TDD

Table of Contents

Introducción

Bienvenidos al curso de Desarrollo Dirigido por Pruebas o TDD por sus siglas en inglés de Test Driven Development.

Que es TDD ?

TDD no es un lenguaje de programación, sino una **técnica de ingeniería de software** cuyo propósito se centra en **tres objetivos básicos**:

- Minimizar el número de bugs que surgen en el software una vez entregado.
- Implementar las funcionalidades justas que el cliente necesita y no más.
- Producir software modular, altamente reutilizable y preparado para el cambio.

Quizás hayáis oído hablar de que el objetivo de TDD es *conseguir el 100% de cobertura de código* con los tests. Pero no es así. Ese objetivo de 100% de cobertura es deseable con cualquier técnica que se aplique.

- Lo que persigue TDD es **que no surjan bugs**. Pero como ya sabemos que eso es imposible en el desarrollo de software, lo que perseguimos es **minimizar el número de bugs**. Cuantos menos bugs surjan en producción, más eficiente habrá sido nuestro trabajo y menos tiempo invertiremos en una tarea que no aporta beneficios directos.

- El segundo de los objetivos es no diseñar ni programar más código del necesario. Muchas veces, guiados por nuestra experiencia o nuestra intuición, dotamos al código de funcionalidades que pensamos que van a ser buenas o van a añadir calidad extra al producto. Pero la mayoría de las veces acaba siendo código que nunca se utilizará o que no se debía comportar como nostros habíamos pensado.

- Y el tercer objetivo busca tener un software de calidad y que no asuste cuando haya que hacer un cambio o una ampliación. Con TDD, hasta el más novel de los programadores será capaz de hacer cambios en un código desconocido, con la absoluta **confianza** de que no va a romper nada.

¿Y cómo se consigue esto?

Pues cambiando la mentalidad tradicional de desarrollo.

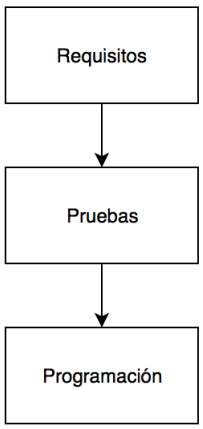
Pasaremos de pensar en implementar tareas, a pensar en ejemplos concretos y certeros que eliminen la ambigüedad creada por la prosa en lenguaje natural (nuestro idioma). Hasta ahora estábamos acostumbrados a que las tareas, o los casos de uso, eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código.

Con TDD intentamos traducir el caso de uso o tarea en **ejemplos concretos**, hasta que el número de ejemplos sea suficiente como para describir la tarea **sin lugar a malinterpretaciones de ningún tipo**.

A lo largo de este curso, iremos viendo cómo aplicar esta técnica en nuestros proyectos.

Flujo de Trabajo

El flujo de trabajo con TDD es radicalmente distinto, desaparecen las fases de análisis y diseño y además, ¡se escriben los tests antes que el código!.



El único análisis necesario antes de empezar es el de los componentes y el lenguaje a utilizar para el desarrollo y el framework (o frameworks) a utilizar para el testeo.

Por ejemplo, Para el desarrollo: Servidor Web Apache con Symfony (PHP) para API REST en el lado servidor, base de datos MySQL y Angular2 (Javascript) en el lado del cliente. Para el testeo: PhpUnit integrado en symfony2 para el código del servidor y Selenium para el el cliente.

¿Cuándo surge TDD?

Aunque es difícil de concretar una fecha, se puede decir que la técnica de TDD tiene sus raíces en la *programación extrema (Extreme Programming)* surgida a finales de los años 90. Uno de los pilares básicos de las misma es el *"Test First"*.

En muy poco tiempo el concepto *"Test First"* derivó en *"Test Driven"* y en 2003, con la publicación del libro *"Test Driven Development: By Example"* de Kent Beck se empezó a expandir y a consolidar alrededor del mundo. Aunque parezca novedosa, en realidad es una técnica bastante madura que además ha alentado innovaciones derivadas de la misma tales como ATDD (Acceptance Test Driven Development) o BDD (Behaviour Driven Development).

Hoy en día es ampliamente utilizada por la Comunidad Ágil (Agile Community).

Algoritmo del TDD

Algoritmo de TDD

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:

1. **Escribir un test que falle.**
2. **Implementar el mínimo código necesario para que el test pase.**
3. **Refactorizar para eliminar duplicidad y hacer mejoras.**

Estos pasos se repiten una y otra vez hasta que se acaban los requisitos.

¿Cómo escribimos un test para un código que todavía no existe?

Respondamos con otra pregunta: ¿Acaso no es posible escribir una especificación antes de implementarla?

El hecho de tener que usar una funcionalidad antes de haberla escrito le da un giro de 180 grados a la forma de trabajar.

Framework de Testeo

Framework de Testeo

Cada lenguaje de programación tiene uno o más frameworks profesionales de testeo. Algunos de los más conocidos son:

Junit (Java), PhpUnit (PHP), Jasmine (JavaScript), Selenium (cliente web), Codeception (PHP)

Permiten programar tests de manera muy sencilla, analizar la cobertura de código, programar dobles para independizar sistemas/componentes en el testeo (Mocks, Stubs).

- Dummy: se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- Fake: tiene una implementación que realmente funciona pero, por lo general, toma algún atajo o cortocircuito que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- Stub: proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas; tal como una pasarela de email que recuerda cuántos mensajes envió.
- Mock: objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles. Las veremos a continuación.

En los sucesivos ejemplos y ejercicios prácticos de este curso, utilizaremos PHP a secas para el desarrollo, sin Apache ni ningún otro tipo de servidor, y PhpUnit para la programación de los tests.

No obstante, el curso se puede seguir sin ninguna dificultad con cualquier otro lenguaje que soporte POO (Programación Orientada a Objetos) y cualquier framework de testeo disponible para el lenguaje elegido.

Desarrollo dirigido por test de Aceptación

ATDD

La gran diferencia entre las metodologías tradicionales y ATDD es la forma en la que se expresan los requisitos de negocio. En lugar de documentos de texto en forma mayoritariamente de párrafos, son listados de ejemplos ejecutables.

Se podría decir que el Desarrollo Dirigido por Test de Aceptación (ATDD), técnica conocida también como Story Test-Driven Development (STDD), es igualmente TDD pero a nivel de captura de requisitos (cliente).

Los tests de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños del producto.

Al principio, la diferencia es sútil. La clave reside en **centrarnos en el qué y no en el cómo**.

Ejemplo

Supongamos el siguiente ejemplo de requisito de una funcionalidad de una tienda:

"(…) Antes de proceder al pago, al cliente le saldrá un resumen detallado de los productos que tiene en la cesta, en los que se deberá aplicar el IVA (21%) o el RE (5.2%) al total, en caso de que el cliente requiera sus facturas con IVA o con RE.

Hum… hemos pedido el máximo detalle en los requisitos y esto es lo que nos ha definido el cliente. Decirle que queremos todavía más detalle no tiene pinta de llevarnos a ningún sitio.

Pero sí que hay una pregunta que podemos hacer al cliente y que nos puede encaminar a conseguir los detalles que queremos:

"¿Qué vas a hacer para comprobar que esta característica *funciona*?"

Ah, muy fácil:

Entraré a la aplicación con un cliente sin IVA y sin RE, añadiré productos a la cesta y en el resumen de la compra previo al pago veré que no se ha aplicado ningún impuesto.

Luego entraré con un cliente con IVA pero sin RE, haré una compra y veré que se esté aplicando el 21% de IVA y que no se le aplica RE.

Después entraré con un cliente con RE pero sin IVA, haré una compra y veré que se esté aplicando el 5.2% de RE y que no se le aplica IVA.

Y finalmente entraré con un cliente que tenga IVA y tenga RE para ver que se aplican ambos impuestos.

Esto parece detalladísimo, sin posibilidad ya de ambigûedades de ningún tipo, pero todavía no es así. Por ejemplo:

- ¿El RE se aplica después de aplicar el IVA?, ¿o antes?
- ¿En el detalle se pone únicamente el precio sin impuestos y el precio final con impuestos o se debe indicar también la cantidad concreta de cada uno de los impuestos?
- ¿Los impuestos se indican en un único concepto “impuestos” o por separado en dos conceptos IVA y RE?

Como al cliente también le parece que lo ha detallado con pelos y señales, lo que hago ahora es llegar a los ejemplos concretos.

Recordemos que **la base del éxito de TDD es tener ejemplos concretos y certeros**, así que vamos a poner ejemplos al cliente.

A nosotros nos servirá para resolver las ambigüedades que quedan y al cliente para tener la certeza de que lo hemos entendido a la perfección y por tanto la confianza de que el producto resultante será tal y como desea:

Entonces, para confirmar que lo he entendido:

Si un cliente con IVA y con RE elige un producto de 100€, en el detalle de la factura le enseñaré:

Precio	antes	de	impuestos:	100€
IVA:				21€
RE:				6.292€
Total: 127.29€				

No, no, no, el RE se calcula sobre la base imponible, no sobre la suma de base imponible más IVA, y debe aparecer el porcentaje de IVA y de RE entre paréntesis. Además las cantidades deben tener siempre 2 decimales. Sería así:

Base	Imponible:	100.00€
I.V.A.	(21%):	21.00€
R.E.	(5.2%):	5.20€
Total: 126.20€		

¡Ahora sí que está bien detallado! Si el cliente no está acostumbrado a dar los ejemplos así, hay que ayudarle a que nos los dé haciendo nosotros mismos los ejemplos. Al menos las primeras veces.

Con este nivel de detalle nos vamos a ahorrar muchísimas revisiones al entregar el código: que siempre hay que poner 2 decimales, que las siglas IVA y RE las quiere con puntos, la etiqueta del precio antes de impuestos debe ser “Base Imponible”, que hay que desglosarlo en las 4 líneas, que tiene que aparecer el porcentaje de cada impuesto entre paréntesis, etc.

Además, tenemos la confianza de que estamos haciendo exactamente lo que el cliente quiere. Y nos evita además numerosas tomas de decisiones que nos pueden parecer obvias, pero que muchas veces no lo son. Al fin y al cabo el cliente es el que mejor conoce su modelo de negocio. Las decisiones que tome el cliente siempre serán más acertadas que las que puedan tomar los desarrolladores, analistas, etc

Pero sigamos, el cliente nos dijo que para validar esta funcionalidad, iba a realizar 4 pruebas, y solamente tenemos el ejemplo de una de ellas. Hagamos los otros tres aprovechando que aún estamos reunidos con el cliente.

Si un cliente con IVA y sin RE elige un producto de 100€, en el detalle de la factura aparecerá:

Base	Imponible:	100.00€
I.V.A.	(21%):	21.00€
R.E.	(0%):	0.00€

Total: 121.00€

No, no, si no lleva RE, esa línea no sale.

¿Entonces así?

Base	Imponible:	100.00€
I.V.A.	(21%):	21.00€

Total: 121.00€

Sí, así, perfecto.

Vale, entonces un cliente sin IVA pero con RE verá:

Base	Imponible:	100.00€
R.E.	(5.2%):	5.20€

Total: 105.20€

¡Uy, no! El IVA siempre tiene que salir en las facturas.

Vaya, eso era imposible de adivinar, el IVA siempre sale aunque sea el 0% y el RE solamente si es del 5.2%. Como mínimo nos hemos ahorrado generar una nueva entrega del software.

¿Entonces quedá así?

Base	Imponible:	100.00€
I.V.A	(0%):	0.00€
R.E.	(5.2%):	5.20€

Total: 105.20€

Sí, así.

(El cliente debe estar ya dándose cuenta en la cuenta del nivel de detalle necesario para que las cosas salgan como él desea y que no cometamos errores o tomemos decisiones de las que no tenemos idea porque se escapan a nuestro conocimiento. En las siguientes funcionalidades su nivel de detalle será mayor sin necesidad de que se lo pidamos. El cliente también aprende inconscientemente a trabajar con ATDD.

Y en el caso de un cliente sin IVA y sin RE entiendo que será así:

Base	Imponible:	100.00€
I.V.A	(0%):	0.00€

Total: 100.00€

Eso es, perfecto.

Genial. Ya tenemos 4 tests de aceptación para la funcionalidad de “mostrar los impuestos en el resumen de la factura”.

Otras consideraciones

Otra ventaja de dirigir el desarrollo por los ejemplos, es que vamos a poder comprobar muy rápido si el programa está cumpliendo los objetivos o no. Conocemos en qué punto estamos y cómo vamos progresando. El Dueño de Producto puede revisar los tests de aceptación y ver cuántos se están cumpliendo, así que nuestro trabajo gana una confianza tremenda.

//*//

Algunas herramientas que facilitan la práctica del ATDD:

- Fit (**http://fit.c2.com/**)
- Fitnesse (**http://fitnesse.org/**)
- Concordion (**http://concordion.org/**)

- Cucumber (**https://cucumber.io/**)
- Robot (**http://robotframework.org/**)

Básicamente lo que proponen es escribir las frases con un formato determinado (HTML, XML…), usando etiquetas de una manera específica para delimitar qué partes de la frase son variables de entrada para el código y cuales son datos para validación del resultado de la ejecución.

Pueden llegar incluso a generar los esqueletos de los tests para frameworks de testeo determinados, ahorrando así adicional.

Características de estos frameworks:

- Orientado al cliente
- Ejecución de especificaciones

Ejemplo practico de Capture de Requerimientos

Enunciados

Supongamos que un cliente nos contrata para desarrollar la siguiente aplicación:

“Quiero lanzar una aplicación monedero para el pago entre amigos. Cada usuario tendrá una cuenta con saldo. La idea es que se puedan hacer transferencias a tus amigos directamente desde la app. La aplicación permitirá al usuario ingresar dinero o retirarlo cuando quiera.”

Pero aunque parece bastante claro, hay muchos detalles no especificados. No se puede aplicar TDD si no tenemos ejemplos concretos y certeros.

El ejercicio es el siguiente:

Imaginémonos que tenemos la oportunidad de entrevistarnos con el cliente para detallar mejor la aplicación. Nuestra tarea es obtener un listado de especificaciones ATDD. Es decir, un **listado de ejemplos concretos y certeros, checkeable, y que cubra todas las posibilidades para que no haya lugar a malinterpretaciones de ningún tipo sobre las funcionalidades**.

Evidentemente, los detalles los conoce el cliente, y no nosotros, así que para este ejercicio, nosotros mismos haremos también de cliente, dialogando con nosotros mismos hasta tener la lista de especificaciones ATDD completa.

Dedicadle un buen rato a completar la lista antes de pasar al siguiente vídeo.

En el siguiente vídeo enumeraré la lista que yo he obtenido, que evidentemente no coincidirá con la vuetra porque mi cliente no es el mismo que el vuestro, así que los requisitos de la aplicación de mi cliente no coincidirán con los de la aplicación de vuestro cliente.

Pero el objetivo realemnte es ver si somos capaces de llegar a elaborar un lista de ejemplos válidos para utilizar la técnica TDD.

Resolución

Una aplicación monedero para el pago entre amigos. Cada usuario tendrá una cuenta con saldo.
La idea es que se puedan hacer trasnferencias a tus amigos directamente desde la app.
La aplicación permitirá al usuario ingresar dinero o retirarlo cuando quiera.

Tras profundizar en cada funcionalidad con el cliente, resulta que tengo las siguientes especificaciones, mucho más detalladas:

- Creación de cuentas.
    - Las cuentas siempre se crean con saldo 0. Hay que hacer algún ingreso después si se quiere tener saldo:

- Ingresos.
    - Suman la cantidad ingresada al saldo.
    - No hay comisiones ni nada por el estilo.
    - No se pueden hacer ingresos negativos
    - Los ingresos admiten un máximo de 2 decimales de precisión
    - La cantidad máxima que se puede ingresar es de 6000

- Retiradas.
    - Restan la cantidad ingresada al saldo.
    - No se puede retirar una cantidad mayor a la del saldo disponible
    - No hay comisiones ni nada por el estilo.
    - No se pueden retirar cantidades negativas
    - Las cantidades admiten un máximo de 2 decimales de precisión
    - La cantidad máxima que se puede retirar es de 6000

- Transferencias
    - No se pueden transferir cantidades negativas
    - El límite cantidad transferida es de 3000:

Este es más o menos el nivel de detalle requerido en la metodología tradicional. Con esto ya se puede realizar un análisis y un diseño completos de la aplicación. Sin embargo el

objetivo de ATDD es conseguir ejemplos concretos, para transformarlos posteriormente en tests concretos cuando apliquemos TDD.

Así que le damos todavía una vuelta más de tuerca para definir los ejemplos con los que validaremos las funcionalidades: (en negrita añado los ejemplos)

- Creación de cuentas.
  - Las cuentas siempre se crean con saldo 0. Hay que hacer algún ingreso después si se quiere tener saldo:
    - **Al crear cuenta el saldo es cero**
- Ingresos.
  - Suman la cantidad ingresada al saldo.
  - No hay comisiones ni nada por el estilo.
    - **Al ingresar 100 en cuenta vacía el saldo es 100**
    - **Al ingresar 3000 en cuenta vacía el saldo es 3000**
    - **Al ingresar 3000 en cuenta con 100 el saldo es 3100**
  - No se pueden hacer ingresos negativos
    - **Al ingresar -100 en cuenta vacía, el saldo sigue siendo 0**
  - Los ingresos admiten un máximo de 2 decimales de precisión
    - **Si ingreso 100.45 en una cuenta vacía, el saldo es de 100.45**
    - **Si ingreso 100.457 en una cuenta vacía, el saldo es de 0**
  - La cantidad máxima que se puede ingresar es de 6000
    - **Si ingreso 6000.00 en una cuenta vacía, el saldo es de 6000.00**
    - **Si ingreso 6000.01 en una cuenta vacía, el saldo es de 0**
- Retiradas.
  - Restan la cantidad ingresada al saldo.
  - No hay comisiones ni nada por el estilo.
    - **Al retirar 100 en cuenta con 500 el saldo es 400**
  - No se puede retirar una cantidad mayor a la del saldo disponible
    - **Si retiro 500 en cuenta con 200 no ocurre nada y el saldo sigue siendo 200**
  - No se pueden retirar cantidades negativas
    - **Si retiro -100 en cuenta con 500 no ocurre nada y el saldo sigue siendo 500**
  - Las cantidades admiten un máximo de 2 decimales de precisión
    - **Al retirar 100.45 en cuenta con 500 el saldo es 399.55**
    - **Al retirar 100.457 en cuenta con 500 con 500 no ocurre nada y el saldo sigue siendo 500**
  - La cantidad máxima que se puede retirar es de 6000
    - **Si retiro 6000.00 en una cuenta con 7000, el saldo es de 1000**
    - **Si retiro 6000.01 en una cuenta con 7000, no ocurre nada y el saldo sigue siendo 7000**
- Transferencias
  - **Al hacer una transferencia de 100 desde una cuenta con 500 a una con 50, en la**

primera cuenta el saldo se quedará en 400 y en la segunda se quedará en 150.

- No se pueden transferir cantidades negativas
  - **Al hacer una transferencia de -100 desde una cuenta con 500 a una con 50, los saldos se quedan en 500 y 50 respectivamente**
- El límite de transferencias en un mismo día desde una misma cuenta es de 3000:
  - **Al hacer una transferencia de 3000 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 500 y en la segunda se quedará en 3050.**
  - **Al hacer una transferencia de 3000.01 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 3500 y en la segunda se quedará en 50.**
  - **Al hacer una transferencia de 2000 desde una cuenta con 3500 a una con 50, y justo después otra de 1200, en la primera cuenta el saldo se quedará en 1500 y en la segunda se quedará en 2050.**

Y ahora sí, hemos acabado de capturar las especificaciones. Con esto ya podemos empezar a escribir tests y a programar siguiendo el algoritmo TDD visto con anterioridad. Cada ejemplo marcado en negrita, se convertirá en un test que habrá que pasar posteriormente.

Ejemplo practico de desarrollo de una aplicación con TDD

Enunciado

Una aplicación monedero para el pago entre amigos. Cada usuario tendrá una cuenta con saldo.
La idea es que se puedan hacer trasnferencias a tus amigos directamente desde la app.
La aplicación permitirá al usuario ingresar dinero o retirarlo cuando quiera.

Tras profundizar en cada funcionalidad con el cliente, resulta que tengo las siguientes especificaciones, mucho más detalladas:

- Creación de cuentas.
  - Las cuentas siempre se crean con saldo 0. Hay que hacer algún ingreso después si se quiere tener saldo:

- Ingresos.
  - Suman la cantidad ingresada al saldo.
  - No hay comisiones ni nada por el estilo.
  - No se pueden hacer ingresos negativos

    - Los ingresos admiten un máximo de 2 decimales de precisión
    - La cantidad máxima que se puede ingresar es de 6000

- Retiradas.
    - Restan la cantidad ingresada al saldo.
    - No se puede retirar una cantidad mayor a la del saldo disponible
    - No hay comisiones ni nada por el estilo.
    - No se pueden retirar cantidades negativas
    - Las cantidades admiten un máximo de 2 decimales de precisión
    - La cantidad máxima que se puede retirar es de 6000

- Transferencias
    - No se pueden transferir cantidades negativas
    - El límite cantidad transferida es de 3000:

Este es más o menos el nivel de detalle requerido en la metodología tradicional. Con esto ya se puede realizar un análisis y un diseño completos de la aplicación. Sin embargo el objetivo de ATDD es conseguir ejemplos concretos, para transformarlos posteriormente en tests concretos cuando apliquemos TDD.

Así que le damos todavía una vuelta más de tuerca para definir los ejemplos con los que validaremos las funcionalidades: (en negrita añado los ejemplos)

- Creación de cuentas.
    - Las cuentas siempre se crean con saldo 0. Hay que hacer algún ingreso después si se quiere tener saldo:
        - **Al crear cuenta el saldo es cero**
- Ingresos.
    - Suman la cantidad ingresada al saldo.
    - No hay comisiones ni nada por el estilo.
        - **Al ingresar 100 en cuenta vacía el saldo es 100**
        - **Al ingresar 3000 en cuenta vacía el saldo es 3000**
        - **Al ingresar 3000 en cuenta con 100 el saldo es 3100**
    - No se pueden hacer ingresos negativos
        - **Al ingresar -100 en cuenta vacía, el saldo sigue siendo 0**
    - Los ingresos admiten un máximo de 2 decimales de precisión
        - **Si ingreso 100.45 en una cuenta vacía, el saldo es de 100.45**
        - **Si ingreso 100.457 en una cuenta vacía, el saldo es de 0**
    - La cantidad máxima que se puede ingresar es de 6000
        - **Si ingreso 6000.00 en una cuenta vacía, el saldo es de 6000.00**
        - **Si ingreso 6000.01 en una cuenta vacía, el saldo es de 0**
- Retiradas.
    - Restan la cantidad ingresada al saldo.
    - No hay comisiones ni nada por el estilo.
        - **Al retirar 100 en cuenta con 500 el saldo es 400**
    - No se puede retirar una cantidad mayor a la del saldo disponible

        - **Si retiro 500 en cuenta con 200 no ocurre nada y el saldo sigue siendo 200**
    - No se pueden retirar cantidades negativas
        - **Si retiro -100 en cuenta con 500 no ocurre nada y el saldo sigue siendo 500**
    - Las cantidades admiten un máximo de 2 decimales de precisión
        - **Al retirar 100.45 en cuenta con 500 el saldo es 399.55**
        - **Al retirar 100.457 en cuenta con 500 con 500 no ocurre nada y el saldo sigue siendo 500**
    - La cantidad máxima que se puede retirar es de 6000
        - **Si retiro 6000.00 en una cuenta con 7000, el saldo es de 1000**
        - **Si retiro 6000.01 en una cuenta con 7000, no ocurre nada y el saldo sigue siendo 7000**
- Transferencias
    - **Al hacer una transferencia de 100 desde una cuenta con 500 a una con 50, en la primera cuenta el saldo se quedará en 400 y en la segunda se quedará en 150.**
    - No se pueden transferir cantidades negativas
        - **Al hacer una transferencia de -100 desde una cuenta con 500 a una con 50, los saldos se quedan en 500 y 50 respectivamente**
    - El límite de transferencias en un mismo día desde una misma cuenta es de 3000:
        - **Al hacer una transferencia de 3000 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 500 y en la segunda se quedará en 3050.**
        - **Al hacer una transferencia de 3000.01 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 3500 y en la segunda se quedará en 50.**
        - **Al hacer una transferencia de 2000 desde una cuenta con 3500 a una con 50, y justo después otra de 1200, en la primera cuenta el saldo se quedará en 1500 y en la segunda se quedará en 2050.**

Y ahora sí, hemos acabado de capturar las especificaciones. Con esto ya podemos empezar a escribir tests y a programar siguiendo el algoritmo TDD visto con anterioridad. Cada ejemplo marcado en negrita, se convertirá en un test que habrá que pasar posteriormente.

Primer test: Introducción Escribiendo Test

Pongámonos manos a la obra. Lo primero que hay que hacer es coger el listado de ejemplos de las especificaciones. Es conveniente que se hayan ordenado por funcionalidad, agrupando todos los ejemplos de la misma funcionalidad. En nuestro caso, fuimos diligentes y organizados y los tenemos agrupados por funcionalidades.

- Creación de cuentas.
  - Las cuentas siempre se crean con saldo 0. Hay que hacer algún ingreso después si se quiere tener saldo:
    - **Al crear cuenta el saldo es cero**
- Ingresos.
  - Suman la cantidad ingresada al saldo.
  - No hay comisiones ni nada por el estilo.
    - **Al ingresar 100 en cuenta vacía el saldo es 100**
    - **Al ingresar 3000 en cuenta vacía el saldo es 3000**
    - **Al ingresar 3000 en cuenta con 100 el saldo es 3100**
  - No se pueden hacer ingresos negativos
    - **Al ingresar -100 en cuenta vacía, el saldo sigue siendo 0**
  - Los ingresos admiten un máximo de 2 decimales de precisión
    - **Si ingreso 100.45 en una cuenta vacía, el saldo es de 100.45**
    - **Si ingreso 100.457 en una cuenta vacía, el saldo es de 0**
  - La cantidad máxima que se puede ingresar es de 6000
    - **Si ingreso 6000.00 en una cuenta vacía, el saldo es de 6000.00**
    - **Si ingreso 6000.01 en una cuenta vacía, el saldo es de 0**
- Retiradas.
  - Restan la cantidad ingresada al saldo.
  - No hay comisiones ni nada por el estilo.
    - **Al retirar 100 en cuenta con 500 el saldo es 400**
  - No se puede retirar una cantidad mayor a la del saldo disponible
    - **Si retiro 500 en cuenta con 200 no ocurre nada y el saldo sigue siendo 200**
  - No se pueden retirar cantidades negativas
    - **Si retiro -100 en cuenta con 500 no ocurre nada y el saldo sigue siendo 500**
  - Las cantidades admiten un máximo de 2 decimales de precisión
    - **Al retirar 100.45 en cuenta con 500 el saldo es 399.55**
    - **Al retirar 100.457 en cuenta con 500 con 500 no ocurre nada y el saldo sigue siendo 500**
  - La cantidad máxima que se puede retirar es de 6000
    - **Si retiro 6000.00 en una cuenta con 7000, el saldo es de 1000**
    - **Si retiro 6000.01 en una cuenta con 7000, no ocurre nada y el saldo sigue siendo 7000**
- Transferencias
  - **Al hacer una transferencia de 100 desde una cuenta con 500 a una con 50, en la primera cuenta el saldo se quedará en 400 y en la segunda se quedará en 150.**

- No se pueden transferir cantidades negativas
  - **Al hacer una transferencia de -100 desde una cuenta con 500 a una con 50, los saldos se quedan en 500 y 50 respectivamente**
- El límite de cantidad transferida es de 3000:
  - **Al hacer una transferencia de 3000 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 500 y en la segunda se quedará en 3050.**
  - **Al hacer una transferencia de 3000.01 desde una cuenta con 3500 a una con 50, en la primera cuenta el saldo se quedará en 3500 y en la segunda se quedará en 50.**

Empezaremos con la primera funcionalidad, con el primer test de la misma: *Al crear cuenta el saldo es cero*

Yo utilizaré PHP y el framework PhpUnit. Pero podéis hacer el ejercicio con cualquier otro lenguaje y framework.

Adelante, escribid el primer test antes de pasar al siguiente vídeo.

Primer test: Escribiendo Test

En este paso a veces no hay nada que hacer, a veces sí. Depende del estado del código y también de la experiencia de los programadores para detectar necesidades de refactorización.

En la clase Cuenta no hay nada que refactorizar. Son las 3 líneas que el test necesita: el nombre de la clase, el nombre del método y que devuelva 0. Ya nos encontraremos situaciones en las que haya que refactorizar el código.

src/Cuenta.php

```
<?php

class Cuenta {

    public function getSaldo() {
        return 0;
    }
}
```

Y el test (no olvidemos que los test también se deben refactorizar) tampoco parece que se preste.

test/EjemploTDDTest.php

```php
<?php
use PHPUnit\Framework\TestCase;
require_once 'src/Cuenta.php';

class EjemploTDDTest extends TestCase
{

    public function testAlCrearCuentaElSaldoEsCero()
    {
        $c = new Cuenta();
        $this->assertEquals(0, $c->getSaldo());
    }

}
```

Pero al crear el archivo y la clase EjemploTDDTest les puse ese nombre porque no tenía ni idea todavía de qué elementos iban a testear y cómo se iban a llamar dichos elementos. Ahora ya sé que este archivo va a contener tests sobre las funcionalidades de la clase *Cuenta* así que para seguir buenas prácticas relativas a la organización de los test de mi aplicación, voy a cambiar el nombre de la clase por CuentaTest y por consiguiente, el nombre del archivo por CuentaTest.php

test/CuentaTest.php

```php
<?php
use PHPUnit\Framework\TestCase;
require_once 'src/Cuenta.php';

class CuentaTest extends TestCase
{

    public function testAlCrearCuentaElSaldoEsCero()
    {
        $c = new Cuenta();
        $this->assertEquals(0, $c->getSaldo());
    }

}
```

He refactorizado. Tengo que asegurarme de que no he roto nada. Vuelvo a ejecutar los tests.

OK (1 test, 1 assertion)

Perfecto. Hemos acabado con el primer test. Pasemos al segundo.

package tdd;

```
import org.junit.Test;

import ec.com.tdd.dto.Cuenta;
import junit.framework.TestCase;

public class CuentaTest extends TestCase{

	@Test
	public void test() {
		fail("Not yet implemented");
	}


	public void testAlCrearCunetaElSaldoEsCero() {
		Cuenta c = new Cuenta();

		assertSame(0, c.getSaldo());



	}

}
```

Resto de Tests

Change Risk Anti-Patterns (CRAP) Index

El índice CRAP se calcula en base a la complejidad ciclomática y a la cobertura de código.

Aquel código que no sea muy complejo y esté bien cubierto por los tests, tendrá un índice CRAP bajo. El CRAP se puede reducir, por lo tanto, escribiendo tests y refactorizando el código para reducir la complejidad

Un método con un índice CRAP de más de 30 se considera CRAPpy (es decir, inaceptable, ofensivo, etc.).

http://gmetrics.sourceforge.net/gmetrics-CrapMetric.html

Los tests como parte de la documentación

../phpunit-5.7.phar test —testdox doc.html

- Importancia de los nombres de los tests
- Importancia de que cada tests se dedique a una úncia especificación

Errores comunes y anti patrones

Errores comunes

Empezar con TDD no es coser y cantar. A lo largo del proceso surgen muchas dudas y se cometen muchos errores.

A continuación enumero una pequeña lista de errores comunes al empezar con TDD.

- El nombre del test no es suficientemente descriptivo

  Recordemos que el nombre de un método y de sus parámetros son su mejor documentación. En el caso de un test, su nombre debe expresar con total claridad la intención del mismo.

  Los nombres de los tests se deben parecer a esto:

  - testIngresoCantidadMasDe2DecimalesNoEsValido
  - testAlRetirar100EnCuentaCon500ElSaldoEs400
  - testAlRetirar200EnCuentaCon1200ElSaldoEs1000YAlRetirarOtros150ElSaldoEs850

  y no a esto:

  - testRetirada1
  - testRetirada2
  - testRetirada3
  - testForBUG128
- Escribir demasiados tests de una vez

  La técnica establece que se debe escribir un test, y luego el código para hacerlo pasar. Luego otro test, y luego el código para hacerlo pasar… Siempre de uno en uno. De esta forma al programar el código que debe pasar el test, nos aseguramos que cada decisión de diseño (de clases, métodos, relaciones entre clases, etc,) tomada en los tests es acertada, viable, válida…

Al coger práctica y experiencia con TDD, es bastante habitual escribir varios tests seguidos y luego el código que los hace pasar. Pero si escribimos decenas de tests seguidos, corremos el riesgo de acarrear malas decisiones de diseño que no hemos contrastado al escribir el código que los hace pasar.

- Adopción parcial de TDD

  A veces sucede, por diversos motivos, que no todos los desarrolladores del equipo usan TDD.

  El proyecto fracasará con toda seguridad a menos que todo el equipo al completo esté
  aplicando TDD.

- No sabemos qué es lo que queremos que haga el SUT (System Under Test)

  Nos hemos lanzado a escribir un test pero no sabemos en realidad qué es lo que el código bajo prueba tiene que hacer.

  En algunas ocasiones, lo resolvemos hablando con el dueño de producto y, en otras, hablando con otros desarrolladores. Hay que tener en cuenta que se están tomando decisiones de diseño al escribir los tests por lo que las especificaciones tienen que estar muy claras antes de empezar para no acabar diseñando un software que no cumple las especificaciones.

- No sabemos quién es el SUT y quién es el colaborador

  Es muy común que cuando necesitamos un colaborador, aquel que representamos mediante un doble (un mock, un stub…), para testear una funcionalidad del SUT, acabamos testeando al colaborador en lugar de al SUT.

- Un mismo método de test está haciendo múltiples afirmaciones

  Cuando practicamos TDD correctamente, apenas tenemos que usar el depurador. Cuando un test falla, lo encontramos directamente y lo corregimos en dos minutos. Para que esto sea así, cada método debe probar una única funcionalidad del SUT.

A veces utilizamos varias afirmaciones (asserts) en el mismo test, pero sólo si giran en torno a la misma funcionalidad. Un método de test raramente excede las 10 líneas de código.

- Se nos olvida refactorizar

No sólo por tener una gran batería de tests, el código ya es más fácil de mantener. Si el código no está limpio, será muy costoso modificarlo, y también sus tests. No hay que olvidar buscar y corregir código duplicado después de hacer pasar cada test. El código de los tests debe estar tan limpio como el código de producción.

- No eliminamos código muerto

A veces, tras cambios en las especificaciones, queda código en desuso. Puede ser código de producción o pueden ser tests. Puesto que normalmente disponemos de un sistema de control de versiones que nos permite volver atrás si alguna vez volviese a hacer falta el código, debemos eliminar todo código que creamosen desuso. El código muerto induce a errores antes o después. Se suele menospreciar cuando se trata de tests pero, como hemos hecho notar antes, el código de los tests es tan importante como el código que testean.

Anti patrones

James Carr **(https://blog.james-carr.org/)** recopiló una lista de antipatrones ayudado por la comunidad TDD.

Ese listado ya no está disponible en su blog, pero la comunidad de TDD mantiene un **catálogo de antipatrones en stackoverflow**:

Los nombres que les pusieron tienen un carácter cómico y no son en absoluto oficiales pero su contenido dice mucho. Algunos de ellos ya están recogidos en los errores comentados en el vídeo anterior.

Como en tantas otras áreas, las reglas tienen sus excepciones. El objetivo es tenerlos en mente para identificar posibles malas prácticas al aplicar TDD.

He traducido algunos de ellos y enumerado a continuación:

- El Mentiroso

Un test completo que cumple todas sus afirmaciones (asserts) y parece ser válido pero que cuando se inspecciona más de cerca, muestra que realmente no está probando su cometido en absoluto.

- Setup Excesivo

Es un test que requiere un montón de trabajo para ser configurado. A veces se usan varios cientos de líneas de código para configurar el entorno de dicho test, con varios objetos involucrados, lo cual nos impide saber qué es lo que se está probando debido a tanto “ruido”.

- El Gigante

Aunque prueba correctamente el objeto en cuestión, puede contener miles de líneas y probar muchísimos casos de uso. Esto puede ser un indicador de que el sistema que estamos probando es un Objeto Todopoderoso.

- El Imitador

A veces, usar mocks puede estar bien y ser práctico pero otras, el desarrollador se puede perder imitando los objetos colaboradores. En este caso un test contiene tantos mocks, stubs y/o falsificaciones, que el SUT ni siquiera se está probando. En su lugar estamos probándo lo que los mocks están devolviendo.

- Sobras Abundantes

Es el caso en que un test crea datos que se guardan en algún lugar y otro test los reutiliza para sus propios fines. Si el generador de los datos se ejecuta después, o no se llega a ejecutar, el test que usa esos datos falla por completo.

- El Héroe Local

Depende del entorno de desarrollo específico en que fue escrito para poder ejecutarse. El resultado es que el test pasa en dicho entorno pero falla en cualquier

otro sitio. Un ejemplo típico es poner rutas que son específicas de una persona, como una referencia a un fichero en su escritorio.

- El Cotilla Quisquilloso

  Compara la salida completa de la función que se prueba, cuando en realidad sólo está interesado en pequeñas partes de ella. Esto se traduce en que el test tiene que ser continuamente mantenido a pesar de que los cambios sean insignificantes. Este es endémico de los tests de aplicaciones web. Ejemplo, comparar todo un HTML de salida cuando solo se necesita saber si el title es correcto.

- El Cazador Secreto

  A primera vista parece no estar haciendo ninguna prueba por falta de afirmaciones (asserts).
  El test está en verdad confiando en que se lanzará una excepción en caso de que ocurra algún accidente desafortunado y que el framework de tests la capturará reportando el fracaso.

  Ejemplo: Test de conexión a base de datos. Se confía en que si no se establece conexión, el propio sistema lanzará una excepción provocando que falle el test.

- El Escaqueado

  Un test que hace muchas pruebas sobre efectos colaterales (presumiblemente fáciles de hacer) pero que nunca prueba el auténtico comportamiento deseado.

- El Bocazas

  Un test o batería de tests que llenan la consola con mensajes de diagnóstico, de log, de depuración, y demás forraje, incluso cuando los tests pasan.
  A veces, durante la creación de un test, es necesario mostrar salida por pantalla, y lo que ocurre en este caso es que, cuando se termina, se deja ahí aunque ya no haga falta, en lugar de limpiarlo.

- El Cazador Hambriento

Captura excepciones y no tiene en cuenta sus trazas, a veces reemplazándolas con un mensaje menos informativo, pero otras incluso registrando el suceso en un log y dejando el test pasar.

- El Secuenciador

Un test unitario que depende de que aparezcan, en el mismo orden, elementos de una lista sin ordenar.

- Dependencia Oculta

Un primo hermano del Héroe Local, un test que requiere que existan ciertos datos en alguna parte antes de correr. Si los datos no se rellenaron, el test falla sin dejar apenas explicación, forzando al desarrollador a indagar por acres de código para encontrar qué datos se suponía que debía haber.

- El Enumerador

Una batería de tests donde cada test es simplemente un nombre seguido de un número, ej, test1, test2, test3. Esto supone que la misión del test no queda clara y la única forma de averiguarlo es leer todo el test y rezar para que el código sea claro.

- El Extraño

Un test que ni siquiera pertenece a la clase de la cual es parte. Está en realidad probando otro objeto (X), muy probablemente usado por el que se está probando en la clase actual (objeto Y), pero saltándosela interacción que hay entre ambos, donde el objecto X debía funcionar en base a la salida de Y, y no directamente. También conocido como La Distancia Relativa.

- El Evangelista de los Sistemas Operativos

Confía en que un sistema operativo específico se está usando para ejecutarse. Un buen ejemplo sería un test que usa la secuencia de nueva línea de Windows en la afirmación (assert), rompiéndose cuando corre bajo Linux.

- El que Siempre Funciona

Se escribió para pasar en lugar de para fallar primero. Como desafortunado efecto colateral, sucede que el test siempre funciona, aunque debiese fallar.

- El Libre Albedrío

En lugar de escribir un nuevo test para probar una nueva funcionalidad, se añade una nueva afirmación (assert) dentro de un test existente.

- El Unico

Una combinación de varios antipatrones, particularmente El Libre Albedrío y El Gigante. Es un sólo test unitario que contiene el conjunto entero de pruebas de toda la funcionalidad que tiene un objeto. Una indicación común de eso es que el test tiene el mismo nombre que su clase y contiene múltiples líneas de setup y afirmaciones.

- El Macho Chillón

Debido a recursos compartidos puede ver los datos resultantes de otro test y puede hacerlo fallar incluso aunque el sistema a prueba sea perfectamente válido. Esto se ha visto comúnmente en fitnesse, donde el uso de variables de clase estáticas, usadas para guardar colecciones, no se limpiaban adecuadamente después de la ejecución, a menudo repercutiendo de manera inesperada en otros tests. También conocido como El huésped no invitado.

- El Excavador Lento

Un test que se ejecuta de una forma increíblemente lenta. Cuando los desarrolladores lo lanzan, les da tiempo a ir al servicio,tomar café, o peor, dejarlo corriendo y marcharse a casa al terminar el día.

- Ciudadanos de segunda clase

  El código de los tests no se refactoriza tan cuidadosamente como el código de producción, acabando con un montón de código duplicado, y haciendo que los tests sean difícil de mantener.

- El Inspector

  Viola la encapsulación en un intento de conseguir el 100 % de cobertura de código y por ello sabe tanto del objeto a prueba que, cualquier intento de refactorizarlo, rompe el test.

Ventajas y Desventajas del TDD

Ventajas y Desventajas

- Cuando se utiliza TDD en un proyecto virgen, en raras ocasiones se tiene la necesidad de utilizar el depurador o debugger.
- A pesar de los elevados requisitos iniciales de aplicar esta metodología, el desarrollo guiado por pruebas (TDD) puede proporcionar un gran valor añadido en la creación de software, produciendo aplicaciones de más calidad y en menos tiempo.
- Proporciona al programador una gran confianza en el código desarrollado.
- Minimiza el número de bugs en producción.
- El equipo de desarrollo se focaliza en lo que el cliente quiere (todavía mucho más si se aplica ATDD).

Desventajas

Cuesta muchísimo trabajo encontrar en la literatura y en la bibliografía las desventajas del uso de TDD. Aquí pongo un listado de las que he encontrado:

- Aprender bien la técnica es todo un reto.
- Muchos tests pueden ser difíciles de escribir, sobretodo los tests no unitarios.
- Casi imposible de aplicar a código “legacy”.
- TDD incrementa enormemente la confianza, pero puede crear una falsa sensación de seguridad.
    - Al igual que introducimos bugs al programar el código, es posible introducir bugs al programar los tests.

    - Si el programador malinterpreta la especificación entonces se acabará creando un test que llevará a un código que pasa el test pero que está mal.
- Los tests automatizados no son sustitutos de testeadores reales. Siempre es necesaria la mezcla de ambos.
- Dado que los programadores que escriben el código, testean el código, se pierden la buena práctica de que el testeador sea distinto del programador.

Otras consideraciones

Principiante

- Capaz de escribir un test unitario previo al código correspondiente.
- Capaz de escribir código suficiente para hacer pasar un test que previamente falla.

Intermedio

- Practica “test driven bug fixing”: Cuando se encuentra un bug, escribe un test que reproduce el defecto antes de corregirlo.
- Capaz de descomponer una funcionalidad de un programa en una secuencia de varios test unitarios.
- Conoce y puede nombrar diferentes tácticas para guiar en la escritura de tests a sus compañeros (por ejemplo: “para testear un algoritmo recursivo, primero escribe un test para el caso correspondiente al fin de la recursión”)
- Capaz de detectar elementos reutilizables en los tests
- Capaz de proporcionar herramientas de prueba específicas para cada situación

Experto

- Capaz de establecer un “roadmap” de tests planificados para funcionalidades macroscópicas (y revisarlo si es necesario)
- Capaz de aplicar TDD con diferentes paradigmas de diseño: orientación a objetos, funcional, dirigido por eventos…
- Capaz de aplicar TDD con diferentes dominios técnicos: interfaz de usuario, acceso persistente a base de datos…