Twoixted againLet's get twoixted again…

- Pages
  - Why Twoixter?
- Categories
- RSS

Test Driven Development

Test Driven Development (TDD), o como se traduce en Español "Desarrollo Guiado por Pruebas" es una práctica de programación muy usada en la metodología Agile Development. Podeis encontrar más información en la Wikipedia en Español o en Inglés.

Lo que pretendo en este post es "guiar" o "introducir" TDD para aquellos que no comprenden del todo el concepto. Para entender TDD, debes saber que requiere escribir las pruebas PRIMERO, no DESPUES, y enfatiza la refactorización para conseguir todo esto. ¿Cómo se pueden escribir las pruebas primero? Si estás acostumbrado a hacer las pruebas (en caso de que las hagas) después de escribir el código, este concepto te será raro. Si las pruebas se hacen después del código, estás comprobando que el código funciona pero dicho código no está inducido por las pruebas, simplemente has hecho un "test" para comprobar que lo que has hecho es correcto.

Puede parecer una perogrullada, pero esto no es lo mismo. TDD hace posible que tu código esté guiado por las pruebas inciales, que al fin y al cabo es un conjunto de requerimientos. Es decir, en TDD empiezas primero por definir el comportamiento al que debes adherirte y asegurarte de cumplir estos comportamientos. Al hacer primero los tests, se evitan una serie de comportamientos y se aseguran una serie de ventajas: por ejemplo el sobredimensionamiento del código, sólo implementamos lo suficiente para pasar los test (cumplir los requerimientos) y también obliga al programador primero a entender bien el problema a resolver y a pensar como cliente al enfocarse en los interfaces.

Como se que todo esto está muy bien, pero es difícil entenderlo, voy a hacer un pequeño ejercicio de TDD para que lo comprendais mejor.

Números de Fibonacci

Los requerimientos son hacer una clase en C++ con un método que devuelve un número de la serie de Fibonacci. Bueno, tampoco vamos a controlar una central nuclear, así que el ejemplito clásico de la serie de Fibonacci para un ejemplo de TDD es perfecto. He utilizado gcc en OSX, los ejemplos deberían funcionar también en Linux.

Primero tenemos que usar alguna herramienta para hacer los tests. Para este ejemplo yo he creado la mía, que simplemente es una fución "test" que le paso una cadena y un "bool" como resultado de la comprobación de la prueba. En un entorno real se debería usar una librería de tests, aunque como podeis ver tampoco hace falta nada del otro mundo para hacer una batería de tests.

```
#include &lt;stdio.h&gt;
#include "fibb.h"

void test(char *str, bool pass)
{
    printf("%-11s %s\n", pass ? "OK" : "***ERROR***", str);
}

int main()
{
    Fibb f;

    test("Fibb debe existir.", true);

    return 0;
}
```

Vale, he incluido la cabecera de mi "futura" librería que crea un objeto "Fibb". Como todabía no la he creado (recordad, los test se hacen ANTES del código) ya tengo mi primer requerimiento: "*Fibb debe existir*".

Como era de esperar, esto falla:

```
Twoixter:pruebas josemiguel$ make
g++    -c -o test.o test.cpp
test.cpp:3:18: error: fibb.h: No such file or directory
test.cpp: In function 'int main()':
test.cpp:12: error: 'Fibb' was not declared in this scope
test.cpp:12: error: expected `;' before 'f'
test.cpp:15: error: 'f' was not declared in this scope
make: *** [test.o] Error 1
Twoixter:pruebas josemiguel$ _
```

Se me olvidaba, este es el fichero Makefile:

```
.SUFFIXES:
.SUFFIXES: .cpp .o
.PHONY: clean

all: test

test: test.o fibb.o
        $(CXX) -o test $^

clean:
        -rm *.o
```

Y esta es la clase esqueleto para hacer pasar el primer test: Vamos a crear el objeto "Fibb":

```
/* Fichero: fibb.h */
class Fibb {
public:
        int dame(int numero);
};
```

```
/* Fichero: fibb.cpp */
#include "fibb.h"

int Fibb::dame(int no)
{
    return 0;
}
```

Bueno, ya tenemos todo en su sitio y ahora vamos a ver qué pasa:

```
Twoixter:pruebas josemiguel$ make
g++    -c -o test.o test.cpp
g++    -c -o fibb.o fibb.cpp
g++ -o test test.o fibb.o
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
Twoixter:pruebas josemiguel$ _
```

Bueno, como suponíamos, ahora pasa el test. Nuestro requerimiento se ha cumplido.

Fijaros que aún no hemos hecho nada para calcular la serie de Fibonacci, estamos haciendo requerimientos y los estamos cumpliendo programando nuestra clase para que pase los test. Esta es la clave del Test Driven Development no hacemos nada aparte de cumplir nuestros tests. Recordad, es Programación Dirigida por Pruebas, son los test los que nos indican qué tenemos que hacer.

Vamos a añadir unos cuantos tests más. Vamos a añadir unos cuantos requerimientos a nuestra "*calculadora fibonacci*":

```
#include &lt;stdio.h&gt;
#include "fibb.h"

void test(char *str, bool pass)
{
    printf("%-11s %s\n", pass ? "OK" : "***ERROR***", str);
}

int main()
{
    Fibb f;
```

```
    test("Fibb debe existir.", true);
    test("Fibb.dame(0) debe ser 0.", f.dame(0) == 0);
    test("Fibb.dame(1) debe ser 1.", f.dame(1) == 1);

    return 0;
}
```

Es importante hacer notar aquí que, obviamente aparte de que cumplan con los detalles del problema, tenemos que procurar hacer los test para que fallen. En nuestro caso, la clase Fibb está vacía, sólo devuelve 0. Nuestro primer requerimiento (fibb de 0 == 0) por razones obvias va a cumplirse, pero a partir de aquí, los demás requerimientos SABEMOS que van a fallar, hacemos los test sabiendo que van a fallar y nuestro cometido es hacer que pasen.

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Fibb.dame(0) debe ser 0.
***ERROR*** Fibb.dame(1) debe ser 1.
```

Como esperábamos, el número 0 de la serie es 0, pero al comprobar el número 1 de la serie no es 1, como debería.

Pues venga, vamos a cumplirlo:

```
/* fichero: fibb.cpp */
#include "fibb.h"

int Fibb::dame(int numero)
{
    if (numero == 1) return 1;

    return 0;
}
```

Ya está. Perfecto. Nuestro programa pasa los tests…

```
Twoixter:pruebas josemiguel$ make
g++    -c -o fibb.o fibb.cpp
g++ -o test test.o fibb.o
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
Twoixter:pruebas josemiguel$
```

Bueno, aquí el lector avezado empezará a decir: "*Espera, espera, no estás programando ninguna serie de Fibonacci, me estás mintiendo*". No, la respuesta es que **estamos dando respuesta a nuestros requerimientos**.

Si nuestros requerimientos fueran sólamente estos, ya habríamos terminado. No se, por ejemplo para un programa tonto que sólo saque los 2 primeros números de la serie de fibonacci esto bastaría. Las claves son las siguientes:

- Nuestra clase funciona **según los requerimientos** indicados.
- Nuestra clase no tiene ninguna funcionalidad extra, con lo cual **no está sobrecargado** con código sobrante.
- Si un futuro programador lee nuestro código, **sabe perfectamente** (por los requerimientos) lo que hace.

Vamos a seguir ampliando requerimientos porque "parece" que nos falta algo para que sea una serie real de Fibonacci. Vamos a incluir lo siguiente:

1. Para cualquier número negativo, devuelve -1 (Esto es un poco arbitrario, lo ponemos como requerimiento).
2. Para el número 2, debe devolver 1.
3. Para el número 3, debe devolver 2.
4. Para el número 4, debe devolver 3.
5. Para el número 5, debe devolver 5.

Venga, manos a la obra. Recordad PRIMERO hacemos los TEST para que FALLEN…

```
/* fichero: test.cpp */
#include &lt;stdio.h&gt;
#include "fibb.h"

void test(char *str, bool pass)
{
    printf("%-11s %s\n", pass ? "OK" : "***ERROR***", str);
```

```
}

int main()
{
    Fibb f;

    test("Fibb debe existir.", true);
    test("Para cualquier negativo, debe ser -1", f.dame(-1) == -1);
    test("Fibb.dame(0) debe ser 0.", f.dame(0) == 0);
    test("Fibb.dame(1) debe ser 1.", f.dame(1) == 1);
    test("Fibb.dame(2) debe ser 1.", f.dame(2) == 1);
    test("Fibb.dame(3) debe ser 2.", f.dame(3) == 2);
    test("Fibb.dame(4) debe ser 3.", f.dame(4) == 3);
    test("Fibb.dame(5) debe ser 5.", f.dame(5) == 5);

    return 0;
}
```

…como era de esperar:

```
Twoixter:pruebas josemiguel$ make
g++    -c -o test.o test.cpp
g++ -o test test.o fibb.o
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
***ERROR*** Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
***ERROR*** Fibb.dame(2) debe ser 1.
***ERROR*** Fibb.dame(3) debe ser 2.
***ERROR*** Fibb.dame(4) debe ser 3.
***ERROR*** Fibb.dame(5) debe ser 5.
Twoixter:pruebas josemiguel$ _
```

Vale, tenemos que hacer cumplir estos tests… así que modificamos el programa principal de esta forma:

```
/* fichero: fibb.cpp */
#include "fibb.h"

int Fibb::dame(int numero)
{
    switch (numero) {
        case -1: return -1;
        case 1: return 1;
        case 2: return 1;
        case 3: return 2;
        case 4: return 3;
        case 5: return 5;
    }

    return 0;
}
```

¡Pero que estafa es esta! ¡Seguimos sin programar un generador de números de Fibonacci!. No, esperad, no funciona así… Estamos cumpliendo los requerimientos, y para los requerimientos que hemos puesto, este programa funciona perfectamente, como lo demuestran los tests:

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
OK          Fibb.dame(2) debe ser 1.
OK          Fibb.dame(3) debe ser 2.
OK          Fibb.dame(4) debe ser 3.
OK          Fibb.dame(5) debe ser 5.
Twoixter:pruebas josemiguel$ _
```

Como veis, hemos pasado todos los tests. Nuestro programa es simple, fácil de entender, y pasa los tests.

Moraleja importante: TDD, o Desarrollo Asistido/Guiado por Pruebas, basa todo en los test al contrario que en la forma tradicional de programación. Si no usaramos TDD, empezaríamos por hacer un generador de números de Fibonacci seguramente de forma recursiva. Nos centraríamos en cosas que no tienen que ver con los requerimientos. En este ejemplo de la serie, insisto, es muy básico pero creo que cumple perfectamente con el objetivo de ver cuán diferente

puede ser esta metodología de programación con respecto al método “clásico”.

Como veis, los detalles de implementación pasan a un plano secundario y lo importante son los requerimientos, el comportamiento que queremos que tenga nuestro programa. Con una batería de test correcta, con todos los requerimientos bien definidos, la implementación pasa a un segundo plano.

Hablando de requerimientos. Hay un fallo importante en los test, si os fijais, el test de los negativos pone “*Para CUALQUIER negativo*“, y sin embargo sólo comprobamos con menos uno. Vamos a cumplimentar mejor la batería de tests:

```
/* fichero: test.cpp */
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include "fibb.h"

void test(char *str, bool pass)
{
    printf("%-11s %s\n", pass ? "OK" : "***ERROR***", str);
}

int main()
{
    Fibb f;

    test("Fibb debe existir.", true);
    test("Para cualquier negativo, debe ser -1", f.dame(-rand()) == -1);
    test("Fibb.dame(0) debe ser 0.", f.dame(0) == 0);
    test("Fibb.dame(1) debe ser 1.", f.dame(1) == 1);
    test("Fibb.dame(2) debe ser 1.", f.dame(2) == 1);
    test("Fibb.dame(3) debe ser 2.", f.dame(3) == 2);
    test("Fibb.dame(4) debe ser 3.", f.dame(4) == 3);
    test("Fibb.dame(5) debe ser 5.", f.dame(5) == 5);

    return 0;
}
```

Hemos cambiado el test para **cualquier negativo** incluyendo un número aleatorio. No es estrictamente “científico”, porque en una pasada de test no podemos comprobar TODOS los números negativos. Para la mayoría de propósitos, un número aleatorio en un rango suficientemente grande nos asegurará que en cada ejecución de los test tengamos muchas posibilidades de que falle el test.

Moraleja: Como veis, seguimos enfocados en que los test fallen. Sería una perdida de tiempo en TDD hacer test para cosas que sabemos que funcionan, o cosas redundantes. Al enfocarnos en hacer test que fallen vamos “dirigiendo” nuestros esfuerzos a mejorar el desarrollo.

El test de los números negativos, como suponíamos, falla:

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
***ERROR*** Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
OK          Fibb.dame(2) debe ser 1.
OK          Fibb.dame(3) debe ser 2.
OK          Fibb.dame(4) debe ser 3.
OK          Fibb.dame(5) debe ser 5.
Twoixter:pruebas josemiguel$ _
```

Ahora introducimos un concepto importante en TDD, la **refactorización**. Una vez que nuestro código pasa los test, debemos refactorizar. Refactorizar es cambiar la programación por cualquier motivo, por ejemplo para que sea más eficiente, más rápido, o hacer el código más simple, pero siempre teniendo la seguridad de pasar los tests.

En nuestro caso, hacer que valide el test de los negativos no sería una refactorización en sí, puesto que hay un test que falla y tenemos que enfocarnos en que valide. Sí que hacemos una refactorización para hacer que lo que antes era un “switch”, pase a ser una tabla.

```
/* fichero: fibb.cpp */
#include "fibb.h"

int Fibb::dame(int numero)
{
```

```
    int tabla_fibb[6] = { 0, 1, 1, 2, 3, 5 };

    if (numero &lt; 0) return -1;
    if (numero &gt; 5) return 0;
    return tabla_fibb[numero];
}
```

Vale, hemos cumplido el test de los negativos haciendo que cualquier número menor que 0 devuelva -1. Después también ha habido la refactorización importante de pasar de un switch a una tabla. Ahora todos los tests pasan, incluso el de los negativos:

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
OK          Fibb.dame(2) debe ser 1.
OK          Fibb.dame(3) debe ser 2.
OK          Fibb.dame(4) debe ser 3.
OK          Fibb.dame(5) debe ser 5.
Twoixter:pruebas josemiguel$ _
```

Bien, como decíamos antes, si nuestros requerimientos fueran estos, ya habríamos terminado. **¡Y sin hacer el algoritmo de Fibonacci!**. Merece la pena recapitular lo que hemos visto hasta ahora:

- Programar usando Test Driven Development significa "dar la vuelta" a la forma de pensar cuando programamos normalmente, ya que PRIMERO se hacen los test (pruebas) en forma de requerimientos.
- Los requerimientos por tanto **deben ser sólidos**, y estar bien fundados ya que nuestro programa va a ser una representación literal de esos requerimientos.
- Los pasos que debemos dar por tanto son: *test* > *implementación* > *probar test* > *refactorización*. Y así continuamente hasta que todos los requerimientos se cumplan.

Para terminar, y como tengo la sensación de que a esta serie de Fibonacci le falta algo, imaginemos que una vez hecho todo lo anterior y ya estamos contentos (nuestro programa cumple con los requerimientos), viene el "jefe" y nos dice:

- No, no, hasta 5 no, debe sacar los números de la serie hasta 40 como mínimo.
- A ver, a ver, entonces cómo se hace eso?
- Muy fácil, un número "n" de la serie es la suma del número "n-1″ más "n-2″…
- Ahhhh… Vale.

Entonces, hacemos el siguiente test:

```
/* fichero: test.cpp */
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include "fibb.h"

void test(char *str, bool pass)
{
    printf("%-11s %s\n", pass ? "OK" : "***ERROR***", str);
}

int main()
{
    Fibb f;

    test("Fibb debe existir.", true);
    test("Para cualquier negativo, debe ser -1", f.dame(-rand()) == -1);
    test("Fibb.dame(0) debe ser 0.", f.dame(0) == 0);
    test("Fibb.dame(1) debe ser 1.", f.dame(1) == 1);
    test("Fibb.dame(2) debe ser 1.", f.dame(2) == 1);
    test("Fibb.dame(3) debe ser 2.", f.dame(3) == 2);
    test("Fibb.dame(4) debe ser 3.", f.dame(4) == 3);
    test("Fibb.dame(5) debe ser 5.", f.dame(5) == 5);

    int n = 20;
    test("Fibb de 'n' debe ser fibb(n-1) + fibb(n-2)", f.dame(n) == f.dame(n-1) + f.dame(n-2) );

    return 0;
}
```

Incluimos un último test donde decimos exactamente eso, que el número "n" de la serie es la suma del n-1 más n-2. Si

corremos la batería de tests, ocurre esto:

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
OK          Fibb.dame(2) debe ser 1.
OK          Fibb.dame(3) debe ser 2.
OK          Fibb.dame(4) debe ser 3.
OK          Fibb.dame(5) debe ser 5.
OK          Fibb de 'n' debe ser fibb(n-1) + fibb(n-2)
Twoixter:pruebas josemiguel$ _
```

¿Comorrrlll? ¿Ha pasado la prueba? Bieennn… Nuestro programa funciona para cualquier número natural positivo con sólo una tabla de 6 números. Bueno, evidentemente, esto está mal. Y está mal porque si recordais, hemos dicho antes que los tests tienen que hacerse PARA QUE FALLEN inicialmente. Si no, pasan estas cosas.

Nota: La explicación de por qué pasa el test debe ser evidente, pero si no, lo que ocurre es que nuestra función devuelve 0 para cualquier número mayor de 5. Entonces, 0 = 0 + 0.

Tenemos MAL el test.

Deberíamos replantear el test de esta forma: "*Fibb de 'n' debe ser fibb(n-1) + fibb(n-2) y* ***mayor que cero***".

```
Twoixter:pruebas josemiguel$ ./test
OK          Fibb debe existir.
OK          Para cualquier negativo, debe ser -1
OK          Fibb.dame(0) debe ser 0.
OK          Fibb.dame(1) debe ser 1.
OK          Fibb.dame(2) debe ser 1.
OK          Fibb.dame(3) debe ser 2.
OK          Fibb.dame(4) debe ser 3.
OK          Fibb.dame(5) debe ser 5.
***ERROR*** Fibb de 'n' debe ser igual a fibb(n-1) + fibb(n-2) y &gt;0
Twoixter:pruebas josemiguel$ _
```

Vale, ahora podemos ponernos manos a la obra:

```
/* fichero: fibb.cpp */
#include "fibb.h"

int Fibb::dame(int numero)
{
    if (numero &lt; 0) return -1;
    if (numero == 0) return 0;
    if (numero == 1) return 1;

    return dame(numero-1) + dame(numero-2);
}
```

Que es un algoritmo de Fibonacci más o menos estandard. (Bueno, no me critiqueis mucho, esto es un post sobre Test Driven Develpment, no sobre cómo hacer un algoritmo de Fibonacci)

De hecho, habiendo refactorizado lo anterior, pasan todos los test. Ahora podríamos hacer un test que comprobara uno a uno todos los números de la serie hasta el 40, que es nuestro tope.

Espero que hayais leido hasta aquí, este ha sido uno de mis posts más largos. La intención ha sido hacer entender de una forma práctica las bases de TDD. En futuros posts, pondré enlaces más interesantes sobre frameworks y librerías que podemos usar para hacer tests y usaré otros lenguajes aparte de C++, lo prometo.

Tags: Agile Programming, TDD

This entry was posted on Domingo, Junio 14th, 2009 at 12:38 am and is filed under C++, OSX, Programación. You can follow any comments to this entry through the RSS 2.0 feed. You can leave a comment, or trackback from your own site.

One comment

1. *Juanjo Falcon*

9 de Octubre de 2009 en 5:45 pm

Me ha gustado mucho la explicacion y como lo has planteado. Me parece interesante tambien, que mediante un ejemplo indiques la confianza que te da tener los tests.

Por ejemplo si implementas fibb de otra manera estaras seguro que esta bien implementado si pasa los tests. En la siguiente presentacion http://www.testingtv.com/2009/08/31/tdd-code-without-fear/# hicieron esta prueba y me gusto mucho la sensacion que produce.

Enhorabuena por el blog…

Leave a comment

Name (required) Juanjo Falcon

Email (required) jjfalcon@somesi.com

Website http://www.somesi.com

Comment

Submit Comment

-

Archivos

- Septiembre 2009
- Agosto 2009
- Junio 2009
- Mayo 2009
- Marzo 2009
- Febrero 2009
- Enero 2009
- Diciembre 2008
- Noviembre 2008
- Octubre 2008
- Agosto 2008

Cosas varias

- Registrarse
- Iniciar sesión
- RSS de Entradas
- RSS de los comentarios
- WordPress.org