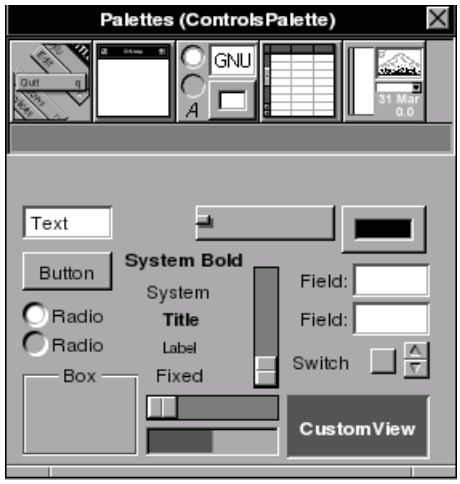
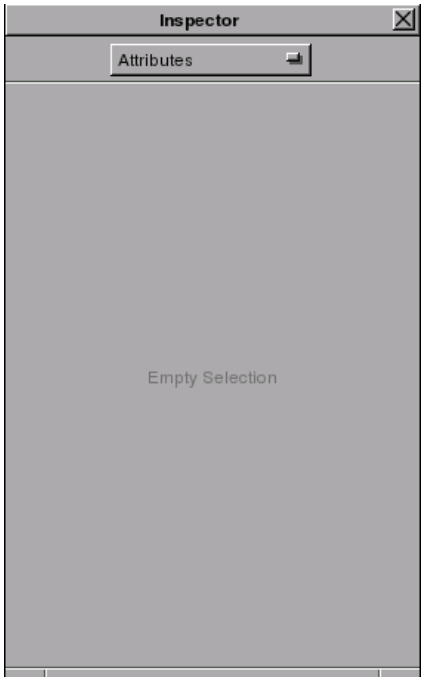


Imagen 7. Componentes de la paleta *Controls Palette*.

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

Imagen 8. **Inspector** de GORM:



Por el momento, nuestra aplicación solamente consta del pequeño menú con el titulo **main menu** y del icono con el logo de GNUstep que aparece en la esquina inferior izquierda. Sin embargo, nuestra aplicación necesitara de una ventana, por lo que deberemos agregar una. Para hacerlo, seleccionamos de la ventana **Palettes** la paleta *Windows Palette*. Y haciendo clic izquierdo sobre el componente *Window*, y manteniendolo presionado, lo arrastramos hacia la ventana **main panel**, la cual debe tener la pestaña *Objects* seleccionada.

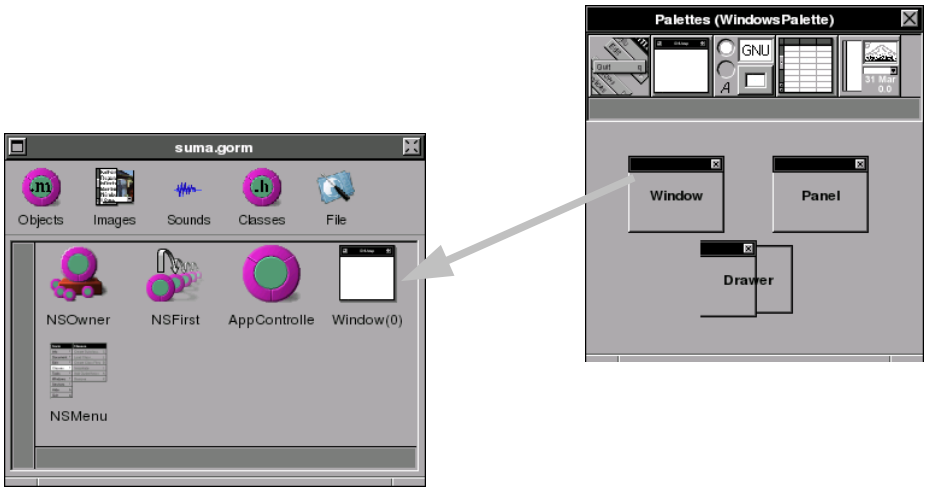


Imagen 9. Arrastrando un componente *Window*.

Hecho esto, se creara una ventana en nuestra aplicación, la cual tendrá el título *Window*. Seleccionada la ventana, podemos modificar sus propiedades en la ventana **Inspector**. Coloquemosle el título de *Aplicación de suma* y seleccionemos un color para nuestra ventana con el icono de selección de color.

Imagen 10. Icono de selección de color.

Que nos muestra en el centro el color que actualmente esta seleccionado, el cual podemos cambiar dando un clic izquierdo sobre el borde para abrir la ventana de selección de color.

Por defecto, la ventana no es visible al momento de iniciar una aplicación. Para hacer que la ventana de nuestra aplicación sea visible desde el primer momento, debemos seleccionar la opción **Visible at launch time** en el **Inspector** de objetos. Podemos redimensionar la ventana de nuestra aplicación con el mouse, o elegir la opción *size* de la lista desplegable y editar las dimensiones.

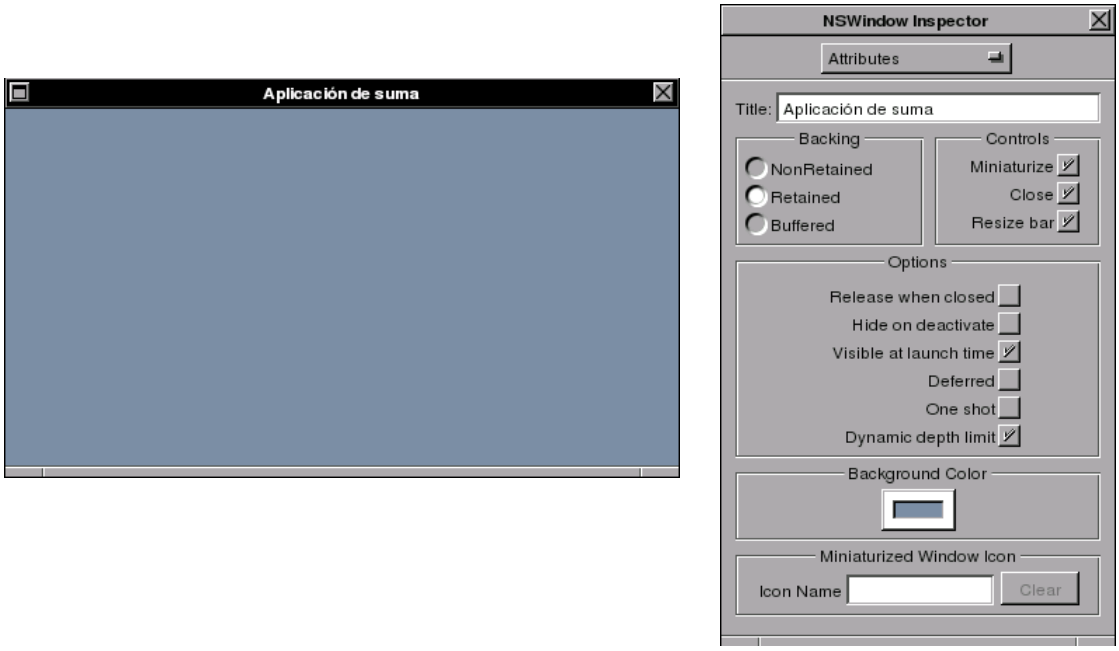


Imagen 11. Estableciendo propiedades para nuestra ventana.

Coloquemos ahora tres componentes *Text*, dos *Title* y un *Button* en nuestra ventana. Para ello, primero seleccionamos la paleta *Controls Palette*, y dando un clic izquierdo sobre el componente deseado, y manteniendo el clic izquierdo presionado, lo arrastramos a la posición deseada en la ventana. Ubiquemos estos componentes como muestra la imagen 12.

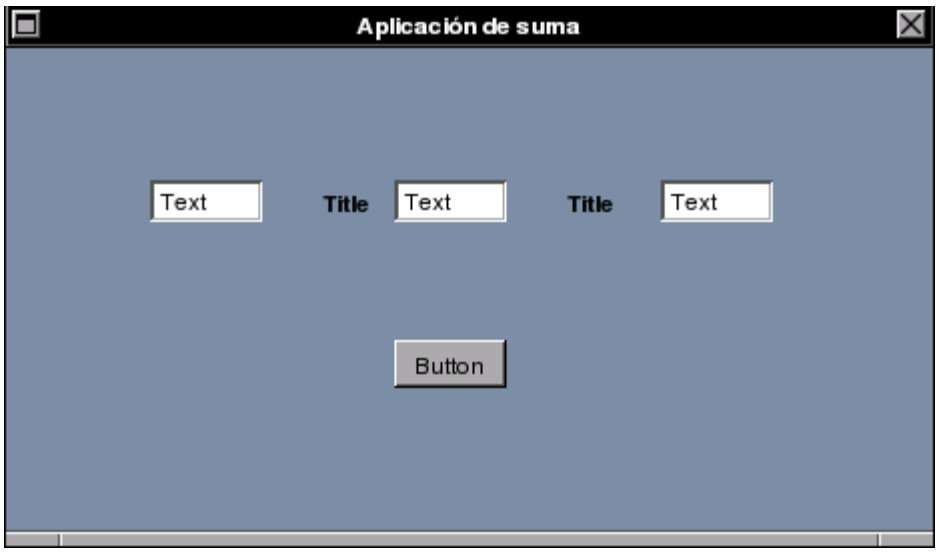


Imagen 12. Ubicación de los componentes.

Haciendo doble clic sobre cada uno de estos componentes, podemos modificar sus textos. En los componentes *Text* coloquemos el número **0**, en el componente *Button* el texto **Sumar** y en los componentes *Title* los caracteres **+** e **=** respectivamente. Y con esto tenemos terminada la apariencia de nuestra ventana.

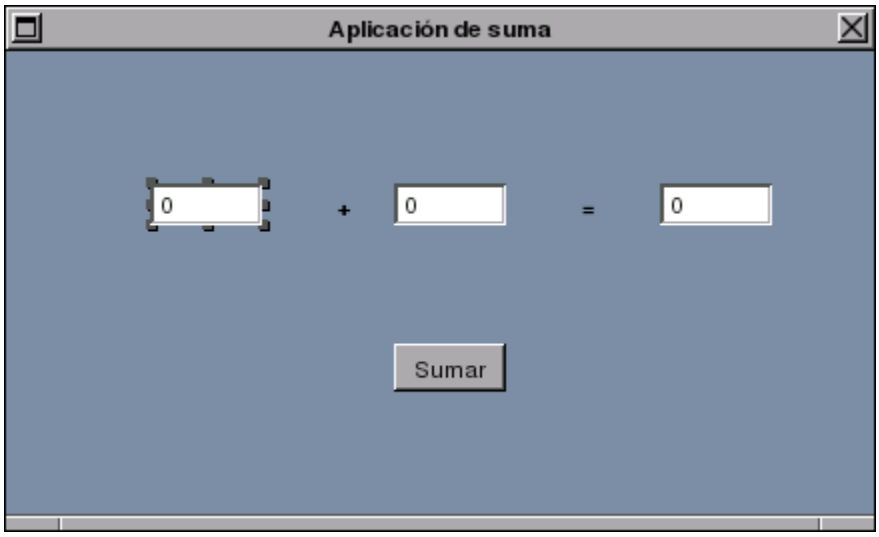


Imagen 11. Apariencia final de nuestra ventana.

Crearemos ahora un clase que derive de la clase NSObject. Y a partir de esta clase, crearemos un Objeto no visual que llevara a cabo la suma cuando el usuario de un clic sobre el botón *Sumar*. Llevar a cabo la suma no es algo complicado, es por esto que elegimos la clase más básica (la clase raíz NSObject) para crear nuestro Objeto. En la ventana **main panel** de GORM, seleccionemos la pestaña *Classes*, y luego la clase NSObjetc (imagen 12). Con esta clase seleccionada, vamos al menú de GORM y seleccionamos la opción *Classes* y, dentro de esta, la opción *Create SubClass...*. Hecho esto, veremos entre las subclases de NSObject una clase seleccionada y con el nombre NewClass. En el Inspector podemos modificar el nombre de esta nueva clase, cambiemos el nombre a MCSuma y presionemos la tecla ENTER. Veremos una advertencia donde se nos informa de que si cambiamos el nombre a la clase, las conexiones que esta tenga serán destruidas. Sin embargo, como esta clase es nueva, aun no tiene ninguna conexión, así que demos clic izquierdo sobre **OK**. (en realidad podemos darle cualquier nombre a la clase, sin embargo, es bueno tener un código para nombrar las clases, en este caso MC es de **Mi Clase**). A esta nueva clase, le agregaremos tres *Outlets* y un *Actions*. De los tres *Otlets*, dos serán para los datos que necesite de los dos componentes *Text* donde el usuario ingresara los sumandos. Y el tercero, para el componente *Text* donde deberá mostrar el resultado de la suma (figura 5). Para crear estos Outlets seleccionamos la pestaña Outlets en el Inspector (con la clase MCSuma todavía seleccionada en **main panel**). Y dando un clic izquierdo sobre el botón **Add Outlet** agregamos un Outlet, al cual podemos cambiarle el nombre dando doble clic sobre el nombre que GORM le da por defecto. De esta forma creamos tres Outlets llamados *sumando1*, *sumando2* y *resultado*.

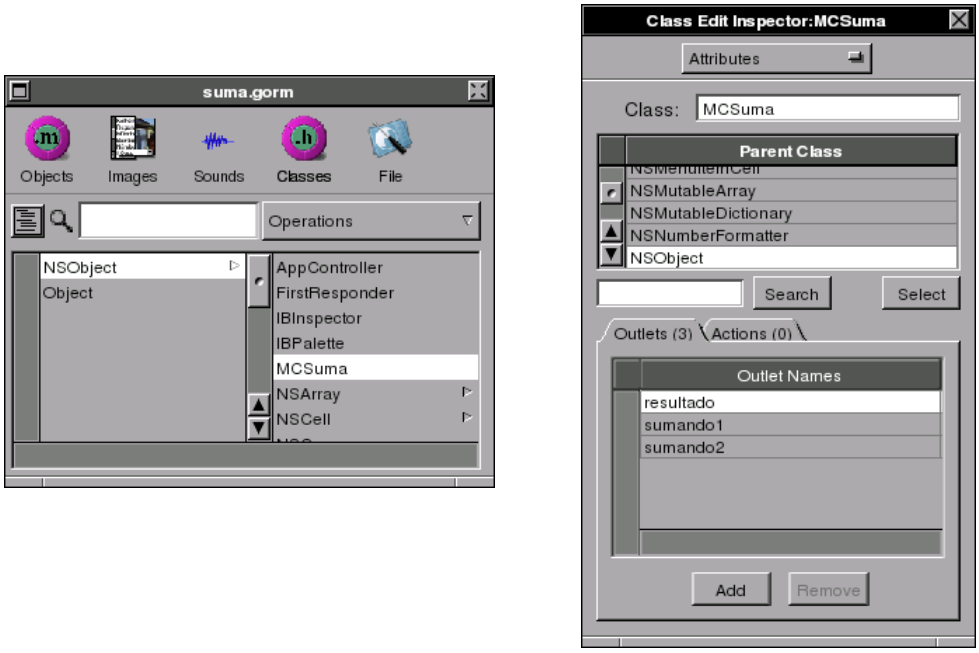


Imagen 12. Clase MCSuma y sus tres Outlets.

Seleccionando ahora la pestaña Actions en el Inspector, crearemos un Action para el botón Sumar. Este action sera el encargado de hacer que nuestro Objeto no visual realise la suma[6]. Dando un clic derecho sobre el botón **Add Action** agregamos un Action, al cual podemos cambiarle el nombre dando doble clic sobre el nombre actual. Llamemos a este Action *sumar:* (los dos puntos al final, no son parte del nombre, pero son obligatorios). Y con esto tendremos la estructura de nuestra clase.

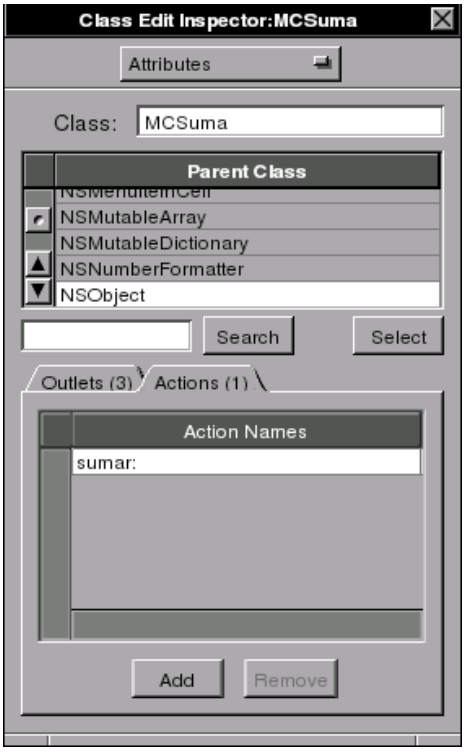


Imagen 13. Action de la clase MCSuma.

6 GORM crea automáticamente, en nuestro Objeto MCSuma, un método con el mismo nombre que le demos al Action. De tal forma, que cuando el usuario de nuestro programa de un clic en el botón *Sumar*, este enviara un mensaje que ejecutara dicho método.

Crearemos ahora un Objeto a partir de nuestra clase MCSuma. Para ello, primero seleccionamos nuestra clase en el **main panel** de GORM. Luego vamos al menú de GORM y seleccionamos la opción *Classes* y, dentro de ella, la opción *Instantiate*. En el **main panel** la pestaña *Objects* se seleccionara automáticamente, y veremos un Objeto llamado MCSuma, que es el Objeto creado a partir de nuestra clase MCSuma. Conectaremos, ahora, este Objeto con los tres componentes *Text* y con el componente *Button*.

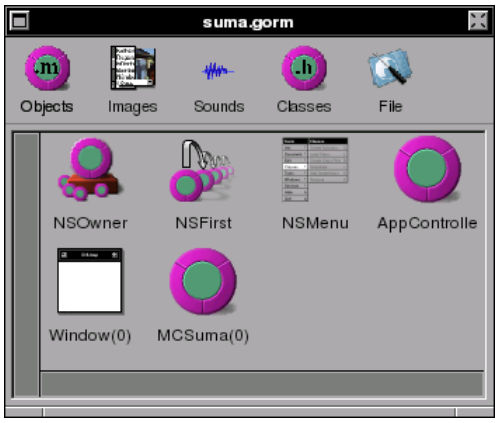


Imagen 14. El Objeto MCSuma.

La conexión se realiza de la siguiente forma. Seleccionamos el Objeto MCSuma en el **main panel**, y manteniendo el clic izquierdo presionado y la tecla CTRL, arrastramos el Objeto hasta el primer componente *Text* (el del primer sumando). Hecho esto, deberá aparecer un pequeño circulo rojo con la letra **T** en el componente *Text*, la **T** es de Target (objetivo).

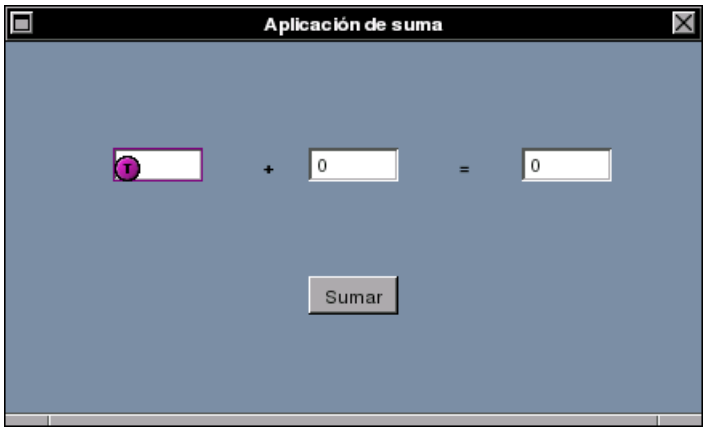


Imagen 15. La primera conexión.

En la ventana Inspector, seleccionemos el Outlet *sumando1*, con lo que veremos un pequeño circulo

verde con la letra **S** cerca del Objeto MCSuma, la **S** es de Source (fuente). Para establecer la conexión, damos un clic izquierdo sobre *Connect*. La nueva conexión aparecerá en la sección *Connections* del Inspector.

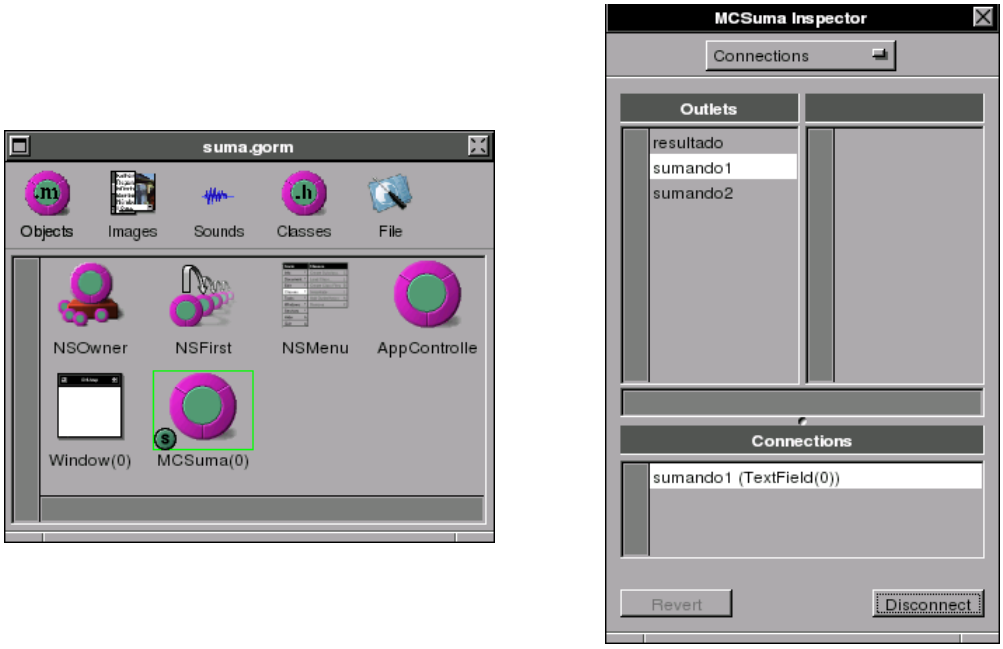


Imagen 16. La primera conexión Outlet.

Damos un clic izquierdo sobre cualquier parte de la ventana para hacer desaparecer los círculos verde y rojo. Y procediendo de forma similar, realizamos las conexiones para los Outlets *sumando2* y *resultado*. Hecho esto, realicemos la conexión del Action. Para ello, damos clic izquierdo sobre el componente *Button* de nuestra ventana, y manteniendo el clic izquierdo presionado y la tecla CTRL, arrastramos el componente hasta el Objeto MCSuma en el **main panel**. Deberemos ver un pequeño circulo verde con la letra **S** (fuente) cerca del componente *Button*, y un pequeño circulo rojo con la letra **T** (objetivo) cerca del Objeto MCSuma.

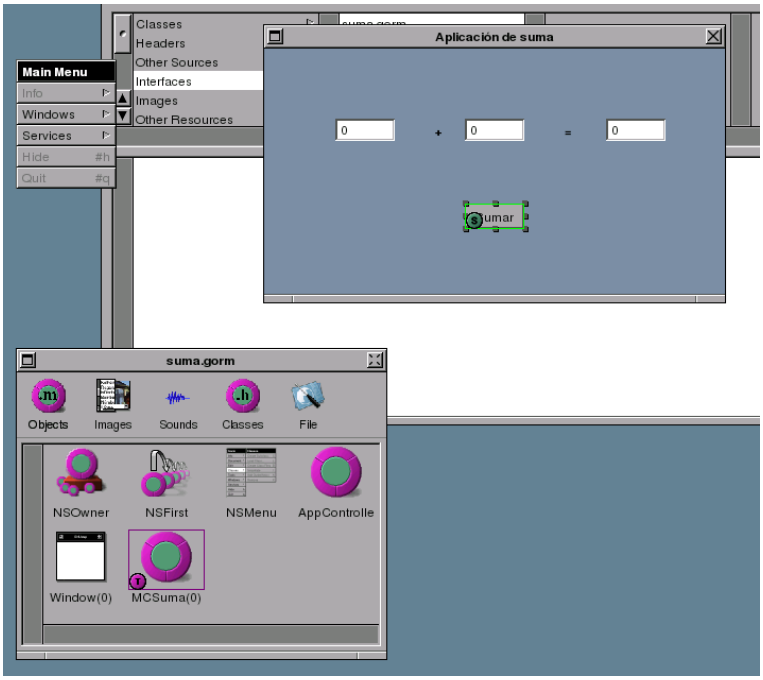


Imagen 17. La conexión del Action.

Ahora en el Inspector, en la sección Outlets, seleccionamos la opción *target*, y dentro de ella el Action que queremos conectar, en esta caso el único Action *sumar:*. Para establecer la conexión damos un clic sobre *Connect*, y veremos la conexión creada en la sección *Connections* del Inspector.

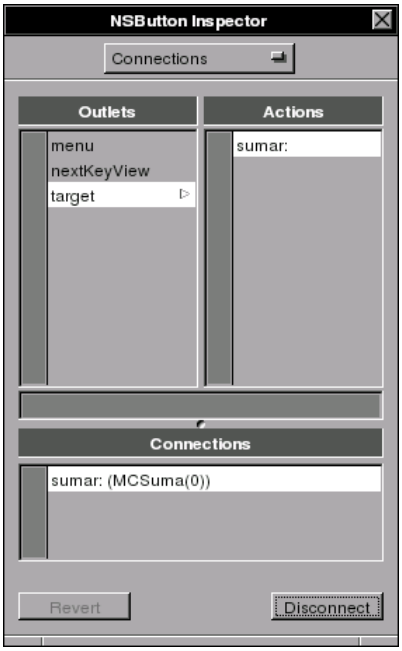


Imagen 18. Estableciendo la conexión del Action.

Ahora necesitamos crear los archivos de la interfaz y la implementación de nuestro Objeto. Para ello, seleccionamos nuestro Objeto en el **main panel** (en la pestaña Classes, no en la Objects), y en la opción *Classes* del menú de GORM, seleccionamos la opción *Create Class Files*. Una ventana nos preguntara por el nombre y la ubicación del archivo de implementación, simplemente demos clic en **OK**. E inmediatamente una ventana similar nos preguntara lo mismo para el archivo de la interfaz, nuevamente sólo damos clic en **OK**.

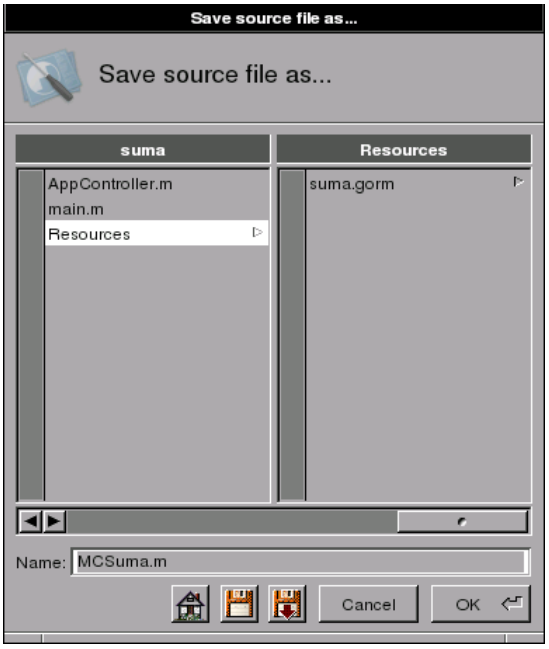


Imagen 19. Guardando el archivo de la implementación.

Por último, cambiemos el título de nuestro menú. Para hacerlo, seleccionemos el Objeto NSMenu en el **main panel** (en la pestaña *Objects*), y en el Inspector pongamosle el título **Suma**. Para guardar nuestra interfaz gráfica, en el menú de GORM seleccionemos la opción *Document* y dentro de ella la opción de *Save all*. Una ventana de advertencia nos dirá que debemos tener precaución con la compatibilidad de nuestra interfaz gráfica y las librerías de GNUstep, simplemente demos clic en **OK**, y cerremos GORM con un clic en la opción *Quit* del menú.

4.2 Escribiendo código para nuestro Objeto

En la sección anterior, hemos creado la interfaz gráfica de nuestra aplicación, y el Objeto que llevara a cabo la suma. Sin embargo, en ningún momento le hemos dicho al Objeto MCSuma como llevar a cabo dicha operación. Esto es lo que haremos ahora. Primero, debemos agregar a nuestro proyecto los archivos de la interfaz y de la implementación de nuestro Objeto MCSuma. Para hacer esto, en Project Center seleccionamos la carpeta virtual *Classes*, y luego en el menú de Project Center la opción *Project*, y dentro de esta la opción *Add Files...*. En la ventana que aparece, debemos buscar la carpeta

de nuestro proyecto. Dentro de esta hay una carpeta virtual llamada *Resources*, en la cual se encuentra el archivo MCSuma.m. Una vez seleccionado este archivo, damos clic en **OK** (el archivo de la interfaz es incluido automáticamente al agregar el archivo de la implementación).

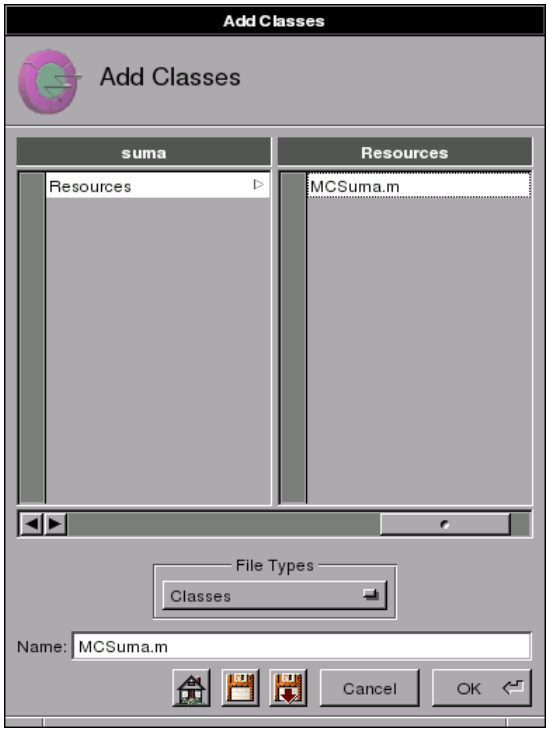


Imagen 20. Agregando el archivo MCSuma.m.

Una vez agregados estos archivos, seleccionemos el archivo MCSuma.m en la carpeta virtual *Classes*. Este archivo contiene un comentario que dice “insert your code here” (inserte su código aquí). Es entre las dos llaves que contienen este comentario, donde debemos agregar nuestro código. Observese que estas dos llaves son el cuerpo del método de instancia *sumar* (este es el método que GORM creo automáticamente cuando agregamos el Action). Este método es el que se ejecutara cuando el usuario de un clic izquierdo sobre el botón Sumar de nuestra aplicación.

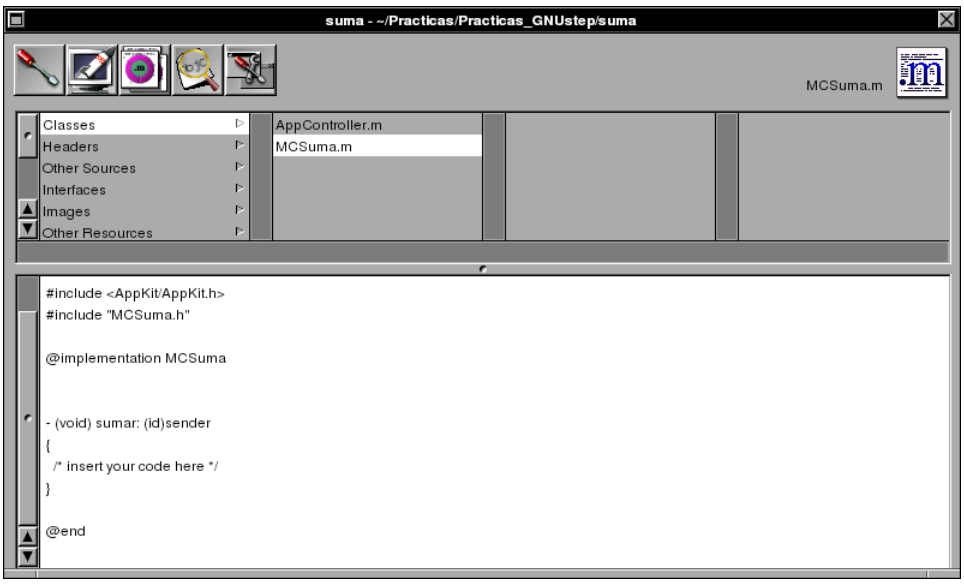


Imagen 21. Contenido del archivo MCSuma.m.

El código para realizar la suma, podría ser de la siguiente forma

```
float numero1, numero2, suma ;

numero1 = [sumando1 floatValue] ;
numero2 = [sumando2 floatValue] ;

suma = numero1 + numero2 ;

[resultado setFloatValue: suma] ;
```

En la primera linea declaramos tres variables de tipo float. Seguidamente, en la variable *numero1*, almacenamos el dato retornado por el mensaje **[sumando1 floatValue] ;**. Este mensaje es enviado por la conexión *sumando1*, la que conecta con el primer componente *Text* de nuestra aplicación, y ejecuta el método floatValue del componente *Text*. Este método retorna el dato de tipo float que contenga el componente. La siguiente linea hace lo mismo para la variable *numero2*. Luego se realiza la suma de estas variables, y el resultado se almacena en la variable *suma*. Seguidamente se enviá el mensaje **[resultado setFloatValue: suma] ;**. Este mensaje se enviá por la conexión *resultado* y ejecuta el método setFloatValue[7] que requiere un parámetro de tipo float, en este caso la variable *suma*, y muestra el contenido de este parámetro en el componente. El código anterior funciona, sin embargo, podemos obtener el mismo resultado en una sola línea de código, y sin tener que declarar variables. Esto se hace

7 La clase de los componentes *Text* es NSTextFiel, la cual deriva de la clase NSCrontrol. Y, por lo tanto, tienen métodos floatValue y setFloatValue (consultar el sistema de documentación de GNUStep).

de la siguiente forma

[resultado setFloatValue: [sumando1 floatValue] + [sumando2 floatValue]] ;

¿Se comprende este código? En este caso el parámetro que se enviá es el resultado de la operación: **[sumando1 floatValue] + [sumando2 floatValue]**. Insertando esta línea de código, el archivo MCSuma.m queda como muestra la imagen 22.

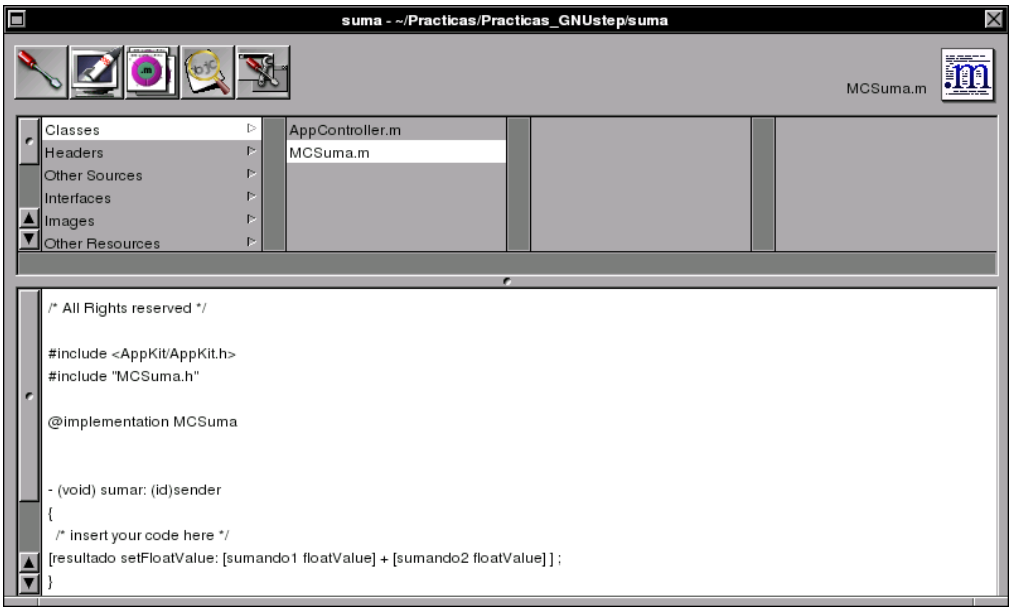


Imagen 22. El nuevo archivo MCSuma.m.

Debemos guardar este cambio al archivo. Para ello, con el archivo MCSuma seleccionado, seleccionamos en el menú de Project Center la opción *File*, y dentro de esta la opción *Save File*. Y con esto tendremos terminado nuestro proyecto.

4.3 Compilando y ejecutando nuestra aplicación

Para compilar nuestro proyecto, demos un clic izquierdo en el icono que muestra un destornillador, el cual abre la ventana **Project Build**. Esta ventana también contiene un icono con un destornillador, y dando un clic sobre este, comienza la compilación de nuestro proyecto. Si no hay ningún error en nuestro proyecto, veremos el mensaje *Build succeeded!*.

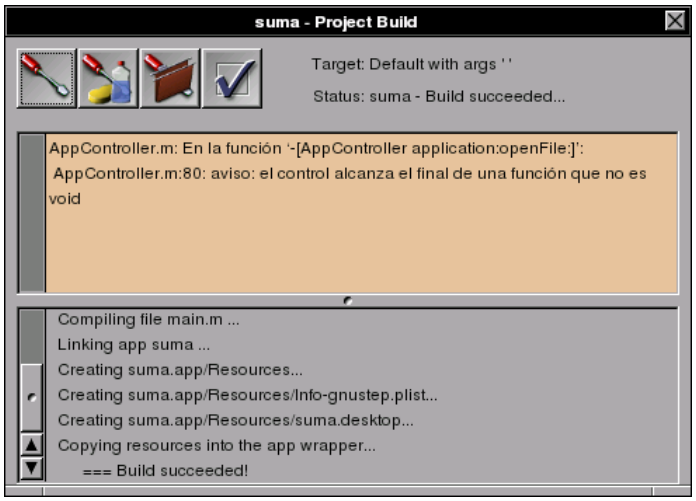


Imagen 23. Compilando nuestro proyecto.

Si todo salio bien, cerremos la ventana **Project Build**, y demos un clic en el icono que muestra un cohete saliendo de un monitor. Esto abre la ventana **Launch**, la cual tiene un icono igual. Dando un clic sobre este icono, nuestra aplicación es **lanzada** o ejecutada.

Imagen 24. Ventana **Launch**.

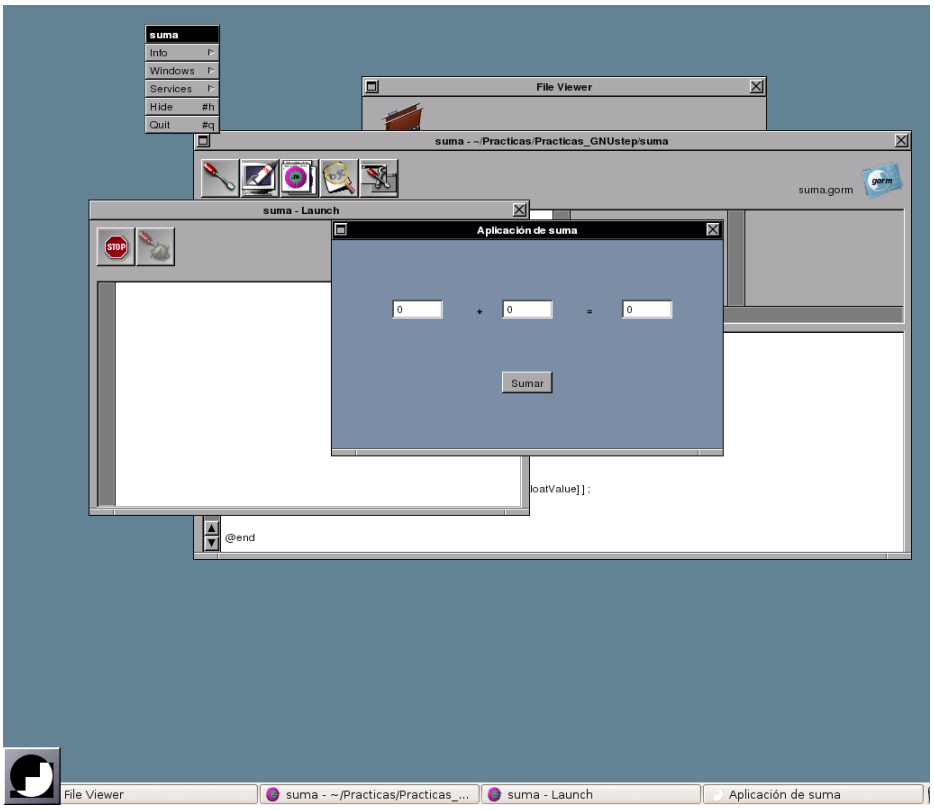


Imagen 25. Nuestra aplicación ejecutándose.

Ingresemos un par de valores para probar la aplicación.

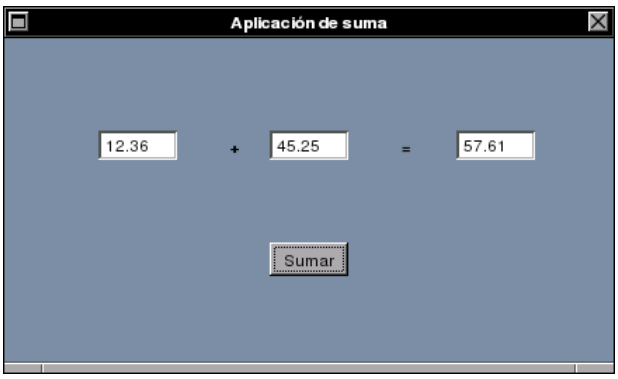


Imagen 26. Probando nuestra aplicación.

Por último, para cerrar la aplicación, damos un clic en la opción *Quit* del menú.

Capítulo 5

Nuevos conceptos y algunas aplicaciones de ejemplo

El lector ya ha realizado su primera aplicación en el capítulo anterior. De aquí en adelante, en los ejemplos que veamos, no detallaremos la creación de la interfaz gráfica de un programa o de sus conexiones. Ya que con lo expuesto en el capítulo anterior, es suficiente para que el lector haya aprendido como arrastrar Objetos desde las paletas, crear subclases, instanciar Objetos y realizar conexiones. De esta forma, nos centraremos en la creación de código para nuestros Objetos.

Para que el lector pueda sacar el mayor provecho de este capítulo, es necesario que ya este familiarizado con el **Sistema de Documentación de GNUstep**. Si no es así, en el **Apéndice B** se presenta una breve descripción, suficiente para aprender el manejo de este sistema. Asimismo, en el **Apéndice A**, se presentan algunas de las funciones más comunes de la librería math.h. Esto porque el **Sistema de Documentación de GNUstep**, solamente documenta las librerías de clases, no las de funciones.

5.1 Strings

Se conoce como **string** a un dato que consiste en una serie de caracteres. Por ejemplo

45rt5 5%fh &/alkjkjaskj2764#”&%$7dk

Es una serie de caracteres o **cadena de texto**, donde el espacio en blanco se considera también un carácter. Entre las clases no visuales de GNUstep, existe una clase que nos permite manejar datos de tipo **string**, esta clase es NSString. Por ejemplo, el siguiente código

```
NSString *nombre ;
nombre = @“Claudia” ;
```

Asigna el texto *Claudia* al puntero a Objeto *nombre* de tipo NSString. Observese que la cadena de texto que se desea asignar debe ir entre comillas y precedida por el carácter @. Conocer el uso de la clase NSString, nos sera útil en los siguientes ejemplos, para establecer las propiedades de algunos Objetos.

5.2 Métodos de Instancia y Métodos de Clase

Al proceso de crear un Objeto a partir de una clase, se le conoce como **Instanciar** la clase. De esta forma, los **Métodos de Instancia** se refieren a los métodos que se efectúan sobre un Objeto. Mientras que los **Métodos de Clase**, se refieren a los métodos que se efectúan sobre la clase misma. Los mensajes que ejecutan métodos de clase, son de la forma

```
[nombre_clase nombre_método] ;
```

Donde solo se requiere del nombre de la clase, y del nombre del método de clase que se desee ejecutar. Es decir, que no se necesitan conexiones Outlet para ejecutar métodos de clase. Bien pero ¿Que significa ejecutar un método de clase? Veamos un sencillo ejemplo.

Realicemos una aplicación que consista de una ventana con un componente *Text* y un componente *Button*. Al dar el usuario un clic sobre el botón, el programa deberá mostrar la hora actual en el componente *Text*. El siguiente es un modelo del programa

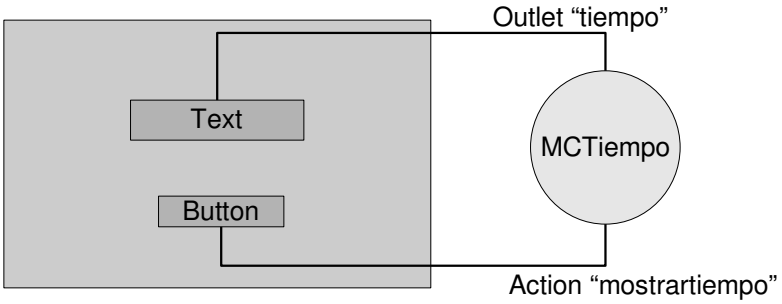


Figura 6. Modelo del programa Tiempo.

Una vez creada la interfaz gráfica, la clase MCTiempo (creada a partir de la clase NSObject), las conexiones correspondientes y agregado el archivo MCTiempo.m (o el nombre que se le haya dado) a nuestro proyecto, lo que nos resta por hacer es escribir el código de nuestro Objeto.

En esta aplicación, haremos uso de la clase no visual NSCalendarDate, la cual nos permite manejar fechas y horas. Sin embargo, antes de escribir el código, recordemos que a una variable le podemos asignar un valor al momento de ser declarada. Por ejemplo, para la variable *x* podemos escribir

```
float x = 1.56 ;
```

Algo similar podemos hacer al momento de declarar un puntero a Objeto, con la diferencia de que se le debe asignar un Objeto. El código para nuestra aplicación, el cual debe ir donde aparece el comentario "insert your code here" en el archivo MCTiempo.m, es el siguiente

```
NSCalendarDate *fecha = [NSCalendarDate calendarDate];
[fecha setCalendarFormat: @"%H : %M : %S"];
[titulo setStringValue: [fecha description]];
```

En la primera línea, se declara un puntero a Objeto llamado *fecha* y de tipo NSCalendarDate. Al mismo tiempo que se declara este puntero a Objeto, se le asigna el Objeto devuelto por el mensaje **[NSCalendarDate calendarDate];**, este mensaje se envía a la clase NSCalendarDate y ejecuta el método de clase **calendarDate**, el cual retorna un Objeto de tipo NSCalendarDate con la fecha y hora actual. En la segunda línea, se le envía un mensaje al Objeto contenido en *fecha* para ejecutar el método de instancia setCalendarFormat. Este método recibe un parámetro de tipo **NSString**, el cual consiste en una serie de caracteres especiales separados por dos puntos, indicando que datos de la fecha y hora nos interesan. En este caso, solamente nos interesan la hora (%H), los minutos (%M) y los segundos (%S). Seguidamente, enviamos un mensaje por la conexión *titulo*, la conexión al componente *Text*, para ejecutar el método de instancia setStringValue, el cual requiere un parámetro de tipo **NSString**, cuyo contenido se muestra en el componente[8]. El parámetro que se le enviá a este método es el dato, de tipo **NSSTring**, devuelto por el mensaje **[fecha description]**. Este mensaje se envía al Objeto fecha y ejecuta el método de instancia **description**, el cual retorna la fecha y hora con el formato que se le ha asignado (el formato asignado en la línea anterior).

Terminada la escritura del código, solo falta guardar los cambios, compilar y probar la aplicación.

8 Los métodos floatValue, intValue y stringValue retornan, respectivamente, los datos de tipo **float**, **int** y **string** que contenga un componente. Mientras que los métodos setFloatValue, setIntValue y setStrinValue permiten pasar datos de tipo **float**, **int** y **string**, respectivamente, los cuales serán mostrados por el componente.

5.3 Nuevamente Outlets y Actions

Anteriormente dijimos que los Outlets son conexiones que le permiten a un Objeto dado enviar mensajes a otros Objetos, y que los Actions son conexiones que le permiten recibir mensajes de otros Objetos. Esto no es totalmente cierto, ya que un Objeto puede enviar un mensaje a través de una conexión Action, al Objeto que le envía mensajes. Esto se logra utilizando la variable **sender**, que actúa como un Outlet hacia el Objeto que envía el mensaje. El siguiente ejemplo muestra como hacer esto.

La siguiente aplicación, consiste de una ventana con un componente *Text* y un componente *Button*. Inicialmente, el componente *Text* tiene el valor 0, y el componente *Button* el título “Escribir 10”. Cuando el usuario de un clic sobre el botón, el componente *Text* cambiara su contenido a 10, y el título del botón cambiara a “Escribir 0”. El modelo de este programa es similar al mostrado en la figura 6, con la diferencia de que ahora el nombre del Objeto es MCBoton, el nombre del Outlet *texto* y el nombre del Action *boton*.

Una vez creada la interfaz gráfica y agregado el archivo MCBoton.m a nuestro proyecto, lo que resta es escribir el código de nuestro Objeto. A continuación, se muestra el contenido completo del archivo MCBoton.m, donde el código que nos corresponde escribir se muestra en color negro.

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "MCBoton.h"

@implementation MCBoton


- (void) boton: (id)sender
{
  /* insert your code here */
int estado = [sender tag] ;

if ( estado == 0 )
  {   [texto setIntValue: 10] ;
      [sender setTitle: @"Escribir 0"] ;
      [sender setTag: 1] ;  }
else
  {   [texto setIntValue: 0] ;
      [sender setTitle: @"Escribir 10"] ;
      [sender setTag: 0] ;  }
}
```

```
@end
```

Antes de analizar este código, mencionemos que los componentes *Button*, como muchos otros componentes, tienen una variable **Tag** para almacenar un dato de tipo **int**. El valor que por defecto tiene esta variable es 0, y este valor lo asociaremos en nuestra aplicación con el titulo que inicialmente tiene el botón, “Escribir 10”.

En el código anterior, el método de instancia **boton** es el que se ejecuta cuando el usuario da un clic en el botón de nuestra aplicación. En la primera línea que nos corresponde escribir, se declara la variable *estado* de tipo **int**. Al mismo tiempo, se le asigna el dato de tipo **int** retornado por el mensaje **[sender tag] ;**. Este mensaje se envía al Objeto que ejecuta el método, el componente *Button* de nuestra aplicación, y ejecuta su método de instancia *tag*. Este método devuelve el valor de la variable **Tag** del componente *Button*, que inicialmente es 0. Seguidamente viene una sentencia condicional **if**, que evalúa el valor de la variable *estado*. Si este es 0, como al inicio, se envían tres mensajes. El primero de ellos, se envía por el Outlet *texto*, la conexión al componente *Text*, y ejecuta el método de instancia setIntValue pasandole el valor de 10. El segundo mensaje se envía hacia el botón, utilizando la variable *sender*, y ejecuta el método de instancia setTitle pasandole el **string** “Escribir 0”. El tercer mensaje se envía también hacia el botón, y ejecuta el método de instancia setTag, estableciendo el valor de la variable **Tag** a 1.

Y, como era de esperarse, los mensajes que se envían en la sección **else** de la sentencia (cuando el valor de la variable *estado* no es igual a 0), restablecen el valor inicial del componete *Text*, el valor inicial de la variable **Tag** y el título inicial del botón.

5.4 NSPoint, NSSize y NSRect

Las clases NSPoint y NSRect nos permiten crear, respectivamente, puntos y rectángulos. Los cuales, entre otras cosas, son útiles cuando deseamos realizar animaciones. La declaración de puntos y rectángulos es muy sencilla. Un punto requiere dos parámetros, correspondientes a sus coordenadas de posición ***x*** y ***y***. Y un rectángulo requiere cuatro parámetros, los primeros dos son las coordenadas ***x*** y ***y*** de la esquina inferior izquierda del rectángulo, y los últimos dos son, respectivamente, el alto y el ancho del rectángulo. En GNUstep, el origen del sistema coordenado se encuentra en la esquina inferior izquierda de la pantalla, y el eje **Y** es positivo hacia arriba y el eje **X** positivo hacia la derecha. Las clases NSPoint y NSRect tienen, sin embargo, una diferencia con el resto de las clases, y es que al definir objetos basados en estas clases, no es necesario utilizar el carácter *. Por ejemplo, seria un error escribir

```
NSPoint *punto1 ;
```

lo correcto es

```
NSPoint punto1 ;
```

Y deforma similar para la clase NSRect. Por ejemplo el siguiente código, crea un objeto NSPoint, llamado **punto**, y un objeto NSRect llamado **cuadro**.

```
NSPoint punto ;
NSRect cuadro ;
```

Ahora, para asignarle datos a estos objetos, utilizamos las funciones o macros NSMakePoint y NSMakeRect respectivamente. Por ejemplo

```
punto = NSMakePoint (5, 8) ;
cuadro = NSMakeRect (12, 18, 20, 30) ;
```

Estos datos asignados pueden, posteriormente, ser consultados de la siguiente forma

```
coordenada_x = punto.x ;
coordenada_y = punto.y ;
cuadro_origen_x = cuadro.origin.x ;
cuadro_origen_y = cuadro.origin.y ;
cuadro_alto = cuadro.size.height ;
cuadro_ancho = cuadro.size.width ;
```

En el código anterior, en las variables **coordenada_x** y **coordenada_y** se almacenan, respectivamente, las coordenadas ***x*** y ***y*** del objeto **punto**. En las variables **cuadro_origen_x** y **cuadro_origen_y** se almacenan, respectivamente, las coordenadas ***x*** y ***y*** de la esquina inferior del objeto **cuadro**. Y, por último, en las variables **cuadro_alto** y **cuadro_ancho**, se almacenan, respectivamente, el alto y el ancho del objeto **cuadro**.
En realidad, un rectángulo esta conformado por un punto, que es el origen del rectángulo, y por un objeto NSSize que le da sus dimensiones. Un objeto NSSize se crea de igual forma que un punto o un rectángulo, sólo que se hace uso de la función NSMakeSize. Veamos, por ejemplo, el siguiente código,

```
NSPoint origen ;
NSSize dimensiones ;
NSRect cuadro

origen = NSMakePoint (5, 8) ;
```

```
dimensiones = NSMakeSize (20, 30) ;
cuadro.origin = origen ;
cuadro.size = dimensiones ;
```

Primero se declaran los objetos *origen*, *dimensiones* y *cuadro*, de clases NSPoint, NSSize y NSRect respectivamente. Luego se le asignan valores a los objetos *origen* y *dimensiones*. Y, por último, se asignan los datos de estos objetos a *cuadro*.

Por el momento no haremos uso de estas clases, Pero más adelante, después de ver un par de temas importantes en el próximo capítulo, tendremos ocasión de verlas en acción.

5.5 El método awakeFromNib

El método awakeFromNib se ejecuta inmediatamente después de que nuestra aplicación lee el contenido del archivo **gorm** de la interfaz gráfica. Por lo tanto, aquí podemos establecer los valores o propiedades iniciales de los distintos objetos que componen la interfaz. Sin embargo, no debemos crear o liberar objetos con este método.

Este método no recibe ningún parámetro y no devuelve nada. Su estructura es

```
-(void)awakeFromNib
{
//Aquí van los cambios que queramos hacer.
}
```

Más adelante haremos uso de este método.

Capítulo 6

Profundizando en Objective-C

En este capítulo abordaremos varios temas importantes en Objective-C, como los son el ciclo de vida de los objetos, la gestión de la memoria y la re-definición de métodos. Estos tres temas son importantes si queremos realizar programas más complejos y, por lo tanto, es crucial que el lector los conozca.

6.1 Métodos

Hasta ahora, los métodos que hemos utilizado en nuestras aplicaciones, han sido creados por GORM. Limitándonos nosotros a escribir código en el cuerpo de los mismos. Sin embargo, es importante que conozcamos la forma en que se implementa un método. De hecho, esto lo comentamos ya en la sección 3.3, pero veamos aquí unos cuantos ejemplos para aclarar. Por ejemplo, un método de instancia (es decir, precedido por -) que no recibe ningún parámetro y que no devuelve nada seria de la siguiente forma

```
- (void) nombre_metodo
{
      Aquí va el cuerpo del método.
}
```

Observese que la palabra **void** entre paréntesis, es necesaria para indicar que el método no devuelve nada. Otro ejemplo seria un método que no reciba ningún parámetro, pero que devuelva un objeto cuya clase aun no esta determinada

```
- (id) nombre_metodo
{
      Aquí va el cuerpo del método.
            ...
      return nombre_objeto ;
}
```

Observese que en este caso, la palabra **id** entre paréntesis, indica que se devolverá un objeto cuya clase aun no se conoce. Notese también, que en la parte final del cuerpo, el objeto es devuelto con **return**, de forma similar a las funciones.

Veamos ahora como seria un método que recibe un objeto de clase NSSTring y que devuelve un objeto de clase NSBox

```
- (NSBox *) nombre_metodo: (NSString *) titulo
{
      Aquí va el cuerpo del método.
            ...
      return nombre_objeto ;
}
```

En este ejemplo, el parámetro lleva el nombre de **titulo**. Adviertanse los caracteres * que van después de las clases, tanto del parámetro como del objeto devuelto. Notese 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

```
- (NSWindow *) crearVentanaAncho: (int) ancho ventanaAlto: (int) alto
{
      Aquí va el cuerpo del método.
            ...
      return objeto_ventana ;
}
```

En este ejemplo, el método usa la etiqueta **ventanaAlto:** para el segundo parámetro. Suponiendo que la conexión hacia el objeto que implementa este método se llama miObjeto, la utilización de este método seria de la siguiente forma

```
[miObjeto crearVentanaAncho: x ventanaAlto: 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.

Por último, no esta de más recordar que si queremos que un método este disponible para otros objetos, este debe declararse en la interfaz (poniendo un punto y coma al final).

6.2 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 de la computadora 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 el objeto ya no es útil, esta parte de la memoria debe ser **liberada** para poder ser utilizada por otros programas. Si una parte de la memoria no es liberada cuando el objeto que almacena ya no es útil, esta parte de la memoria no podrá volver a ser utilizada por ningún otro programa, y sera un desperdicio de los recursos de la computadora (sólo podremos liberarla reiniciando la computadora).

Hasta aquí, los objetos que hemos utilizado en nuestros programas han sido creados mediante GORM. Y en estos casos, no hemos tenido que preocuparnos por el ciclo de vida de dichos objetos, ya que GORM se encarga de esto por nosotros. Sin embargo, se presentan ocasiones en que debemos 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 forma en que se administra esta memoria, depende de la forma en que fue creado el objeto.

Existen tres formas de crear un objeto. Las cuales analizaremos a continuación, con sus correspondientes métodos para administrar la memoria. No es necesario que el lector comprenda totalmente estos conceptos, sino solamente que tenga la idea. En el siguiente capítulo, donde los aplicaremos, todo quedara más claro.

6.2.1 Constructores convenientes

Estos son los más sencillos de utilizar, puesto que al crear un objeto mediante un constructor conveniente, 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 método que crea al objeto). Esto se entiende mejor con un ejemplo. Y, de hecho, ya realizamos anteriormente, en la sección 4.2, una aplicación que crea un objeto mediante un constructor conveniente. A continuación, se reproduce el

código contenido en el archivo MCTiempo.m (ver sección 4.2)

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "MCTiempo.h"

@implementation MCTiempo

- (void) mostrartiempo: (id)sender
{
  /* insert your code here */
NSCalendarDate *fecha = [NSCalendarDate calendarDate] ;
[fecha setCalendarFormat: @"%H : %M : %S"] ;
[titulo setStringValue: [fecha description]] ;
}

@end
```

En esta aplicación, cuando el usuario da un clic sobre el botón de la interfaz, se ejecuta el método **mostrartiempo** del objeto MCTiempo. Este método crea un objeto llamado **fecha** el cual deriva de la clase NSCalendarDate. Para esto hace uso del constructor conveniente **calendarDate**. Como se ve, los constructores convenientes son métodos de clase que devuelven objetos de dicha clase. Este objeto se destruye al terminar la ejecución del método **mostrartiempo**. Cuando el usuario da otro clic sobre el botón, nuevamente se crea y destruye el objeto **fecha**. Mientras este objeto existe se le da formato y se le extrae la fecha.

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.

6.2.2 alloc e init

No todos los objetos tienen constructores convenientes. En estos casos, deberemos crear los objetos haciendo uso de los métodos **alloc** e **init**. Por ejemplo, para crear un objeto llamado **caja** cuya clase sea NSBox tendríamos

```
NSBox *caja = [[NSBox alloc] init] ;
```

El método **alloc**, es un método de clase que, en este caso, devuelve un objeto de tipo NSBox. Sin embargo, este objeto devuelto aun no es utilizable. Para ello debemos inicializarlo mediante algún método para tal fin. Como veremos más adelante, existen varios métodos que nos permiten inicializar un objeto y, entre ellos, el más sencillo es el método **init**, el cual usamos aquí. Este paso de creación de un objeto, puede simplificarse si se utiliza el método de clase **new**, de la siguiente forma

```
NSBox *caja = [NSBox new] ;
```

Como veremos en ejemplos posteriores, ambas formas son útiles dependiendo de lo que necesitemos.

Lo importante a tener en cuanta al crear un objeto mediante alguna de estas formas, es que el objeto queda almacenado en la memoria de la computadora, y somos nosotros los responsables de liberarla. En este ejemplo, la liberación de la memoria que ocupa el objeto **caja**, puede realizarse ejecutando el método **release** del objeto

```
[caja release] ;
```

O también ejecutando

```
RELEASE(caja) ;
```

Observese que RELEASE esta escrito en mayúsculas.

En general, se recomienda que el objeto sea liberado por el mismo método que lo creo. Aunque en algunas situaciones esto no puede ser así.

6.2.3 autorelease

En algunos casos, deberemos crear métodos que devuelvan objetos. Es decir, métodos que al ser llamados devuelvan un objeto. Si el objeto a devolver es creado mediante un constructor conveniente, este sera liberado automáticamente después de haber devuelto el objeto, así que no tenemos que preocuparnos por liberar la memoria que este utiliza. Un ejemplo de esto es el siguiente código

```
- (NSCalendarDate *) devolverFecha
{
NSCalendarDate *fecha = [NSCalendarDate calendarDate] ;
return fecha ;
}
```

En este ejemplo, el método **devolverFecha**, devuelve un objeto de tipo NSCalendarDate. Este es creado mediante un constructor conveniente y luego es devuelto con **return**. Sin embargo, si el objeto es creado con **alloc** e **init**, o con **new**, debemos asegurarnos de que el objeto sea liberado después de ser devuelto (esto es lo recomendado). Para ello, hacemos uso del método de instancia **autorelease**, que como su nombre lo indica, se encarga de liberar automáticamente la memoria que utiliza el objeto. Un ejemplo de esto es el siguiente código

```
- (NSString *) devolverNombre
{
NSString *nombre = [[NSString alloc] initWithString: @"Germán Arias."] ;
[nombre autorelease] ;
return nombre ;
}
```

En este ejemplo el objeto **nombre** es inicializado con el método initWithString, el cual recibe un parámetro de tipo **string** que es transferido al objeto. Es decir que el objeto **nombre** es creado conteniendo la cadena “Germán Arias.”. Seguidamente se ejecuta el método **autorelease** que se encargara de liberar posteriormente al objeto. Y, por último, el objeto es retornado.

En ambos casos, el objeto devuelto solamente tiene una existencia temporal y luego es destruido por **autorelease**. Como veremos más adelante, esto es suficiente en muchos casos. Sin embargo, en otros, deberemos **retener** al objeto para que exista durante un mayor tiempo.

6.3 Mas sobre el ciclo de vida de los objetos

Es momento de profundizar en la forma en que los objetos son creados y destruidos, para tener una idea más clara de como es que funciona una aplicación. Comencemos por mencionar que todos los objetos son creados mediante los métodos **alloc** e **init**, o mediante **new**. Incluso cuando utilizamos un constructor conveniente el objeto es creado de esta forma, lo que sucede es que el constructor conveniente se encarga de esto por nosotros. Creado el objeto, podemos mandar mensajes a cualquiera de sus métodos de instancia para interactuar con el. Y una vez que este ya no nos es útil, lo liberamos y esta acción de liberar el objeto, hace que se envié un mensaje al método **dealloc**.

Para comprender todo esto utilicemos una analogía con los objetos o clases que componen una casa. Básicamente supongamos que tenemos tres clases: *cimientos*, *paredes* y *techo*. Donde la clase *paredes* es una subclase de la clase *cimientos*, y la clase *techo* es una subclase de la clase *paredes*. Esto tiene sentido, puesto que para poner un techo primero se necesita que hallan paredes, y para poner estas se

necesitan los cimientos. Supongamos que estas son las únicas clases en nuestro Framework, y que deseamos crear, para nuestra aplicación, un objeto antena. Bien, puesto que la antena va colocada en el techo, lo más lógico es derivar la clase de nuestro objeto antena de la clase *techo*. Llamaremos a esta nueva clase *antena*, y una vez creada esta, crearemos un objeto llamado MIAntena a partir de esta. Surge ahora la pregunta ¿donde va a ir colocada nuestra antena? ¿no necesitamos un techo para colocarla? ¿y no necesitamos paredes para colocar ese techo? ¿y cimientos para estas paredes? Se comprende, entonces, que al crear un objeto, este se debe encargar de crear todos los demás objetos que necesite para funcionar. En el caso de nuestro objeto MIAntena, al momento de ser creado, se dirige a la clase *techo* para crear un techo, a su vez este objeto techo se dirige a la clase *paredes* para crear las paredes, y estas, por último, se dirigen a la clase *cimientos* para crear los cimientos. De esta forma, nuestro objeto MIAntena se asegura de construir toda la infraestructura necesaria para poder funcionar. Se comprenderá también, que al momento de liberar a nuestro objeto MIAntena, este debe asegurarse de liberar el techo, las paredes y los cimientos que creo para su funcionamiento. Al momento de liberar nuestro objeto MIAntena, este se dirige al objeto techo y lo libera. Este, a su vez, se dirige al objeto paredes y lo libera, y este, por último, se dirige al objeto cimientos y lo libera. De esta forma, quedan liberados todos los objetos que utilizo nuestro objeto MIAntena.

Cuando un objeto es creado, el método **alloc** se encarga, entre otras cosas, de reservar la memoria necesaria para el funcionamiento del objeto. Hecho esto, se inicializa el objeto con algún método para tal fin. Estos métodos (cuyos nombres comienzan con **init**), se encargan de establecer valores para algunas de las variables utilizadas por el objeto. Una vez hecho esto, ejecutan el método de inicializacion de la superclase. Es decir, de la clase de la cual deriva el objeto. Este método de la superclase, después de establecer valores para algunas variables, ejecuta el método de inicializacion de su superclase, y así sucesivamente hasta llegar a la clase raíz.

Volviendo a la analogía, al crear nuestro objeto MIAntena, el método **alloc** se encarga de reservar la memoria necesaria para el funcionamiento de nuestro objeto. Seguidamente se inicializa el objeto con algún método **init**. Este establece valores para algunas variables (color, tamaño, forma, etc.) y luego ejecuta el método de inicializacion de su superclase, es decir de la clase *techo*. Este método crea un objeto techo estableciendo valores para algunas de sus variables (color, tamaño, etc.), y luego ejecuta el método de inicializacion de su superclase, es decir de la clase *paredes*. Este, a su vez, crea un objeto paredes estableciendo valores para algunas de sus variables, y luego ejecuta el método de inicializacion de su superclase, la clase *cimientos*. Y este crea, por último, el objeto cimientos estableciendo valores para algunas de sus variables. Es importante resaltar que el método **alloc**, solamente se ejecuta con la creación de nuestro objeto MIAntena, y no con el resto de los objetos. Esto se debe a que el método **alloc** se encarga de reservar la memoria suficiente para toda la cadena de objetos que necesite el objeto por crear.

Una vez un objeto ya no es útil se libera, y esta acción ejecuta el método **dealloc** de dicho objeto. Este método se encarga de liberar cualquier otro objeto que haya sido creado por el objeto que se esta liberando (si los hay), y luego ejecuta el método **dealloc** de su superclase. Este a su vez, ejecuta el

método **dealloc** de su superclase y así sucesivamente hasta llegar a la clase raíz.

Volviendo a la analogía, al liberar nuestro objeto MIAntena, se ejecuta su método **dealloc** el cual únicamente (MIAntena no creo ningún otro objeto) se encarga de ejecutar el método **dealloc** de su superclase, es decir de la clase *techo*. De esta forma, se libera el objeto techo que inicialmente creo nuestro objeto. Este techo, al ser liberado, ejecuta el método **dealloc** de su superclase, es decir de la clase *paredes*. Lo cual libera el objeto paredes, y al hacerlo, ejecuta el método **dealloc** de su superclase, la clase *cimientos*. Lo cual libera el objeto cimientos. De esta forma, todos los objetos que creo MIAntena, son liberados al ser liberado este. Y la memoria queda entonces disponible para otras aplicaciones.

Ahora contestemos la pregunta ¿De que sirve saber esto? Como ya sabemos (capítulo 3) cuando creamos un objeto a partir de una clase, este objeto se crea a partir de los planos de dicha clase. Si este objeto, creado según los planos de la clase, no es útil así, no hay problema. Si deseamos agregarle algunos nuevos métodos, tampoco hay problema. Si queremos modificar alguno de sus métodos (que no sean **alloc**, **init** o **dealloc**), en general, no hay problema. Pero, si queremos re-definir un método **init** o **dealloc** (es poco común la re-definición del método **alloc**), debemos tener cuidado para evitar que el programa vaya a cometer un grave error. Abordaremos esto en la sección 5.5, después de profundizar en el tema de los inicializadores en la siguiente sección.

6.4 Inicializadores de máxima genericidad

Anteriormente dijimos que los objetos pueden tener varios métodos de inicializacion. Estos métodos tienen una jerarquía que establece el orden en que se ejecutaran. Esta jerarquía se establece en relación a la cantidad de parámetros que reciba la función. Es decir, que en la parte más alta, se encuentra la función inicializadora que recibe más parámetros, seguida por las que reciben menos parámetros, hasta llegar a la función que menos parámetros recibe o a la que no recibe ningún parámetro. El método de inicio que esta en la parte más alta de la jerarquía es llamado *inicializador de máxima genericidad*. En la sección anterior, vimos que los métodos de inicializacion se encargan de establecer valores para algunas de las variables utilizadas por el objeto y luego ejecutan el método de inicializacion de la superclase. En realidad esto sólo lo hacen los inicializadores de máxima genericidad. El resto de inicializadores, después de establecer valores para algunas variables, ejecutan el método de inicializacion que se encuentra arriba de ellos en la jerarquía. El siguiente esquema aclara esto.

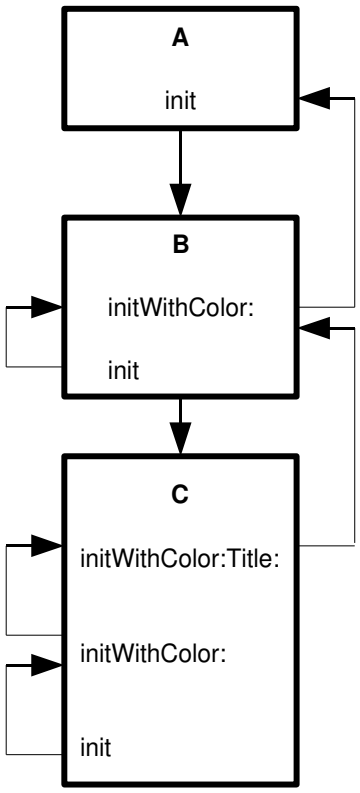


Figura 7. Jerarquía de los inicializadores.

En este esquema, del objeto (o clase) A deriva el objeto B y de este, deriva el objeto C. El objeto A solo tiene un método de inicio **init** que no recibe ningún parámetro, y que es el de máxima genericidad. El objeto B tiene dos métodos de inicio **init**, que no recibe ningún parámetro, e **initWithColor:** que recibe un parámetro y que, por lo tanto, es el de máxima genericidad. Y el objeto C tiene tiene, ademas de los métodos del objeto B, el método **initWithColor:Title:** el cual recibe dos parámetros y es el de máxima genericidad. Observese que el método de máxima genericidad de un objeto dado, ejecuta el método de máxima genericidad de la clase superior. Mientras que el resto de inicializadores, ejecutan el método de inicializacion que se encuentra arriba de ellos en la jerarquía, pero en la misma clase u objeto. Para comprender el porque de varios métodos de inicio, supongamos que creamos un objeto de la clase C y que lo inicializamos con el método **init**. Este método ejecuta el método initWithColor: el cual le asigna al objeto el color por defecto. Seguidamente, se ejecuta el método **initWithColor:Title:** el cual le asigna, nuevamente, el color por defecto pero ademas agrega el titulo por defecto. Pero, si quisiéramos crear el objeto con un color especifico, los mejor seria inicializarlo con el método **initWithColor:** para asignarle el color deseado. Y, claro esta, si quisiéramos crearlo con un color y un titulo especifico, utilizaríamos el método **initWithColor:Title:** para asignarle el color y el titulo deseados. Ya en el próximo capítulo, tendremos ocasión de utilizar métodos como estos.

6.5 Re-definición de métodos init y alloc y los receptores self y super

En situaciones en las que tengamos necesidad de re-definir un método de inicializacion, lo más común es re-definir el método de máxima genericidad. Sin embargo, si el objeto deriva de una clase que no posee un método de inicializacion, podemos buscar un método de inicializacion más arriba en la jerarquía hasta encontrar un método que se ajuste a nuestras necesidades.

Anteriormente, en relación a los métodos de máxima genericidad, dijimos que estos primero establecen valores para algunas de las variables y luego ejecutan el método de máxima genericidad de la superclase. Pero, de hecho, es al revés, primero se ejecuta el método de máxima genericidad de la superclase, y luego se establecen las variables. Algo que debemos tener presente, es que al re-definir un método de máxima genericidad debemos asegurarnos de ejecutar el método de máxima genericidad de la superclase. Ya que de lo contrario, no se creara toda la infraestructura para el buen funcionamiento de nuestro objeto. Para ejecutar el método de la superclase, se utiliza el receptor **super**, el cual hace referencia a la superclase. Mientras que el receptor **self**, se refiere al objeto que implementa la clase. El receptor **self** actuá como una variable de objeto y, por lo tanto, se le pueden asignar objetos. El siguiente es un ejemplo típico de re-definición de un método de máxima genericidad.

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

// Aquí van nuestras modificaciones al inicializador.

return self ;
}
```

En este ejemplo, se re-define el inicializador **init**. El cual, como todo método inicializador, devuelve un objeto. En la primera línea, se ejecuta el método **init** de la superclase con el mensaje [super init], y el objeto devuelto se almacena en la variable **self**. Esto crea toda la infraestructura para el funcionamiento de nuestro objeto. Seguidamente van nuestras modificaciones al objeto. Y, por último, se retorna el objeto creado. Algunas veces resulta conveniente verificar si el método de máxima genericidad 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 por lo tanto el objeto no pueda ser creado. El siguiente código es un ejemplo de esto

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

  if (self)
```

```
  {
  // Aquí van nuestras modificaciones al inicializador.
  }

return self ;
}
```

En este caso, el método initWithContentsOfFile: recibe un parámetro de tipo **string** que contiene la ruta del archivo a partir del cual se creara el objeto, y este mismo parámetro se le pasa al método de la superclase al ser llamado. Si **self** es verdadero, es decir, si el archivo se encontró y se pudo crear el objeto, entonces se realizan la modificaciones al inicializador. Observese que **return self** se encuentra fuera de la sentencia condicional. Esto significa que, se encuentre o no el archivo, el método devolverá un objeto. Aunque, por supuesto, si no se encontró el archivo, el objeto devuelto no sera el deseado. Más adelante, tendremos ocasión de aplicar la re-definición de un método de inicializacion de máxima genericidad. Por el momento, pasemos a los métodos **dealloc**.

El método **dealloc** se encarga únicamente de ejecutar, mediante **super**, el método **dealloc** de la superclase. Sin embargo, si nuestro objeto crea otros objetos que utilizara durante toda su existencia (es decir, que estos otros objetos deben existir tanto tiempo como nuestro objeto), debemos re-definir este método de tal forma que primero libere a todos estos objetos y luego ejecute el método **dealloc** de la superclase. Por ejemplo, supongamos que nuestro objeto crea tres objetos llamados *objeto1*, *objeto2* y *objeto3*, entonces, la re-definición del método **dealloc** seria de la siguiente forma.

```
-(void)dealloc
{
RELEASE(objeto1) ;
RELEASE(objeto2) ;
RELEASE(objeto3) ;
[super dealloc] ;
}
```

De esta forma, la memoria queda correctamente liberada.

Capítulo 7

Integrando todo

En este capítulo se presentan dos ejemplos a modo de integrar todo lo visto en los capítulos anteriores. El primero de ellos se presenta inicialmente de forma sencilla y luego se va modificando para hacerlo cada vez más complejo.

7.1 Un cronómetro

Construyamos primero la interfaz gráfica del cronómetro, similar a la que se muestra en la imagen 27. Seleccionada la ventana (en GORM) seleccionamos la opción *Visible at launch time* para que la ventana sea visible al momento de lanzar la aplicación. Y para la caja de texto, seleccionamos la alineación centrada como se ve en la imagen.

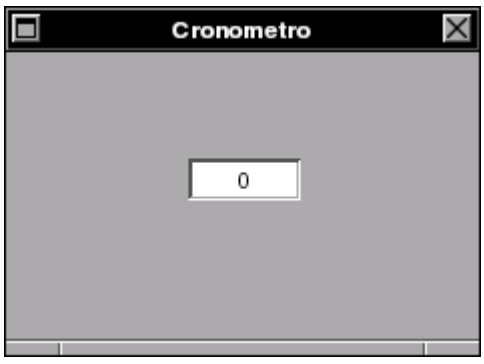


Imagen 27. Interfaz del cronómetro.

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 8.

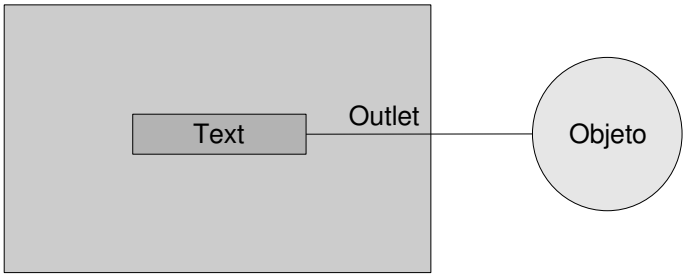


Figura 8. Modelo del cronómetro.

Así, necesitamos un objeto que controle el avance del tiempo y que comunique este avance a la caja de texto. Este proceso no es muy complicado, por lo que nuestro objeto puede crearse a partir de una clase que derive de la clase raíz NSObject. Por lo tanto, comencemos creando una clase derivada de la clase raíz NSObject, y a la cual llamaremos *Cronometro*. Hecho esto, instanciamos la clase y le creamos un Outlet con el nombre **tiempo**, el cual conectaremos con la caja de texto. Realizada esta conexión, generamos los archivos de la clase *Cronometro*, guardamos todo y volvemos a Project Center.

Una vez agregado el archivo Cronometro.m, procedemos a escribir el código de nuestra aplicación. Para el funcionamiento de nuestro cronómetro, utilizaremos un objeto NSTimer. Estos objetos pueden ejecutar algún método especificado cada cierto tiempo. El archivo Cronometro.m debe quedar de la siguiente forma

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "Cronometro.h"

@implementation Cronometro

-(id)init
{
   self = [super init] ;
   NSTimer *reloj ;
   segundos = 0 ;
```

```
    reloj = [NSTimer scheduledTimerWithTimeInterval:1
    target:self
    selector:@selector(segundero:)
    userInfo:nil
    repeats:YES] ;

    return self ;
}


- (void) segundero: (NSTimer *) timer
{
  segundos = segundos + 1 ;
  [tiempo setIntValue: segundos] ;
}

@end
```

Analicemos este código. El primer método, es la re-definición del método de inicializacion **init**. Por lo tanto, primero se ejecuta el método **init** de la superclase haciendo uso del receptor **super**, y el objeto devuelto se almacena en el receptor **self**. Hemos re-definido el método **init**, porque queremos que el objeto NSTimer exista desde el momento en que nuestro objeto Cronometro sea creado (es decir, desde el momento en que la aplicación es lanzada). Seguidamente, esta la declaración de un puntero a objeto llamado *reloj* y de clase NSTimer, y luego se le asigna el valor de 0 a la variable **segundos**. Esta última variable es la que llevara el conteo de los segundos. Luego se hace uso del constructor conveniente scheduledTimerWithTimeInterval: target: userInfo: repeats: para crear un objeto NSTimer que se almacena en la variable *reloj*. Este método recibe cinco parámetros, el primero de ellos, **WithInterval**, es el intervalo de tiempo entre cada ejecución del método que le indiquemos, en este caso 1 segundo; el segundo, **target**, es el nombre del objeto (o de la conexión) donde se encuentra el método que queremos que ejecute. En este caso es el mismo objeto Cronometro, de ahí que se le envíe como parámetro **self**); el tercero, **selector**, es el nombre del método que debe ejecutar, en este caso se le envía @selector(segundero:), donde **segundero:** es el nombre del método a ejecutar y **@selector()** convierte en tipo *selector* el nombre del método entre paréntesis[9]. El cuarto, **userInfo**, no lo utilizamos en este caso, por lo que se le envía *nil* (*nil* significa nulo). El quinto, **repeats**, sirve para indicar si queremos que el método se ejecute repetidamente o no, en este caso se le envía YES. Por último, el objeto creado es retornado.

9 Así como los números pueden ser de tipo **int** o **float**, las cadenas de texto pueden ser de tipo **string** o **selector**, todos los nombres de métodos son de tipo **selector**. Consultando el sistema de documentación, puede verse que el método **scheduledTimerWithTimeInterval: target: userInfo: repeats:** solicita el nombre del método en tipo **selector**, de ahí que la conversión sea necesaria.

El siguiente método, **segundero:**, es el método que el objeto *reloj* ejecutara repetidamente. Para que este método pueda ser ejecutado por el objeto *reloj*, este debe recibir un parámetro de tipo NSTimer. En este ejemplo, este parámetro tiene el nombre *timer*. Este método no devuelve nada, y lo que hace es aumentar el valor de la variable **segundos** en 1, para luego establecer el contenido de la caja de texto con el valor de dicha variable. Para hacer esto hace uso del Outlet **tiempo**.

Guardados los cambios al archivo Cronometro.m, solo falta agregar la variable **segundos** como atributo de nuestro objeto en el archivo Cronometro.h. Esto es así porque la variable **segundos** es usada por los dos métodos de nuestro objeto. El archivo Cronometro.h queda, entonces, de la siguiente forma

```
/* All Rights reserved */

#include <AppKit/AppKit.h>

@interface Cronometro : NSObject
{
  id tiempo;
  int segundos ;
}
@end
```

Hecho esto, guardamos los cambios, compilamos y probamos la aplicación.

Modifiquemos ahora esta aplicación para agregarle un botón que nos permita detener o reanudar la marcha del cronómetro. El titulo del botón deberá cambiar para indicar la acción que realizara al darle un clic. La imagen 28 muestra el nuevo aspecto de la interfaz gráfica.

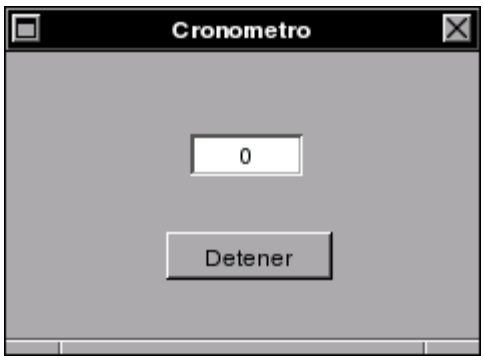


Imagen 28. Nueva interfaz del cronómetro.

Hecho este cambio, en la pestaña Objects seleccionamos nuestro objeto *Cronometro* y lo eliminamos

seleccionando **Delete** en la opción **Edit** del menú de GORM. Seleccionamos ahora nuestra clase *Cronometro* en la pestaña Classes, y le agregamos un Action llamado **boton**, el cual conectaremos con el botón de la interfaz. Hecho esto, instanciamos nuestra clase, y realizamos la conexión del Action **boton** y, nuevamente, la del Outlet **tiempo**. Por último, generamos los archivos de nuestra clase *Cronometro*, guardamos los cambios, y regresamos a Project Center.

En Project Center, primero eliminamos el archivo Cronometro.m. Para ello lo seleccionamos y elegimos la opción **Remove Files...** de la opción **Project** del menú de Project Center. Esto abrirá un cuadro de dialogo donde elegimos la opción **...from Project and** para eliminar completamente el archivo. El mismo procedimiento se realiza para eliminar el archivo Cronometro.h. Eliminados estos archivos, agregamos el nuevo archivo Cronometro.m a nuestro proyecto. Este nuevo archivo tiene ahora un método llamado **boton:** que es el que se ejecutara cuando el usuario de un clic en el botón. A este nuevo archivo le agregamos el mismo código que al archivo anterior, con algunas modificaciones, más el código del método **boton:**. El contenido de este archivo queda entonces de la siguiente forma

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "Cronometro.h"

@implementation Cronometro

-(id)init
{
   self = [super init] ;
   NSTimer *reloj ;
   segundos = 0 ;
   estado = 0 ;

   reloj = [NSTimer scheduledTimerWithTimeInterval:1
   target:self
   selector:@selector(segundero:)
   userInfo:nil
   repeats:YES] ;

   return self ;
}

- (void) segundero: (NSTimer *) timer
{
 if (estado == 0)
```

```
 {
 segundos = segundos + 1 ;
 [tiempo setIntValue: segundos] ;
 }
}

- (void) boton: (id)sender
{
 /* insert your code here */

 if (estado == 0)
 { [sender setTag: 1] ;
   [sender setTitle: @"Reanudar"] ; }
 else
 { [sender setTag: 0] ;
   [sender setTitle: @"Detener"] ; }

 estado = [sender tag] ;
}


@end
```

La primera diferencia se da en la re-definición del método **init**, ya que después de asignarle el valor de 0 a la variable **segundos**, a la variable **estado** también se le asigna el valor de 0. La segunda diferencia se da en el método **segundero:**, donde ahora hay una sentencia condicional **if** que verifica el valor de la variable **estado**. Si este valor es 0, se procede a incrementar la variable **segundos** y comunicar este incremento a la caja de texto. Y si el valor es diferente de 0 no se hace nada. Por último, se encuentra el método **boton:** cuyo funcionamiento es similar al del método utilizado en la sección 4.3. Si el valor de la variable **estado** es igual a 0, entonces a la variable **tag** se le establece el valor 1, y se cambia el titulo del botón a "Reanudar". Si el valor es igual a 1, la variable **tag** se establece a 0, y se cambia el titulo del botón a "Detener". Y la última línea le asigna a la variable **estado** el valor actual de la variable **tag**[10]. Por último, no debemos olvidar agregar la variable **estado** (de tipo **int**) como atributo de nuestro objeto en el archivo Cronometro.h. La figura 9 muestra el modelo de esta aplicación.

10 Debido a que el parámetro **sender** pertenece al método **boton:**, este sólo puede ser utilizado en dicho método. Es decir, la actualización de la variable **estado** mediante el código *estado = [sender tag]* no hubiera podido llevarse a cabo en otro método.

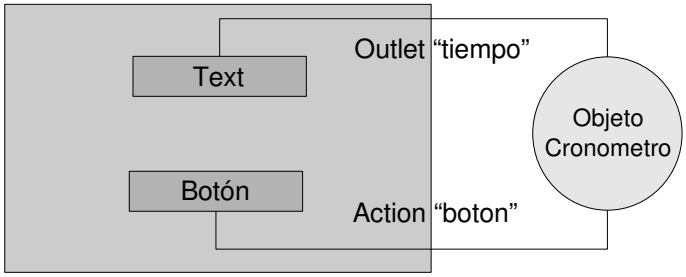


Figura 9. Nuevo modelo de nuestra aplicación.

Guardados los cambios, compilamos y probamos la aplicación.

Hagamos una última modificación a esta aplicación, agregándole un aguja que muestre el avance de los segundos. Para ello, agreguemos un componente **CustomView** a la interfaz gráfica. La imagen 29 muestra como debe quedar nuestra interfaz vista en GORM (al correr nuestra aplicación, el componente **CustomView** debe mostrar el avance de la aguja).

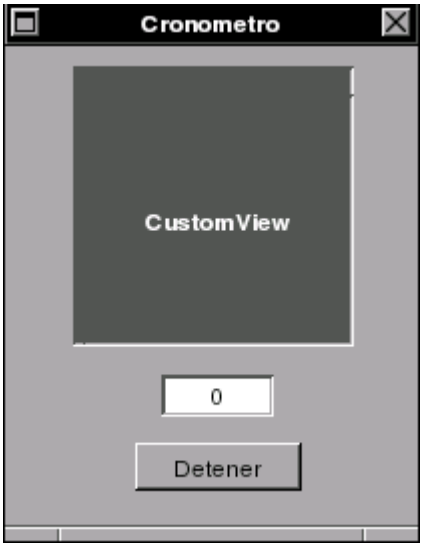


Imagen 29. Nueva interfaz del cronometro.

El componete CustomView deriva de la clase NSView, la cual nos provee facilidades para dibujar en una ventana. Sin embargo, como vamos a necesitar realizar algunas modificaciones a esta clase, primero creemos una clase llamada *Aguja* que derive de la clase NSView. Para ello, buscamos la clase NSResponder y dentro de esta la clase NSView. Creada nuestra clase *Aguja*, le agregamos un Action (en la pestaña Actions) llamado **avanzar:**. Entonces seleccionamos el componente CustomView de

nuestra interfaz y en el inspector seleccionamos la opción Custom Class de la lista desplegable. Veremos una lista de clases que podemos elegir para el componente CustomView. Seleccionemos nuestra clase *Aguja*, con lo que el nombre del componente debe cambiar a *Aguja*.

Hecho esto, eliminamos el objeto Cronometro, y a la clase *Cronometro* le añadimos un nuevo Outlet llamado **tiempoAguja**. Entonces instanciamos la clase *Cronometro*, y realizamos las conexiones del Outlet **tiempo** y el Action **boton** como antes[11]. El nuevo Outlet, **tiempoAguja**, lo conectamos al componente CustomView de nuestra interfaz (ahora con el titulo *Aguja*). Por último, creamos los archivos de nuestras clases *Cronometro* y *Aguja*, guardamos los cambios y regresamos a Project Center.

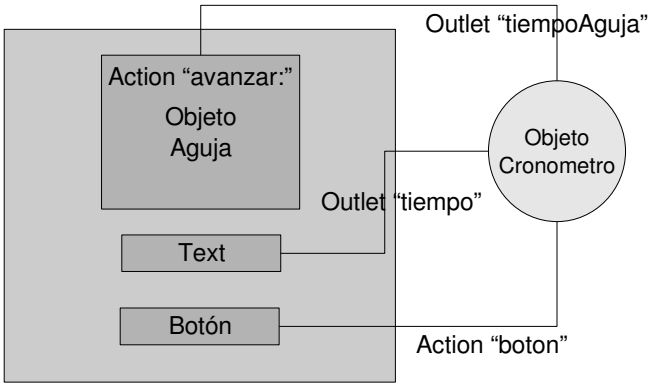


Figura 10. Modelo final del cronometro.

La razón de conectar el objeto Cronometro con el objeto Aguja, es para utilizar el mismo NSTimer del objeto Cronometro. Pudo haberse proveído al objeto Aguja de su propio NSTimer y conectarlo directamente con el botón. De esta forma, los dos NSTimer marcharían a la par, aunque no estarían necesariamente sincronizados. Sin embargo, aquí implementaremos el modelo mostrado en la figura 10.

Estando en Project Center eliminamos los archivos Cronometro.h y Cronometro.m del proyecto, y agregamos el nuevo archivo Cronometro.m y el archivo Aguja.m. La única diferencia en el archivo Cronometro.m ocurre en el método **segundero:** el cual queda de la siguiente forma

```
- (void) segundero: (NSTimer *) timer
{
  if (estado == 0)
  {
  segundos = segundos + 1 ;
  [tiempo setIntValue: segundos] ;
```

11 Adviertase que no es necesario instanciar el objeto *Aguja*, puesto que este actuara a través del componente CustomView.

```
  [tiempoAguja avanzar: segundos] ;
  }
}
```

Donde al final de la sentencia condicional se envía un mensaje por la conexión **tiempoAguja** para ejecutar el método **avanzar:** pasandole como parámetro la variable **segundos**. Guardamos los cambios de este archivo y pasamos al archivo Aguja.m. El cual debe quedar de la siguiente forma

```
/* All Rights reserved */
#include <AppKit/AppKit.h>
#include "Aguja.h"

@implementation Aguja

-(id)initWithFrame: (NSRect) frame
{
 self = [super initWithFrame: frame] ;
 origen = NSMakePoint(frame.size.width/2, frame.size.height/2) ;
 punta = NSMakePoint(0,60) ;
 return self ;
}

- (void) drawRect: (NSRect) frame
{
   NSBezierPath *segundera = [NSBezierPath bezierPath];
   [segundera setLineWidth: 1];
   [[NSColor blackColor] set] ;
   [segundera moveToPoint: origen];
   [segundera relativeLineToPoint: punta];
   [segundera stroke];
}

- (void) avanzar: (int)seg
{
  /* insert your code here */
  float segundo_x = 60*sin(M_PI*seg/30);
  float segundo_y = 60*cos(M_PI*seg/30);
  punta = NSMakePoint(segundo_x, segundo_y);
  [self setNeedsDisplay: YES] ;
}
```

```
@end
```

Primero se encuentra la re-definición del método de máxima genericidad initWithFrame:. Este método recibe un objeto NSRect al momento de ser llamado, en este caso llamamos a este objeto **frame**. Este **frame** es el área rectangular de que disponemos para poder dibujar. Mediante los receptores **self** y **super**, ejecutamos el método initWithFrame: de la superclase. El cual también requiere un parámetro de tipo NSRect. Es por ello que se le envía el mismo objeto que recibe nuestro método al ser llamado, es decir el objeto **frame**. Los objetos **origen** y **punta** son de tipo NSPoint, y deben ser declarados como atributos en el archivo Aguja.h. Estos puntos representan, respectivamente, el origen y la punta de la aguja segundera. La segunda línea le asigna al punto **origen** la coordenada del centro del **frame**. Para ello se accede al alto (height) y al ancho (width) del **frame** y luego se divide el dato correspondiente en dos (recuérdese la sección 4.4). Seguidamente a **punta** se le asigna la coordenada (0,60). Y, por último, se retorna el objeto creado.

El segundo método, drawRect:, se encarga de dibujar el contenido del objeto NSRect que reciba como parámetro. En este caso del objeto **frame**. Este método es llamado automáticamente, después de initWithFrame:. La primera linea en este método crea un objeto NSBezierPath llamado **segundera**, este objeto se crea mediante el constructor conveniente bezierPath. Los objetos NSBezierPath nos permiten construir una matriz para almacenar puntos, para luego dibujar una imagen a través de esos puntos (como los dibujos para niños donde se deben unir los puntos con líneas para obtener el dibujo). La siguiente línea, ejecuta el método setLineWidth de nuestro objeto **segundera**, este método asigna el ancho de la linea que se utilizara para unir los puntos, en este caso se le pasa el valor 1. En la siguiente línea el mensaje [NSColot blackColor] ejecuta el constructor conveniente blackColor de la clase NSColor, este devuelve un objeto que representa el color negro. Seguidamente se ejecuta el método **set** de este objeto, lo que hace que el color negro sea el utilizado por cualquier método de dibujo. En la línea que sigue, se utiliza el método moveToPoint para asignar el punto **origen** (el centro del **frame**) como el primer punto de nuestro objeto (o dibujo) **segundera**. En la siguiente línea se agrega el punto **punta**, utilizando el método relativeLineToPoint, que hace que las coordenadas sean relativas al punto anterior (el punto **origen**). Esto hace que la aguja este inicialmente apuntando hacia arriba (hacia las doce en nuestro reloj) y que tenga una longitud de 60 pixeles. El último mensaje ejecuta el método **stroke** de nuestro objeto **segundera**. Este método dibuja el contorno de nuestro dibujo, en este caso la línea que une los dos puntos (otro método disponible, si el dibujo es cerrado, es **fill**, que se utiliza para rellenar el dibujo).

Es la coordenada del punto **punta** el que se modificara para hacer que la aguja avance. De esto se encarga el método **avanzar:**, este método recibe como parámetro un dato de tipo **int** que, en este caso, llamamos **seg**. Y el cual, si recordamos el método **segundera:** en el archivo Cronometro.m, es el número de segundos transcurridos[12]. Las primeras dos líneas de este método establecen dos variables de

12 GORM crea este método asignándole un parámetro de tipo **id** llamado **sender**. Sin embargo, nosotros necesitamos pasarle un parámetro de tipo **int** (recuérdese el método **segundera:** en el archivo Cronometro.m), por lo que debemos modificar el tipo de parámetro recibido, tal y como se muestra en el código.

tipo **float** llamadas **segundo_x** y **segundo_y** que son, respectivamente, las coordenadas **x** y **y** de la punta de la aguja segundera. Las operaciones realizadas en estas líneas son fáciles de entender si se recuerda que la aguja mide 60 pixeles de longitud y que, como una vuelta tiene 60 segundos, un segundo representa 6 grados sexagesimales. **M_PI** es una constante que contiene el valor de Pi, necesario para convertir el angulo a radianes que es el utilizado por las funciones **sin()** y **cos()**. La tercera línea le asigna a **punta** las coordenadas calculadas que, recordemos, son relativas al centro del **frame**, puesto que se utiliza el método relativeLineToPoint: para asignar este punto al objeto **segundera** (usamos, por comodidad, coordenadas relativas para simplificar los cálculos).

El último mensaje **[self setNeedsDisplay: YES]** le dice al objeto Aguja (**self**) que ejecute el método setNeedsDisplay: con el parámetro YES. Esto hace que el método drawRect: se vuelva a ejecutar, haciendo que el **frame** de nuestro objeto Aguja se vuelva a dibujar tomado en cuenta la nueva coordenada de **punta**. Si llamáramos directamente al método drawRect: tendríamos que crear un objeto NSRect para pasárselo como parámetro. Sin embargo, el método setNeedsDisplay: nos evita hacer esto.

Sólo nos falta escribir el código del archivo Aguja.h, el cual debe quedar de la siguiente forma

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include <math.h>

@interface Aguja : NSView
{
NSPoint origen, punta ;
}
- (void) avanzar: (int)seg;
@end
```

Observese que debe incluirse la librería math.h la cual nos permite usar las funciones **sin()** y **cos()**, así como la constante **M_PI**. Además, debemos declarar como atributos los objetos **origen** y **punta**, y modificar el parámetro recibido por el método **avanzar:**. Hecho todo lo anterior, guardamos los cambios, compilamos y probamos la aplicación. La imagen 30 muestra la aplicación final en ejecución.

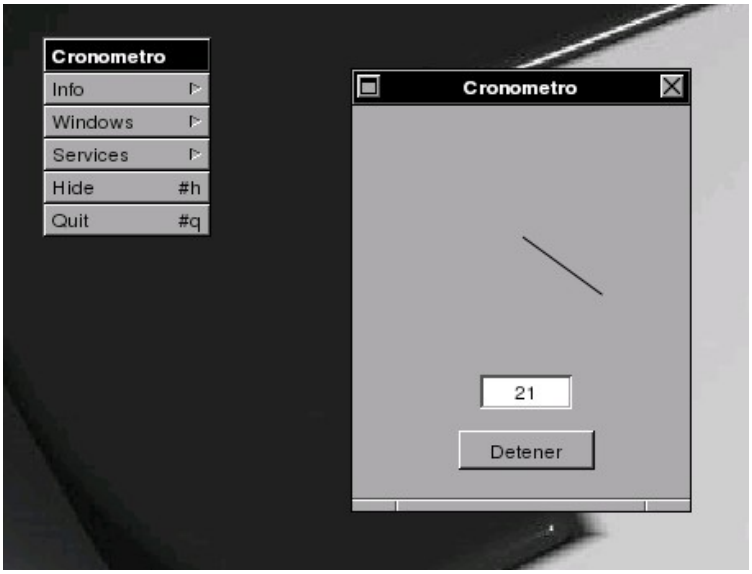


Imagen 30. Aplicación Cronómetro junto a su menú.

En la imagen 31 se muestra la misma aplicación con una imagen añadida como fondo del reloj. La imagen se agrega primero a GORM, seleccionando la opción **Load Image** en la opción **Document** del menú de GORM. Seguidamente se agrega a la interfaz un componente NSImageView desde la paleta *Data Palette* y en el inspector, en el campo con el titulo **Icon**, se escribe el nombre de la imagen que se agrego (sin extensión). Seleccionado el componente NSImageView, se elige la opción **Send To Back** en la opción **Layout** del menú de GORM, para hacer que la imagen quede detrás del objeto Aguja y no encima.

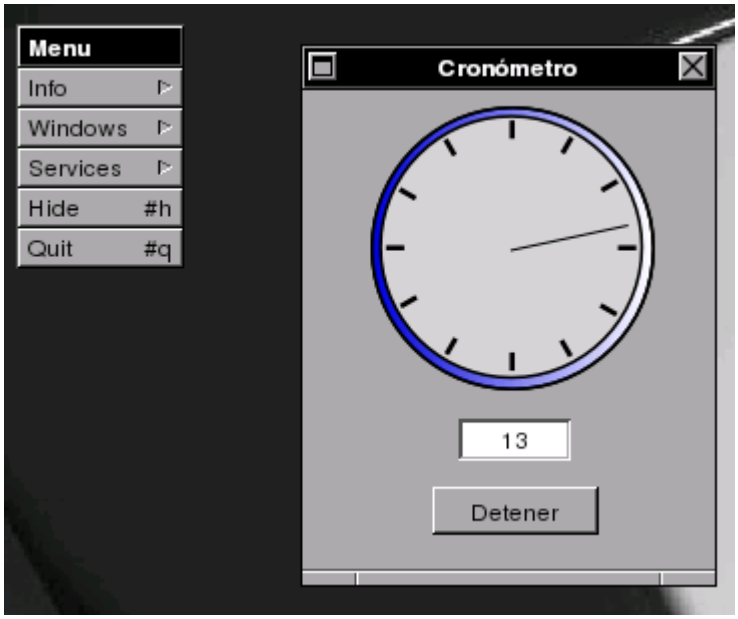


Imagen 31. Aplicación Cronómetro con imagen de fondo.

7.2 Un simple editor gráfico

La imagen 32 muestra la ventana de la aplicación. Esta tiene tres botones, un componente NSColorWell y un componente CustomView.

Imagen 32. Ventana del editor.

Vamos ahora a crear el menú de nuestra aplicación. Sin embargo, primero debemos borrar el menú que se crea automáticamente. Para ello, lo seleccionamos en la ventana **main panel** en la sección **Objects** y luego elegimos **Delete** en la opción **Edit** del menú de GORM. Ahora de la paleta **Menus** arrastramos un nuevo menú (el pequeño icono en la parte inferior) a la ventana **main panel**. Este nuevo menú tiene solamente dos opciones o **items**, Hide y Quit. Seleccionando el item Hide en dicho menú, le cambiamos el titulo a *Ocultar* en el Inspector. De forma similar, al item Quit le cambiamos el nombre a *Salir*. Hecho esto, vamos a agregar un nuevo item a nuestro menú. Este item sera el correspondiente a la información de nuestra aplicación. Para agregarlo, de la paleta **Menus** arrastramos el item con el titulo *item* a nuestro menú y lo colocamos arriba del item *Ocultar*. Luego le cambiamos el nombre a *Información*. La siguiente imagen muestra como debe quedar el menú.

Imagen 33. El menú de nuestro editor.

Para que este nuevo item presente la información de nuestra aplicación, debemos conectarlo con el método correspondiente. Para ello seleccionamos el item *Información* y, manteniendo presionada la tecla Ctrl, lo arrastramos al objeto NSFirst en el **main panel**. El objeto NSFirst es el encargado de gestionar el funcionamiento de nuestra aplicación. Ahora en el **Inspector**, en la sección **connections**, seleccionamos *target* y buscamos el método *orderFrontStandarInfo*, que es el encargado de crear la ventana de información. Seleccionado este método damos un clic en la opción *Connec*t, para establecer la conexión. Y con esto tenemos terminada la creación de nuestro menú (la información la agregamos más adelante).

Crearemos ahora la clase para nuestro objeto CustomView. Primero, en el **main panel** seleccionamos la opción **Classes**, y creamos una subclase de la clase NSView a la que llamaremos MILienzo. Hecho esto, le agregamos tres Outlets a nuestra clase llamados: *linea*, *ovalo* y *rectangulo* y un Action llamado *color*. Creados estos Outlets y el Action, seleccionamos el objeto CustomView de la ventana de nuestra aplicación, y en el **Inspector**, en la sección **Custom Class**, le establecemos como clase nuestra clase MILienzo. Ahora realizamos las conexiones de los Outlets *linea*, *ovalo* y *rectangulo*, los cuales debemos conectar con los tres botones, y por último, el Action *color* lo conectamos al componente NSColorWell.

Hecho todo esto, creamos los archivos MILienzo.h y MILienzo.m de nuestra clase, guardamos los cambios y cerramos GORM.

Estando ahora en Project Center, modificamos el contenido del archivo MILienzo.m, el cual debe quedar de la siguiente forma,

```
/* All Rights reserved */

#include <AppKit/AppKit.h>
#include "MILienzo.h"

@implementation MILienzo

- (void) linea: (id)sender
{
  /* insert your code here */
  vboton = 1 ;
  vColor = [color color] ;
}


- (void) ovalo: (id)sender
{
  /* insert your code here */
  vboton = 2 ;
  vColor = [color color] ;
}
```

```
- (void) rectangulo: (id)sender
{
  /* insert your code here */
  vboton = 3 ;
  vColor = [color color] ;
}


- (void)mouseDown: (NSEvent *) evento
{
    locA = [self convertPoint:[evento locationInWindow] fromView:nil];
    BOOL clic = YES;

    while (clic)
      {
       evento = [[self window] nextEventMatchingMask: NSLeftMouseUpMask | NSLeftMouseDraggedMask];

      switch ([evento type])
        {
         case NSLeftMouseDragged:
           {
             locB = [self convertPoint:[evento locationInWindow] fromView:nil];
             NSSize dimensiones =  NSMakeSize(locB.x - locA.x, locB.y - locA.y) ;
            rectSeleccionado.origin = locA ;
            rectSeleccionado.size = dimensiones;
            seleccion = YES;

               [self setNeedsDisplay: YES] ;
           }
            break;

         case NSLeftMouseUp:
           {
             clic = NO;
           }
            break;

         }
      }
}


- (void)drawRect: (NSRect) rect
{
   if (seleccion)
      {
        [vColor set] ;

        switch(vboton)
```

```
        {
        case 1:
          {
          ruta = [NSBezierPath bezierPath] ;
          [ruta moveToPoint: locA] ;
          [ruta lineToPoint: locB] ;
          [ruta stroke] ;
          }
          break ;

       case 2:
          {
          ruta = [NSBezierPath bezierPathWithOvalInRect: rectSeleccionado] ;
          [ruta fill] ;
          }
          break ;
      case 3:
          {
          ruta = [NSBezierPath bezierPathWithRect: rectSeleccionado] ;
          [ruta fill] ;
          }
          break ;

    }

    }
}


@end
```

Los métodos *linea*, *ovalo* y *rectangulo*, lo único que hacen es establecer el valor de la variable *vboton* a 1, 2 o 3 respectivamente para, más adelante, saber que es lo debe dibujarse. Es decir, si *vboton* vale 1, entonces debe dibujarse una linea, si vale 2 debe dibujarse un ovalo, etc. Y mediante la conexión **color** (la conexión hacia el componente NSColorWell) ejecutar el método *color* el cual devuelve el color seleccionado en el componente. El color devuelto es almacenado entonces en la variable *vColor*.

El siguiente método, *mouseDown*, es mucho más complicado y se comenta aquí brevemente. Pero se anima al lector a revisarlo cuidadosamente para entender su funcionamiento. Incluso a cambiar algunas cosas para entender "que hace que". La primera línea, le asigna al punto *locA* la coordenada en el NSView donde ocurre la acción de hacer clic izquierdo. Es decir, la coordenada desde la cual se comienza a dibujar la línea, ovalo o rectángulo. La siguiente línea, declara la variable booleana[13] *clic* a YES. Esta variable nos permite controlar la sentencia iterativa que viene a continuación. La sentencia

13 Las variables booleanas son variables lógicas que pueden tomar el valor YES o NO, y se declaran de la misma forma que las variables **int** y **float**, haciendo uso de la palabra reservada BOOL. Al ser variables lógicas, no necesitan ser comparadas. Es decir, seria un error escribir algo como:

while (clic == YES)

while, la cual se ejecuta si la variable *clic* es verdadera (es decir, si su valor es YES), evaluá primero el siguiente evento del mouse que ocurra. Como solo nos interesan los eventos de mover o arrastrar el mouse (con el botón izquierdo presionado) y el de soltar el botón izquierdo, al método nextEventMatchingMask se le indica que detecte únicamente los eventos NSLeftMouseUpMask y NSLeftMouseDraggedMask. La barra, |, indica que detecte cualquiera de los dos eventos[14]. Seguidamente viene una sentencia *switch* que evaluá el tipo de evento que ocurre.

Si el mouse es arrastrado, es decir si el tipo del eventos es NSLeftMouseDragged, entonces primero se almacena en el punto *locB* la nueva ubicación del mouse en el NSView. Seguidamente se crea un objeto NSSize con las dimensiones del rectángulo que tiene por esquinas opuestas los puntos *locA* y *locB*. Seguidamente al rectángulo *rectSeleccionado* (este es el que se utilizara para dibujar las figuras) se le establece como origen el punto *locA* y como dimensiones, las dimensiones del objeto *dimensiones*. Se establece la variable booleana *seleccion* a YES (esta variable sirve más adelante) y, por último, se vuelve a dibujar el frame del NSView.

En el caso de que el tipo de evento sea NSLeftMouseUp, la variable *clic* se establece a NO, lo que permite salir de la sentencia *while*.

El último método, *drawRect*, revisa primero si la variable *seleccion* es verdadera, si es así, procede a dibujar la línea, el ovalo o el rectángulo. Primero establece como color de dibujo, el color seleccionado en el componente NSColorWell y luego procede a dibujar la figura seleccionada.

Por último, el archivo MILienzo.h debe modificarse para declarar las variables a utilizar. Debe quedar de la siguiente forma,

```
/* All Rights reserved */

#include <AppKit/AppKit.h>

@interface MILienzo : NSView
{
  id color;
  int vboton ;
  NSColor *vColor ;
  NSRect rectSeleccionado ;
  NSPoint locA, locB ;
  BOOL seleccion ;
  NSBezierPath *ruta ;
}
- (void) linea: (id)sender;
```

14 La barra | simplemente sirve para separar los eventos que nos interesan, puede decirse que es como una coma. Y no tiene ninguna relación con el operador lógico || (OR).

```
- (void) ovalo: (id)sender;
- (void) rectangulo: (id)sender;
@end
```

Aquí, solamente se han agregado seis lineas para declarar las variables necesarias. Guardados todos los cambias, compilamos y probamos la aplicación. La imagen 34 muestra la aplicación en acción.

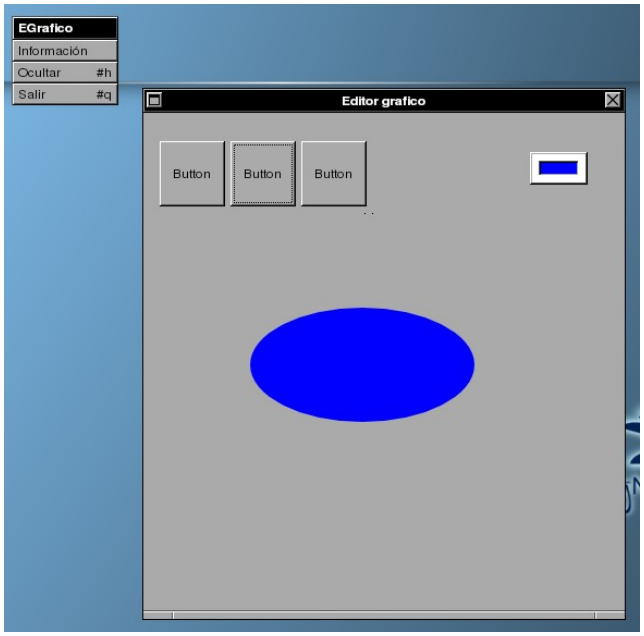


Imagen 34. El editor en acción.

Como se ve, este editor no guarda los dibujos realizados previamente. Solamente el último dibujo realizado es visible.

Ahora vamos a realizar algunos últimos cambios a la interfaz de nuestra aplicación. Como hemos visto al ejecutar nuestra aplicación, no es posible distinguir el borde del objeto NSView, por lo que vamos a ponerle un marco consistente en un objeto NSBox. Para ello, en GORM, seleccionamos el NSView y en la opción **Edit** del menú de GORM, seleccionamos la opción **Group** y, dentro de esta, la opción **In Box**. Hecho esto, seleccionamos el objeto NSBox agregado, y le cambiamos el titulo en el **Inspector**.

Imagen 35. NSView dentro de un NSBox.

Ahora, para los botones, necesitamos tres imágenes, estas hay que crearlas mediante algún editor gráfico. En mi caso he creado tres imágenes de 38x38 pixeles en Gimp. El resultado final se ve en la imagen 36. Por último, para agregar la información de nuestra aplicación, con el proyecto abierto en Project Center, seleccionamos el icono con un destornillador y una llave. Esto abre la ventana del inspector del proyecto. En la lista desplegable elegimos la opción **Project Description**, introducimos los datos correspondientes, cerramos el inspector, guardamos los cambios, compilamos y probamos nuestra aplicación.

Dando un clic en la opción **Información** del menú de nuestra aplicación, podemos ver como quedo el panel de información (imagen 37).

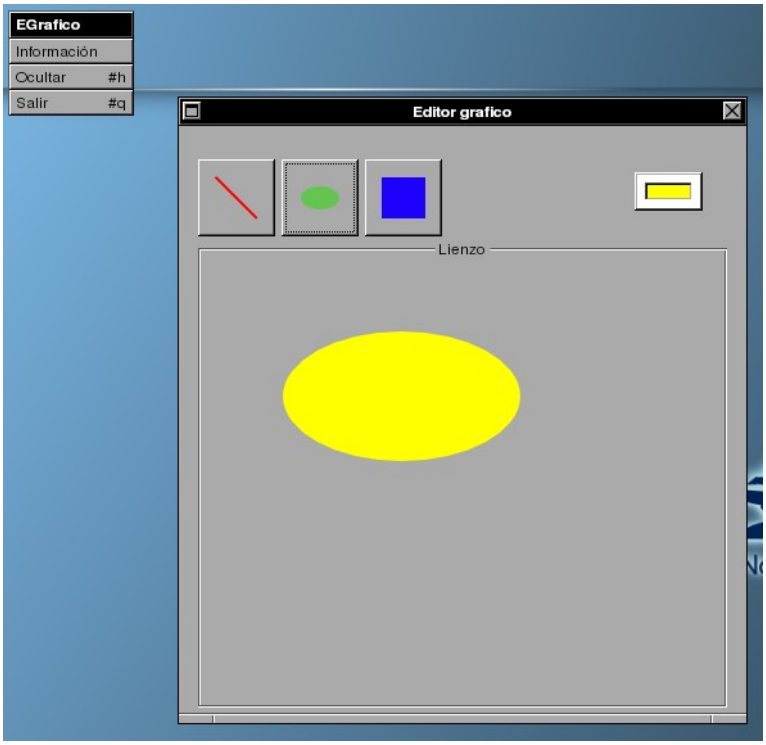


Imagen 36. Nuestra aplicación final en acción.

Imagen 37. El panel de información de nuestra aplicación.

Como ejercicio adicional, el lector puede agregar el método *awakeFromNib* al archivo MILienzo.m para establecer la opción de dibujar líneas desde el inicio. El código a añadir seria el siguiente,

```
-(void) awakeFromNib
{
vboton = 1 ;
}
```

También puede agregar etiquetas de ayuda (Tool Tips) a los botones, de tal forma que al momento de que el usuario coloque el mouse sobre alguno de los botones, aparezca un mensaje con la descripción de lo que hace el botón. Para hacer esto en GORM, simplemente se selecciona el botón al que se le va a agregar el Tool Tip, luego se selecciona en el **Inspector** la opción **Help**, y en el campo con el titulo **Tool Tips** se escribe la descripción de lo que realiza el botón.

Capítulo 8

Apariencia y comportamiento de las aplicaciones

Este capítulo presenta, en forma general, como podemos modificar la apariencia y el comportamiento de nuestras aplicaciones. Estas modificaciones se limitan al No se presentan todas las formas posibles de modificar

8.1 Modificaciones en GORM

Nos limitaremos a las modificaciones que podemos hacer a las ventanas de nuestra aplicación. Por defecto, GORM crea las ventanas con tres controles: un botón para minimizar (Miniaturize), un botón para cerrar (Close) y la barra para redimensionar la ventana (Resize bar). Sin embargo, en el Inspector podemos desactivar los controles que no necesitemos o que no queramos. Al desactivar alguno de estos controles, el cambio no es visible en GORM. Este es apreciable hasta que la aplicación esta ejecutándose.

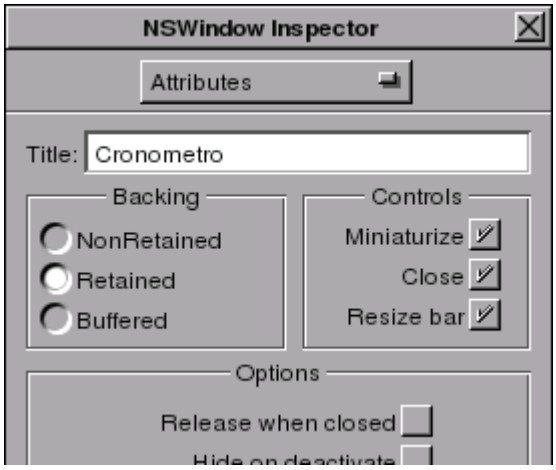


Imagen 38. Miniaturize, Close y Resize bar activados.

8.2 Configuración de las librerías GUI y Back

Existe un conjunto de variables que nos permiten configurar el comportamiento de las librerías GUI y Back. El valor de estas variables se puede modificar mediante la aplicación System Preferences en la sección **Defaults**, o modificando directamente el archivo oculto .GNUstepDefaults que se encuentra en la carpeta Defaults dentro de la carpeta GNUstep. Modificando el valor de estas variables mediante la aplicación System Preferences y viendo luego los cambios en el archivo .GNUstepDefaults, se puede aprender como escribir directamente las modificaciones en el archivo.

Imagen 39. Aplicación System Preferences.

Entre las variables que controlan el aspecto de las aplicaciones, se encuentra la variable *GSX11HandlesWindowDecoration* la cual es aplicable únicamente cuando la aplicación corre en un servidor gráfico X11. Por defecto, esta variable tiene el valor booleano NO, lo que significa que son las librerías de GNUstep las que se encargan de la decoración de las ventanas de nuestra aplicación. El valor YES hace que la decoración de las ventanas sea manejado por el servidor gráfico, lo que significa que las ventanas tendrán la decoración establecida por el tema que el usuario tenga seleccionado para estas. Esto significa también que cuando la ventana sea minimizada no aparecerá el Miniwindow, sino que la ventana sera minimizada en la barra de estado. Significa, también, que nuestra ventana tendrá un control para maximizarla. Pero, por ejemplo, si la ventana de nuestra aplicación a sido establecida para no tener un control de minimizar y una barra para redimensionar la ventana, esto sera respetado por el servidor gráfico que hará que la ventana no sea redimensionable y que no tenga un control para minimizar. Sin embargo, el aspecto del menú no se modificara en absoluto. La variable *GSSuppressAppIcon* nos permite suprimir el icono de nuestra aplicación si la establecemos a YES. Esta variable también es aplicable únicamente cuando el servidor gráfico es el X11.

Por otro lado, la variable *GSUseWMTaskBar*, aplicable únicamente en el sistema operativo Windows, esta establecida por defecto a YES. Lo que significa que la ventana de nuestra aplicación se minimizara en la barra de estado, y que ni el AppIcon ni el Miniwindow se mostraran. Pero, el menú, sera visible todo el tiempo, aun cuando la ventana este minimizada.

La variable NSInterfaceStyleDefault nos permite cambiar la forma en que el menú sera mostrado. Por defecto, esta variable tiene el valor NSNextStepInterfaceStyle, lo que hace que el menú aparezca separado de la ventana. El valor NSMacintoshInterfaceStyle hace que el menú se muestre al estilo del sistema MacOS X. También se encuentra el valor NSWindos95InterfaceStyle que permitirá (a la fecha aun no esta implementado) hacer que el menú se integre a la ventana.

Por último, en la sección **Color Schemes** de la aplicación System Preferences, podemos modificar el color de diferentes partes de nuestra aplicación (controles, texto, frames, resaltado, sombras, etc.), así como el estilo en general de nuestra aplicación (actualmente solo hay disponibles cuatro estilos). En la siguiente imagen, se presenta el editor gráfico del capitulo anterior. Se ha establecido la variable *GSX11HandlesWindowDecoration* a YES, y se ha modificado la combinación de colores.

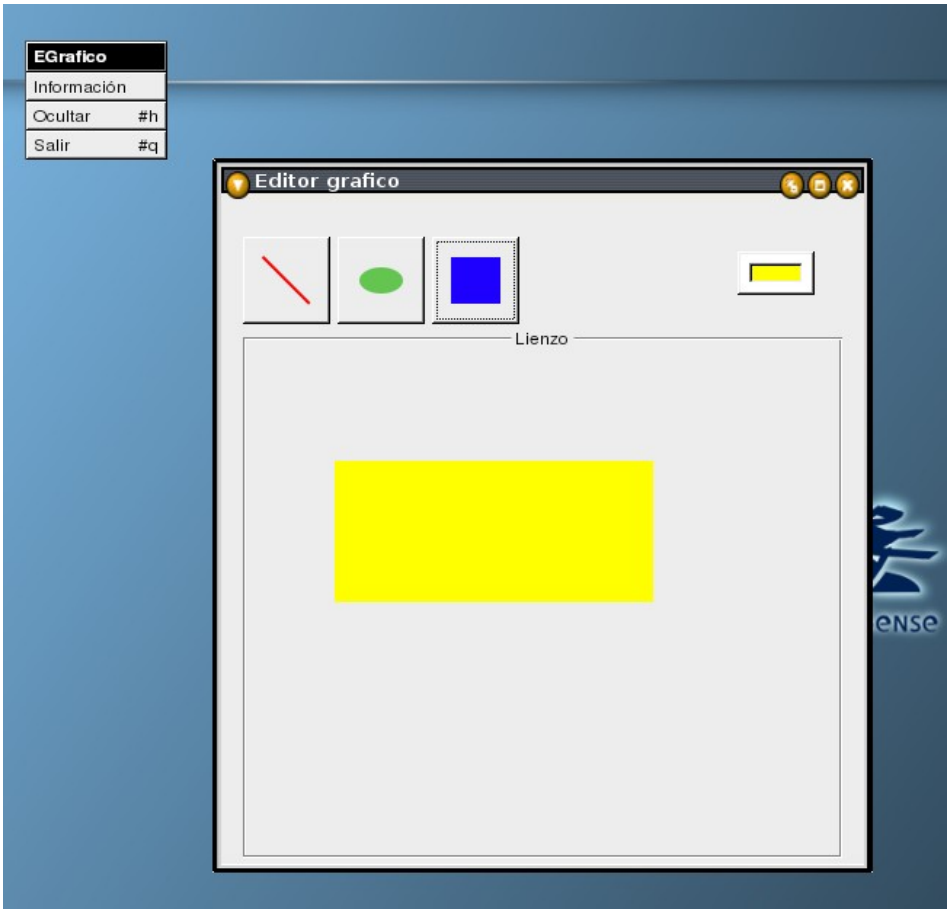


Imagen 40. El editor gráfico con su apariencia modificada.

Apéndice A

Librerías de funciones

A continuación se listan algunas de las funciones de la librería **math.h**.

Función	*Descripción*	*Librería*
acos(x)	Devuelve el coseno inverso de la variable x.	math.h
asin(x)	Devuelve el seno inverso de la variable x.	math.h
atan(x)	Devuelve la tangente inversa de la variable x.	math.h
cos(x)	Devuelve el coseno de la variable x.	math.h
sin(x)	Devuelve el seno de la variable x.	math.h
tan(x)	Devuelve la tangente de la variable x.	math.h
sqrt(x)	Devuelve la raíz cuadrada de la variable x.	math.h
fabs(x)	Devuelve el valor absoluto de la variable x,	math.h
exp(x)	Devuelve **e** elevado a la potencia x.	math.h
pow(x,y)	Devuelve x elevado a la potencia y.	math.h

Apéndice B

El sistema de documentación de GNUstep

El sistema de documentación de GNUstep contiene, además de algunos manuales en ingles, toda la información relacionada a los frameworks Base y GUI, y a las variables de configuración de GNUstep. Este sistema es un conjunto de archivos **html** para visualizar mediante un navegador web. La mayor parte de esta documentación puede ser generada a partir de los archivos instalados por el paquete **startup**, haciendo uso de la herramienta **autogsdoc**. Sin embargo, es más fácil bajarse los paquetes en la pagina de la distribución que se este utilizando. Estos paquetes son: **gnustep-core-doc**, **gnustep-make-doc**, **gnustep-base-doc**, **gnustep-gui-doc**, **gnustep-back-doc** y **gnustep-tutorial-pdf** (o en su lugar **gnustep-tutorial-html**). Instalados estos paquetes se pueden crear accesos a las páginas principales (las que llevan por nombre *index*), o a los documentos, que generalmente se encuentran dentro de las carpetas creadas en la ruta /usr/share/GNUstep/Documentation, y tener así un acceso rápido al sistema de documentación. Si se desea puede también instalarse el paquete **gnustep-icons** el cual provee de una buena cantidad de iconos para nuestras aplicaciones.