Test Driven Development

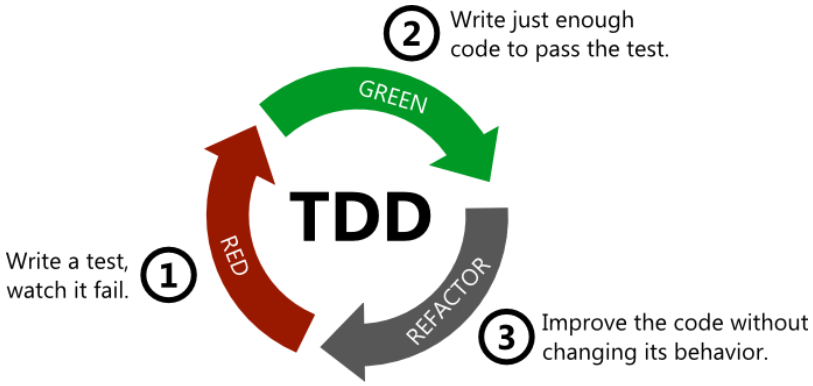
En este apartado vamos a resumir los elementos más importantes de Test Driven Development (TDD) y a presentar un ejemplo completo de desarrollo de código usando la práctica.

Introducción a TDD

La práctica de TDD fue introducida por Kent Beck como una de las prácticas propias de XP. Se denominó inicialmente *test-first programming*. Con el paso del tiempo se ha convertido en una de las prácticas más populares de la metodología y también es usada por desarrolladores y equipos que no utilizan el resto de técnicas de XP.

La técnica se basa en escribir el código de forma iterativa, paso a paso, y siempre comenzando por los tests.

Supongamos que tenemos que implementar una determinada funcionalidad. La analizamos y pensamos en cómo implementarla, dividiéndola en pequeños elementos necesarios para poder completarla. Escribimos estos elementos en una lista y empezamos a codificarlos utilizando el siguiente ciclo:

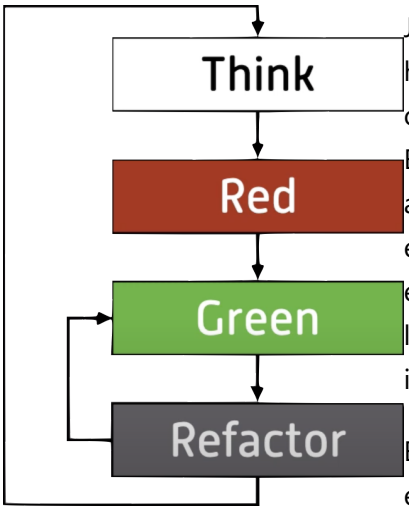


1. Se escribe un único test en el que se especifica lo que debe realizar el código para implementar el pequeño incremento de funcionalidad deseado. El test constituye un ejemplo funcionamiento en el que se invoca el código (que todavía no existe) y se realiza una aserción, una hipótesis, de qué debe suceder como resultado de la ejecución de ese código. Se añade el test a los ya existentes y se lanzan todos. El test que acabamos de añadir falla (rojo).
2. Se escribe únicamente el código necesario que hace que el test pase (verde). No hay que añadir código adicional, sólo el necesario para que el test pase.
3. Ahora que tenemos una base de tests que funciona perfectamente, se analiza el código y los tests, se detectan problemas de diseño y se realiza una refactorización (profundizaremos más adelante en esto). Se vuelven a lanzar los tests para asegurarnos que la refactorización no ha roto nada.

Se repite el ciclo, cada vez añadiendo nuevos tests que vayan acercando el código a la nueva funcionalidad. En el proceso de añadir pequeños tests y ampliar el código, nos iremos dando cuenta de cosas que hay que implementar y que no habíamos tenido en cuenta. Las anotamos en la lista de tareas a hacer y en algún momento tocará resolverlas con el ciclo de TDD.

Esta labor de diseño y de reflexión para elaborar el test mínimo y seleccionar cual es el siguiente test a implementar está implícita en el ciclo original anterior. Pero algún autor, como James Shore, la ha incluido en el propio ciclo, tal y como se ve en la imagen a la izquierda.

Este paso inicial de "pensar" consiste en analizar qué pequeño paso nos puede servir para avanzar en la funcionalidad que estamos desarrollando. Anotamos las ideas que se nos ocurran como pequeños pasos que hay que implementar. Se escoge el paso más básico, el inicial que sirve de base para todo lo demás.



James Shore dice que la base de TDD es la realización de una serie de hipótesis y su validación. ¿Qué quiere decir con eso? Cuando escribimos código siempre tenemos en la cabeza qué queremos que ese código haga. Eso es lo que Shore denomina una hipótesis. Es una predicción sobre qué va a pasar cuando el código se incorpore al programa. En TDD lo que hacemos es hacer explícitas estas hipótesis o predicciones en formas de tests. Y las escribimos antes que el propio código, con la idea de que al escribirlo y lanzar el test se valide la predicción que el test dice que el código va a implementar.

En este proceso pueden haber sorpresas. A veces el código no hace lo que esperamos que haga. O el test no expresa correctamente el propósito que pretendemos que haga el código. Las sorpresas son continuas cuando programamos. Significa que hemos escrito mal algo o que estamos haciendo suposiciones que no son correctas. La ventaja fundamental de TDD es que estas sorpresas aparecerán pronto en el proceso y tendré la obligación de corregirlas.

Los tests deben representar pequeños pasos hacia adelante

Los tests y el código que generan deben ser muy pequeños y concretos. Sólo se generalizará cuando se detecte que sea necesario. Se hará en la fase de refactorización. Es importante que los tests vayan haciendo crecer la funcionalidad en pequeños incrementos. Al ser los cambios pequeños hay menos posibilidades de estropear cosas.

Si el incremento de código es pequeño también es más sencillo encontrar un error cuando se algo no sale como esperábamos. Si te acostumbras a usar TDD verás que no será necesario utilizar el debugger para encontrar dónde están los errores. Los encontrarás rápidamente en el momento que los hayas cometido.

Esta idea de buscar el incremento más pequeño que puede hacer avanzar nuestro programa es una de las ideas más importante de TDD. Y se puede usar fuera de TDD. De hecho, Kent Beck ha creado recientemente una técnica basada en ella denominada TCR (*Test && Commit || Revert*).

En TCR se avanza el desarrollo del programa escribiendo código y haciendo tests (no es necesario escribir el test antes). Si el test pasa, automáticamente se hace un commit y el código se graba. Lo original de la técnica es lo que pasa cuando el test falla. Si el test falla, se revierte automáticamente el código al último commit, haciendo un `reset --hard`. Sí, has leído bien, el código se elimina. Desaparece. Borrado. Esto obliga a tener que volverlo a escribir.

Evidentemente la técnica te obliga a hacer incrementos pequeños validados continuamente con tests. ¡Nadie se arriesga a escribir un trozo grande de código y después perderlo!

Beck escribió esta técnica en septiembre de 2018 en un post titulado test && commit || revert. Después ha seguido publicando algún artículo más y algunos vídeos en YouTube demostrando el uso de la técnica en problemas sencillos. Por ejemplo Substring, TCR style o una serie de vídeos publicados en abril de 2020 explicando ejemplos de TCR en Python.

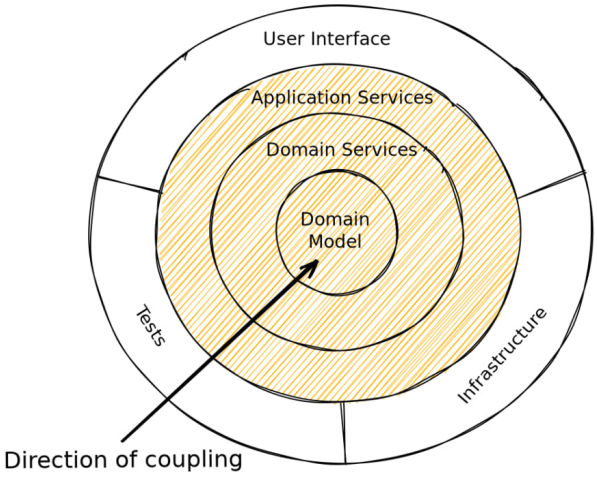
Una de las frases más usadas por Beck en los últimos años es la siguiente:

> "Make the change easy and then make the easy change"
>
> Kent Beck

Significa que cuando queramos hacer un cambio en nuestra aplicación debemos ir refactorizando poco a poco el programa de forma que los tests sigan funcionando y acercarnos hasta una situación en la que el programa sea fácil de cambiar y sea fácil introducir el cambio deseado.

Los tests se implementan de dentro a afuera

En TDD se utiliza un enfoque *bottom-up*: primero se programan los elementos y funcionalidades básicas y después se van programando funcionalidades de mayor nivel que se basan en las anteriores. Muchas veces se utiliza la metáfora de la cebolla. Las capas de la cebolla represen las capas de nuestro proyecto software. En TDD construiremos la cebolla de dentro a afuera: primero las capas más internas y después las superiores, basándonos en las ya construidas anteriormente.



En TDD no nos preocupamos demasiado de que los tests sean unitarios ni de usar mocks para aislar funcionalidades. Como la batería de tests se construye de forma incremental, cuando hacemos los tests de más alto nivel no hace falta aislar los elementos de bajo nivel porque ya hemos comprobado que funcionan correctamente.

En ocasiones sí que se usarán mocks por motivos de eficiencia y para que los tests pasen más rápido. Sobre todo en los casos de tests de integración en los que hay que conectar con servicios externos.

A la hora de implementar un algoritmo usando TDD se suelen definir los tests en el siguiente orden:

1. Interfaz central
2. Cálculos y condicionales
3. Bucles y generalizaciones
4. Casos especiales y manejo de errores

En primer lugar escribimos los tests que definen la interfaz central de nuestro problema: nombres de métodos, de parámetros, tipos de datos de los parámetros y de los valores devueltos, etc.

En segundo lugar escribimos tests que obligan a implementar la parte más sencilla del algoritmo, la que define los cálculos y los condicionales necesarios.

En tercer lugar pasamos a especificar los bucles y las generalizaciones. Podría ser que el cálculo del paso 2 lo hayamos definido para un solo elemento y ahora tengamos que hacer un bucle para aplicarlo a *n*. Y además puede que tengamos que generalizar para tratar distintos tipos de elementos. Con esto tendremos el algoritmo casi terminado a falta del paso 4 en el que trataremos los casos especiales y el manejo de errores.

En el vídeo de James Shore *Incremental Test Driven Develpment* se muestra esta técnica con un sencillo ejemplo de uso de TDD para implementar un algoritmo que codifica texto usando el algoritmo ROT13 en JavaScript.

Ventajas y críticas del uso de TDD

Utilizando esta técnica las pruebas no sólo sirven para comprobar que el software funciona correctamente, sino que sirven para especificarlo y diseñarlo.

La utilización de TDD tiene muchas ventajas:

- En todo momento tenemos una especificación clara de lo que nuestro código tiene que hacer.
- Cohesión y acoplamiento - si es difícil escribir el test, es una señal de que tenemos un problema de diseño. El código débilmente acoplado y altamente cohesionado es más fácil de probar.
- Confianza: escribiendo código limpio que funciona y demostrando tus intenciones con las pruebas construye una relación de confianza con tus compañeros.
- Ritmo: es muy fácil perderse durante horas cuando se está programando. Con el enfoque de test-first, está claro lo que hay que hacer a continuación: o escribimos una prueba o hacemos que funcione una prueba rota (broken test). El ciclo que se genera se convierte pronto en algo natural y eficiente: test, code, refactor, test, code, refactor.

A pesar de sus beneficios, TDD tiene bastantes críticos. Es una técnica que no es fácil y que obliga a una disciplina que al principio no gusta a muchos programadores. Al principio, cuando uno no está acostumbrado, se siente que la velocidad de desarrollo es mucho peor, que haríamos las cosas mucho más rápido sin la obligación de ir paso a paso.

A esta crítica se suele responder con dos argumentos. En primer lugar, en muchos ejercicios se suelen exagerar los pequeños pasos y se hacen más pequeños de lo que en realidad podrían ser. Los pasos deben ser pequeños, pero no minúsculos. Estos pasos también serán algo más grandes en programadores con más experiencia, que tengan más dominio de los patrones de diseño.

En segundo lugar, la velocidad de desarrollo es muy relativa y puede cambiar mucho cuando la medimos a medio-largo plazo. Lo que puede parecer un desarrollo rápido, en el que hemos introducido un cambio importante en poco tiempo, se puede convertir con el tiempo en mucho más lento debido al aumento de bugs y a la necesidad de arreglarlos. En TDD la velocidad de desarrollo es mucho más constante, sin altibajos, en la línea del principio de XP de desarrollo sostenido.

Entre las críticas que han tenido más difusión se encuentra la de David Heinemeir Hanson (creador de Ruby on Rails), en su artículo de 2014 TDD is dead. Long live testing. Hanson comenta que durante muchos años ha estado usando TDD pero que ha llegado un momento en el que se ha dado cuenta de que no se sentía cómodo con este estilo y de que era más productivo no usándolo. Sobre todo en la parte en la que tenía que hacer tests que no eran unitarios, en los que tenía que tocar la base de datos o la entrada salida. En sus palabras:

> Test-first units leads to an overly complex web of intermediary objects and indirection in order to avoid doing anything that's "slow". Like hitting the database. Or file IO. Or going through the browser to test the whole system. It's given birth to some truly horrendous monstrosities of architecture. A dense jungle of service objects, command patterns, and worse.

La alternativa que propone Hanson es que prefiere en esos casos hacer tests que toquen los servicios reales (bases de datos, entrada-salida, etc.) o mocks de esos servicios (como se hace en Rails y en Spring Boot) y lanzar esos tests del sistema (aunque sean lentos) en servicios especializados en la nube.

El artículo tuvo mucha repercusión por si mismo, pero también porque a raíz de su publicación se organizó una conversación pública en YouTube entre el propio Hanson, Beck y Fowler. La conversación se dividió en seis vídeos publicados en YouTube, con una duración total de alrededor de 3 horas y media. En el post de Fowler *Is TDD Dead* se resumen todos los vídeos y se proporcionan los enlaces a YouTube.

Ejemplo práctico: cambio de monedas

Vamos a presentar un ejemplo tomado del libro de Kent Beck "Test-Driven Development By Example". El ejemplo está codificado en Java y puedes descargarlo y comprobar su realización completa en el repositorio de GitHub https://github.com/domingogallardo/tdd-monedas.

Tenemos un sistema de gestión de fondos de inversión y acciones que produce informes como este:

Acciones	**Cantidad**	**Precio**	**Total**
IBM	1000	25	25.000
Apple	400	100	40.000
		Total	65.000

Queremos ampliar el sistema para que pueda tener en cuenta distintas monedas:

Acciones	**Cantidad**	**Precio**	**Total**
IBM	1000	25 USD	25.000 USD
Acerinox	400	120 EUR	48.000 EUR

Necesitamos definir el cambio entre dólares y euros

|----------------| | 1 USD = 1,5 EUR |

Y podremos hacer la conversión en una determinada moneda:

Acciones	**Cantidad**	**Precio**	**Total**	**Total en USD**
IBM	1000	25 USD	25.000 USD	25.000 USD
Acerinox	400	120 EUR	48.000 EUR	32.000 USD
			Total	56.000 USD

Queremos diseñar un conjunto de clases que permitan gestionar una cartera de valores con múltiples monedas usando TDD.

Cada ciclo de TDD consta de 3 pasos:

- Test sencillo con el que añadimos una nueva comprobación.
- Código que hace pasar el test.
- Refactorización para eliminar duplicación.

A este ciclo de 3 pasos es importante añadir un paso fundamental, que hay que hacer continuamente: pensar. Tenemos que pensar en los objetivos que queremos cumplir, en los problemas del diseño actual y en cómo solucionarlos.

Como resultado de esta reflexión continua mantendremos una lista de cosas por hacer, que nos ayudará a mantenernos concentrados, y decidir qué hacer cuando hayamos terminado cada uno de los tests.

Pondremos en negrita lo siguiente a hacer y eliminamos lo que hayamos terminado. Cuando pensemos en que necesitamos escribir otro test, lo añadiremos a la lista.

Pensamos

Del ejemplo en la tabla vemos que inicialmente hay dos cosas importantes que tenemos que hacer:

1. Multiplicar precio por cantidad en una determinada moneda.
2. Sumar monedas distintas.

Escribimos estos dos objetivos en forma de tests, en la lista de cosas a hacer.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
5 USD * 2 = 10 USD

De los dos objetivos, comenzamos por el segundo, que parece más básico que el primero. Recordemos que en TDD programamos el sistema de dentro a afuera o, lo que es lo mismo, de abajo a arriba. Primero las capas más básicas y después las capas superiores, que estarán basadas en la implementación ya testeada de las primeras.

En primer lugar, para poder multiplicar una moneda, debemos poder crearla. Añadimos una tarea más a la lista.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
5 USD * 2 = 10 USD
Crear una moneda con 5 dólares

Test creación de 5 dólares

Vamos a especificar el test para el objetivo de crear una moneda de 5 dólares.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestMonedas {
    @Test
    public void testCrearMoneda() {
        Dolar cinco = new Dolar(5);
        assertEquals(5, cinco.cantidad);
    }
}
```

Fijaros que en el test ya hemos realizado decisiones de diseño. Hemos definido una clase `Dolar` con un atributo `cantidad`. El hecho de comenzar escribiendo un test obliga a concretar el objetivo inicial (algo

difuso) en decisiones concretas en el lenguaje y modelo de programación con el que vamos a trabajar. En nuestro caso, el lenguaje Java y su paradigma orientado a objetos nos lleva al diseño anterior.

Vamos a exagerar un poco la metodología TDD. TDD nos dice que debemos escribir el código mínimo para que el test pase. En este caso, el mínimo código sería el siguiente:

```
public class Dolar {
    int cantidad = 5;

    public Dolar(int cantidad) {
    }
}
```

El test pasa (verde). Pero tenemos un código fuertemente acoplado al test. Cualquier cambio mínimo en el test hace que el test se rompa. En este caso es muy evidente porque estamos exagerando. Pero vamos a usar el ejemplo para mostrar la tercera parte de TDD: debemos buscar duplicidades y refactorizar para eliminarlas. La duplicación es un indicador de dependencia.

> "If dependency is the problem, duplication is the symptom. Duplication most often takes the form of duplicate logic—the same expression appearing in multiple places in the code.
>
> Kent Beck

En este caso hemos acoplado el valor 5 del código directamente al valor introducido en el test.

Refactorización código creación dólares

Vamos entonces a refactorizar para eliminar la duplicidad del valor `5` entre el test y el código. Empezamos moviendo la asignación de la cantidad dentro del constructor (pequeños pasos):

```
public class Dolar {
    int cantidad;

    public Dolar(int cantidad) {
        this.cantidad = 5;
    }
}
```

Lanzamos los tests y sigue en verde. Pero seguimos con la duplicidad del 5 en el test y en el código. Lo arreglamos asignando el parámetro `cantidad` del constructor al atributo. De esta forma eliminamos la duplicidad y generalizamos. Realmente, lo que queremos que se guarde en la `cantidad` no es `5`, sino el valor que le pasemos como `cantidad` en el test (sea el que sea).

```
public class Dolar {
    int cantidad;

    public Dolar(int cantidad) {
```

```
        this.cantidad = cantidad;
    }
}
```

Pensamos

Con esto hemos cumplido el objetivo y lo eliminamos de la lista. Ahora podemos ya podemos centrarnos en el test de la multiplicación.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
5 USD * 2 = 10 USD

Test multiplicación en dólares

Añadimos el test de multiplicación a la clase de test. En principio vamos a usar un diseño típicamente orientado a objetos, en el que las operaciones mutan el estado del objeto. En este caso, la operación `multuplicadoPor(int)` modifica la cantidad almacenada en la instancia:

```
    @Test
    public void testMultiplicacion() {
        Dolar cinco = new Dolar(5);
        cinco.multiplicadoPor(2);
        assertEquals(10, cinco.cantidad);
    }
```

Creamos la cantidad de 5 dólares y llamamos al método `multiplicadoPor` para que se multiplique por 2. El valor resultante deberá estar guardado en el atributo `cantidad` y deberá ser 10.

El código mínimo que soluciona el test es el siguiente:

```
public class Dolar {
    int cantidad;

    public Dolar(int cantidad) {
        this.cantidad = cantidad;
    }

    public void multiplicadoPor(int multiplicador) {
        cantidad *= multiplicador;
    }
}
```

Con eso ya hemos pasado el test de la multiplicación.

Pensamos

Analizando el diseño del código y de los tests, nos damos cuenta de que hay varias cosas que se podrían mejorar:

- Hacer `cantidad` privado.
- Podría haber efectos laterales en `Dolar` por utilizar programación orientada a objetos tradicional. ¿Usamos un objeto valor?
- Estamos limitando los valores y las cantidades a multiplicar a enteros.

Las anotamos en la lista:

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Hacer "cantidad" privado
Efectos laterales en Dolar?
Usar punto flotante - redondeo?

Decidimos empezar con los efectos laterales. Un objeto llamado `cinco` no debería tener un valor de `10`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Hacer "cantidad" privado
Efectos laterales en Dolar?
Usar punto flotante - redondeo?

Refactorización de test con patrón value object

Vamos a usar el patrón value object para mejorar el diseño. Modificamos el test para usar este patrón (también hay que refactorizar los tests):

```
    @Test
    public void testMultiplicacion() {
        Dolar cinco = new Dolar(5);
        Dolar resultado = cinco.multiplicadoPor(2);
        assertEquals(10, resultado.cantidad);
        resultado = cinco.multiplicadoPor(3);
        assertEquals(15, resultado.cantidad);
    }
```

Y modificados el código para que el test pase:

```
public class Dolar {

-    public void multiplicadoPor(int multiplicador) {
```

```
+    public Dolar multiplicadoPor(int multiplicador) {
-        cantidad *= multiplicador;
+        return new Dolar(cantidad * multiplicador);
    }
}
```

Con esto se pasa el test y se consiguen eliminar los efectos laterales en `Dolar`.

Pensamos

En el patrón Value Object es necesario poder comparar objetos según su valor. Debemos implementar `equals` y `hashCode`.

Empezamos por `equals`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Hacer "cantidad" privado
Usar punto flotante - redondeo?
equals()
hashCode()

Test equals

El test para implementar `equals` es muy sencillo:

```
    @Test
    public void testIgualdad() {
        assertTrue(new Dolar(5).equals(new Dolar(5)));
    }
```

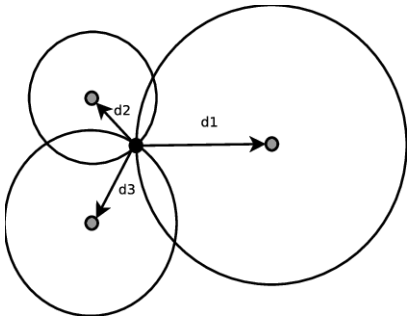
Siguiendo la filosofía de TDD (¡exagerándola!), el mínimo código que hace que pase es haciendo que el método devuelva `true`:

```
    public boolean equals(Object object) {
        return true;
    }
```

El test pasa, pero tenemos la sensación de que no es la solución correcta. De hecho, está pasando lo mismo que al principio, hay duplicación de código (el `true` que devuelve el método está duplicando el `assertTrue` del test).

Para comprobar si todo está bien introducimos una segunda aserción en el test, que obliga a generalizar el código. Kent Beck llama "triangular" a esta técnica de introducir nuevas aserciones para obligar a generalizar el código.

El nombre de "triangulación" viene de la técnica para localizar una posición en un mapa a partir de la distancia a balizas conocidas. Para localizar unívocamente una posición es necesario obtener la distancia a tres balizas.



Es una técnica que se utiliza continuamente en nuestros teléfonos móviles para obtener su localización usando satélites como balizas (GPS) o antenas de Wifi (localización por Wifi).

En el caso del TDD, muchas veces no es suficiente con una única aserción para definir un test. Necesitamos más aserciones para obligar a eliminar la dependencia entre el test y el código.

Triangulamos el test equals

Triangulamos, añadiendo otro ejemplo al test:

```
@Test
public void testIgualdad() {
    assertTrue(new Dolar(5).equals(new Dolar(5)));
    assertFalse(new Dolar(5).equals(new Dolar(6)));
}
```

El código falla y tenemos que generalizarlo:

```
public boolean equals(Object object) {
    Dolar dolar = (Dolar) object;
    return cantidad == dolar.cantidad;
}
```

El test ya pasa. Perfecto, seguimos avanzando.

Pensamos

¿Qué pasa si comparamos con `null` o con otro objeto? Lo añadimos a la lista de cosas por hacer y lo dejamos para después.

Miramos el código y nos damos cuenta de que ahora que tenemos implementada la igualdad, podemos refactorizar el test de la multiplicación para usar esta implementación. No tenemos que acceder a `cantidad`. De esta forma podremos hacer el atributo privado.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Hacer "cantidad" privado

Cosas por hacer
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object

Refactorización tests para usar equals

Tenemos que refactorizar los tests en los que se usa la cantidad para comparar y pasar a comparar usando el nuevo método equals:

```
    @Test
    public void testCrearMoneda() {
        assertEquals(new Dolar(5), new Dolar(5));
    }


    @Test
    public void testMultiplicacion() {
        Dolar cinco = new Dolar(5);
        assertEquals(new Dolar(10), cinco.multiplicadoPor(2));
        assertEquals(new Dolar(15), cinco.multiplicadoPor(3));
    }
```

Probamos que los tests siguen funcionando.

Refactorización código

Y ahora ya podemos hacer privado el atributo `cantidad`, porque no lo necesitamos en ninguno de los tests.

Código:

```
-    int cantidad;
+    private int cantidad;
```

Una cosa menos en la lista.

Pensamos

El primer ítem de la lista está todavía muy lejos. No es posible implementarlo con un pequeño cambio. Necesitamos como prerrequisto tener un objeto `Euro`, similar al `Dolar`. Añadimos otro ítem que nos obliga a crear la clase `Euro`.

Cosas por hacer

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object

5 EUR * 2 = 10 EUR

Tests igualdad y multiplicación en euros

Añadimos dos tests de igualdad para euros. Y copiamos y modificamos el test de la multiplicación en dólares.

Tests:

```
    @Test
    public void testIgualdad() {
        assertTrue(new Dolar(5).equals(new Dolar(5)));
        assertFalse(new Dolar(5).equals(new Dolar(6)));
+       assertTrue(new Euro(5).equals(new Euro(5)));
+       assertFalse(new Euro(5).equals(new Euro(6)));
    }
```

```
    @Test
    public void testMultiplicacionEuro() {
        Euro cinco = new Euro(5);
        assertEquals(new Euro(10), cinco.multiplicadoPor(2));
        assertEquals(new Euro(15), cinco.multiplicadoPor(3));
    }
```

Para pasar los tests copiamos y modificamos el código de `Dolar`.

El código es el siguiente:

```
public class Euro {
    private int cantidad;

    public Euro(int cantidad) {
        this.cantidad = cantidad;
    }

    public Euro multiplicadoPor(int multiplicador) {
        return new Euro(cantidad * multiplicador);
    }
```

```
    public boolean equals(Object object) {
        Euro dolar = (Euro) object;
        return cantidad == dolar.cantidad;
    }
}
```

Hemos dado una solución rápida y pequeña.

Pensamos

Vemos que hay mucho código duplicado. El objetivo final es eliminar la duplicación entre `Dolar` y `Euro`, creando una clase común que contenga el código repetido (`equals` y `multiplicadoPor`).

Vamos a comenzar creando una clase común que elimine la duplicación de `equals`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object
Duplicación Dolar Euro
equals duplicado
multiplicadoPor duplicado

Refactorización eliminar duplicación equals

Vamos a refactorizar el código para crear una clase común que contenga el código duplicado. Comenzaremos por `equals`.

Vamos a ir haciendo poco a poco pequeños cambios en el código, comprobando en cada momento que pasan los tests.

Creamos la clase `Moneda` y hacemos que `Dolar` y `Euro` la extiendan:

```
+ public class Moneda {
+ }

- public class Dolar {
+ public class Dolar extends Moneda {

- public class Euro {
+ public class Euro extends Moneda {
```

Los tests siguen pasando.

Movemos la variable de instancia `cantidad` a la clase superior:

```
public class Moneda {
+    protected int cantidad;
}

public class Dolar extends Moneda {
-    private int cantidad;

public class Euro extends Moneda {
-    private int cantidad;
```

Los tests siguen pasando.

Cambiamos en ambas clases los métodos `equals`, generalizándolo para que trabaje con monedas:

```
    public boolean equals(Object object) {
        Moneda moneda = (Moneda) object;
        return cantidad == moneda.cantidad;
    }
```

Los tests siguen pasando.

Por último, ya tenemos el mismo código de `equals` en ambas clases `Dolar` y `Euro`. Podemos moverlo a la clase padre `Moneda`:

```
public class Moneda {
    protected int cantidad;

    public boolean equals(Object object) {
        Moneda moneda = (Moneda) object;
        return cantidad == moneda.cantidad;
    }
}

public class Dolar extends Moneda {

    public Dolar(int cantidad) {
        this.cantidad = cantidad;
    }

    public Dolar multiplicadoPor(int multiplicador) {
        return new Dolar(cantidad * multiplicador);
    }
}
```

```
public class Euro extends Moneda {

    public Euro(int cantidad) {
        this.cantidad = cantidad;
    }

    public Euro multiplicadoPor(int multiplicador) {
        return new Euro(cantidad * multiplicador);
    }
}
```

Y los tests siguen pasando.

Pensamos

Ya que estamos con `equals` tenemos que asegurarnos que la comparación entre euros y dólares es `false`:

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado
Comparar euros con dólares

Test comparación euros con dólares

Añadimos una nueva línea al test de igualdad:

```
    @Test
    public void testIgualdad() {
        assertTrue(new Dolar(5).equals(new Dolar(5)));
        assertFalse(new Dolar(5).equals(new Dolar(6)));
        assertTrue(new Euro(5).equals(new Euro(5)));
        assertFalse(new Euro(5).equals(new Euro(6)));
+       assertFalse(new Euro(5).equals(new Dolar(5)));
    }
```

El test falla. Modificamos el código para que pase el test:

```
    public boolean equals(Object object) {
        Moneda moneda = (Moneda) object;
        return cantidad == moneda.cantidad
+               && this.getClass().equals(moneda.getClass());
    }
}
```

Pensamos

La solución anterior es un poco sucia, porque utilizamos las clases de Java y no un concepto del dominio, por ejemplo guardar de alguna forma la denominación de la moneda. Añadimos un elemento más a la lista de cosas por hacer y seguimos eliminando la duplicación de `Dolar` y `Euro`.

Para ello nos fijamos como objetivo eliminar la duplicación del método `multiplicadoPor`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado
Denominación moneda?

Como este cambio no se puede hacer con un pequeño paso, aplicamos otros patrones que nos acerquen a ese objetivo. Vamos a hacer algunas refactorizaciones basadas en el principio de inversión de dependencias de SOLID. Este principio nos dice que deberíamos depender de abstracciones, en lugar de implementaciones concretas.

Para ello vamos a eliminar en la medida de lo posible el uso de clases concretas `Dolar` y `Euro` y usar la clase más abstracta `Moneda`. Convertiremos la clase `Moneda` en una factoría con métodos que devuelvan euros y dólares.

Empezamos por refactorizar para añadir los métodos factoría.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null

Cosas por hacer
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado
Denominación moneda?
Eliminar uso de las clases concretas
Métodos factoría en Moneda

Refactorización: Métodos factoría en Moneda

Vamos ahora a intentar eliminar del código el uso de las clases `Dolar` y `Euro`, de forma que no tengamos que hacer explícitamente un `new` de esas clases. Así ganamos en abstracción.

Para ello vamos a convertir `Moneda` en una factoría, con un método `dolar` que devuelva un dolar. Empezamos refactorizando el test de multiplicación para obligar a construir el método de factoría.

```
    @Test
    public void testMultiplicacion() {
        Dolar cinco = Moneda.dolar(5);
        assertEquals(Moneda.dolar(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.dolar(15), cinco.multiplicadoPor(3));
    }
```

Al eliminar las llamadas a los constructores, hemos desacoplado los tests de la relación de herencia entre `Dolar` y `Moneda`.

Escribimos el código para que el test pase correctamente:

```
public class Moneda {
    ...

    static Dolar dolar(int cantidad) {
        return new Dolar(cantidad);
    }

    ...
}
```

Pasan los tests.

Cambiamos en los tests todas las llamadas a `new Dolar()` por llamadas al método factoría `Moneda.dolar()`.

Probamos los tests y comprobamos que pasan.

Y hacemos lo mismo para crear el método factoría que construye euros.

Primero cambiamos `testMultiplicacionEuro` para obligar a implementar el método factoría:

```
    @Test
    public void testMultiplicacionEuro() {
        Euro cinco = Moneda.euro(5);
        assertEquals(Moneda.euro(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.euro(15), cinco.multiplicadoPor(3));
    }
```

El test falla. Implementamos el método factoría en la clase `Moneda` para arreglarlo:

```
    public static Euro euro(int cantidad) {
        return new Euro(cantidad);
    }
```

Probamos que el test pasa. Y cambiamos en los tests todas las llamadas a `new Euro()` por llamadas al método factoría `Moneda.euro()`. Y volvemos a lanzar los tests y probar que funcionan.

Por último, cambiamos en los métodos `multiplicadoPor` las llamadas a los constructores por llamadas a los métodos factoría:

```
    public Dolar multiplicadoPor(int multiplicador) {
        return Moneda.dolar(cantidad * multiplicador);
    }

    ...

    public Euro multiplicadoPor(int multiplicador) {
        return Moneda.euro(cantidad * multiplicador);
    }
```

Pensamos

Una vez añadidos los métodos factoría, ya podemos pasar a refactorizar para eliminar el uso de las clases concretas `Dolar` y `Euro`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null

Cosas por hacer
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado
Denominación moneda?
Eliminar uso de las clases concretas

Refactorización para usar monedas en lugar de clases concretas

Empezamos por el código de `multiplicaPor` de ambas clases, haciendo que devuelvan una `Moneda`:

```
public class Dolar extends Moneda {
    ...
     public Moneda multiplicadoPor(int multiplicador) {
         return Moneda.dolar(cantidad * multiplicador);
     }
}

public class Euro extends Moneda {
     ...
     public Moneda multiplicadoPor(int multiplicador) {
         return Moneda.euro(cantidad * multiplicador);
     }
}
```

Los tests siguen pasando.

Pero queremos eliminar el uso de las clase `Dolar` y `Euro` en los tests `testsMultiplicacionEuro` y `testsMultiplicacion`.

Empezamos por el `testsMultiplicacion`, cambiando el objeto `cinco` para que sea de tipo `Moneda`:

```
    @Test
    public void testMultiplicacion() {
        Moneda cinco = Moneda.dolar(5);
        assertEquals(Moneda.dolar(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.dolar(15), cinco.multiplicadoPor(3));
    }
```

El método `multiplicador` no está definido en `Moneda`. Lo arreglamos haciendo la clase `Moneda` abstracta y declarando ahí el método. También podemos cambiar lo devuelto por el método estático `Moneda.dolar()`.

El código queda así:

```
abstract public class Moneda {
    protected int cantidad;

    static Moneda dolar(int cantidad) {
        return new Dolar(cantidad);
    }

    abstract Moneda multiplicadoPor(int multiplicador);

    ...
}
```

Los tests pasan, por lo que no hemos roto nada.

Hacemos lo mismo en el test `testMultiplicacionEuro`, para que la variable `cinco` sea también de tipo Moneda:

```
    @Test
    public void testMultiplicacionEuro() {
        Moneda cinco = Moneda.euro(5);
        assertEquals(Moneda.euro(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.euro(15), cinco.multiplicadoPor(3));
    }
```

Los tests pasan.

Y cambiamos el método factoría `euro()` para que devuelva una `Moneda`:

```
    public static Moneda euro(int cantidad) {
        return new Euro(cantidad);
    }
```

Pensamos

Seguimos intentando eliminar la duplicación de los dos multiplicadores. El código es ya muy parecido, pero por ahora no es posible hacerlo con un paso sencillo. Nos decidimos entonces atacar el objetivo de la denominación de la moneda.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null

Cosas por hacer
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado
Denominación moneda?

Test denominación monedas

Especificamos el objetivo con el siguiente test:

```
    @Test
    public void testDenominacion() {
        assertEquals("USD", Moneda.dolar(1).denominacion());
        assertEquals("EUR", Moneda.euro(1).denominacion());
    }
```

El código que lo hace pasar es muy directo:

```
abstract public class Moneda {
    abstract String denominacion();
}


public class Euro extends Moneda {

    public String denominacion() {
        return "EUR";
    }
}

public class Dolar extends Moneda {

    public String denominacion() {
        return "USD";
    }
}
```

Refactorización denominación

Refactorizamos el código anterior, para intentar obtener la misma implementación en ambas clases. Usaremos una variable de instancia.

```
public class Euro extends Moneda {

    private String denominacion;

    public Euro(int cantidad) {
        this.cantidad = cantidad;
        this.denominacion = "EUR";
    }

    public String denominacion() {
        return denominacion;
    }
}


public class Dolar extends Moneda {

    private String denominacion;

    public Dolar(int cantidad) {
        this.cantidad = cantidad;
        this.denominacion = "USD";
    }

    public String denominacion() {
        return denominacion;
    }
}
```

Pasan los tests correctamente.

Vemos que el código del método `denominacion` es ahora exactamente el mismo en ambas clases. Podemos entonces subir la declaración de la variable y el método a la clase padre. Y eliminarlos de las clases hijas.

```
abstract public class Moneda {
    protected String denominacion;

    String denominacion() {
        return denominacion;
    }
}
```

Si nos fijamos en los constructores de `Dolar` y `Euro` vemos que podemos refactorizarlos y hacerlos idénticos, haciendo que la denominación se pase como parámetro del constructor.

Tenemos que refactorizar también la creación en los métodos de factoría para que se pasen los identificadores correctos.

```
public class Euro extends Moneda {

    public Euro(int cantidad, String denominacion) {
        this.cantidad = cantidad;
        this.denominacion = denominacion;
    }
}


public class Dolar extends Moneda {

    public Dolar(int cantidad, String denominacion) {
        this.cantidad = cantidad;
        this.denominacion = denominacion;
    }
}


abstract public class Moneda {

    static Moneda dolar(int cantidad) {
        return new Dolar(cantidad, "USD");
    }

    static Moneda euro(int cantidad) {
        return new Euro(cantidad, "EUR");
    }

}
```

Lanzamos los tests y comprobamos que siguen funcionando correctamente.

Como último paso, ahora ya tenemos los constructores idénticos y podemos subir la implementación a la clase `Moneda`.

```
public class Euro extends Moneda {
    ...
    public Euro(int cantidad, String denominacion) {
        super(cantidad, denominacion);
    }
    ...
}

public class Dolar extends Moneda {
    ...
    public Dolar(int cantidad, String denominacion) {
        super(cantidad, denominacion);
    }
    ...
}
```

```
abstract public class Moneda {
    ...
    public Moneda(int cantidad, String denominacion) {
        this.cantidad = cantidad;
        this.denominacion = denominacion;
    }
    ...
}
```

Lanzamos los tests, comprobamos que funcionan y con esto hemos conseguido el objetivo que estábamos buscando.

Refactorización para eliminar duplicidad de multiplicadoPor

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object
Duplicación Dolar Euro
multiplicadoPor duplicado

Ahora ya estamos muy cerca de poder eliminar la duplicidad del código de `multiplicadoPor`. Y también de eliminar la duplicidad entre las clases `Dolar` y `Euro`, eliminando la necesidad de usar subclases.

Si miramos el código vemos que, al añadir la denominación de la moneda como una propiedad, ya no es necesario usar la propia clase para identificar si tenemos una moneda de un tipo o de otro. El tipo de moneda lo tenemos en su denominación.

De esta forma, podemos hacer que la multiplicación devuelva un objeto `Moneda`, con el mismo identificador del objeto que estamos multiplicando. El código sería entonces el mismo en ambas clases y ya podemos subirlo a la clase `Moneda`.

Necesitamos también quitar el `abstract` de la clase `Moneda`, para poder crear objetos de la propia clase:

```
public class Moneda {
    ...
    public Moneda multiplicadoPor(int multiplicador) {
        return new Moneda(cantidad * multiplicador, denominacion);
    }
    ...
}

public class Euro extends Moneda {
```

```
    public Euro(int cantidad, String denominacion) {
        super(cantidad, denominacion);
    }
}

public class Dolar extends Moneda {
    public Dolar(int cantidad, String denominacion) {
        super(cantidad, denominacion);
    }
}
```

Hay tests que fallan después de este cambio. Son los tests de igualdad, porque el método `equals` hace una comparación de clases. Ahora las clases ya no son distintas; son todos objetos de la clase `Moneda`.

Modificamos el código de `equals` para que funcione correctamente, comparando las denominaciones en lugar de las clases:

```
    public boolean equals(Object object) {
        Moneda moneda = (Moneda) object;
        return cantidad == moneda.cantidad
                && this.denominacion().equals(moneda.denominacion());
    }
```

Y los tests vuelven a pasar correctamente. Hemos conseguido unificar el método `multiplicadoPor`.

Refactorización para eliminar las clases Dolar y Euro

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object

Duplicación Dolar Euro

Por último, vamos a hacer una refactorización para eliminar las clases `Dolar` y `Euro`. Veremos que, con todas las refactorizaciones anteriores, el código que habrá que cambiar es ya muy poco.

Quitamos las referencias a las clases en los métodos factoría y sustituirlas por `Moneda`:

```
public class Moneda {
    ...
    static Moneda dolar(int cantidad) {
        return new Moneda(cantidad, "USD");
```

```
    }

    static Moneda euro(int cantidad) {
        return new Moneda(cantidad, "EUR");
    }
    ...
}
```

Los tests pasan correctamente.

Y ya podemos eliminar las clases `Dolar` y `Euro`, ya que nadie las usa. Cambiamos también las variables de instancia de `Moneda` a `private`.

Las borramos, lanzamos los tests y vemos que todos pasan.

Y con esto hemos terminado la unificación de `Dolar` y `Euro`.

Cosas por hacer
5 USD + 10 EUR = 10 USD si el cambio es 2:1
Usar punto flotante - redondeo?
hashCode()
Equal null
Equal object

Quedan todavía bastantes cosas por hacer. La mas importante la primera funcionalidad de sumar cantidades de distintas monedas. Pero no nos da más tiempo a terminarlo todo. Con lo que hemos visto hasta aquí creo que es suficiente para tener una idea bastante completa del funcionamiento práctico del TDD.

Si estás interesado en ver como termina todo el ejercicio, puedes consultar el libro de Kent Beck que está en las referencias.

Código final

Después de todos los tests y refactorizaciones, el código queda de la siguiente manera:

Clase `Moneda`:

```
public class Moneda {
    private int cantidad;
    private String denominacion;


    static Moneda dolar(int cantidad) {
        return new Moneda(cantidad, "USD");
    }

    static Moneda euro(int cantidad) {
```

```
        return new Moneda(cantidad, "EUR");
    }

    public Moneda(int cantidad, String denominacion) {
        this.cantidad = cantidad;
        this.denominacion = denominacion;
    }

    String denominacion() {
        return denominacion;
    }

    public Moneda multiplicadoPor(int multiplicador) {
        return new Moneda(cantidad * multiplicador, denominacion);
    }

    public boolean equals(Object object) {
        Moneda moneda = (Moneda) object;
        return cantidad == moneda.cantidad
                && this.denominacion().equals(moneda.denominacion());
    }
}
```

Tests:

```
import org.junit.Test;

import static org.junit.Assert.*;

public class TestMonedas {

    @Test
    public void testMultiplicacion() {
        Moneda cinco = Moneda.dolar(5);
        assertEquals(Moneda.dolar(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.dolar(15), cinco.multiplicadoPor(3));
    }

    @Test
    public void testIgualdad() {
        assertTrue(Moneda.dolar(5).equals(Moneda.dolar(5)));
        assertFalse(Moneda.dolar(5).equals(Moneda.dolar(6)));
        assertTrue(Moneda.euro(5).equals(Moneda.euro(5)));
        assertFalse(Moneda.euro(5).equals(Moneda.euro(6)));
        assertFalse(Moneda.euro(5).equals(Moneda.dolar(5)));
    }

    @Test
    public void testMultiplicacionEuro() {
        Moneda cinco = Moneda.euro(5);
        assertEquals(Moneda.euro(10), cinco.multiplicadoPor(2));
        assertEquals(Moneda.euro(15), cinco.multiplicadoPor(3));
```

```
    }

    @Test
    public void testDenominacion() {
        assertEquals("USD", Moneda.dolar(1).denominacion());
        assertEquals("EUR", Moneda.euro(1).denominacion());
    }
}
```

Referencias

- Kent Beck (2002) *Test-Driven Development By Example*
- James Shore (2020) *Incremental Test-Driven Development* - Vídeo con un ejemplo de código interesante: implementación de la codificación ROT13.