**Aplicación transferencia bancaria usando TDD**

**Requisitos**

El cliente quiere un programa que le permita transferir un importe desde una cuenta origen a una cuenta destino. Una vez realizada la transferencia, se tendrá que mostrar el nuevo balance de las cuentas.

Es necesario que las cuentas puedan expresarse tanto en euros como en dólares. El usuario no tendrá que preocuparse en ningún momento del cambio entre divisas, ya que será el sistema quien lo aplicará en cada transacción según el valor actual.

**Tests funcionales**

El personal técnico del cliente ha preparado una serie de pruebas funcionales que ayudarán a comprobar si la aplicación realiza lo esperado. Este tipo pruebas ve al software como una caja negra, ya que el único aspecto que considera es que se cumpla un determinado caso de uso.

Nuestra aplicación es muy básica, por lo que sólo tiene un caso de uso o requisito.

A continuación se muestra gráficamente el test funcional elaborado por el cliente. Supongamos que partimos de la siguiente situación:

**Cuenta origen 24.000,50 EUROS**

**Cuenta destino 46.620,00 DOLARES**

**Importe a transferir 8.290,10 EUROS**

Al transferir 8.290.10 euros, dado que la cuenta origen está en esa moneda y suponiendo que en ese momento 1 euro equivale a 1,48 dólares, el programa informará que una vez realizada la operación las cuentas quedarán de la siguiente manera:

**Cuenta origen**

**Cuenta destino**

El saldo resultante para la cuenta origen: 24.000,50€ - 8.290,10€ = **15.710,40€**

El saldo resultante para la cuenta destino: 8.290,10€ \* 1,48 = 12.269,35$;

por tanto, 46.620,00$ + 12.269,35$ = **58.889,35$**

**Tests unitarios**

Un test unitario consiste en una clase cuya única misión es probar que otra funciona correctamente. Por tanto, siempre tenemos la clase real (o de implementación) y la clase de test que opera sobre la real.

Lo que comprueba una clase de test es la interfaz pública de la clase real, es decir, que en función de una llamada y un (optativo) paso de parámetros se obtiene una determinada salida.

Una clase de test no comprueba directamente la implementación de la clase real, es decir, sus métodos privados o protegidos. El funcionamiento correcto de la implementación de una clase queda indirectamente comprobado a través de su interfaz, ya que ésta se basa en aquélla.

A grandes rasgos, si una clase real tiene 5 métodos públicos que forman su interfaz, la clase de test tendrá como mínimo 5 métodos para probar el funcionamiento de la clase real. Decimos “como mínimo”, ya que es normal tener más de un método en la clase de test para probar un método de la clase real. Por ejemplo, podríamos tener un método de test para probar que sucede si se le pasa un número muy grande, otro para ver qué ocurre si se le pasa un número muy pequeño, otro para ver qué sucede si se le pasa un nulo, etc.

Por tanto: una batería de tests unitarios consiste en un conjunto de métodos que prueban que los métodos de una clase funcionan correctamente.

Es muy importante destacar que los tests unitarios deben probar cada una de las funcionalidades que proporciona la clase de manera aislada, es decir, deben probar cada uno de sus métodos y lo deben hacer siguiendo un principio fundamental: la ejecución de un test no puede alterar el comportamiento de otros, ni directa ni indirectamente. Por tanto, si un test modifica el entorno, este entorno debe restaurarse a su estado original antes de que un nuevo test se ejecute.

También es muy importante que la clase se pruebe sin ninguno de sus colaboradores (dependencias; por ejemplo si probamos la clase Vehiculo, no podemos permitir que utilice la clase Matricula), ya que en caso contrario no estaríamos probando la clase de manera aislada, dado que los colaboradores podrían alterar los resultados.

Para que una clase pueda ser probada unitariamente, en lugar de probarla con sus colaboradores genuinos se utilizan los denominados stubs y mocks.

TDD y test unitarios

Los test unitarios son la base para el desarrollo de software conducido por test (TDD). La clase es la unidad funcional más básica en un lenguaje de programación orientado a objetos, es decir, es la unidad de trabajo y es precisamente por esto que las pruebas unitarias operan sobre las clases, para demostrar la robustez de las mismas. Una clase que haya pasado una buena batería de test, es una clase que nos dará confianza y que podremos reutilizar en tantos proyectos como necesitemos.

En la práctica del TDD no aplicamos los tests para comprobar si la clase que acabamos de construir funciona correctamente. TDD significa comenzar primero por crear un determinado test para un método de una clase que todavía no existe.

Esta forma de trabajar puede parecer desconcertante pero tiene beneficios innegables:

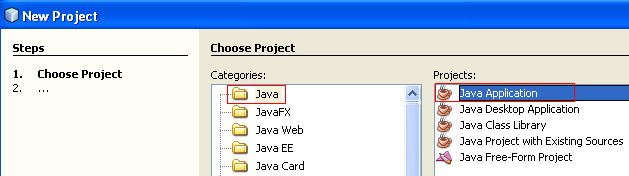
* Aplicaciones muy robustas que proporcionan al programador mucha paz. Se tiene la tranquilidad de que al añadir una nueva funcionalidad no se perderá el control de la aplicación aunque esta falle.
* Aplicaciones que no hacen nada que no sea estrictamente lo necesario (importantísimo).

La creación de tests unitarios pasa por una serie de iteraciones que van haciendo madurar los requisitos de negocio. Comenzamos por los test más básicos que se nos ocurran.

**Creación del proyecto**

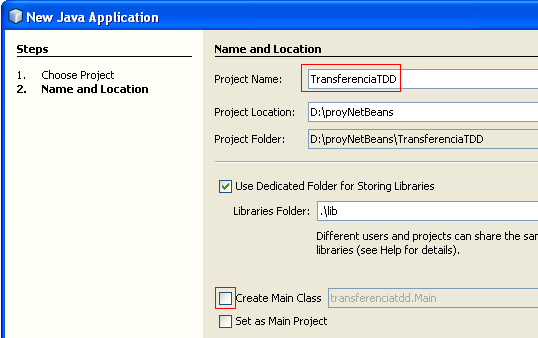
Para esta práctica nos basaremos en NetBeans, aunque Eclipse o cualquier otro IDE semejante nos valdría perfectamente. No obstante, NetBeans, al integrarlo todo de base ofrece un plus de comodidad para trabajar con pruebas unitarias.

Creamos un proyecto Java “normal”:



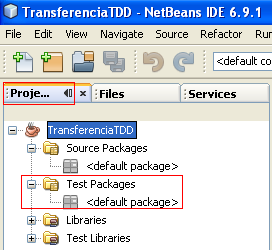
Pulsamos “Next”.

Proporcionamos un nombre para el proyecto y desmarcamos la opción de crear una clase Main.



Pulsamos “Finish”.

Las plantillas de proyecto de NetBeans contemplan el uso de pruebas unitarias como parte esencial de la construcción de las aplicaciones. Por tanto, podremos ver en la pestaña ‘Projects’ la existencia de la carpeta ‘Test Packages’ como una estructura paralela a la carpeta ‘Source Packages’, que es el lugar donde creamos las clases de implementación:



Cuando practicamos TDD, la estructura ‘Test packages’ cobra especial relevancia, pues es nuestro punto de partida en lo que se conoce como la triangulación: test, implementación, refactorización.

Test: Creación de un test para probar una característica de una clase. El test no compilará, ya que la característica todavía no existe.

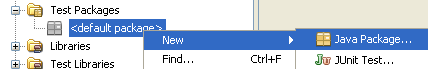
Implementación: Creación del código en la clase que permitirá que el test pase.

Refactorización: Ahora que todo funciona, se mira de mejorar el código elaborado, tanto de la implementación real como del test.

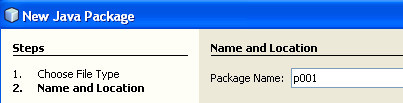
**Iteración 1**

Empezamos:

Situados en la estructura ‘Test Packages’ creamos un nuevo paquete:

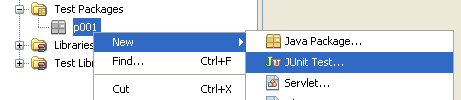


Como nombre proporcionamos ‘p001’:

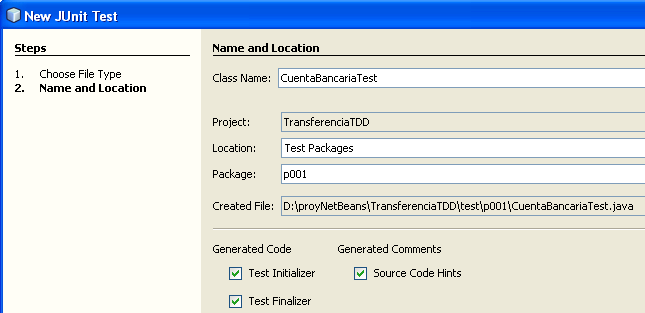


Finalizamos el asistente.

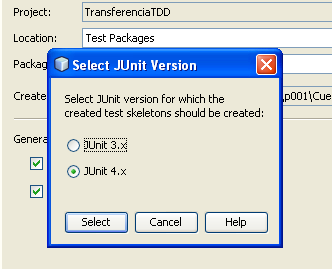
A continuación, situados en el paquete creado anteriormente, hacemos clic derecho y seleccionamos ‘New->Unit Test…’:



En el cuadro de diálogo que aparece proporcionamos como nombre para la clase: ‘CuentaBancariaTest’:



Finalizamos. En el cuadro de diálogo que aparece seleccionamos ‘JUnit 4.x’:



Pulsamos ‘Select’.

Veremos que el asistente de creación de clases de test ha generado una serie de métodos, cada uno con un anotación distinta: @BeforeClass, @AfterClass, etc.

Estas anotaciones las proporciona el framework JUnit y sirven para que éste sepa a qué secuencia dentro del ciclo de vida de una prueba pertenece un determinado método.

Veamos el significado de estas anotaciones:

@BeforeClass

El código que escribamos en un método prefijado con esta anotación se ejecutará antes de realizar el primer test de la batería de test. Esto suele ser el caso de la creación de algunos (costosos) objetos, de la apertura de ficheros, etc).

@AfterClass

Se ejecutará el código del método prefijado con esta anotación una vez finalizada la ejecución de todos los tests.

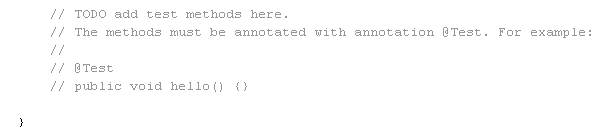
@Before

El código que escribamos en un método prefijado con esta anotación se ejecutará antes de realizar cada uno de los tests de la batería de test. En este método se restaura el entorno para cada nuevo test de la clase. Es frecuente, ubicar aquí la creación del SUT (Subject Under Test), es decir la instancia de la clase que estamos probando, para que cada test opere con un SUT ‘limpio’.

@After

Hace que el código para el método se ejecute siempre que finalice la ejecución de cada test de la clase.

Si nos fijamos en el código generado automáticamente por NetBeans veremos que al final de la clase de test aparece cierto código entre comentarios:

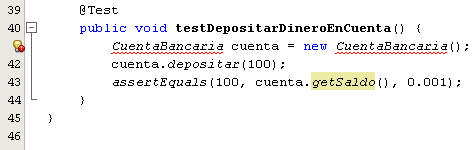


Esta es la signatura típica de un método de prueba, por lo que podemos quitar el comentario y utilizar este fragmento como punto de partida.

Nuestro primer test probará lo siguiente:

* Se crea una cuenta
* Se realiza un depósito
* Expectativa: El saldo recuperado de la cuenta debe coincidir con el depósito efectuado anteriormente.

Lo llamaremos testDepositarDineroEnCuenta y queda como sigue:



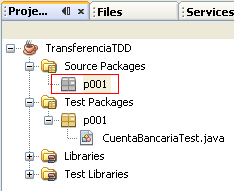
El método assertEquals() pertenece al framework JUnit y nos permite comparar dos números reales ignorando el caso de que entre ellos haya la diferencia establecida por el tercer parámetro (valor delta). Así, lo que le estamos diciendo en este caso es que compruebe si el valor 100 (valor esperado) es igual al valor devuelto por el método getSaldo() (valor real).

La documentación de JUnit dice lo siguiente:

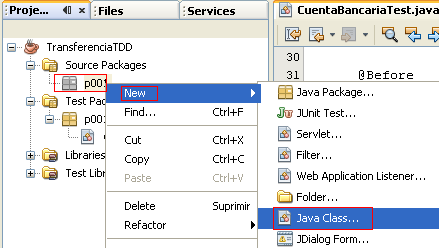
assertEquals(double expected, double actual, double delta) checks whether (Math.abs(expected-actual) <= delta) is true. This is becomes floating point variables don't work perfectly with decimal numbers, and you might get 0.9999999 where you'd expect 1. So if that's the case for you and it's OK then you can call assertEquals with a delta of 0.00000001.

Podemos observar cómo NetBeans señala en rojo la clase CuentaBancaria, indicando que es un tipo desconocido.

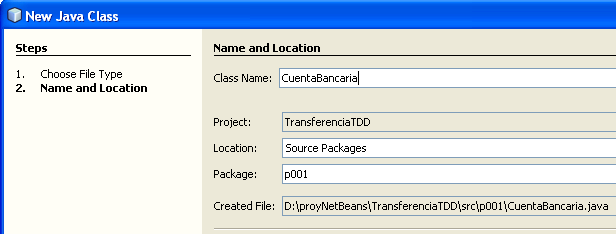
Bien, es lógico, ya que no aún no la hemos creado. Por tanto, aplicando la técnica de la triangulación, ahora lo que toca es hacer que el test pase, ya que por el momento ni siquiera compila. Por tanto, tenemos que proceder con la creación de la clase (el SUT). Para ello nos dirigimos a la estructura ‘Source Packages’ y creamos el paquete ‘p001’:



A continuación creamos la clase:

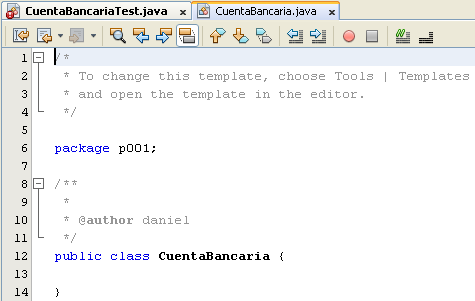


Como nombre tenemos que escribir ‘CuentaBancaria’:

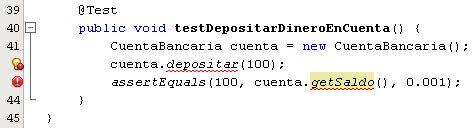


Finalizamos el asistente de creación de clases.

El asistente nos habrá creado algo como esto:

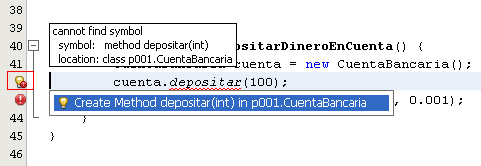


Bien, pues si ahora grabamos todo el proyecto (icono de los disquetes) y regresamos a la clase de test, comprobaremos que NetBeans ya no emite el mensaje de error sobre el tipo CuentaBancaria:

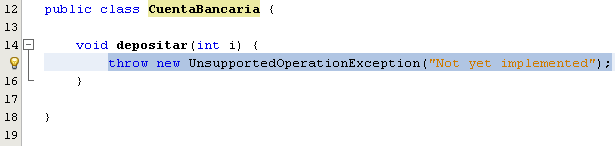


Ahora el error es relativo a los métodos depositar() y getSaldo(), lo cual era de esperar, ya que el SUT CuentaBancaria no tiene aún ninguno de estos métodos.

Por consiguiente, tenemos que crearlos para que el test pase. La idea es crear la implementación mínima para que los test se ejecuten correctamente, por lo que nos apoyaremos en el propio NetBeans, ya que los IDE son fantásticos para este tipo de trabajo. Hacemos clic derecho sobre el icono de la barra lateral izquierda con la bombilla y en el pop-up menú que aparece seleccionamos ‘Create Method depositar…’:

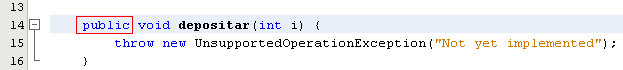


Esto habrá creado el método en la clase CuentaBancaria:



Se trata de una implementación trivial, pero que consigue que el test pase en lo que a él respecta y, precisamente, estamos en esa fase de la triangulación.

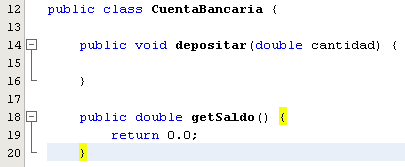
Un pequeño detalle: hacemos el método sea parte de la interfaz de la clase:



Y hacemos que reciba un parámetro ‘double’ y que no lance ninguna excepción:

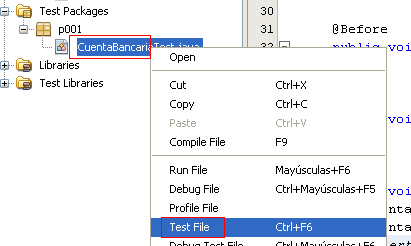
A description...

Ahora hacemos lo propio para el método getSaldo(). Sin embargo, al tratarse de un método de acceso -un getter- la mecánica para la generación de código difiere del caso anterior. Por el momento creamos el método manualmente, con lo que nos queda la clase de la siguiente manera:

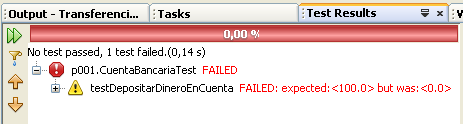


Notad lo rudimentaria que es la implementación de getSaldo(). Hemos hecho que siempre devuelva el valor 0.0. Esto es lo correcto por ahora, ya que sólo buscamos que el test compile.

Si grabamos todo y volvemos a la clase de test veremos que NetBeans ya no señala ningún problema en el código. Es momento de ejecutar el test unitario: hacemos clic derecho sobre la clase CuentaBancariaTest y seleccionamos ‘TestFile’

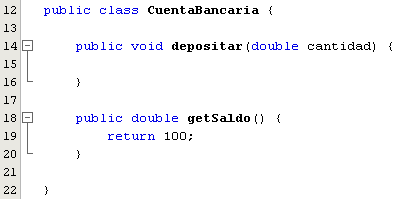


Si todo va según lo previsto obtendremos una ventana como la siguiente:



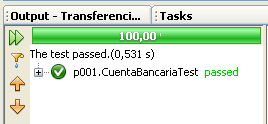
Vemos que el test ha detectado problemas (estamos en rojo). Notad que JUnit está indicando que se esperaba que el saldo fuera 100, en cambio el valor devuelto por el método del SUT ha sido 0. Por tanto, tenemos que revisar la clase CuentaBancaria para ver por qué el test no ha pasado.

No hay que ser un lince para ver que el problema es por el return 0.0 que habíamos escrito anteriormente a toda prisa para que el test compilara. Pero ahora que el test compila tenemos que procurar que además pase correctamente, por lo que realizamos el siguiente cambio en el método getSaldo() del SUT:



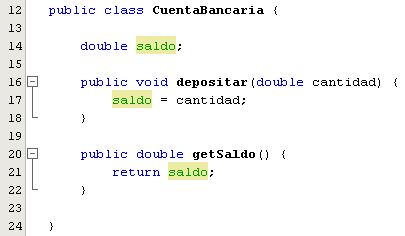
Lo anterior se denomina una implementación falsa, dado que sabemos que no es la correcta. Sin embargo, ahora es lo que toca, pues nuestro objetivo es que el test pase.

Si ahora ejecutamos el test, tendremos que ver en la ventana de salida algo parecido a lo siguiente:



Perfecto, ya tenemos nuestro primer test unitario en verde. Ahora, siguiendo con la técnica de la triangulación, tenemos que refactorizar, tanto el SUT como el test.

Por consiguiente, modificamos CuentaBancaria para que funcione de manera general y no sólo cuando se haga un importe de 100:



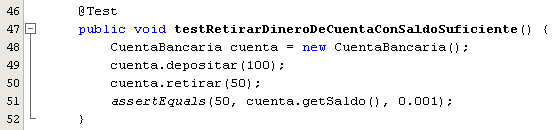
Notad que sigue siendo una implementación trivial, ya que en cada nuevo depósito estamos machacando el saldo que hubiera en ese momento por el recibido. No obstante, la implementación actual es la adecuada en estos momentos. Podemos estar tranquilos, porque más adelante tendremos un test que detectará esta circunstancia.

Vamos ahora con el reintegro. La primera prueba consistirá en efectuar un depósito y a continuación un reintegro cuya cantidad sea menor al depósito. Posteriormente, crearemos otro método de test para probar el caso de una retirada mayor que el depósito.

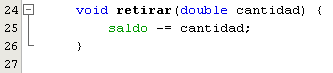
*Nota*

*A partir de ahora no se detallará tanto cada paso de la triangulación como se ha hecho hasta el momento, dado que una vez que se ha visto como se hace es fácil aplicarlo en nuestros desarrollos.*

El nuevo test queda como sigue:



Y el nuevo método del SUT queda así:



Ejecutamos los dos tets que llevamos hasta el momento:

