

¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
 Ese apoyo que siempre quiso tener...

1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
 HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
 Gestor de contenidos (Alfresco)
 Aplicaciones híbridas

Tareas programadas (Quartz)
 Gestor documental (Alfresco)
 Inversión de control (Spring)

Control de autenticación y
 acceso (Spring Security)
 UDDI
 Web Services
 Rest Services
 Social SSO
 SSO (Cas)

JPA-Hibernate, MyBatis
 Motor de búsqueda empresarial (Solr)
 ETL (Talend)

Dirección de Proyectos Informáticos.
 Metodologías ágiles
 Patrones de diseño
 TDD

BPM (jBPM o Bonita)
 Generación de informes (JasperReport)
 ESB (Open ESB)

AdictosAlTrabajo

Final de Terrakas
¡¡Ven al estreno!!
terrakas.com



autentia
Soporte a desarrollo informático
Hosting patrocinado por
enREDados

Entra en Adictos a través de  

E-mail

Contraseña

Entrar [Deseo registrarme](#)
[Olvidé mi contraseña](#)



[Inicio](#) [Quiénes somos](#) [Formación](#) [Comparador de salarios](#) [Nuestros libros](#) [Más](#)

» Estás en: [Inicio](#) [Tutoriales](#) [Xcode Guía Básica Test Unitarios](#)



Saúl García Díaz

Consultor tecnológico de desarrollo de proyectos informáticos.

Charla sobre [LiquiBase](#)

Charla sobre [Alfresco Community Edition](#)

Puedes encontrarme en [Autentia](#): Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

[Ver todos los tutoriales del autor](#)

Fecha de publicación del tutorial: 2013-05-13

Tutorial visitado 71 veces [Descargar en PDF](#)

Xcode - Guía Básica Tests Unitarios

0. Índice de contenidos.

- 1. Entorno
- 2. Introducción
- 3. OCUnit
 - 3.1 Añadiendo Tests Unitarios a nuestro proyecto
 - 3.2 Configuración Test Lógicos
 - 3.3 Configuración Test de Aplicación
- 4. Primeros Tests Unitarios
- 5. Alternativas OCUnit y complementos
- 6. Conclusiones

1. Entorno

- Hardware: Portátil MacBook Pro 15' (2.6 GHz Intel Core i7, 8GB DDR3 SDRAM, 250GB HDD).
- Sistema Operativo: Mac OS X Mountain Lion 10.8.2
- Xcode 4.6.2

2. Introducción

Independientemente del software que estemos desarrollando el uso de TDD (Test Driven Development) es un enfoque de desarrollo ágil en el que primero se escriben las pruebas y después el código necesario para que las pruebas se ejecuten de manera satisfactoria. El objetivo del tutorial no es explicar cuáles son las ventajas de aplicar TDD en los desarrollos actuales, incluidos los desarrollos para plataformas móviles, en este caso para dispositivos iOS, sino aprender que herramientas pone a nuestra disposición dicha plataforma para probar nuestras apps. En cualquier caso no esta mal recordar algunas de las principales ventajas de esta técnica, como por ejemplo:

- Mayor calidad y robustez del código.
- Diseño más simple y enfocado a las necesidades reales.
- Aumento de la productividad.
- Menos tiempo en la detección y corrección de errores.

3. OCUnit

OCUnit es un framework de pruebas para Objective C. Esta desarrollado por Sen:Te y la principal ventaja frente a otras alternativas disponibles es que Apple lo ha integrado en Xcode desde la versión 2.1, por lo que resulta más fácil y rápido comenzar a trabajar. Es multiplataforma, lo cual es útil si estamos usando Objective C en plataformas distintas a iOS o Mac OS X.

Otra característica importante de OCUnit es que pone a nuestra disposición SenTestingKit, framework open-source que nos permitirá diseñar conjuntos de pruebas para probar nuestro código. La perfecta integración con Xcode nos ayudará a incluir TDD en nuestros desarrollos de una manera fácil, permitiéndonos realizar dos tipos de pruebas:

- Pruebas lógicas: estas pruebas comprueban el correcto funcionamiento de una pieza de código por sí mismo (no en el contexto de una aplicación). Con las pruebas lógicas se puede diseñar casos de prueba específicos para probar una pieza de código de manera independiente. También puede utilizar las pruebas lógicas para llevar a cabo pruebas de estrés de nuestro código asegurándonos que se comporta correctamente en situaciones extremas que probablemente se den en la ejecución de la aplicación en un entorno real. Estas pruebas ayudan a producir un código robusto que funciona correctamente cuando se dan casos que no se han previsto.
- Pruebas de aplicación: estos exámenes prueban piezas de código en el contexto real de la aplicación. Podemos utilizar estas pruebas de aplicación para asegurarnos de que las conexiones entre los controles de la interfaz de usuario (IBOutlet y IBAction) y los objetos de negocio funcionan correctamente. También podemos utilizar estas

Catálogo de servicios Autentia



Síguenos a través de:



Últimas Noticias

» [Atención, APLAZADO](#)
Estreno último capítulo de Terrakas

» [Vendedor: Soy inseguro, filtra o elige por mi: si quieres que te compre.](#)

» [Comentando el libro: El arte de pensar, de Rolf Dobelli](#)

» [Ya está a la venta mi segundo libro: Planifica tu éxito, de aprendiz a empresario](#)

» [Ya esta disponible en eBook mi primer libro: Informática Profesional](#)

[Histórico de noticias](#)

Últimos Tutoriales

» [Haciendo BDD con Cucumber](#)

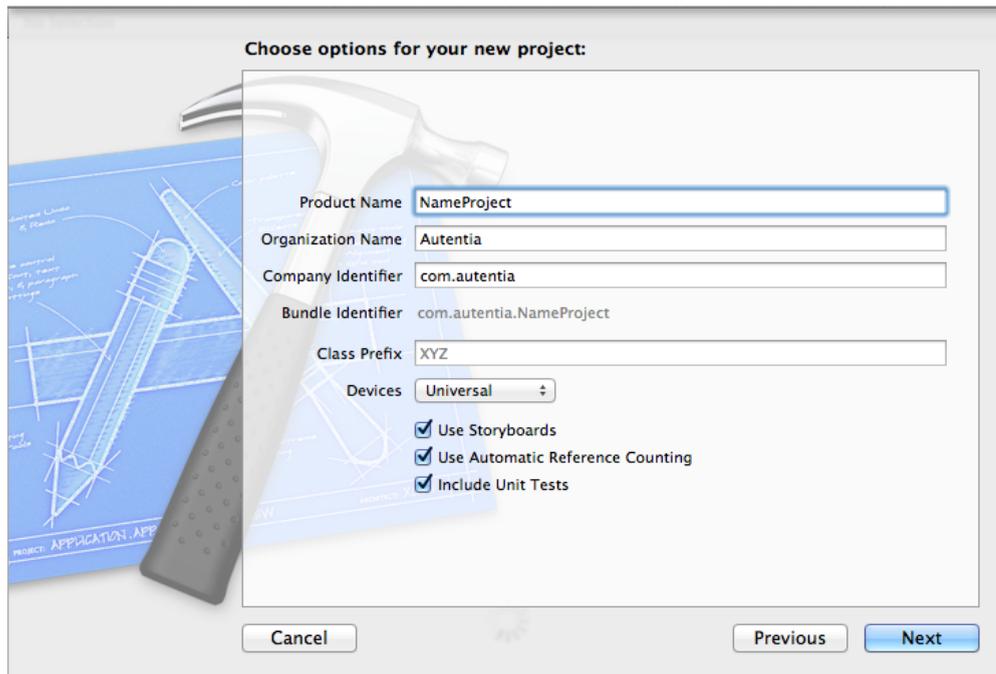
» [Manejo de test con TestLink](#)

» [Prototipado de pantallas con Pencil](#)

tests para realizar pruebas de hardware, como por ejemplo, obtener la ubicación del dispositivo en el que se ejecuta la aplicación.

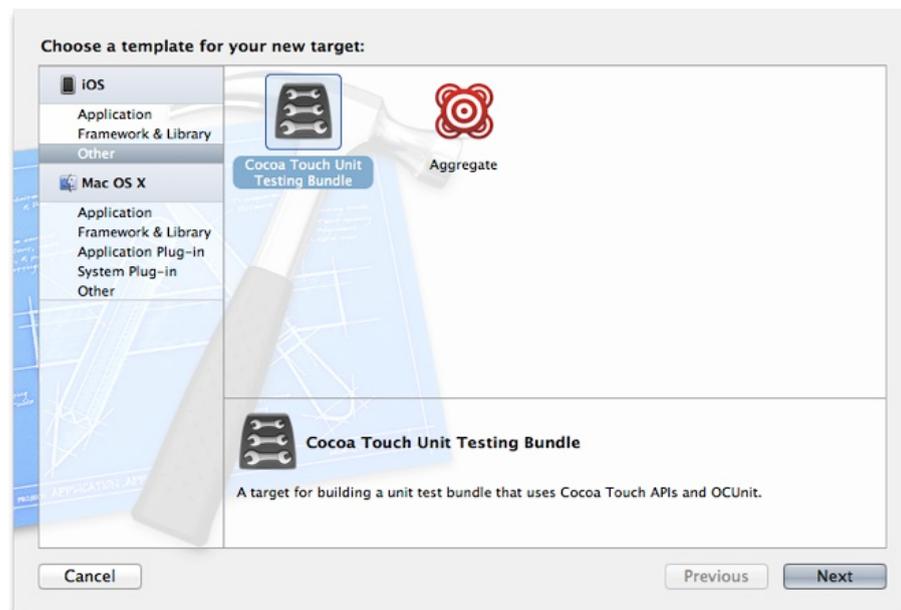
3.1 Añadiendo Tests Unitarios a nuestro proyecto

La forma más cómoda de añadir tests unitarios a un proyecto es en el momento de crearlo. Si selecciona la opción "Include unit tests" al crear un proyecto, Xcode incluye un nuevo "target" para las pruebas unitarias en el proyecto.



Si este no es el caso podemos seguir los siguientes pasos:

- 1 - Desde el proyecto al que queremos añadir la unidad de pruebas elegimos File > New > New Target.
- 2 - Seleccionamos una de las plantillas que nos proporciona Xcode en función de la plataforma:
 - IOS : En esta sección seleccionar Otros > Cocoa Touch Unit Testing Bundle.
 - Mac : En esta sección seleccionar Otros > Cocoa Unit Testing Bundle.



- 4 - Seleccionamos la opción Next.
- 5 - Especificamos la información básica y necesaria para el nuevo "target".

» Como testear aplicaciones en Ember.js

» Internacionalizar una aplicación creada con Ember

Últimos Tutoriales del Autor

» Mountain Lion - Git "Command Not Found"

» Chrome Remote Desktop

» Cómo implementar un datatable editable con el soporte de primefaces

» Cómo trabajar con branch en SmartGit

» Cómo trabajar con JSF2 y el soporte de inyección de dependencias de Spring

Últimas ofertas de empleo

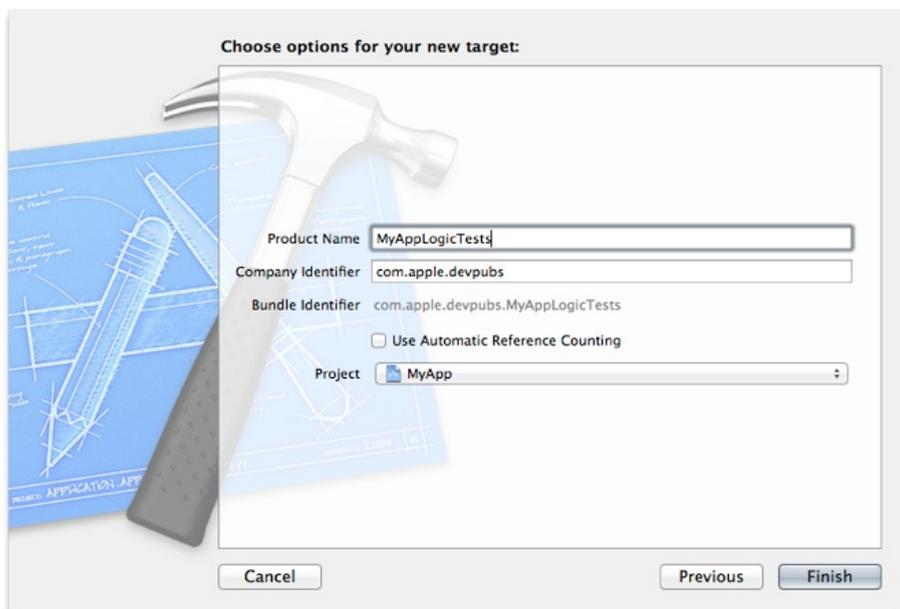
2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

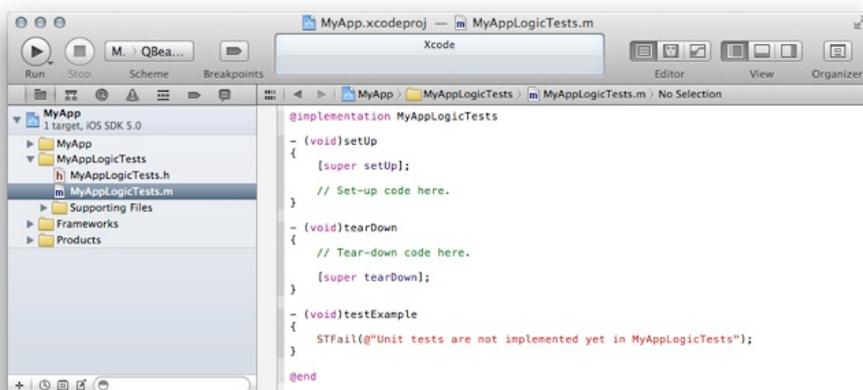
2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.



- 6 - Pulsamos finish.

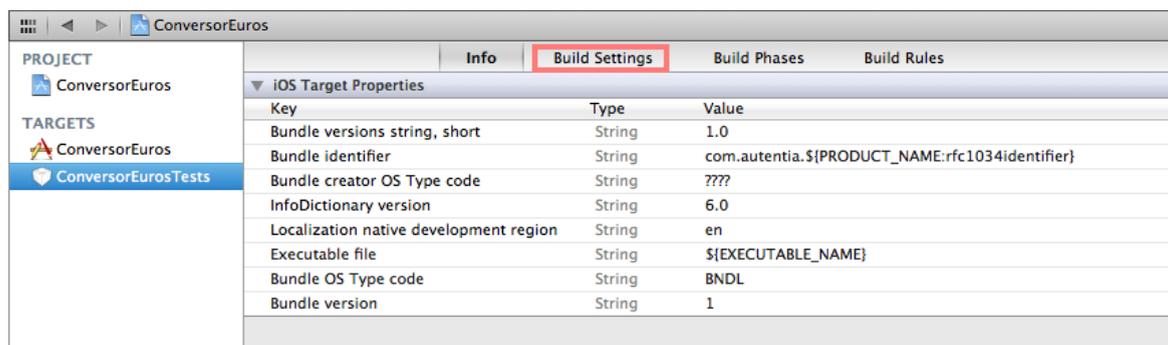
Desde el navegador podemos ver el nuevo "target" en la jerarquía del proyecto.



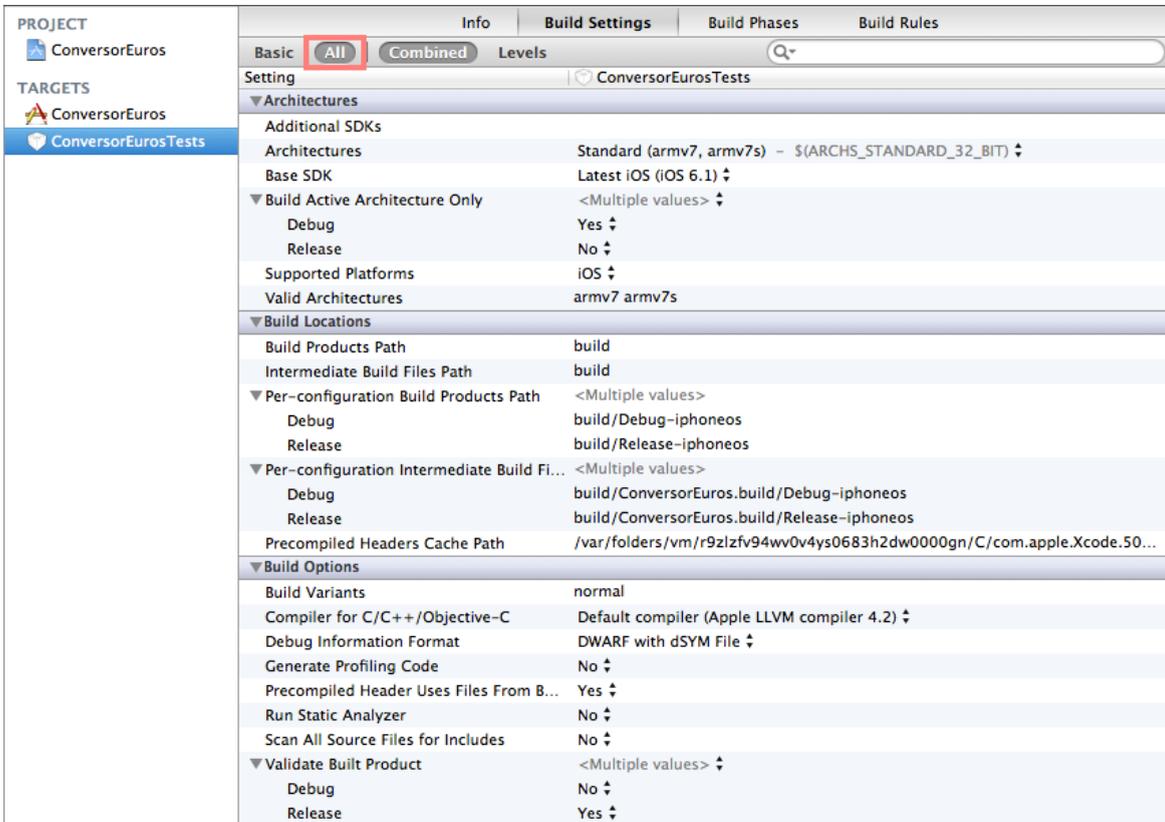
3.2 Configuración Test Lógicos

Si añadimos un nuevo "target" para los tests unitarios tal y como os he descrito en el punto anterior, ya está configurado para realizar pruebas lógicas. Para realizar la configuración manualmente podemos seguir los siguientes pasos:

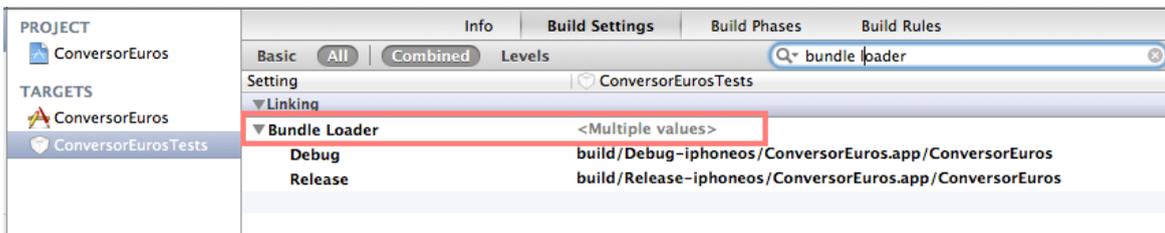
- 1 - Desde el editor de proyecto seleccionamos el "target" a configurar y pulsamos "Build settings".



- 2 - Desde el panel "Build settings", click "All".

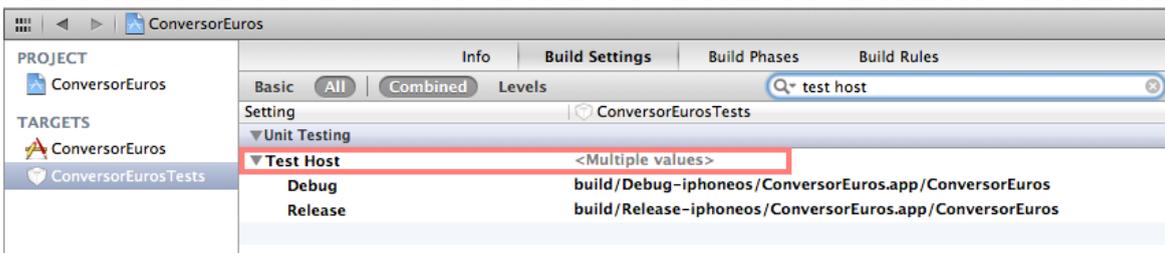


3 - En el campo de búsqueda escribimos "bundle loader".



4 - Si la propiedad "bundle loader" aparece en negrita la eliminamos.

5 - En el campo de búsqueda escribimos "test host".



6 - Si la propiedad "test host" aparece en negrita la eliminamos.

3.3 Configuración Test de Aplicación

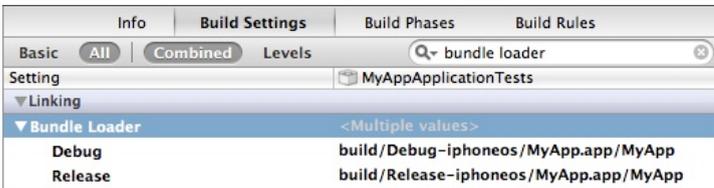
Si hemos creado un proyecto, marcando la opción "include unit test" la unidad de tests ya está configurada para realizar pruebas unitarias de aplicación. De cualquier otro modo para configurar la unidad de test podemos seguir los siguientes pasos:

1 y 2 - Exactamente igual que en el punto anterior. Con la unidad de pruebas que queremos configurar seleccionada abrimos el panel "Build settings".

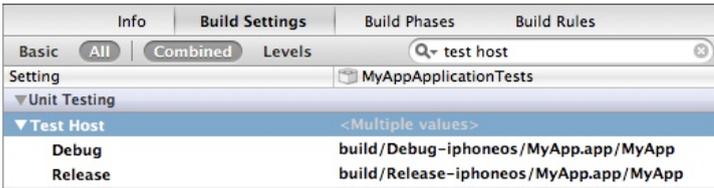
3 - Asignamos valor a la propiedades de la sección "bundle loader" en función de la plataforma:

- iOS: \$(BUILT_PRODUCTS_DIR)/.app/
- Mac: \$(BUILT_PRODUCTS_DIR)/.app/Contents/MacOS/

donde "app-name" es el nombre de nuestra aplicación.

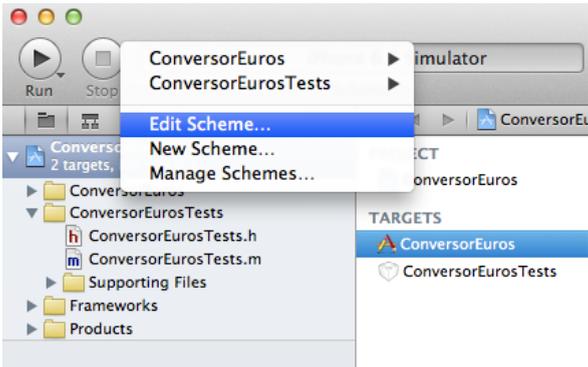


4 - Asignamos valor a las propiedades de la sección "test host" a \$(BUNDLE_LOADER):

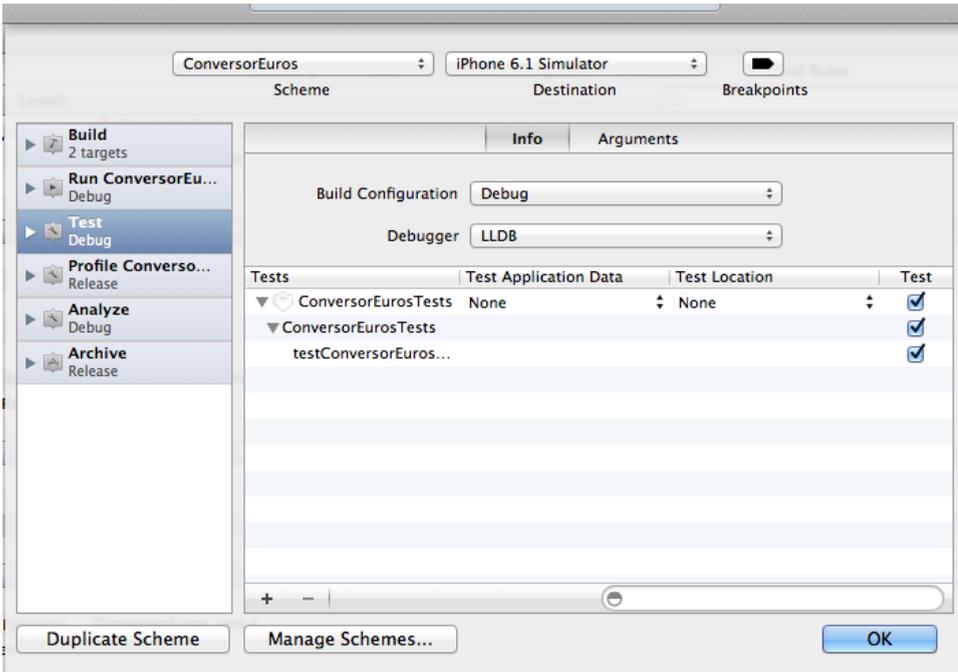


5 - Hacemos dependiente el "target" para las pruebas del "target" en el que se basa nuestra aplicación

6 - Nos aseguramos que el "target" para las pruebas esta correctamente asociado al schema de nuestra aplicación. Desde el panel de edición editamos el schema de nuestra aplicación:



Seleccionamos la opción "Test" del menú lateral y tenemos que ver la unidad de pruebas asociada.



4. Primeros Tests Unitarios

El primer paso a la hora de escribir pruebas unitarias y por tanto el primer paso para comenzar el diseño de nuestra aplicación es identificar los "requisitos", es decir , cuál es la necesidad de nuestro software. Después podemos decidir que código nos gustaría usar para satisfacer la necesidad requerida. Este código es el que usaremos para conformar el cuerpo de nuestro test. Para poder seguir el tutorial más cómodamente os dejo aquí los fuentes.

Una característica importante de cualquier test unitario es que debe ser repetible. La ejecución de los tests, independientemente del entorno en el que se ejecuten debe terminar satisfactoriamente si el código de la prueba es correcto y fallar en caso contrario. Los factores ambientales externos tales como la configuración del entorno en el que se ejecutan las pruebas o la integración de software externo como base de datos no deben influir en la en la correcta ejecución de los mismos.

El ejemplo propuesto para el tutorial es una aplicación sencilla que convierte Euros a pesetas , llamadme nostálgico si queréis :-)



No nos darán el premio al mejor diseño del año pero como ejemplo será suficiente. Recibimos por tanto un dato de entrada que será convertido y devolveremos un dato de salida. Tendremos una clase encargado de realizar el negocio de nuestra app y devolver el resultado de la operación realizada. Lo primero que voy a probar es que la clase tiene la propiedad en la que se retornará el resultado.

```
- (void) testConversorEurosShouldReturnNullForValuePtsWhenInitObject
{
    ConversorEuros *conversor = [[ConversorEuros alloc] init];
    XCTAssertNil(conversor.valuePts, @"Valor en pts no es Null cuando se inicializa el objeto");
}
```

Lógicamente tenemos errores de compilación ya que todavía no existe nuestra clase ConversorEuros. Recordad que estamos haciendo TDD y primero vamos escribiendo los test, lo que nos irá guiando para escribir nuestro código. Ahora nuestro test no funciona (red flag). El siguiente paso es refactorizar nuestro código para que el test pase satisfactoriamente (green flag). Creamos entonces ConversorEuros e incluimos en la declaración la propiedad valuePts.

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

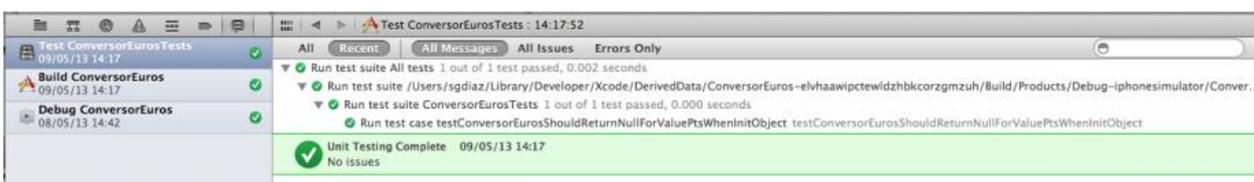
@interface ConversorEuros : NSObject
@property (nonatomic, retain) NSNumber *valuePts;
@end
```

Quedando la implementación:

```
#import "ConversorEuros.h"

@implementation ConversorEuros
@synthesize valuePts;
@end
```

Desde el test veremos que los errores de compilación se han solucionado así que vamos a ejecutar el test. Para ello solo tenemos que pulsar Product > Test. El resultado de la ejecución se muestra en la vista de "Log" del menú lateral de Xcode.



De manera sencilla hemos seguido el flujo de trabajo de TDD:

- Elegir un requisito: se elige de una lista el requisitos el que se creó que nos dará mayor conocimiento del problema y que a la vez sea fácilmente implementable.
- Escribir una prueba: se comienza escribiendo una prueba para el requisito. Para ello el programador debe entender claramente las especificaciones de la funcionalidad que va a implementar.
- Verificar que la prueba falla: si la prueba no falla es porque el requisito ya estaba implementado o porque la prueba es errónea.
- Escribir la implementación: escribir el código más sencillo que haga que la prueba funcione. Se usa la metáfora "Hazlo fácil" ("Keep It Simple, Stupid" (KISS)).
- Ejecutar las pruebas automatizadas: verificar si todo el conjunto de pruebas funciona correctamente.
- Refactorización: el paso final es la refactorización, que utilizaremos normalmente para eliminar código duplicado y dejar nuestro código más limpio y legible.
- Actualización de la lista de requisitos: se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requerimientos de diseño, por ejemplo, que una funcionalidad esté desacoplada de otra.

Seguimos escribiendo nuestro código escribiendo un nuevo test. En este caso probamos que el `ConversorEuros` es capaz de recibir un parámetro de entrada y convertirlo a pesetas.

```

- (void) testConversorEurosShouldReturnValuePtsFromEuros
{
    float number = 123.18;
    float euroDefault = 186.66;
    NSNumber *resultPts = [[NSNumber alloc] initWithFloat:number * euroDefault];

    NSNumber *eurosParameter = [[NSNumber alloc] initWithFloat:number];
    ConversorEuros *conversor = [[ConversorEuros alloc] init];
    [conversor convertirEuros: eurosParameter];
    XCTAssertTrue([conversor.valuePts isEqualToNumber: resultPts], [NSString stringWithFormat:@"No se
    han convertido correctamente. Expected: %@ Obtained: %@", resultPts, conversor.valuePts]);
}

```

Vemos que nuestro test esta marcado con "red flag" ya que el método no esta declarado en `ConversoEuros`. Realizamos las modificaciones oportunas en nuestro código:

```

#import <Foundation/Foundation.h>

@interface ConversorEuros : NSObject
    @property (nonatomic, copy) NSString *valuePts;
    -(void) convertirEuros: (NSNumber*) valueEuros;
@end

```

Y la implementación:

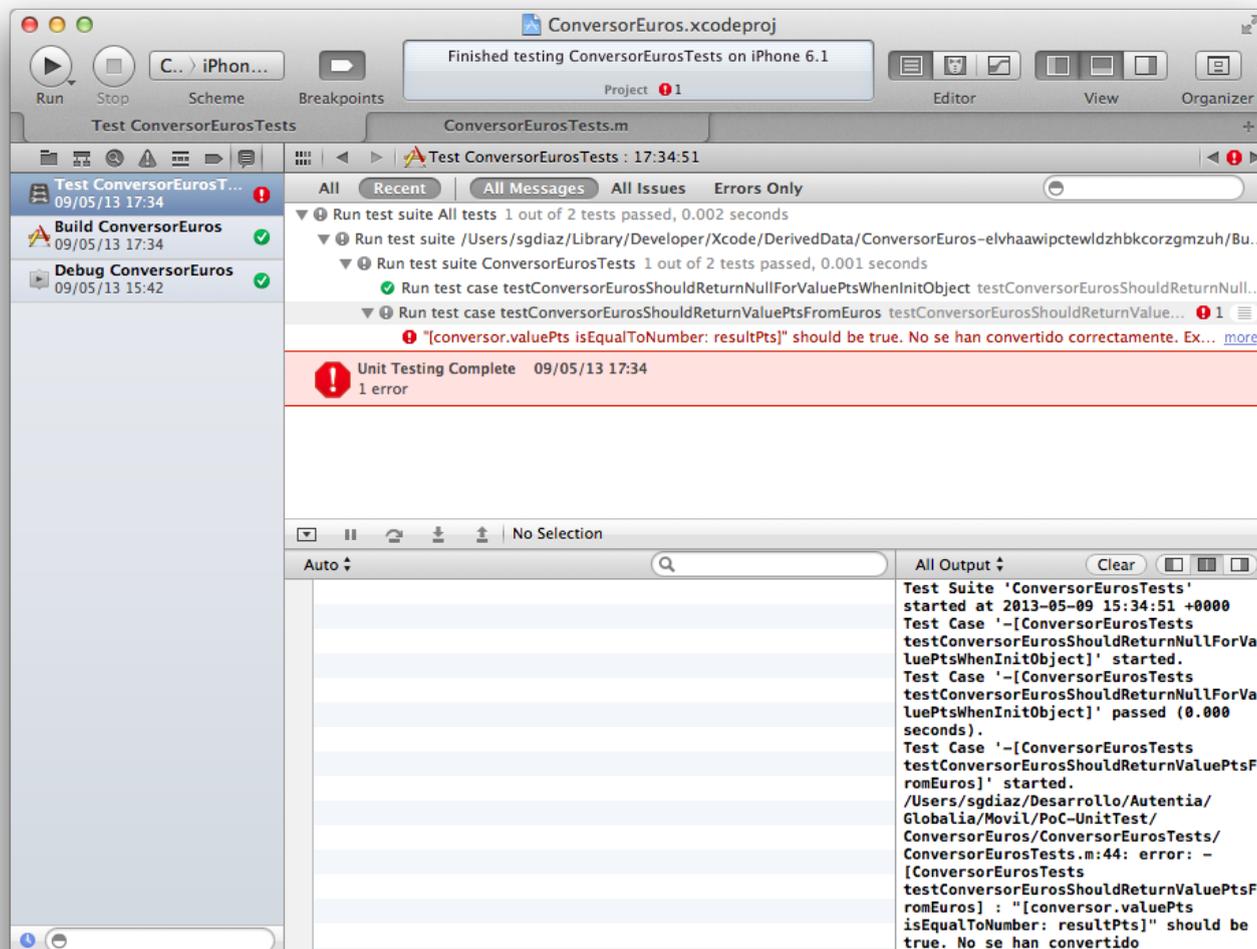
```

#import "ConversorEuros.h"

@implementation ConversorEuros
    @synthesize valuePts;
    -(void) convertirEuros: (NSNumber*) valueEuros{
        valuePts=[NSNumber numberWithInt:0];
    }
@end

```

A continuación ejecutamos los test, Product > Test



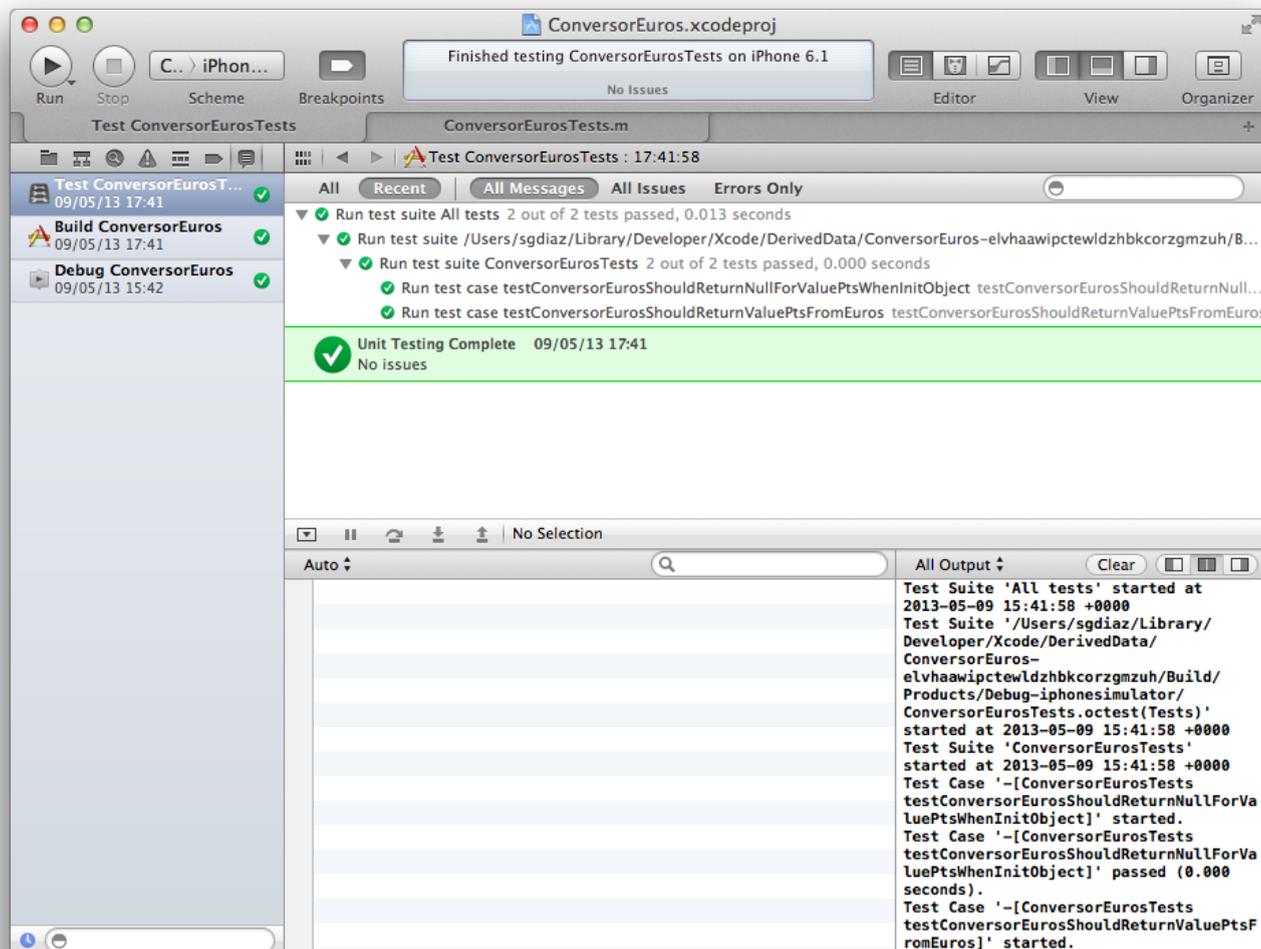
Podemos comprobar que el nuevo test falla, con lo que continuamos con el siguiente paso en flujo de TDD y refactorizamos nuestro código para que la prueba pase satisfactoriamente (green flag).

```
#import "ConversorEuros.h"

@implementation ConversorEuros
    @synthesize valuePts;
    - (void) convertirEuros: (NSNumber*) valueEuros{

        NSNumber *eurosDefaultValue = [[NSNumber alloc] initWithFloat:UN_EURO_PESETAS];
        self.valuePts = [NSNumber numberWithIntFloat: valueEuros.floatValue * eurosDefaultValue.floatValue];
    }
@end
```

Volvemos a lanzar los tests y voilà , los tests se han ejecutado satisfactoriamente (green flag):



Así podríamos seguir escribiendo los tests y por tanto diseñando y escribiendo el código de nuestra aplicación para cada una de las piezas que la conforma. Esto es un ejemplo y por eso no voy a incluir en el tutorial los tests del resto de las piezas que conforman la app, como podría ser los tests para el ViewController. Sin embargo es importante que sepamos que "asserts" disponemos para validar el resultado de nuestro código.

Table 4.1 The Macros Made Available to Unit Tests by OCUit, and the Conditions Needed to Pass the Test in Each Case

Test Macro	Success Criteria
<code>STAssertTrue(expression, msg, ...)</code>	Expression does not evaluate to 0.
<code>STAssertEqualObjects(a1, a2, msg, ...)</code>	Either the object pointers <code>a1</code> and <code>a2</code> refer to the same object (<code>a1 == a2</code>), or <code>[a1 isEqual: a2] == YES</code> .
<code>STAssertEquals(a1, a2, msg, ...)</code>	The arguments <code>a1</code> and <code>a2</code> are C datatypes (for example, primitive values or structs) of the same type with equal values.
<code>STAssertEqualsWithAccuracy(a1, a2, accuracy, msg, ...)</code>	The C scalar values <code>a1</code> and <code>a2</code> are of the same type and have the same value to within \pm <code>accuracy</code> .
<code>STFail(msg, ...)</code>	Never successful.
<code>STAssertNil(a1, msg, ...)</code>	The object <code>a1</code> is <code>nil</code> .
<code>STAssertNotNil(a1, msg, ...)</code>	The object <code>a1</code> is not <code>nil</code> .
<code>STAssertTrueNoThrow(expression, msg, ...)</code>	Expression does not evaluate to 0 and does not throw an exception.
<code>STAssertFalse(expression, msg, ...)</code>	Expression does evaluate to 0.
<code>STAssertFalseNoThrow(expression, msg, ...)</code>	Expression evaluates to 0 and does not throw an exception.
<code>STAssertThrows(expression, msg, ...)</code>	Expression must throw an exception.
<code>STAssertThrowsSpecific(expression, exception, msg, ...)</code>	Expression must throw an exception of the same class as the <code>exception</code> parameter, or a subclass of that class. In other words, <code>[expression isKindOfClass: exception]</code> must be true.
<code>STAssertThrowsSpecificNamed(expression, exception, name, msg, ...)</code>	Expression must throw an exception of the same class as or a subclass of the <code>exception</code> parameter, and with the name passed in the <code>name</code> parameter.
<code>STAssertNoThrow(expression, msg, ...)</code>	Expression does not throw an exception.
<code>STAssertNoThrowSpecific(expression, exception, msg, ...)</code>	Expression either doesn't throw an exception, or if it does, the exception isn't an instance of the <code>exception</code> parameter or its subclasses.
<code>STAssertNoThrowSpecificNamed(expression, exception, name, msg, ...)</code>	Expression either doesn't throw an exception or throws one that does not have the same type and name as the <code>exception</code> and <code>name</code> parameters.

Os dejo el [enlace](#) a la documentación oficial.

5. Alternativas OCUit y complementos

En este punto comentaré brevemente algunas alternativas para OCUit y también un complemento indispensable como OCMock para poder probar nuestras aplicaciones Apple.

- 1 - Google Toolkit for Mac : el kit de herramientas de Google para Mac (GTM) esta lleno de utilidades interesantes y útiles que aumentan las capacidades de prueba para iOS. Las unidades descritas en [GTM Testing](#) sólo es una de sus características. GTM nos proporciona una colección de macros adicionales que permiten entre otras cosas que los métodos de prueba sean más cortos y más descriptivos de lo que son cuando se escribe para los test con OCUit. También proporciona "mock object" para verificar que los mensajes de registro coincide con lo que cabría esperar, y cuenta con categorías para las pruebas de gráficos e imágenes.

Table 4.2 Test Assertion Macros Provided by GTM's Unit Testing Capabilities

Test Macro	Success Criteria
<code>STAssertNoErr(expression, msg, ...)</code>	Expression is an <code>OSStatus</code> or <code>OSErr</code> equal to the constant <code>noErr</code> .
<code>STAssertErr(expression, err, msg, ...)</code>	Expression is an <code>OSStatus</code> or <code>OSErr</code> equal to the value of <code>err</code> .
<code>STAssertNotNULL(expression, msg, ...)</code>	Expression is a pointer, the value of which is not <code>NULL</code> .
<code>STAssertNULL(expression, msg, ...)</code>	Expression is a pointer, the value of which is <code>NULL</code> .
<code>STAssertNotEquals(a1, a2, msg, ...)</code>	The C types <code>a1</code> and <code>a2</code> are not equal.
<code>STAssertNotEqualObjects(a1, a2, msg, ...)</code>	The Objective-C objects <code>a1</code> and <code>a2</code> are not equal.
<code>STAssertOperation(a1, a2, op, msg, ...)</code>	The expression <code>a1 op 'a2'</code> must be true, where <code>a1</code> and <code>a2</code> are simple C types. E.g. if <code>op</code> is <code>&</code> , then <code>a1 & a2</code> must not be equal to 0.
<code>STAssertGreaterThan(a1, a2, msg, ...)</code>	<code>a1 > a2</code>
<code>STAssertGreaterThanOrEqual(a1, a2, msg, ...)</code>	<code>a1 >= a2</code>
<code>STAssertLessThan(a1, a2, msg, ...)</code>	<code>a1 < a2</code>
<code>STAssertLessThanOrEqual(a1, a2, msg, ...)</code>	<code>a1 <= a2</code>
<code>STAssertEqualStrings(a1, a2, msg, ...)</code>	The <code>NSString</code> instances <code>a1</code> and <code>a2</code> represent the same sequence of characters.
<code>STAssertNotEqualStrings(a1, a2, msg, ...)</code>	The <code>NSString</code> instances <code>a1</code> and <code>a2</code> represent different sequences of characters.
<code>STAssertEqualCStrings(a1, a2, msg, ...)</code>	The C strings <code>a1</code> and <code>a2</code> represent the same sequence of characters.
<code>STAssertNotEqualCStrings(a1, a2, msg, ...)</code>	The C strings <code>a1</code> and <code>a2</code> represent different sequences of characters.

- 2 - GHUnit: El marco [GHUnit](#) está diseñado con [OCUnit](#) y [Google compatibilidad Toolkit](#) en mente. De hecho, es posible tomar conjuntos de pruebas escritas para cualquier framework de pruebas y utilizarlos en [GHUnit](#) sin modificaciones. La principal característica de [GHUnit](#) es que posee un frontend personalizado tanto para [Mac](#) y [iOS](#), que proporciona la capacidad de filtrar los resultados de pruebas basadas en palabras clave, y ofrece un mayor control sobre la presentación de los resultados de las pruebas que [Xcode](#) permite. Originalmente, se ejecuta en una aplicación, lo que hace más fácil depurar las pruebas unitarias de lo que es posible con [OCUnit](#). El [GHUnit](#) GUI para [iOS](#) se muestra en siguiente imagen.



- 3 - [OCMock](#) : [OCMock](#) es un framework para la creación sencilla de objetos simulados. Utiliza las capacidades de

introspección de Objective-C en tiempo de ejecución para crear automáticamente objetos simulados que pueden sustituir a las instancias de los objetos reales. Al usar "mocks" en nuestros tests, crearemos objetos indicando qué métodos pueden ser llamados, cuáles deben ser los parámetros y el valor que deben ser devueltos. En la fase de verificación, evaluará la llamada a los métodos previstos (y que no ejecutó nada mas) con los parámetros configurados previamente. Este es un complemento imprescindible para el desarrollo de aplicaciones basados en pruebas (TDD).

6. Conclusiones

La conclusión, no tenemos excusa para no probar nuestros desarrollos en la plataforma de la manzanita dando un valor añadido a las misma y a los usuarios que la utilicen ya que nuestra aplicación será mucho más fiable al haber sido diseñada mediante TDD.

Espero que este tutorial os haya sido de ayuda. Un saludo.

Saúl García Díaz
sgdiaz@autentia.com

A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)

Por favor, vota +1 o compártelo si te pareció interesante

Share | 0

Anímate y coméntanos lo que pienses sobre este **TUTORIAL**:

» **Regístrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo licencia [Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

PUSH THIS Page Pushers Community Help?

---- 0 people brought clicks to this page

no clicks + + + + + + + +

powered by [karmacracv](#)

