Depuración y pruebas

Índice

La depuración y las pruebas son dos procesos indispensables a la hora de comprobar y solucionar comportamientos erróneos en la ejecución de nuestras aplicaciones. En esta sesión comenzaremos tratando los dos tipos de depuración existentes en iOS: uso de la consola para mostrar información determinada de los procesos y el manejo de la herramienta de *debugging* de XCode. Después comentaremos el uso de la aplicación *Instruments*, gracias a ella podremos averiguar qué objetos no hemos controlado a la hora de gestionar su memoria así como depurar más en profundida la aplicación. Para terminar la sesión explicaremos, mediante ejemplos, la implementación de pruebas de unidad en XCode.

1. Depuración clásica: Uso de NSLog y Asserts

La depuración clásica consiste en alterar el código fuente de forma temporal añadiendo código para mostrar mensajes informativos en la consola. El comando habitual para enviar mensajes a la consola de XCode es `NSLog`. La directiva NSLog recibe como parámetro un `NSString` con *especificadores de formato*. Un especificador de formato es una cadena que será sustituida en tiempo de ejecución por el valor que se especifique. Todos los especificadores de formato empiezan con el símbolo del porcentaje (%) seguido de un carácter que indica el formato del elemento a mostrar.

Los distintos tipos de especificadores de formato son:

- **%@**: Cadena de texto, objetos.
- **%i**: Entero (integer).
- **%f**: Decimal (float).
- **%.02f**: Decimal (float) con dos decimales.
- **%ld**: Entero Long
- **%p**: Puntero, referencia de un objeto.

El más usado es `%@` ya que, además de mostrar una cadena de texto también puede mostrar la descripción de un método si este está preparado. Un ejemplo podría ser el siguiente:

```
NSLog(@"Descripcion del objeto window: %@", self.window);
```

En este caso se mostrará por la consola la información más relevante del elemento *window* como el tamaño del frame, opacidad, autoresize, etc. Existen más especificadores de formato no tan comunes que se pueden consultar en la documentación de Apple, dentro del apartado "String format Specifiers".

Un error muy común a la hora de especificar una cadena de texto en una directiva `NSLog` es indicar distinto número de especificadores de formato que parámetros tiene la cadena o equivocarse a la hora de especificar un tipo de parámetro determinado, por ejemplo, mostrar un entero cuando en realidad es una cadena. Estos errores pueden derivar en una excepción en tiempo de ejecución o simplemente mostrar un valor que no corresponda en

absoluto con lo esperado.

El uso de este tipo de depuración mediante trazas en el código es bastante usado ya que es fácil de implementar (sólo un comando) y, además, a veces es el único modo de depurar el código. Las directivas NSLog funcionan con cualquier tipo de configuración (Debug o Release) y en cualquier tipo de ejecución (en el simulador o en el dispositivo), incluso funcionan en otros dispositivos a modo de beta tester. Para poder visualizar la salida de consola de una aplicación en fase beta se debe conectar el dispositivo al XCode y ejecutarla, el log de la consola se guardará dentro de la ventana *Organizer* de XCode.

Una vez que tengamos la aplicación lista para generar el binario para distribuir en la *App Store* debermos antes eliminar todas las directivas NSLog del código para dejarla "limpia" y evitar una posible relentización de la ejecución debido a la escritura en la consola.

Otro tipo de llamada con la cual podemos depurar nuestro código por consola es el llamado **Assert**. Los *Asserts* son condiciones que se deben de cumplir para que se continue con la ejecución, deben de devolver "true". En el caso de que algún *assert* no se cumpla se producirá una excepción en fase de ejcución que detendrá la aplicación en el momento almacenando todo el "log" en el XCode para su posterior depuración.

El uso de *asserts* es manera muy buena de asegurarse que una determinada situación dentro de nuestro código cumple con las espectativas. Muchos desarrolladores dejan incluso estas directivas *assert* a la hora de distribuir la aplicación.

La directiva *assert* en Objective-C viene en forma de macro habitualmente `assert()` a la cual se le pasa una condición que será que se tenga que evaluar a verdadero (true) o falso (false). Si se evalua falso la aplicación se parará mostrando en la consola una explicación y si se evalua a verdadero la aplicación continuará su ejecución. Los *asserts* son usados frecuementes en APIs así como en códigos de testeo.

Los *asserts* también se pueden programar mediante la clase de Objective-C `NSAssert2` de más algo nivel que la macro comentada anteriormente. Mediante esta clase podremos especificar de una manera más clara y descriptiva el tipo de condición que no se cumple (en el caso de que esta no se cumpla). A continuación podemos ver una breve comparativa de los dos tipos de assert:

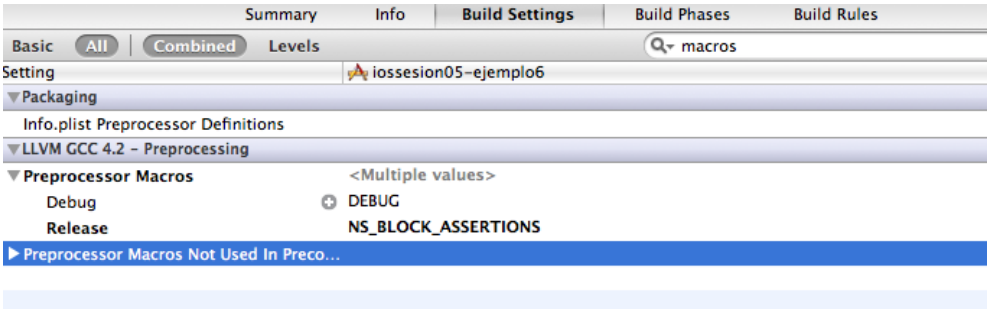
```
// Uso de la macro assert()
assert(valor < maximoValor && @"El valor es demasiado grande!");

// Uso de NSAssert2
NSAssert2(valor < maximoValor, @"El valor %i es demasiado grande
(max.:%i)!", valor,
maximoValor);
```

Como podemos ver, el segundo tipo es más descriptivo que el primero y, por tanto, más adecuado. Internamente `NSAssert2` usa la macro `assert()`.

Por último, para terminar, queda comentar como desactivar todos los asserts que hayamos

escrito en el código mediante `NSSAssert2`. Para desactivar todos los *asserts*, por ejemplo para compilar el proyecto para distribución, deberemos declarar la directiva `NS_BLOCK_ASSERTIONS` dentro de las macros del prepocesador en la configuración de "release" como se muestra en la imagen siguiente.

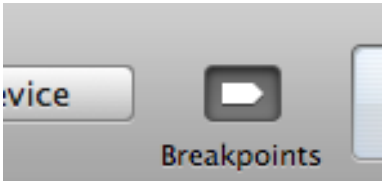


Macros del preprocesador

2. Usando el depurador de XCode

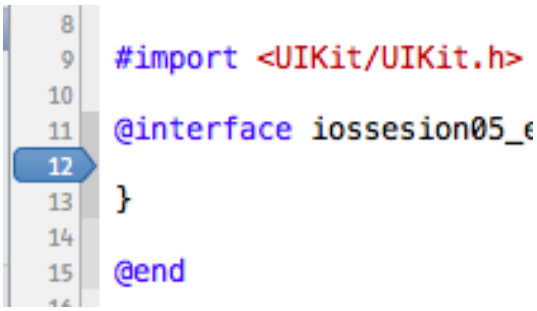
Después de comentar el método básico de depuración usando la consola de XCode pasamos a explicar el funcionamiento del depurador que viene integrado en el propio XCode y que no será de mucha utilidad en determinadas ocasiones. Cuando se está ejecutando la aplicación en modo "debug" podremos pararla y observar el estado en ese momento, como cualquier depurador de código que conozcamos.

Para activar el seguimiento de los *breakpoints* y que el depurador se detenga en ellos deberemos arrancarla con la opción de depuración activada, esto se hace pulsando sobre el botón en forma de flecha que se encuentra en la parte superior de la interfaz de XCode.



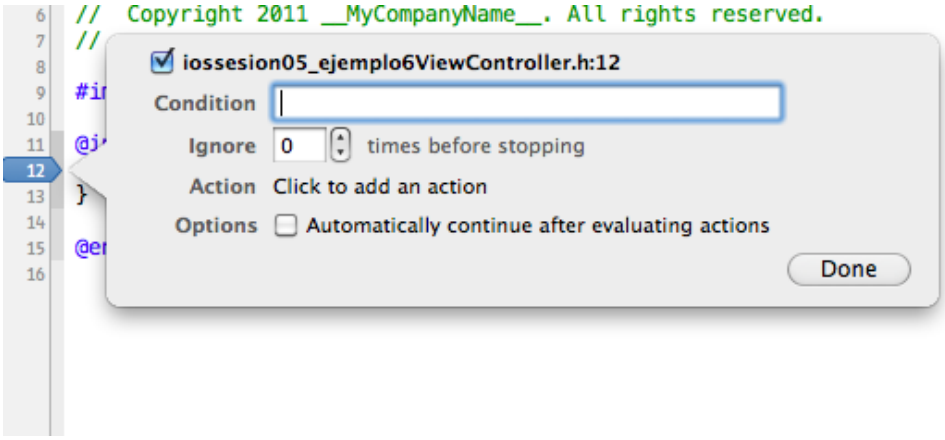
Botón Breakpoints en XCode

Para crear un punto de parada o *breakpoint* seleccionaremos en el editor la línea de código en donde queramos pausar la ejecución y pulsamos sobre *Product > Debug > Add Breakpoint at Current Line*, también podremos crearlo de una forma más rápida simplemente pulsando en la parte izquierda de la línea de código. El *breakpoint* se indica con una flecha de color azul. Si este está desactivado se mostrará semitransparente.



Breakpoint activado

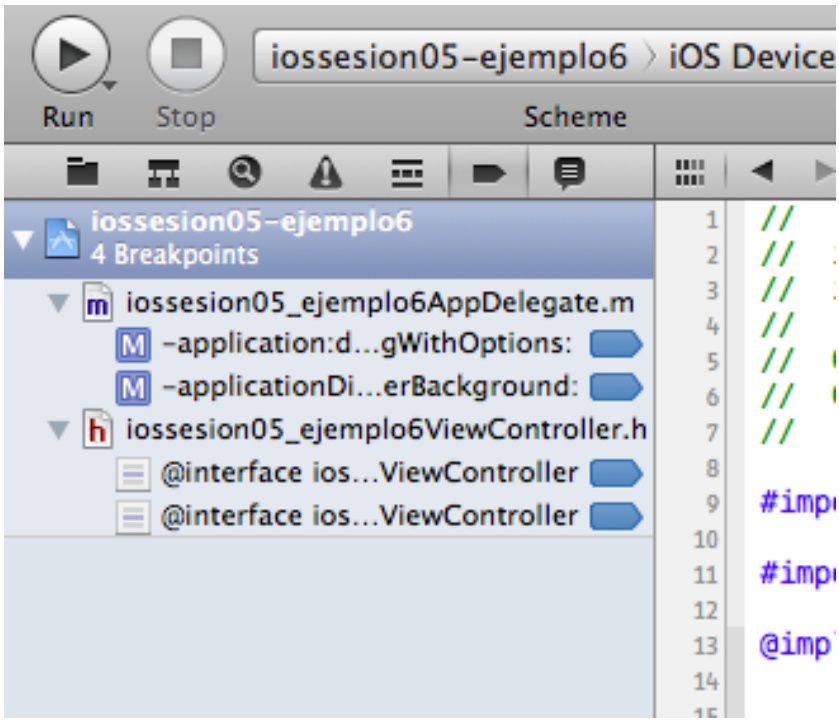
Los *breakpoints* se pueden desactivar o borrar, en estos casos el depurador no se detendrá en esa línea, de esta forma en el caso de que no queramos utilizar un *breakpoint* en un momento dado pero que más adelante puede que si que queramos, lo dejaremos como desactivado pero sin borrarlo. También podremos editar los *breakpoints* para, por ejemplo, indicar que sólo se pare la ejecución cuando se cumple una determinada condición o si sólo ha pasado por el punto un determinado número de veces. Para acceder a estas opciones deberemos hacer *ctrl + click* sobre el punto de parada y seleccionar *Edit breakpoint*. Los *breakpoints* se pueden activar y desactivar pulsando sobre *Product > Debug > Activate/Deactivate Breakpoints*.



Condición en los breakpoints

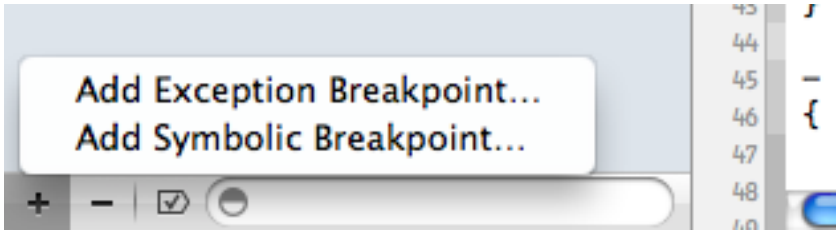
Otra de las características que tienen los puntos de parada en XCode es que estos podemos configurarlos para que no se detenga la ejecución cuando se pasa por ellos. Esto es una alternativa a la depuración clásica comentada en el punto anterior, evitaremos llenar el código con `NSLogs` que después deberemos controlar.

El entorno de XCode 4 dispone además de un navegador de *breakpoints*, al cual se puede acceder desde la pestaña en forma de flecha con color negro que se encuentra en la parte superior de la columna de la izquierda. En este navegador veremos de un golpe de vista todos los breakpoints que tiene nuestro proyecto separados por ficheros. En la parte inferior hay dos botones, uno para añadir nuevos breakpoints y otro para eliminarlos.



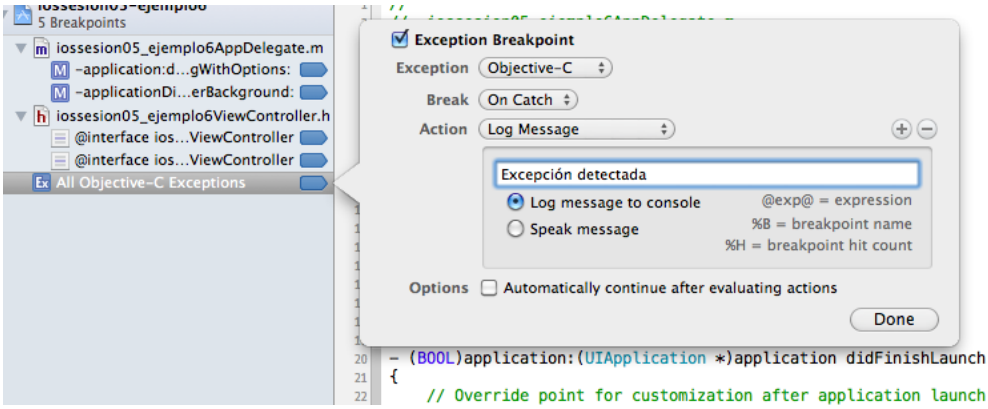
Navegador de breakpoints

Al pulsar sobre el botón "+" nos aparecerá una ventana emergente en la que nos da a elegir si queremos crear un *breakpoint* de tipo excepción o de tipo simbólico. Estos son los dos tipos que existen en XCode junto con el más común que hemos explicado antes.

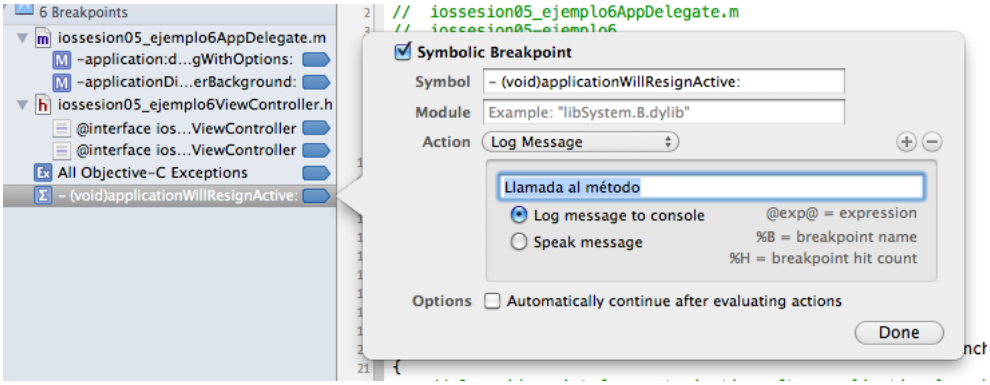


Tipos de breakpoint

- **Exception breakpoint**: Este tipo de breakpoint se ejecuta cuando salta una excepción en la aplicación. Es recomendable crear este tipo de breakpoints dentro del bloque "catch" de la sentencia *try-catch* para que en el momento en el que se ejecute se pueda ver toda la traza de manera completa y así detectar el problema que causa la excepción.
- **Symbolic breakpoint**: Este tipo de breakpoint se usa cuando queremos interrumpir la aplicación al ejecutar un determinado método o función. Deberemos indicar el método concreto por ejemplo `pathsMatchingExtensions:`, un método de una clase: `[SKTLine drawHandlesInView], people::Person::name()`, o un nombre de función: `_objc_msgForward`.



Exception breakpoint



Symbolic breakpoint

A la hora de depurar la aplicación, la ejecutaremos en modo "Debug" y esta se detendrá en cuanto detecte un breakpoint activo. En la ventana principal de XCode se mostrará el fichero que contiene el breakpoint, el cual se mostrará mediante una flecha verde. Esta linea será la que se vaya a ejecutar en cuanto continuemos con la ejecución de la aplicación. Una vez que tenemos parada la ejecución deberemos analizar el estado en que está la aplicación, para ello podremos realizar cualquiera de las siguientes acciones:

- **Situarse en el programa**: Lo primero que debemos hacer cuando la aplicación se detiene en un breakpoint es ver la traza por dónde ha pasado la ejecución hasta llegar a este punto. Los métodos que se listan en la traza que están en color negro y con el icono de usuario son nuestros y los que son de color gris son de la propia API, la cual no tenemos acceso a su código fuente. Para analizar un determinado método nuestro deberemos pulsar sobre el.
- **Analizar los valores de las variables**: Esta es una de las razones más importantes por la que depuramos nuestro código. Cada una de las variables iniciadas en el código se puede analizar y ver los valores de todos sus componentes. El depurador de XCode nos marca las variables han cambiado su valor recientemente ya que estas son las que más nos pueden interesar. También disponemos de un pequeño buscador para filtrar por nombre de variable o por su valor.

- **Crear un *watchpoint***: Un *watchpoint* es similar a un *breakpoint* pero que depende de una variable en concreto y se puede crear de forma dinámica mientras estamos ejecutando el depurador. Este se detendrá siempre que el valor de la variable cambie. Para crear un *watchpoint* haremos *ctrl+click* en la variable, dentro del listado y seleccionaremos *Watch Address of Variable*. Una vez creado se mostrará dentro del listado de breakpoints junto con el resto.
- **Usar la linea de comandos**: El uso de la línea de comandos del depurador de XCode es un método alternativo al uso de la interfaz. Mediante este podremos realizar las mismas funciones que con la interfaz e incluso más. El depurador que utiliza XCode es *GDB*.
- **Editar el listado de breakpoints dinamicamente**: Se pueden crear, editar y borrar los puntos de parada siempre que deseemos, incluso estando la aplicación en ejecución.
- **Continuar hasta la siguiente línea o continuar hasta el siguiente breakpoint**: A estas dos acciones se le llaman *step* y *continue* respectivamente. Cuando queremos avanzar en la depuración hasta el siguiente breakpoint ejecutaremos la opción de "continue" seleccionando *Product > Debug > Continue*. Por otro lado si queremos ir avanzando línea a línea en el código deberemos de seleccionar una de las opciones de *step* (step over, step into o step out) que se encuentran en *Product > Debug*. *Step over* detendrá la ejecución en la siguiente línea, *Step Into* la detendrá dentro del método que se "llame" desde la línea actual y *Step out* detendrá la ejecución cuando "salgamos" del método que se esté ejecutando. A todas estas opciones se puede acceder directamente desde los botones que están en la parte superior del panel de depuración.
- **Empezar de nuevo o abortar la ejecución**: Para detener la ejecución de la aplicación pulsaremos sobre el botón de "Stop" que se encuentra en la barra superior de XCode o desde *Product > Stop*.

Botón de Home

Si pulsamos el botón de *Home* en el simulador la aplicación no se detendrá ya que seguirá ejécutandose en segundo plano. Esto ocurre desde iOS 4 en donde la multitarea apareció. Si queremos detener la ejecución o depuración de una aplicación deberemos pulsar sobre el botón "Stop" que se encuentra en la barra superior de XCode.

Información depurador

El depurador de XCode no es más que una interfaz que nos facilita enormemente la función de depuración de nuestras aplicaciones. XCode hace uso de **GDB**, el famoso depurador de **GNU**. Recuerda que siempre que quieras puedes usar la línea de comandos si te sientes más cómodo. Puedes encontrar toda la documentación en su sitio oficial (http://www.gnu.org/s/gdb/documentation/).

3. Usando Instruments para detectar problemas de memoria

XCode, dentro de su conjunto de utilidades entre las que se encuentra el simulador de iPhone/iPad y el depurador encontramos una serie de aplicaciones para depurar, en un nivel algo más avanzado, nuestras aplicaciones. Entre estas utilidades se encuentra **Instruments**, la cual deberemos de usar al menos una vez al final de nuestro desarrollo para depurar fallos de memoria, la cual es un "bien" muy apreciado en todos los dispositivos móviles incluyendo los iOS.

Los "famosos" *Memory leaks* o "fugas de memoria" son partes de memoria que fueron reservadas o creadas en nuestra aplicación y que el programa pierde en un momento determinado. Estas partes de memoria nunca serán liberados por la aplicación y por tanto puede provocar múltiples fallos a nivel de ejecución que dificilmente podremos detectar si no es con la ayuda de las utilidades de XCode. Esto suele ocurrir cuando usamos dentro de nuestro código `new`, `malloc` o `alloc` y no hacemos posteriormente `delete`, `free` o `release` respectivamente.

Cuando reservamos memoria haciendo uso de uno de los tres métodos comentados anteriormente (new, malloc o alloc) el sistema operativo espera que la liberemos cuando no la necesitemos más (usando free, delete o release). Los *Memory leaks* aparecen cuando no liberamos estos fragmentos de memoria. Esta mala gestión de la memoria puede provocar en el mejor de los casos que simplemente esa memoria se libere cuando paremos la aplicación y no pase nada, en el peor de los casos puede provocar la salida de la aplicación de forma inesperada por falta de memoria en el caso que el problema ocurra con bastante frecuencia durante la ejecución.

Memory Leaks

Puedes encontrar más información sobre los *Memory leaks* en esta dirección de la *Wikipedia*: http://en.wikipedia.org/wiki/Memory_leak

Algunos *Memory leaks* son fáciles de detectar en nuestro código, pero otros no. Para detectar los más complicados utilizaremos la herramienta que incluye el "kit" de desarrollador que se llama **Instruments**. A continuación explicaremos el uso de *Instruments* para la detección de estos problemas relacionados con la memoria, para ello realizaremos un ejemplo sencillo en el que veremos su funcionamiento. ¡A por ello!

Novedades en iOS 5

Con la llegada del SDK de iOS 5, los problemas con las fugas de memoria se han reducido considerablemente ya que no tendremos que preocuparnos nunca más de liberar la memoria reservada (usar "release") o de retenerla ("retain"). Esto se puede gracias a lo que en Apple han llamado *Automatic Reference Counting (ARC)*, activando esta opción en nuestros proyectos indicamos que es el nuevo compilador de Apple LLVM el encargado de hacer estas funciones de gestión de memoria reduciendo en gran medida los problemas comentados anteriormente de "fugas de memoria". Si usamos esta opción en nuestras aplicaciones, estas serán sólo compatibles a partir de la versión de iOS 5.0, algo a tener muy en cuenta.

3.1. Probando la aplicación de ejemplo

Para realizar las pruebas con *Instruments* y el depurador de XCode vamos a crear una aplicación desde cero. Arrancamos XCode (versión 4.2 en adelante) y seleccionamos *File > New > New Project > Single View Application*. Escribimos el nombre "sesion06-ejemplo1" y desmarcamos todas las opciones (incluendo la comentada anteriormente "Use automatic reference counting").

Al haber desmarcado la opción de usar *ARC* la gestión de la memoria corre a nuesta cuenta, por lo que deberemos de tener mucho cuidado al respecto. Ahora abrimos el fichero `ViewController.m` y escribimos lo siguiente dentro del método `viewDidLoad`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
        // Do any additional setup after loading the view, typically from
a nib.

    NSString *str = [NSString stringWithFormat:@"Hola"];

    NSMutableArray *array = [[NSMutableArray alloc] init];
    [array addObject:@"opción 1"];

    NSLog(@"%@. Probando, probando...", str);

    [str release];
}
```

Ahora ejecutamos la aplicación mediante el botón de "Run" de la barra superior de la interfaz. Veremos que después de compilar y al poco de ejecutarse esta se para y aparece de nuevo la ventana de XCode mostrándonos algo similar a esto:

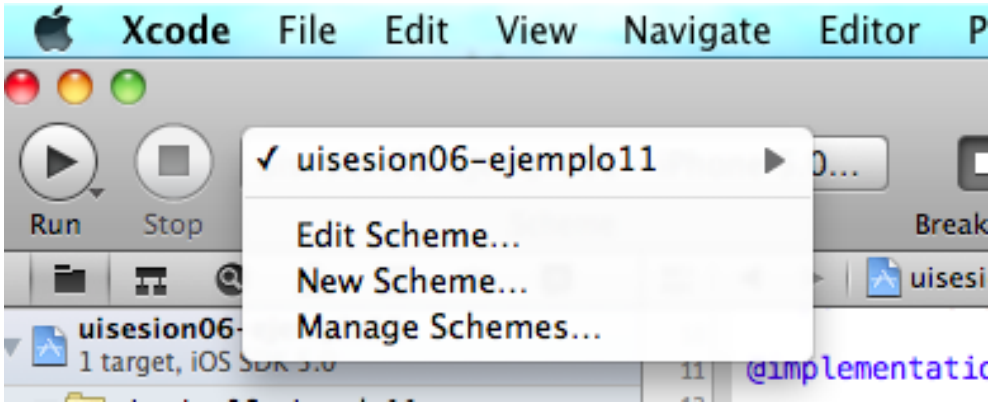
```
0x0131fde1  <+0145>  movl   $0x15,(%esp)
0x0131fde8  <+0152>  call   0x138dc10 <__CFRecordAllocatic
0x0131fded  <+0157>  mov    -0x10(%ebp),%eax
0x0131fdf0  <+0160>  mov    %eax,(%esp)
0x0131fdf3  <+0163>  call   0x13fbbf6 <dyld_stub_sel_getNa
0x0131fdf8  <+0168>  mov    %edi,0x10(%esp)
0x0131fdfc  <+0172>  mov    %eax,0xc(%esp)
0x0131fe00  <+0176>  add    $0xa,%esi
0x0131fe03  <+0179>  mov    %esi,0x8(%esp)
0x0131fe07  <+0183>  lea    0x12e6aa(%ebx),%eax
0x0131fe0d  <+0189>  mov    %eax,0x4(%esp)
0x0131fe11  <+0193>  movl   $0x3,(%esp)
0x0131fe18  <+0200>  call   0x136aad0 <CFLog>
0x0131fe1d  <+0205>  int3
0x0131fe1e  <+0206>  call   0x13fb8b4 <dyld_stub_getpid>
0x0131fe23  <+0211>  mov    %eax,(%esp)
0x0131fe26  <+0214>  movl   $0x9,0x4(%esp)
0x0131fe2e  <+0222>  call   0x13fb908 <dyld_stub_kill>
0x0131fe33  <+0227>  xor    %edi,%edi
0x0131fe35  <+0229>  jmp    0x132012e <___forwarding___+99
0x0131fe3a  <+0234>  lea    0xed29f(%ebx),%eax
0x0131fe40  <+0240>  mov    %eax,0x4(%esp)
0x0131fe44  <+0244>  mov    %esi,(%esp)
0x0131fe47  <+0247>  call   0x13fbc56 <dyld_stub_strcmp>
0x0131fe4c  <+0252>  test   %eax,%eax
0x0131fe4e  <+0254>  mov    %edi,%eax
0x0131fe50  <+0256>  mov    %eax,%edi
0x0131fe52  <+0258>  mov    %eax,-0x14(%ebp)
0x0131fe55  <+0261>  jne    0x131fe63 <___forwarding___+27
```

Pila de llamadas

Como podemos ver, a menos que sepamos programar en ensamblador, es imposible detectar el error de mediante esta traza. Para obtener algo más de información del error de memoria vamos a activar la opción de *Zombie*.

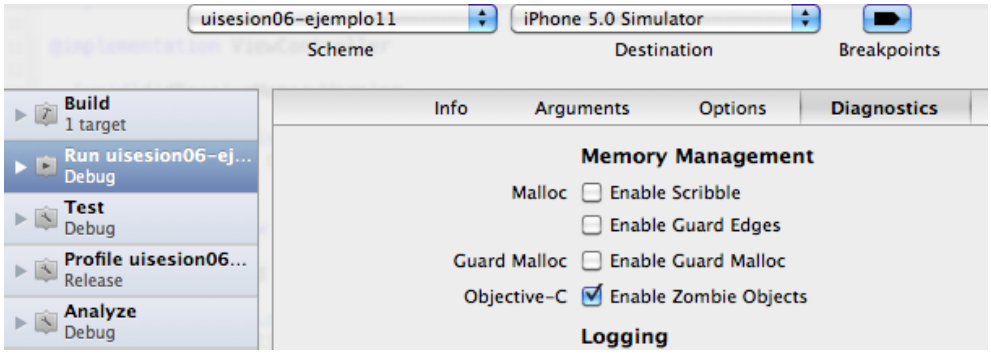
3.2. Usando NSZombieEnabled

`NSZombieEnabled` no es más que un parámetro de compilación cuya finalidad es proporcionarnos advertencias cuando cuando intentemos acceder a un objeto que ya haya sido liberado de memoria, en el caso de que cometamos ese error (bastante común por otro lado), el compilador nos dará algo más de información al respecto que nos ayudará a solucionarlo. Para activar este parámetro en versiones de XCode superiores a la 4.0 deberemos pulsar sobre la barra superior de selección de esquema y seleccionar `Edit Scheme`.



Edit Scheme

Ahora en la ventana emergente que aparece seleccionamos en la columna de la izquierda el objetivo "Run" y en la derecha la pestaña de `Diagnostics`. Ahi deberemos marcar la opción de `Enable Zombie Objects` dentro del bloque de `Memory Management`. Pulsamos *OK* para guardar los cambios y cerrar la ventana.



Activando los objetos Zombie

Ahora volvemos a ejecutar la aplicación y veremos que cuando se para, en la ventana de la consola nos apararecerá algo más de información, como por ejemplo el tipo de objeto que ha provocado el fallo de memoria.

```
This GDB was configured as "x86_64-apple-darwin".
sharedlibrary apply-load-rules all
Attaching to process 11137.
2011-10-16 13:01:13.706 uisesion06-ejemplo11[11137:207] Hola. Probando, probando...
2011-10-16 13:01:13.713 uisesion06-ejemplo11[11137:207] *** -[CFString release]: message sent to
deallocated instance 0x6b1f890
(gdb)
```

Mensaje de consola del objeto zombie

Como podemos ver en la salida de la consola el objeto que está mal gestionado es `NSString` al hacer `release` de el. Esto ocurre porque al crear el `NSString` lo hemos creado usando el método `stringWithFormat` y este tipo de inicialización hace que `NSString` sea "autoliberable", es decir *autorelease*, por lo que es el propio compilador el que gestiona la memoria del objeto. Al liberar nosotros la memoria usando `[str release]` estamos forzando la liberación del objecto de la memoria cuando posiblemente

ya se haya liberado, de ahi el fallo.

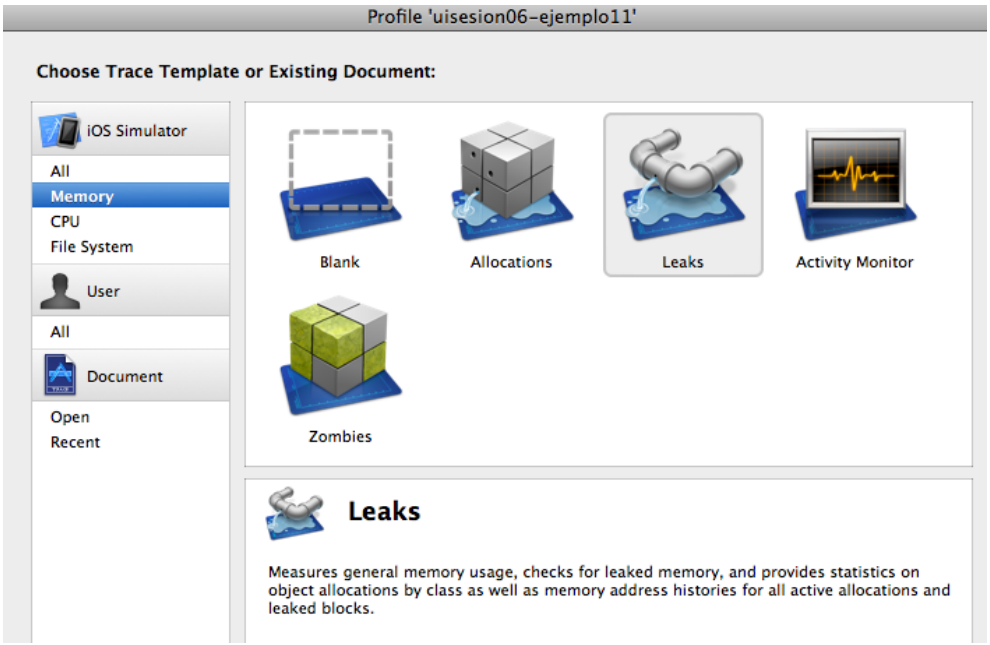
Para corregir este error, simplemente deberemos que suprimir la línea `[str release]`. De esta forma ya debe de funcionar correctamente la aplicación y no debe de saltar en ningún momento. Lo comprobamos volviéndola a ejecutar.

3.3. Encontrando fugas de memoria (memory leaks)

En este apartado vamos a detectar posibles fugas de memoria que se produzcan en nuestras aplicaciones. Estas son causadas, como hemos comentado en puntos anteriores por no liberar objetos de memoria en determinadas ocasiones dentro de nuestro código. A partir de iOS 5, estas liberaciones de memoria se realizarán de forma automática por el compilador y no deberemos de preocuparnos por ello, pero en versiones anteriores sí que lo tendremos que tener en cuenta. Para entender de una forma más clara lo que es una fuga de memoria y lo que implica vamos a continuar con el ejemplo que hemos realizado en el punto anterior.

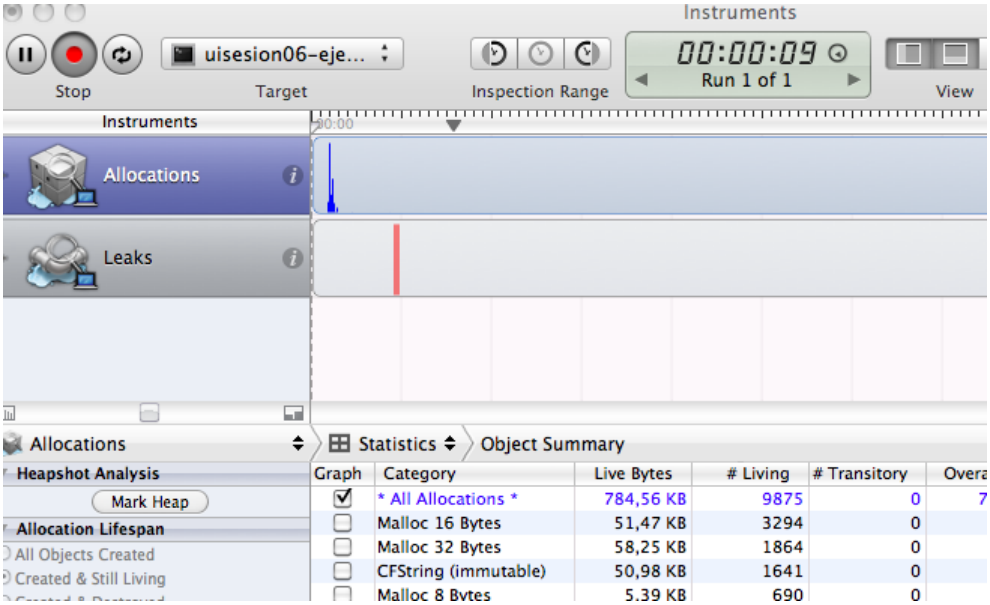
Para la detección y corrección de las posibles fugas de memoria que se produzcan en nuestras aplicaciones usaremos las utilidades de **Instruments** que incorpora XCode y que con la salida de *iOS 5* han mejorado notablemente. Siguiendo con el ejemplo anterior y comentando la línea `[str release]`, volvemos a ejecutar la aplicación. En este caso no notaremos nada, pero internamente sí está pasando algo que a la larga puede perjudicar al rendimiento de la aplicación. Vamos a adivinar qué es...

Para analizar con detalle los *memory leaks* pulsamos sobre la opción del menú principal *Product > Profile*. En ese momento XCode compilará la aplicación y seguidamente abrirá una ventana emergente con distintas opciones, cada una de ellas corresponde a una utilizad del paquete de *Instruments*. En este caso deberemos seleccionar `Leaks`.

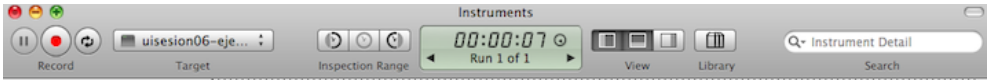


Pantalla principal de Instruments

Al pulsar sobre *Leaks* se abrirá una nueva interfaz en una ventana separada de XCode que estará divida en varias partes. Por un lado en la parte superior de la pantalla se encuentra el menu principal que tendrá los botones de control de ejecución: "pausa", "grabación/ejecución" y "reset", el "target" que estamos analizando, los distintos rangos de inspección, por defecto analizaremos toda la traza de ejecución y los distintos tipos de vista que están activadas.



Ejecución de Instruments

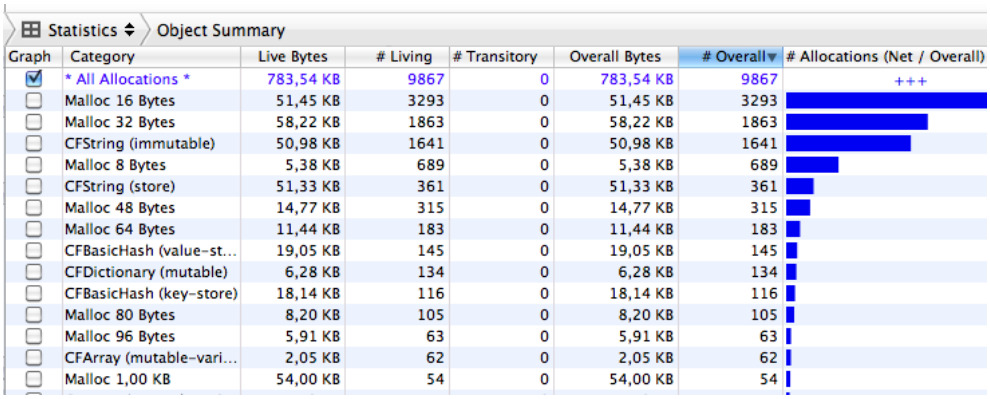


Barra superior de Instruments

En la parte central de la pantalla se encuentra toda la información que *Instruments* va recopilando a medida que la traza de ejecución va pasando. Al seleccionar *Leaks* utilizaremos dos instrumentos distintos: `Allocations` y `Leaks`. El primero nos indica en azul los objetos que la aplicación en ejecución va reservando en memoria. El segundo por otro lado nos indica en rojo los objetos que hemos creado en memoria y no hemos liberado cuando teniamos que haberlo hecho, las *fugas de memoria*.

A medida que la ejecución del programa va pasando, las gráficas de *Allocations* y de *Leaks* se van actualizando cada cierto tiempo, este intervalo de tiempo entre actualizaciones del estado de la memoria de la aplicación se puede cambiar desde el menu de la columna de la izquierda, dentro del bloque de `Snapshots` de *Leaks*. Por defecto el intervalo de actualización es de 10 segundos.

Cuando *Leaks* detecta una fuga de memoria la indica mediante una barra de color rojo dentro de la línea de tiempo. En el caso de que se detecten varios leaks al mismo tiempo, esta barra será mayor y en este caso la vista gráfica se irá alejando automáticamente para representar de una forma más clara los datos. En la parte justo inferior a las barras gráficas de *Allocations* y *Leaks* se encuentra la misma información pero en forma de listado. Ahi se mostrarán todos los objetos a los que se hace referencia en las barras gráficas. Si pulsamos sobre la propia barra de *Leaks* se seleccionará el objeto al que hace referencia dentro del listado.



Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes	# Overall	# Allocations (Net / Overall)
☑	* All Allocations *	783,54 KB	9867	0	783,54 KB	9867	+++
☐	Malloc 16 Bytes	51,45 KB	3293	0	51,45 KB	3293	
☐	Malloc 32 Bytes	58,22 KB	1863	0	58,22 KB	1863	
☐	CFString (immutable)	50,98 KB	1641	0	50,98 KB	1641	
☐	Malloc 8 Bytes	5,38 KB	689	0	5,38 KB	689	
☐	CFString (store)	51,33 KB	361	0	51,33 KB	361	
☐	Malloc 48 Bytes	14,77 KB	315	0	14,77 KB	315	
☐	Malloc 64 Bytes	11,44 KB	183	0	11,44 KB	183	
☐	CFBasicHash (value-st...	19,05 KB	145	0	19,05 KB	145	
☐	CFDictionary (mutable)	6,28 KB	134	0	6,28 KB	134	
☐	CFBasicHash (key-store)	18,14 KB	116	0	18,14 KB	116	
☐	Malloc 80 Bytes	8,20 KB	105	0	8,20 KB	105	
☐	Malloc 96 Bytes	5,91 KB	63	0	5,91 KB	63	
☐	CFArray (mutable-vari...	2,05 KB	62	0	2,05 KB	62	
☐	Malloc 1,00 KB	54,00 KB	54	0	54,00 KB	54	

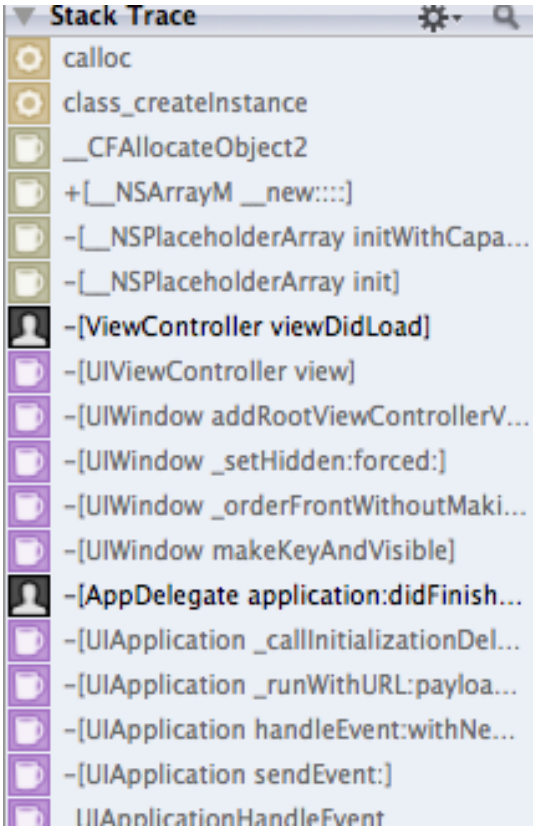
Listado de objetos usados

Si seleccionamos uno de los objetos en los que *Instruments* ha detectado un *leak* y hacemos doble click sobre el accederemos a la línea exacta de nuestro código en donde se referencia a ese objeto. De esta forma podremos situarnos en el objeto en el que no liberamos su memoria y podremos realizar los cambios oportunos en nuestro código para remediarlo. Esto se puede hacer directamente desde *Instruments*.

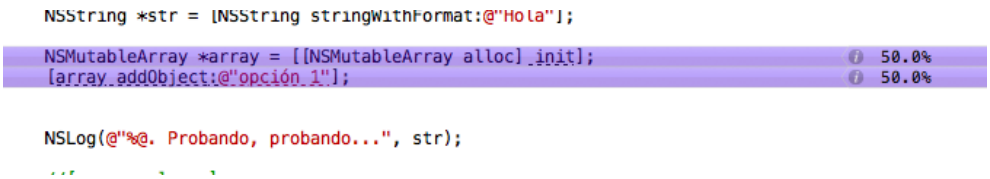
Una vez que hemos visto las principales caracteristicas del funcionamiento de *Instruments* y su detección de *Leaks* vamos a proceder a realizar lo propio con nuestra aplicación de ejemplo. Para ello la arrancamos mediante *Product* > *Profile*. Seleccionamos la opción de `Leaks` y esperamos a que arranque *Instruments*. En ese momento veremos como la línea de tiempo empieza a avanzar detallando en la gráfica superior las reservas de memoria y en la barra inferior los *leaks* encontrados. En este caso *Instruments* detecta dos *leaks* en el segundo seis aproximadamente, aunque el momento puede variar por muchos factores externos.

En cualquier momento de la ejecución podemos pausar la depuración por *Instruments* pulsando sobre el botón de pausa de la barra superior del menu. También podemos detener el análisis y guardar los resultados para más adelante analizarlos. Si después de pausar la ejecución la volvemos a poner en marcha veremos como la linea de tiempo se posicionará en el momento en donde le corresponde y dejará un hueco vacío en medio sin datos.

Nosotros vamos a detener la ejecución pulsando sobre el botón rojo de `stop`. Una vez detenido el análisis y la aplicación nos situamos sobre los *leaks* detectados (la única barra roja que hay) y vemos como en el listado de objetos aparecen dos: `Malloc 16 bytes` y `NSMutableArray`. Si desplegamos la columna de la derecha pulsando sobre el botón de la derecha del todo del apartado "view" del menu superior veremos toda la traza de la aplicación y, en negrita, los métodos en los que se ha detectado el *leak* seleccionado. Como podemos ver el método al que hace referencia es `[ViewController viewDidLoad]`. Haciendo doble click sobre el nombre del método podremos ver el código y subrayado en en color violeta los métodos responsables del *leak*.



Pila de llamadas



Detección de Memory Leak

Volviendo al código y analizándolo con un poco más de detalle vemos que la variable `array` de tipo `NSMutableArray` se crea en memoria usando `alloc` y en ningún momento la liberamos, por lo que inclumplimos la norma básica de la gestión de memoria en iOS: todo lo que se crea en memoria se debe liberar una vez que no lo necesitemos. Por tanto, para solucionar esta *fuga de memoria* deberemos llamar a `release` al final del método quedando de la siguiente manera:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
        // Do any additional setup after loading the view, typically from
a nib.

    NSString *str = [NSString stringWithFormat:@"Hola"];
```

```
    NSMutableArray *array = [[NSMutableArray alloc] init]; //Reserva
memoria
    [array addObject:@"opción 1"];


    NSLog(@"%@. Probando, probando...", str);

    [array release]; //Libera la memoria: evita el leak
}
```

Ahora volvemos a ejecutar la aplicación con *Instruments* y comprobaremos que ya no nos aparece ningún *Leak*. ¡Problema resuelto!

4. Pruebas de unidad

Los test de unidad o unitarios son una forma de probar el correcto funcionamiento de un módulo o método concreto de la aplicación que desarrollemos. La finalidad de este tipo de pruebas es comprobar que cada uno de los módulos de la aplicación funcione según lo esperado por separado. Por otro lado, los tests de integración comprueban que la aplicación funciona correctamente en su totalidad. Dentro de este módulo veremos como implementar las pruebas de unidad en nuestras aplicaciones iOS.

XCode facilita en gran manera la generación de las pruebas de unidad, proporciona un entorno adaptado y simple. Las pruebas de unidad están compuestas por los *tests*, los cuales comprueban que un bloque de código devuelve un determinado resultado, en caso contrario el test falla. Los *grupos de tests* son un conjunto de tests que prueban una determinada característica o funcionalidad de la aplicación. El *framework* en el que se basa XCode para su entorno de testing es de código abierto y se llama *SenTestingKit*.

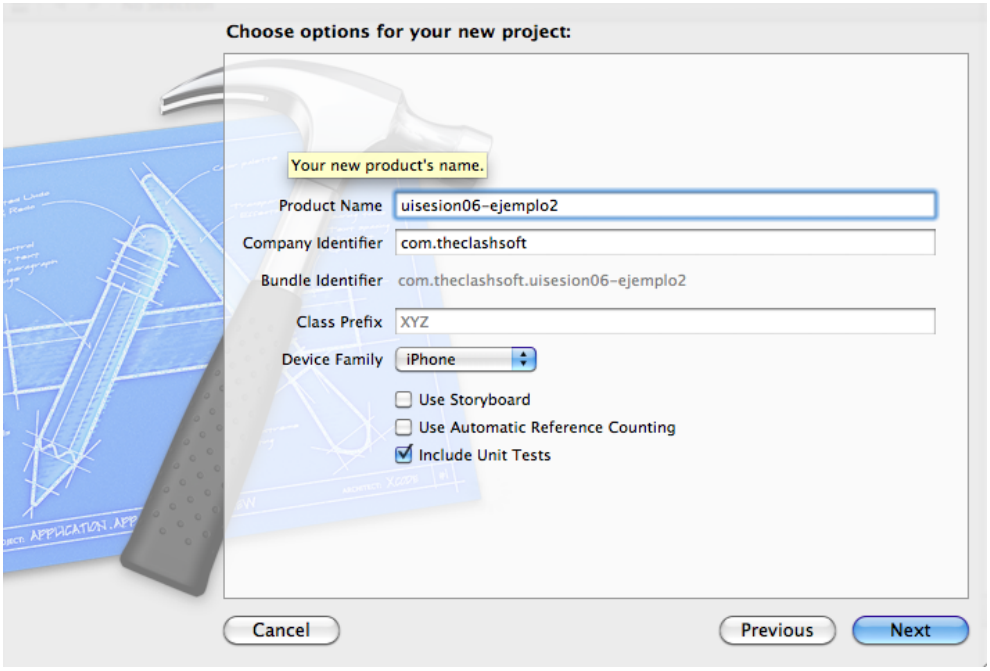
En XCode encontramos dos tipos de tests de unidad:

- **Tests de lógica:** Comprueban el correcto funcionamiento de cada uno de los métodos por separado, no en su conjunto en toda la aplicación. Los tests de lógica se pueden implementar para hacer pruebas de estres, poniendo a prueba la aplicación para situaciones extremas no habituales en una ejecución nomal. Estos tipos de pruebas sólo se pueden probar con el simulador de iOS.
- **Test de aplicación:** Comprueban el correcto funcionamiento de la aplicación en su conjunto. Incluyen tests de conectividad con las interfaces de usuario, tests de sensores, etc. Este tipo de pruebas sí se pueden realizar sobre dispositivos iOS, a parte de los simuladores.

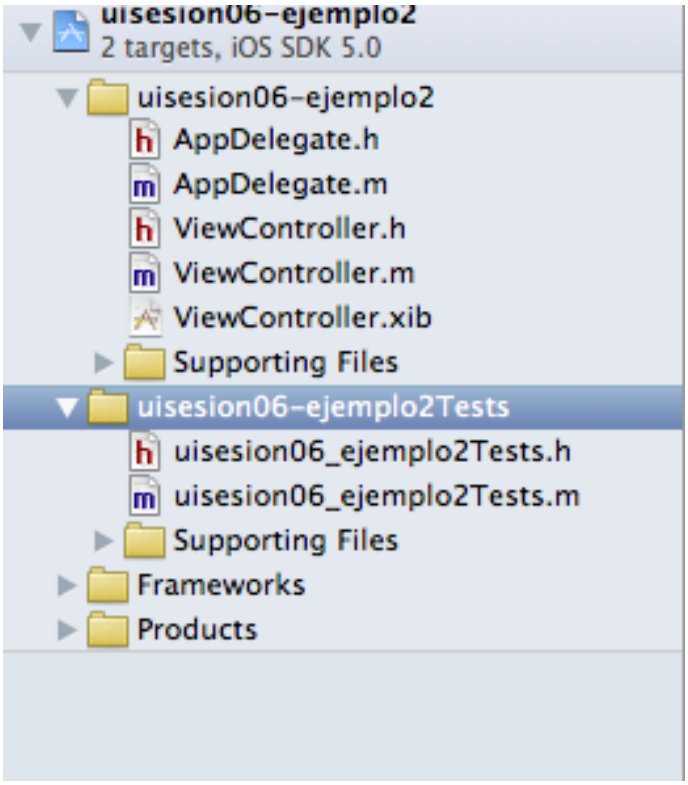
4.1. Configurando XCode para ejecutar tests

En el ejemplo que vamos a realizar a continuación vamos a realizar los dos tipos de pruebas comentadas anteriormente: de lógica y de aplicación. Para ello empezamos creando un nuevo proyecto en XCode de tipo *Single View Application* que llamaremos `uisesion06-ejemplo2`. Debemos de acordarnos de marcar la opción de `Include Unit`

`Tests`, de esta forma el propio XCode nos creará un subdirectorio dentro del proyecto en donde se ubicarán los distintos tests de unidad que realicemos.

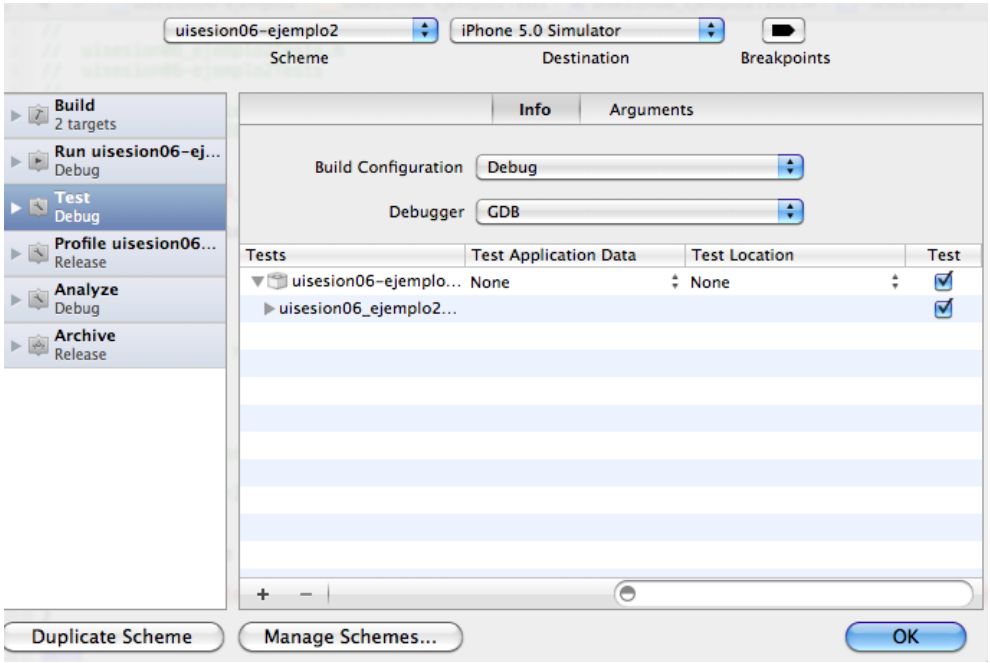


Propiedades del proyecto



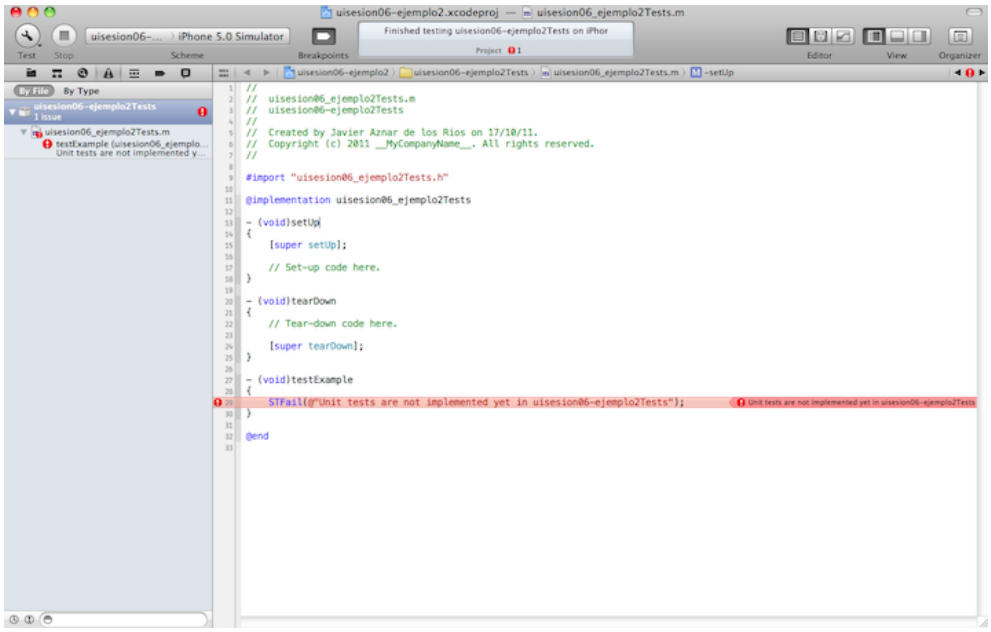
Carpeta de Tests del proyecto

XCode también ha creado de forma automática un *target* nuevo para tests que podremos modificar a nuestro antojo y un *esquema*. Esto lo podemos ver pulsando sobre el raiz del proyecto y pulsando sobre `Edit Scheme` dentro del listado de esquemas del proyecto respectivamente.



Target de Tests en el proyecto

Para ejecutar los tests que se crean por defecto al generar el proyecto deberemos de pulsar sobre *Product* > *Test* y esperar a que se compile el proyecto y arranquen los tests. Veremos que este caso falla uno, el correspondiente al método `(void)testExample`. Como podemos ver los tests que no han pasado se muestran en color rojo tanto en el código como en el navegador de la izquierda:



Fallo de test

Si desplegamos la sección de abajo y el log de salida veremos un resumen de los tests que se han pasado y sus resultados:

```
Test Suite 'All tests' started at 2011-10-21 10:48:33 +0000
Test Suite
'/Users/javikr/Documents/Projects/builds/uisesion06-ejemplo2-eqfkbmmboigtvg
gumflvqvufkhlb/Build/Products/Debug-iphonesimulator/uisesion06-ejemplo2Tests.octest
(Tests)' started at 2011-10-21 10:48:33 +0000
Test Suite 'uisesion06_ejemplo2Tests' started at 2011-10-21 10:48:33 +0000
Test Case '-[uisesion06_ejemplo2Tests testExample]' started.
/Users/javikr/Documents/Projects/CURSO/uisesion06-ejemplo2/uisesion06-ejemplo2Tests/
uisesion06_ejemplo2Tests.m:29: error: -[uisesion06_ejemplo2Tests
testExample] :
Unit tests are not implemented yet in uisesion06-ejemplo2Tests
Test Case '-[uisesion06_ejemplo2Tests testExample]' failed (0.000
seconds).
Test Suite 'uisesion06_ejemplo2Tests' finished at 2011-10-21 10:48:33
+0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.000) seconds
Test Suite
'/Users/javikr/Documents/Projects/builds/uisesion06-ejemplo2-eqfkbmmboig
tvggumflvqvufkhlb/Build/Products/Debug-iphonesimulator/uisesion06-ejemplo2Tests.octest
(Tests)' finished at 2011-10-21 10:48:33 +0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.001) seconds
Test Suite 'All tests' finished at 2011-10-21 10:48:33 +0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.025) seconds
```

Una vez que hemos configurado nuestro proyecto de XCode para ejecutar tests de unidad vamos a ver cómo escribir estos tests de una forma eficiente.

4.2. Escribiendo tests de unidad

Un test no deja de ser una instancia de un conjunto o *suite* de tests, el cual no recibe parámetros ni devuelve ningún tipo de objeto, sólo `void`. El objetivo del test es probar la API de la aplicación y detectar si produce el resultado esperado. Este resultado puede ser una excepción o un valor cualquiera. Los tests hacen uso de una serie de *macros*, las cuales debemos de conocer saber las distintas posibilidades que disponemos a la hora de diseñar los casos de prueba. Para crear una *suite de tests* deberemos crear una clase que herede de `SenTestCase`. Para crear nuevos tests dentro de una *suite de tests* ya creada simplemente deberemos de implementarlo siguiendo la siguiente estructura:

Macros disponibles

El framework de `SenTestingKit` ofrece un amplio listado de macros que podemos utilizar en nuestras pruebas de unidad. El listado completo lo podemos ver dentro de la documentación oficial de iOS en esta dirección. Existen macros para producir fallos de forma incondicional, para comprobar dos valores concretos, para comprobar referencias a valores nulos, comprobaciones de *booleanos*, otros que comprueban si se lanza o no una excepción y de qué tipo...

```
- (void)testMiTestDePrueba {
   ...     // Configuración inicial (setup)
```

```
   ST...   // Asserts
   ...            // Liberación de memoria y variables auxiliares
}
```

Normalmente, para configurar las variables auxiliares que son necesarias para ejecutar los casos de pruebas implementaremos dos métodos: `(void)setup` y `(void)teardown`. El primero creará las variables y las configurará de forma adecuada, el segundo las liberará de memoria. Esta sería la estructura básica de ambos métodos, los cuales están por defecto incluidos al crear los tests de unidad con XCode:

```
- (void)setUp {
   objeto_test = [[[MiClase alloc] init] retain];
   STAssertNotNil(objeto_test, @"No se puede crear un objeto de la clase
MiClase");
}

- (void)tearDown {
   [objeto_test release];
}
```

Fallo en setup o tearDown

Cuando XCode detecta algún fallo en los métodos `Setup` o `tearDown`, estos se reportarán dentro del caso de pruebas que se esté probando y desde el que se hizo la llamada.