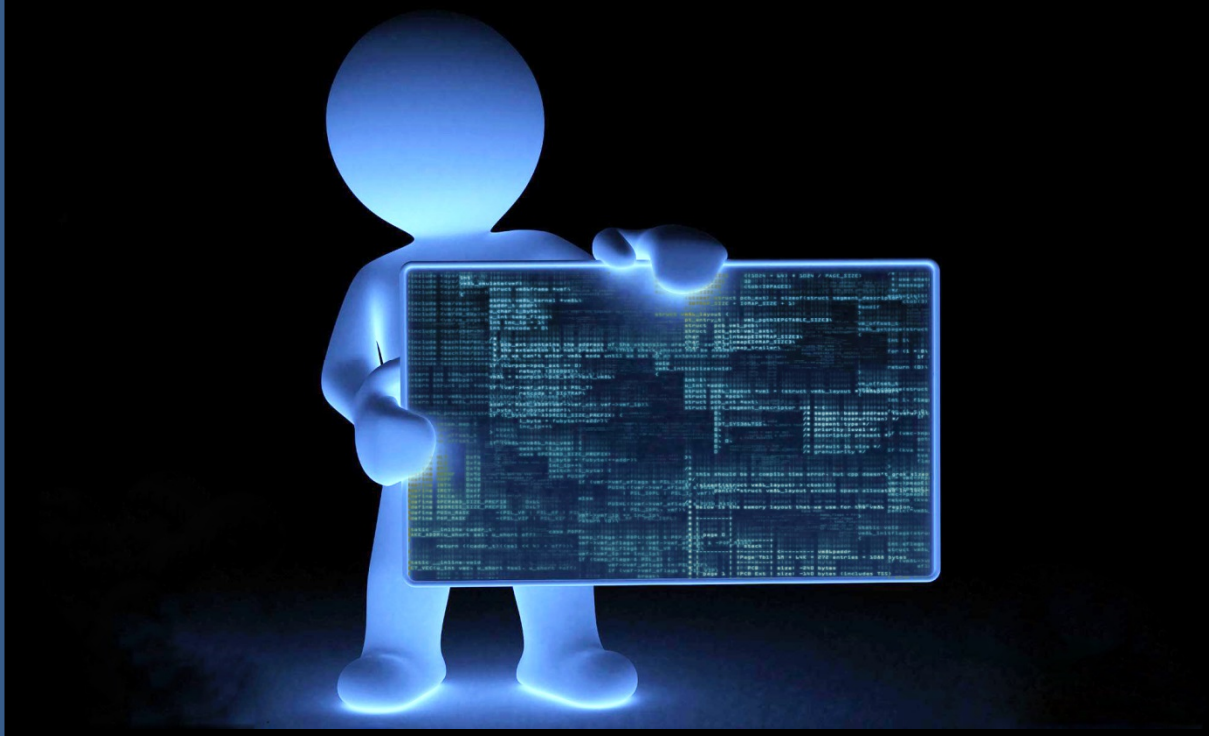
Desarrollo Dirigido por Pruebas Práctico

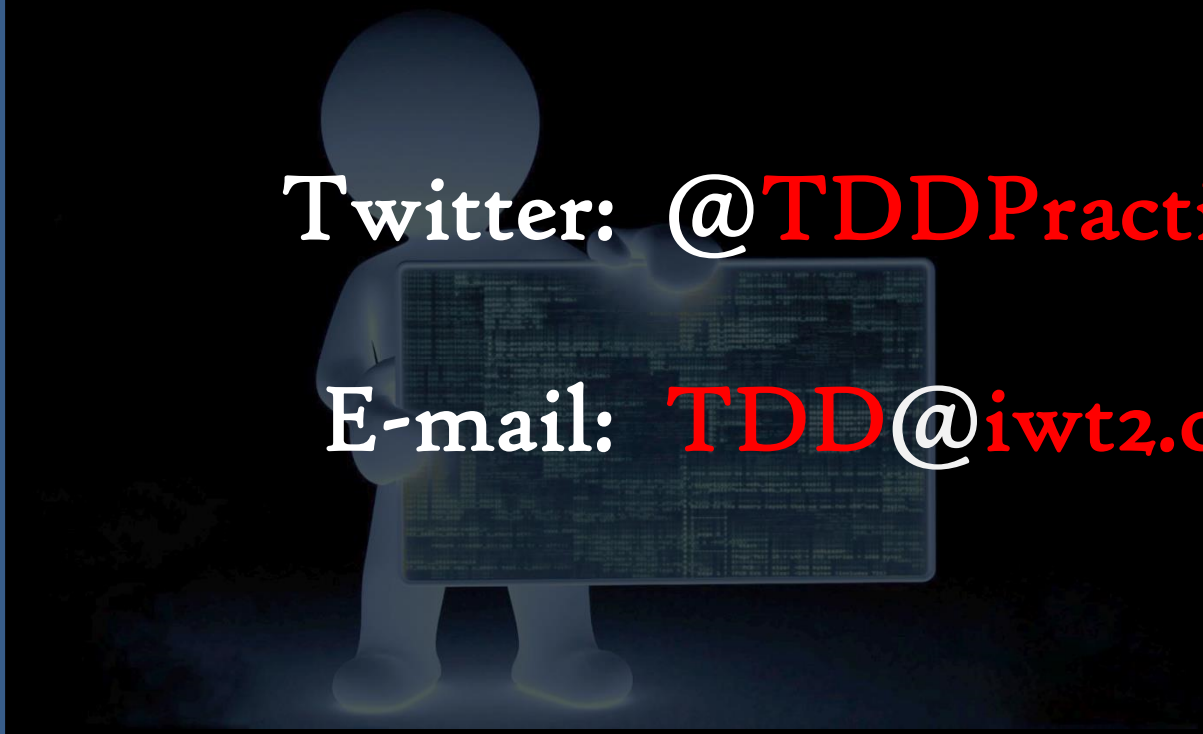


Javier J. Gutiérrez Rodríguez

Prototipo

tdd@iwt2.org

Contacta con nosotros:



Twitter: @TDDPractico

E-mail: TDD@iwt2.org

Encuesta on-line (vía Google):
http://goo.gl/mz6by

Control de cambios.

Fecha – Versión	Cambios
20/04/2013 – Prototipo 01	Primer prototipo público.
28/04/2013 – Prototipo 02	Incluimos capítulo 03. Incluimos índice y enlace de la encuesta Corregimos algunos errores en el capítulo 1 gracias a Guillermo Gutiérrez (vía ScribD).
19/05/2013 – Prototipo 03	Incluimos capítulo 6 (incompleto) y un cuestionario de autoevaluación sobre conocimientos de testing. Arreglamos el índice para que muestre correctamente los títulos de los capítulos.
01/06/2013 – Prototipo 04	Mejoramos los textos de los capítulos 1 y 2. Añadimos un apartado de agradecimientos del libro (además de los agradecimientos de cada capítulo). Añadimos una descripción del contenido de los capítulos 4 y 5. Comentarios son bienvenidos.
03/07/2013 – Prototipo 05	Añadimos el capitulo 4 completo con dos soluciones distintas para la cata Mancala.

Introducción al proyecto

Alberto Savioa (@Pretotyping en Twitter) escribió un interesante y muy recomendable libro sobre prototipos y que, a su vez, también es un prototipo de libro (Pretotype It: http://www.pretotyping.org/pretotype-it---the-book). He decidido adoptar esa misma filosofía en el futuro libro que, sin embargo, ya estás leyendo justo en este momento.

Tengo muy claro que no quiero escribir un libro solo para plasmar mis conocimientos de TDD o mi visión particular. Existen algunos libros muy buenos que explican casi todo lo que necesitas saber, y el resto te lo da la práctica. Mi objetivo es enseñar TDD a través de ejemplos que muestren muchos de los escenarios y dudas que aparecen cuando se aplican y que ofrezcan ideas y soluciones para abordarlos.

Todo el material que se elabore para este libro será de libre descarga lo más rápidamente posible. Creo que vale la pena que el material llegue pronto a los interesados, aunque eso implique que incluya notas de cosas por hacer y arreglar o que el formato no esté pulido. Mi objetivo es que quién tenga interés, pueda ver cómo va a ser el libro y pueda anticiparse y contarme qué cosas quiere que ponga o cambie. Así, tendremos un libro de todos para todos.

Voy a desarrollar el libro en dos fases: prototipos y versiones beta.

Los prototipos serán versiones del libro incompletas. Incluso con capítulos que tampoco estén completos. En contenido de un prototipo no es estable y todo puede cambiar o desaparecer en futuras versiones. Probablemente la versión que estés leyendo ahora mismo sea un prototipo.

Una vez que encontremos la estructura del libro adecuada, los ejemplos más interesantes y la mejor manera de exponerlos, pasaremos a las versiones beta. Una versión beta ya será un libro completo y cerrado con todos (o casi todos) sus capítulos. El objetivo de las versiones beta será puliendo el contenido y corregir erratas.

Además, también espero que esta filosofía de liberar el trabajo rápidamente me permita obtener un feedback que me anime y me motive a trabajar en el libro.

No quiero despedirme sin agradecerte la ayuda que me estás prestando sólo por leer estas líneas. Espero que lo que encuentres aquí te resulte útil y entretenido.

Javier Jesús Gutiérrez

Sevilla – Primavera 2.013

Agradecimientos

Hay muchas personas que me han ayudado de una u otra manera a que este libro vaya creciendo y, sobre todo, se vaya adaptando a los gustos y expectativas de vosotros, sus lectores. Como justo reconocimiento de toda esa ayuda he querido poner en cada capítulo el nombre de las personas y proyectos que han estado involucrados.

Pero también he tenido la enorme suerte de encontrar a personas que se merecen ser citadas en todos y cada uno de los capítulos. Va para ellos también mi agradecimiento.

- A Joaquín Engelmo Moriche (Kinisoft) por sus muchos y muy buenos comentarios durante los primeros prototipos del libro, que eran justo cuando más lo necesitaba.

Índice

Capítulo 1.
Desarrollo Dirigido por Pruebas. Lo Mínimo que necesitas saber

Como veremos más adelante, uno de los pilares del desarrollo dirigido pro pruebas (TDD por sus siglas en inglés) es escribir siempre el mínimo código posible para que una prueba se ejecute con éxito. En este libro vamos a aplicar la misma filosofía a la teoría de TDD, por eso en este capítulo encontrarás los conceptos mínimos necesarios para empezar a aplicar TDD a partir del capítulo siguiente.

No vas a encontrar aquí reflexiones sobre la bondad o no de TDD, ni discusiones filosóficas sobre si TDD está más centrada en el diseño que en pruebas, o cualquier otro tema habitual de debate sobre TDD porque damos por hecho que, si lees esto, estás lo suficientemente convencido para darle una oportunidad a TDD. Solo lo mínimo para entrar en acción.

Aun así, si este capítulo se te hace aburrido, no lo leas. Ve directamente a los ejemplos y comienza a practicar TDD. Vuelve aquí de vez en cuando para repasar y terminar de afianzar los conceptos básicos. Recuerda que para aprender bien TDD hay que practicarlo.

Como complemento a estas píldoras de teoría de TDD, encontrarás al final de cada ejemplo práctico reflexiones sobre la manera de aplicar TDD que te ayudará a profundizar en esta buena práctica.

Para empezar a aprender y practica TDD solo necesitas conocimientos básicos sobre prueba del software. Hemos incluido un conjunto de ejercicios para que también puedas practicar y evaluar tus propios conocimientos. Encontrarás estos ejercicios en los apéndices de este libro
Cuanto más aprendas y practiques sobre prueba del software mejor aplicarás TDD.

Un ejemplo de Test-Driven Development

Empecemos con un ejemplo. Supongamos que estamos escribiendo una clase que funciona como un carrito de la compra que aún no ha sido escrita. Como queremos aplicar TDD debemos escribir una prueba antes de poder empezar a implementar la clase. Un posible caso de prueba podría ser el mostrado en el código 1.1.

Código 1.1

```
3   @test
4   public void añadirUnElementoAlCarrito() {
5       // Set-up
6       X carrito = new X(); // 1
7       Item i = itemDePrueba();
8
9       carrito.x(x...); // 2
10
11      assertX(carrito.x()); // 3
12  }
```

Si analizamos detenidamente la prueba vemos que ya hemos tomado decisiones que tienen un impacto muy grande sobre el código que tendremos que escribir (en este caso la clase carrito). Las primeras decisiones que tomamos al escribir una prueba están en la línea 6 (comentario //1): ¿cómo creamos un objeto carrito? ¿Cuál es el nombre de la clase carrito? ¿Tendrá un constructor sin parámetros?

Después, en la línea 9 (comentario //2), seguimos tomando decisiones, por ejemplo cómo añadimos elementos al carrito: ¿usaremos una llamada a un método? ¿Cuál será el nombre de ese método? ¿Qué parámetros tendrá y de qué tipo serán?

Por último, en la línea 11 (comentario //3), tomamos decisiones sobre cómo comprobamos que el carrito ha almacenado correctamente el producto, por ejemplo: ¿qué métodos incluimos? ¿Cuál es el resultado esperado? ¿Hay que escribir antes una prueba para dicho método? Con TDD, antes de escribir el código nos centramos cómo queremos usarlo y nuestras primeras pruebas son requisitos o declaraciones de cómo debe ser dicho uso.

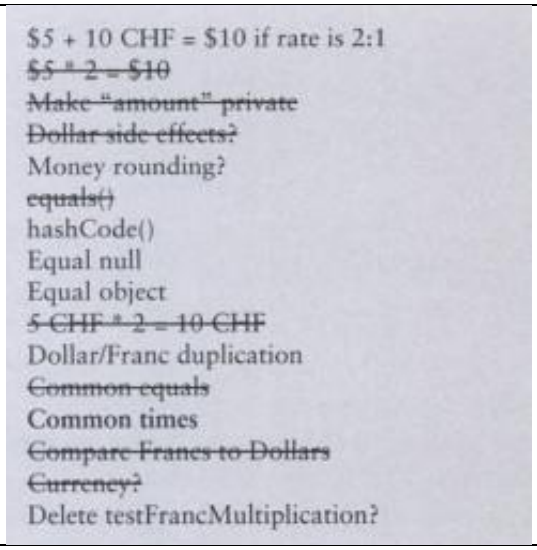
El diario de diseño

El paso previo antes de empezar con un ciclo TDD (que veremos en la próxima sección) es decidir sobre qué vamos a trabajar: ¿acceso a datos? ¿validaciones? ¿diseñar nuevas clases? ¿implementar lógica de negocio? ¿Mejorar cosas que se nos han quedado pendientes? Para centrar nuestro trabajo utilizaremos el diario de diseño.

Este diario es la lista de tareas que tenemos pendientes de hacer. Estas tareas serán, por lo general, código a escribir o funcionalidad a implementar. El principal objetivo del diario de diseño es mantenernos en todo momento con el foco centrado. ¿No te ha pasado nunca que te has concentrado mucho en escribir una prueba y luego no recordabas muy bien qué querías implementar? El diario nos ayudará con esto.

Utilizaremos el diario principalmente en dos momentos: el primero será cuando ya tengamos una funcionalidad implementada (y gracias a TDD probada) para decir la siguiente. El segundo momento será cuando, durante un ciclo de TDD, se nos ocurra nueva funcionalidad o escenarios adicionales (por ejemplo: nulos, cadenas o colecciones vacías, números negativos, etc.) a implementar en nuestro código.

Figura 1.1. Ejemplo de diario de diseño.



En la figura 1.1 tienes un ejemplo de diario de diseño utilizada por Kent Beck en su libro *Test Driven Development: By Example* para implementar una clase que represente una cantidad en un sistema monetario.

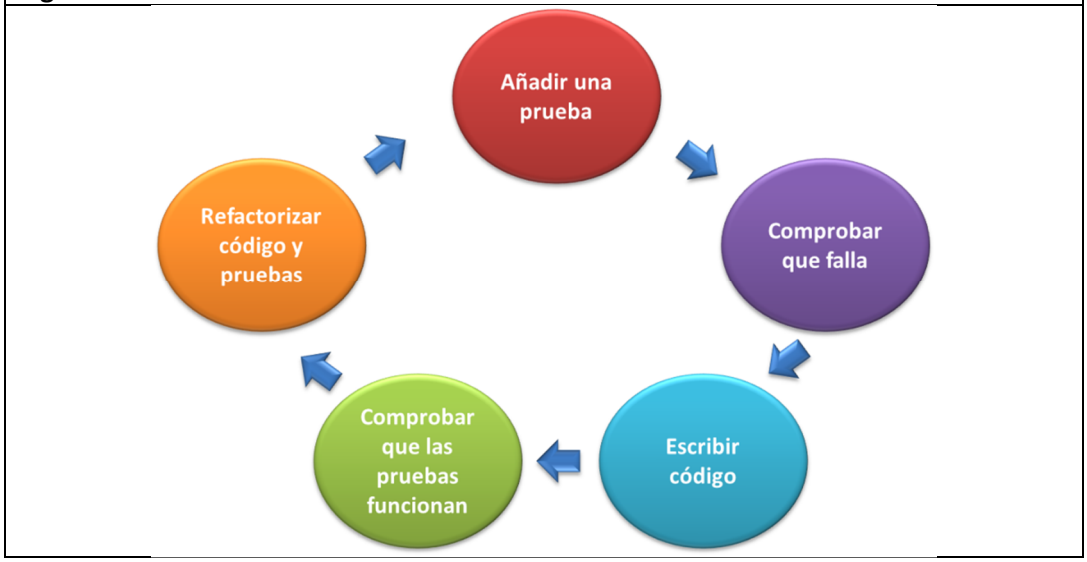
Como puedes ver en la figura 1.1, el diario de diseño no es una lista ordenada ni priorizada. En el momento de elegir una nueva tarea para trabajar, tienes libertad de elegir aquella que consideres más importante. Tampoco tienes que limitarte por posibles dependencias entre tareas, recuerda que las pruebas unitarias rompen las dependencias por lo que estas no condicionan la elección.

Veremos ejemplos de diarios de diseño en los ejemplos de los próximos capítulos.

El ciclo de Test-Driven Development

El ciclo de TDD es el conjunto de pasos que seguimos para implementar un nuevo fragmento de código de producción. El ciclo de TDD utiliza las pruebas para definir qué resultado queremos obtener del código que vamos a escribir. Estas pruebas nos ayudan a tomar las decisiones de diseño de código más adecuadas. El ciclo completo de TDD se muestra en la figura 1.1. A continuación, veremos cada elemento con más detalle.

Figura 1.2. Ciclo TDD.



El primer paso para empezar con el ciclo TDD consiste en escribir una prueba que ponga de relieve funcionalidad que queremos implementar. Esto nos obliga a definir qué es lo que queremos obtener con el código que vamos a escribir.

A continuación ejecutamos la prueba y vemos como falla. Si la prueba no falla estudiamos qué está sucediendo y elegimos otra.

Después, escribimos el código mínimo (más corto) para que la prueba pase con éxito. No nos preocupamos de escribirlo bonito, tenemos libertad para tomar atajos.

A continuación ejecutamos de nuevo todas nuestras pruebas si todo ha ido bien la prueba que falló antes ahora debería ejecutarse con éxito y todas las pruebas que antes se ejecutaban con éxito ahora lo seguirán haciendo. Si no debemos para e investigar qué ha sucedido en el paso anterior.

Por último, quitamos los atajos y refactorizamos el código y las pruebas. Verás varios ejemplos de refactorizaciones en los ejemplos de los próximos capítulos.

Esto es todo TDD, a partir de aquí ya puedes empezar a aplicarlo. Sin embargo es interesante que conozcamos un poco más del por qué hacemos las cosas de esta manera.

Código mínimo para pasar una prueba

En el ciclo de Test-Driven Development, a la hora de implementar nuestro código y hacer que la prueba pase, intentamos implementar sólo el mínimo código necesario y tan sencillo como sea posible. A esto lo llamamos avanzar en pequeños pasos, o *babysteps*.

Puede ser frustrante tener la implementación de un método en la cabeza y tener que parar porque no tenemos todas las pruebas que necesitamos para poder escribirla. Sin embargo siempre podemos introducir un error en el código sin darnos cuenta, por lo que es mejor avanzar más despacio pero más seguro.

También podemos tener una idea muy buena en la cabeza de cómo tiene que ser un método o un API pero luego descubrir que no es la manera más cómoda de utilizar. Por eso es mejor avanzar más despacio pero comprobando que nuestro código es usable.

Aunque la primera vez nos parezca que ese código no tenga valor, sí que lo tiene. Esto lo iremos descubriendo a medida que ganemos experiencia aplicando TDD. Cuando algo falle nos será de mucha ayuda acotar el problema en un pequeño fragmento de código. Además, el tener métodos que no tengan demasiadas líneas de código es una heurística de buen diseño

Un último motivo para escribir pruebas cortas y el mínimo código posible es no perder el foco. Uno de los puntos clave de TDD es ejecutar las pruebas muy a menudo. Si nos detenemos demasiado en una implementación o intentamos abarcar demasiado probablemente surjan muchos errores que nos lleven a entrar en un bucle de ejecución, corrección, ejecución monótono y aburrido. Veremos muchos ejemplos de código mínimo

Empezar fallando

Otro aspecto que puede resultar desconcertante en TDD es el ejecutar la prueba en primer lugar y que falle. Vamos a detenernos en ver los motivos.

Las pruebas tienen que ser simples ya que no escribimos pruebas que prueben las pruebas. Por ello las pruebas tienen que ser rápidas y pequeñas y deben fallar.

Una prueba que funciona nada más escribirla puede indicar o bien que el código no funciona como pensamos o bien que ya hace lo que debería. En ambos casos puede merecer la pena replantear el próximo paso.

Una excepción a esto son pruebas que son redundantes, es decir, que ya verifican algún escenario contemplado por otras pruebas. Aunque no parece lógico añadir pruebas redundantes, pueden ser útiles para documentar bien el código, exponer escenarios más complejos o, simplemente, aumentar nuestra sensación de seguridad y control a la hora de hacer cambios. Siempre que mantengamos el código de pruebas actualizado no habrá ningún problema al contar con pruebas redundantes. TDD no dice nada sobre estas pruebas así que puedes añadirlas si lo deseas siempre que te comprometas a refactorizarlas y a mantenerlas como al resto del código. Vamos a hablar de esto justo a continuación.

La importancia de la refactorización de todo el código

Refactorizar es cambiar la estructura interna del código para mejorar su calidad, pero sin cambiar su funcionalidad e interfaces externas. Refactorizar es una tarea que algunos desarrolladores pueden olvidar presionados por el tiempo. Sin embargo es un punto vital para que funcione TDD ya que esta buena práctica trabaja de manera incremental. Veamos las razones.

Código 1.1

Prueba A

```
2   def test_carritoMAL(self):
3       c = CarritoCompra()
4       p = Producto("producto", 39.5)
5
6       c.add(p)
7       c.add(p)
8       c.add(p)
9
10      self.assertEquals(c.total(), 118.5)
```

Prueba B

```
15  def test_carritoBIEN(self):
16      c = self.crearCarritoVacio()
17      p = self.crearProductoDePrueba()
18      unidades = 3
19
20      c.añadirProducto(p, unidades)
21      self.assertEquals(c.totalAPagar(), (p.precio * unidades))
```

Un desarrollador no pasa la mayor parte de su tiempo escribiendo código nuevo, sino que pasa la mayor parte del tiempo diseñando código, modificando código y haciendo que fragmentos de código funcionen buen juntos. Por tanto un desarrollador tiene que ser capaz de leer y entender el código. Las refactorizaciones nos ayudan a pulir

y mejorar el código para que sea más sencillo de entender y actualizar. Las pruebas nos dan la tranquilidad de que el código va a seguir funcionando correctamente. Veamos un ejemplo, ¿cuál de las dos pruebas del código 1.1 se entiende mejor?

En la primera prueba exponemos detalles de bajo nivel que no todos tienen que conocer, por ejemplo la manera de crear productos. Si repetimos la creación de productos en muchas pruebas y luego la creación de un producto cambia (lo que pasa con más frecuencia de lo que uno se imagina) tendremos que arreglar muchas pruebas. Además, en esta prueba no nos interesa conocer los detalles del producto.

Otro ejemplo: ¿Por qué el total del carrito debe ser exactamente 118.5? ¿Qué significa ese número? Si vemos la prueba B comprobamos que el total del carrito debe ser tres veces el precio del producto porque hemos añadido 3 copias.

Si no refactorizamos el código de nuestra aplicación,cada vez será más difícil de entender, más difícil de modificar y más difícil corregir los errores. Eso nos empujará a buscar alternativas o rodeos que empeorarán la situación.

Si tampoco refactorizamos el código de pruebas también será más difícil de entender y modificar con el paso del tiempo. Cuando una prueba falle nos costará más esfuerzo descubrir qué está fallando. Además cuando el código cambie nos costará más esfuerzo cambiar las pruebas y, poco a poco, tendremos a dejarlas de lado con lo que nuestro código no tendrá la red de protección de las pruebas y perderemos la confianza en él.

Cómo continuar aplicando el ciclo TDD

En el ciclo de TDD, primero tenemos que escribir una prueba que falle, pero ¿qué podemos hacer después? Existen tres estrategias: *Fake*, *Triangulación* e *Implementación Obvia*. No hay ningún orden para aplicar estas estrategias, aunque es muy común comenzar *fake* y luego evolucionar a *triangulación* o *Implementación Obvia*.

La estrategia *fake* consiste en simular el resultado o comportamiento esperado de la forma más rápida y sencilla posible. Para ello podemos incrustar el valor esperado en el cuerpo de la función. Utilizamos la estrategia *fake* cuando no tenemos muy claro cómo implementar un método, o cuando la implementación es demasiado larga o compleja para hacerla en un único paso o cuando queremos priorizar la creación de interfaces fáciles de usar antes de entrar en los detalles de sus implementaciones.

La estrategia *triangulación* consiste en añadir nuevas pruebas a una misma funcionalidad buscado situaciones no implementadas aún. Con esta técnica podemos evolucionar los *fakes* hasta la implementación en pequeños pasos (o babysteps).

En algunas ocasiones, el código a escribir para superar una prueba sigue un patrón muy claro y conocido, o bien es lo suficientemente pequeño para poder escribirlo en poco tiempo. En estos casos se puede implementar el código final directamente. Esta

es la *Implementación Obvia*. Algunos ejemplos de implementaciones obvias son recorrer colecciones, métodos get/set, etc.

Heurísticas TDD

En un desarrollo real no tenemos que obsesionarnos por aplicar TDD al pie de la letra. Al contrario, lo más beneficio es adaptar TDD a nuestra manera de trabajar, complementarlo con otras técnicas para obtener los mejores resultados.

Sin embargo, si estamos aprendiendo o practicando TDD sí es útil saber si lo estamos aplicando bien o no para no coger ningún mal hábito. A continuación veremos algunas ideas para poder autoevaluar si estamos aplicando TDD con éxito.

Empecemos repasando las métricas de código. Algunas métricas pueden ayudarnos a detectar si estamos obteniendo buenos resultados o no. A continuación se muestra un listado de métricas fáciles de medir.

- Tamaño de los métodos (pocas líneas).
- Pocos parámetros en los métodos (no más de 3).
- Clases e interfaces con pocos métodos.
- Alta cohesión y bajo acoplamiento.
- Alta cobertura de pruebas (90% o más).
- Cantidad de código duplicado.

La cobertura es la cantidad de código fuente que es ejecutado por las prueba. El valor del 90% es una referencia y puede variar dependiendo del proyecto / tecnología.

Algunas herramientas para calcular métricas y cobertura de código que puedes utilizar en Java son PMD, checkStyle y EcmaEmas. PMD además una herramienta llamada CPD para detectar código duplicado.

Hay que tener cuidado con las métricas ya que si nos centramos demasiado en ellas podemos desplazar el foco de nuestro trabajo a conseguir buenos números en vez de buscar la felicidad de nuestros clientes y usuarios. Por ejemplo podemos tener una alta cobertura de código o métodos pequeños y clases poco cohesionadas pero un código mal diseñado, difícil de entender y de modificar.

Ejecutar las pruebas con mucha frecuencia y dejar transcurrir poco tiempo entre ejecución y ejecución del conjunto de pruebas son dos indicadores que estamos aplicando TDD adecuadamente, aunque pueden ser más difíciles de medir que las métricas anteriores.

Para terminar

Si es la primera vez que te acercas a TDD utiliza todo lo que has visto en este capítulo para hacerte más fácil aplicarlo. A medida que ganes experiencia probablemente descubras pequeñas variantes, o alternativas, o mejoras para adaptar TDD a tu manera de trabajar. No dudes en aplicarlas y recuerda que TDD es una buena práctica y el contenido de este capítulo solo es una ayuda, nunca una norma de obligado cumplimiento.

Si ya conocías TDD, compara tu experiencia con todo lo que hemos visto en este capítulo. Seguro que encuentras algo en lo que puedas mejorar.

¿Estás preparado para empezar a aplicar TDD? Si lo estás ve al próximo capítulo y si no, ¡ve también al próximo capítulo!

Agradecimientos del capítulo

- A todos los alumnos de la primera edición del curso "Desarrollo Dirigido por Pruebas desde Cero" porque sus ejercicios, dudas y reflexiones me han ayudado mucho a darle forma a este capítulo.
- A Pablo Escribano porque es uno de los mejores *TDD-Practicioner* que tenemos en el sur de España.

Capítulo 2.
Primeros ejemplos de desarrollo dirigido por pruebas

Antes de empezar con ejemplos de mayor tamaño vamos a ver ejemplos pequeños y sencillos de desarrollo de software dirigido por pruebas para empezar a aplicar todo lo visto en el capítulo anterior. Estos ejemplos servirán para introducir los fundamentos de TDD que aplicaremos en los ejercicios de los próximos capítulos.

Listas palíndromas (en Java)

En nuestro primer ejercicio vamos a implementar un método que verifique si una lista es palíndroma, es decir que el primer elemento coincide con el último, el segundo con el penúltimo, etc. La solución que vamos a implementar es recorrer las dos mitades de la lista a la vez, la primera mitad de orden creciente y la segunda mitad en orden decreciente, comparando los elementos de ambas mitades.

En este caso nuestro diario de diseño es muy sencillo y casi coincide al cien por ciento con los requisitos y las pruebas (cosa que no pasará en ejercicios más complejos, como veremos más adelante). Este diario se muestra en el cuadro 2.1.

Cuadro 2.1

- Comprobar listas con números pares.
- Comprobar listas con números impares
- Caso especial: lista vacía no es palíndroma
- Caso especial. Lista con un único elemento no es palíndroma.

Para empezar escribimos nuestra primera prueba (código 2.1). Esta prueba nos sirve para definir cómo va a ser la función: ¿Qué parámetros necesita? ¿Cómos e llamará? Etc.

Código 2.1

```
@Test
	public void testPalindromaDeEnteros_Par() {
		List<Integer> l = Arrays.asList(1,2,2,1);
		Boolean b = ListasPalindromas.isPalindroma(l);

		assertTrue(b);
	}
```

Vamos a escribir ahora el mínimo código que pasa la prueba (código 2.2). En concreto aplicamos la estrategia *fake* vista en el capítulo anterior.

Código 2.2

```
public  static <T >boolean isPalindroma(List<T> lista){
   return true;
}
```

La prueba pasa con éxito. Vamos a *triangular* y, para ello, añadimos una segunda prueba en la que el método detecta que una lista no es palíndroma. Una posible prueba se muestra en el código 2.3.

Código 2.3

```
@Test
	public void testNoPalindromaDeEnteros_Par() {
		List<Integer> l = Arrays.asList(1,2,3,1);
		Boolean b = ListasPalindromas.isPalindroma(l);

		assertFalse(b);
	}
```

A partir de las dos pruebas anteriores vamos a implementar el núcleo de la comparación. El código mínimo que hace que ambas pruebas se ejecuten con éxito se muestra en el código 2.4.

Código 2.4.

```
public  static <T >boolean isPalindroma(List<T> lista){
		for(int i = 0; i < lista.size() / 2; i++){
		if (lista.get(i) != lista.get(lista.size()-1 - i)){
		   return false;
		}
	  }
```

```
        return true;
    }
```

Antes de continuar, vamos a refactorizar para que el algoritmo sea más sencillo de entender. Compara el código del cuadro 2.4 con el código del cuadro 2.5. ¿Es más sencillo de entender? ¿Qué cambios harías para mejorar la legibilidad? En el primer capítulo repasamos el por qué es muy importante que el código sea fácil de aprender cuando aplicamos TDD.

Código 2.5. Código refactorizado.

```
public  static <T >boolean isPalindroma(List<T> lista){
            int mitad = lista.size() / 2;
            int ultimoelemento = lista.size()-1;

            for(int i = 0; i < mitad; i++){
           if (lista.get(i) != lista.get(ultimoelemento - i)){
              return false;
           }
        }
        return true;
    }
```

La solución anterior funciona con listas de elementos pares. Nuestro siguiente paso será añadir pruebas para que el código trabaje también con listas con un número impar de elementos. Añadimos dos pruebas con listas impares (código 2.6 y código 2.7).

Código 2.6. Comprueba que una lista impar es palíndroma.

```
@Test
      public void testPalindromaDeEnteros_Impar() {
            List<Integer> l = Arrays.asList(1,2,3,2,1);
            Boolean b = ListasPalindromas.isPalindroma(l);

            assertTrue(b);
      }
```

Código 2.7. Comprueba que una lista impar no es palíndroma.

```
@Test
      public void testNoPalindromaDeEnteros_ImPar() {
            List<Integer> l = Arrays.asList(1,2,3,3,1);
            Boolean b = ListasPalindromas.isPalindroma(l);

            assertFalse(b);
      }
```

El conjunto de pruebas sigue funcionando correctamente, incluidas las pruebas de los cuadros 2.6 y 2.7. Busquemos ahora pruebas que nos hagan escribir nuevo código de producción. Por ejemplo podemos implementar la gestión de una lista vacía. Para ello escribimos la prueba del código 2.8.

Código 2.8.

```
@Test
	public void testListaVacia() {
		List<Integer> l = Arrays.asList();
		Boolean b = ListasPalindromas.isPalindroma(l);

		assertFalse(b);
	}
```

Y, ahora sí, la nueva prueba falla y podemos escribir código adicional. El resultado se muestra a continuación.

Código 2.9.

```
public  static <T >boolean isPalindroma(List<T> lista) {
		if (lista.size() == 0 ) {
		return false;
	   }

		int mitad = lista.size() / 2;
		int ultimoelemento = lista.size()-1;

		for(int i = 0; i < mitad; i++) {
		if (lista.get(i) != lista.get(ultimoelemento - i)) {
		   return false;
		}
	  }
	  return true;
    }
```

Sin embargo si la lista tiene un único elemento tampoco será posible comprobar si es o no palíndroma, por lo que añadimos una nueva prueba. En este caso tomo la decisión de que una lista con un único elemento no es palíndroma.

Código 2.10.

```
@Test
	public void testListaUnElemento() {
		List<Integer> l = Arrays.asList(1);
		Boolean b = ListasPalindromas.isPalindroma(l);

		assertFalse(b);
	}
```

A continuación, modificamos el código para que todas las pruebas que tenemos hasta ahora se ejecuten con éxito.

Código 2.11.

```
public  static <T >boolean isPalindroma(List<T> lista) {
            if (lista.size() < 2 ) {
            return false;
          }

            int mitad = lista.size() / 2;
            int ultimoelemento = lista.size()-1;

            for(int i = 0; i < mitad; i++) {
           if (lista.get(i) != lista.get(ultimoelemento - i)) {
              return false;
            }
        }
        return true;
    }
```

Y ya está terminado. Después de revisar el diario de diseño vemos que hemos acabado con todas las tareas por lo que podemos pasar al siguiente ejercicio.

Tipo de dato lista (en Python)

En esta sección vamos a ver un ejemplo de cómo utilizar las pruebas como requisitos con una clase muy sencilla que incluya alguna de las operaciones del tipo de dato lista. No vamos a empezar pensando en qué métodos tiene que tener la lista ni en los detalles de su implementación sino en ejemplos de cómo queremos utilizarla y los documentamos en nuestro diario de diseño:

Cuadro 2.2.

- Si creo una lista, entonces el número de elementos es cero.
- Si creo una lista y añado un elemento, el número de elementos es uno.
- Si creo una lista y añado los elementos A, B y C, entonces el elemento de índice 0 es A, el elemento de índice 1 es B y el elemento de índice 2 es C.

Podemos codificar estos ejemplos como pruebas en Python para tomar decisiones sobre cómo vamos a utilizar la clase lista y para ir verificando que todo funciona correctamente a medida que la programamos. Para ello utilizamos el módulo *unittest* que ya viene incluido en la distribución base de Python. Una posible implementación de los ejemplos anteriores se muestra a continuación.

Código 2.12

```
class TestsCasesForMyList(unittest.TestCase):

    def test_GivenANewListThenItHasZeroElements(self):
        l = MyList()
        self.assertEqual(0, l.size())

    def test_GivenANewListWhenIAddAnElementThenItHasOneElement(self):
        l = MyList()
        l.add("A")
        self.assertEqual(1, l.size())

    def test_GivenANewListWhenIAdd_A_B_C_Then_A_HasInex_0_B _1_and_C_2
    (self):
        l = MyList("A", "B", "C")
        self.assertEqual("A", l.get(0))
        self.assertEqual("B", l.get(1))
        self.assertEqual("C", l.get(2))
```

Fijémonos en las decisiones de diseño y requisitos que hemos establecido puede que sin darnos cuenta. Por ejemplo, si nos fijamos en la tercera prueba vemos que, aunque nadie lo había mencionado, nos resulta muy cómodo indicar un grupo de elementos al momento de crear la lista, en vez de añadirlo uno a uno. Si no hubiéramos puesto este ejemplo puede que no lo hubiéramos descubierto.

La implementación de las pruebas anteriores, siguiendo la regla del mínimo código se muestra a continuación.

Código 2.13

```
class MyList:
      def __init__(self, *val):
            self.elements = 0

      def size(self):
            return self.elements

      def add(self, elem):
            self.elements += 1

      def get(self, index):
            if index == 0:
                  return "A"
            if index == 1:
                  return "B"
            return "C"
```

¿Cuál sería tu siguiente paso para continuar aplicando TDD a partir de aquí? Tienes la solución al final de este capítulo.

Para reflexionar

Vamos a comentar algunas decisiones que hemos tomado en los ejercicios anteriores. Empezamos repasando el mínimo código de producción necesario para que una prueba funcione. Aplicando este principio hemos escrito el código 2.14 para implementar la primera prueba del ejercicio de listas palíndroma.

Código 2.14

```
public  static <T >boolean isPalindroma(List<T> lista){
    return true;
}
```

¿No es este código demasiado trivial para aportar algo? No, este código es valioso, porque, en primer lugar me permite pasar la prueba de fallo a éxito y me da seguridad para seguir avanzando. En segundo lugar sirve como comprobante de que la prueba funciona. En tercer y último lugar me permite usar mi propio código y darme cuenta si el nombre del método, parámetros, tipo devuelto, etc. son la mejor elección.

Como vimos en el capítulo anterior, a partir de este código podemos tomar tres caminos: modificarlo por el código en producción, escribir una nueva prueba que me obligue a cambiar el código para que las pruebas pasen, o escribir una prueba que me lleve a implementar otro método.

La tercera opción no tiene sentido aquí ya que no estamos construyendo un API ni similares, pero podría tener sentido si quisiéramos modelar una interfaz externa cómoda de usar antes de entrar en la implementación.

La primera opción, personalmente, no convence en este caso. Creo que el código real de producción es muy complejo para implementarlo en un único pequeño paso (*babystep*). Por este motivo elegí la segunda opción y añadí una segunda prueba como vimos al principio del capítulo. Esta es una opción personal que dicta la práctica, cualquier otra opción probablemente sería igual de interesante ya que conduce a aumentar la experiencia y, por tanto, la capacidad de reflexionar y aprender.

Recuerda también que buscamos avanzar en la implementación con pequeños pasos. Un error común al principio es escribir una implementación obvia como la del código 2.14 y triangular para intentar implementar 30 o 40 líneas código de golpe. Busca que la triangulación te lleve a implementar solo unas pocas líneas de código. Aunque pienses que es una implementación obvia, intenta no escribir demasiado código de producción, recuerda que también tenemos que ejecutar las pruebas muy a menudo

Otro aspecto interesante para reflexionar es qué hacer con pruebas innecesarias. Hemos visto en el primer ejemplo que las pruebas de los cuadros 2.6 y 2.7 no nos hacían escribir nuevo código, es decir, funcionan con el código actual. ¿Por qué escribirlas? Desde el punto de vista de TDD estas pruebas sobrarían y no deberíamos haberlas escrito.

No estamos obligados aplicar TDD durante todo el desarrollo y podemos escribir pruebas por otros motivos además de para escribir nuevo código. Podemos utilizar las pruebas para documentar adecuadamente escenarios adicionales. Observa que, aunque el código tal y como lo tenemos se comporta correctamente, puede no ser obvio cual es el resultado esperado para las listas impares. Las pruebas de los cuadros 2.6 y 2.7 hacen más claro ese resultado.

En general, salimos del ciclo de TDD cuando queremos añadir pruebas adicionales, cuando no seguimos escribiendo nuevo código, por ejemplo para dedicar más tiempo a refactorizar, o cuando consideramos que TDD no nos aporta ningún valor en el código a escribir. Nunca, nunca, nunca (sí, tres veces) debería abandonar TDD porque una prueba te resulte difícil. Justo al contrario, si una prueba te resulta difícil e escribir TDD está haciendo bien su trabajo avisándonos de que debemos volver a pensar en el diseño del código.

Solución al ejercicio planteado

Una posible solución para continuar con el ejercicio de la lista en Python está en el cuadro 2.15.

Código 2.15

```
class MyList:
	def __init__(self, *val):
		self.elements = 0

	def size(self):
		return self.elements

	def add(self, elem):
		self.elements += 1

	def get(self, index):
		if index == 0:
			return "A"
		if index == 1:
			return "B"
		return "C"
```

El método *get* es un *fake*, hemos incrustado los valores que tiene que devolver. Sin embargo es difícil hacer evolucionar esta implementación ya que no hemos definido una prueba que nos lleve a incluir en MyList una manera de guardar los elementos.

Completar.

Agradecimientos del capítulo

- A la comunidad Solveet (www.solveet.com) y al excelente trabajo que hace Rubén Bernárdez con su mantenimiento.
- A Juan Luis Cano y al blog de Pybonacci (pybonacci.wordpress.com)

Capítulo 3.
Criba de Eratóstenes en Java

En este capítulo realizaremos un ejemplo de mayor tamaño que implementaremos también aplicando el ciclo de desarrollo dirigido por pruebas que vimos en el primer capítulo.

El problema

Implementar, aplicando TDD, una versión muy sencilla (y poco optimizada) del algoritmo de la Criba de Eratóstenes para calcular la lista de los números primos desde 2 hasta un número n indicado. El algoritmo de la criba de Eratóstenes se describe a continuación.

1. Se crea una lista con los números desde 2 hasta n.
2. Se elige el siguiente número x.
3. Se marcan todos los múltiplos de dicho número (x*2, x*3, etc.).
4. Se repite desde el paso 2 mientras queden números.

Cuando se ha terminado con todos los números aquellos que queden sin marcar son primos. Veamos las primeras evoluciones aplicando TDD.

Empezamos por un mal camino

Para empezar a aplicar TDD escribimos nuestra primera prueba. Esta prueba la puedes ver en el código 2.1. Una vez que probamos que falla (aunque Java ni siquiera permite ejecutarla) la implementamos con el código mínimo posible, con un *fake*, que se muestra en el código 3.1.

Código 3.1

Prueba

```
@Test
public void testCalculaConValorInicialUno() {
	List<Integer> l = CribaDeEratosthenes.Calcula(1);
	assertTrue(l.isEmpty());
}
```

Implementación

```
static class CribaDeEratosthenes {
	public static List<Integer> Calcula(int i) {
		return new ArrayList<Integer>();
	}
}
```

En este caso hemos empezado definiendo qué sucede cuando intentamos calcular los números primos hasta el 1. Buscamos ahora una nueva prueba que nos haga evolucionar mediante *triangulación*, por ejemplo buscando los primos del número 2, cuyo resultado esperado debe ser el propio número 2. Aplicamos un nuevo ciclo TDD con el par prueba-código de producción del código 3.2.

Código 3.2

Prueba

```
@Test
public void testCalculaConValorInicialDos() {
	List<Integer> l = CribaDeEratosthenes.Calcula(2);
	assertEquals(1, l.size());
	assertEquals(new Integer(2), l.get(0));
}
```

Implementación

```
public static List<Integer> Calcula(int i) {
	List<Integer> l = new ArrayList<Integer>();

	if (i >= 2)
		l.add(2);

	return l;
}
```

La prueba verifica que el resultado es un único número y que dicho número es el 2, que es el resultado esperado al calcular los primos de 2.

Esta prueba falla, ya que el código de producción siempre devuelve una lista vacía. A continuación escribimos el mínimo código para superar esta y todas las demás pruebas que ya tenemos escritas (código 3.2). Al final de la traza veremos una manera más cómoda de escribir este tipo de asserts utilizando la librería de Java.

Vamos a continuar triangulando para añadir más código al método. La siguiente prueba verifica los resultados de calcular los primos de 3, para lo cual la lista resultante debe tener un nuevo elemento (código 3.3).

Código 3.3

Prueba

```
	@Test
	public void testCalculaConValorInicialTres() {
		List<Integer> l = CribaDeEratosthenes.Calcula(3);

		assertEquals(2, l.size());
		assertEquals(new Integer(2), l.get(0));
		assertEquals(new Integer(3), l.get(1));
	}
```

Implementación

```
// Código
		public static List<Integer> Calcula(int i) {
			List<Integer> l = new ArrayList<Integer>();

			if (i >= 2) {
				l.add(2);
				l.add(3);
			}

			return l;
		}
```

En este nuevo paso puedes ver que hemos entrado en un ciclo que no nos aporta nada. Si ahora hiciéramos una prueba con el valor 5, el mínimo código sería añadir el valor 5 a la lista devuelta. Pero en este caso no estamos calculando nada.

Este ciclo se puede romper haciendo una refactorización. Con ella en vez añadir uno a uno los valores esperados los calculamos. Para hacer esta refactorización Tendríamos que dar un paso muy grande con muchos cambios que pueden salir mal para implementar el código del algoritmo. No estaríamos sacándole partido a TDD.

Llegados a este punto ya nos damos cuenta de que los casos de prueba no nos ayudan a evolucionar el código y que refactorizar sería que implementar todo el método de una vez, sin aplicar TDD. Hemos encontrado un mal camino, es decir, una manera de trabajar que nos no lleva a ningún buen resultado y que nos empuja a probar otras maneras distintas de trabar.

Vamos a cambiar la manera de aplicar TDD. No tires el código que hemos escrito hasta ahora. Podremos aprovecharlo más adelante.

Un nuevo comienzo

El primer paso del nuevo comienzo va a ser redactar el diario de diseño (cuadro X). En este ejemplo el diario es los pasos del algoritmo.

3.1. Diario de diseño

- Construir una lista con los números desde 2 hasta n en el que ningún elemento está marcado.
- Marcar los múltiplos de cada número x de la lista (x*2, x*3, etc.).
- Construir una lista con todos los números no marcados (la lista de números primos).

Para avanzar pasos más diminutos, cada paso del algoritmo lo implementaremos en un método con sus propias pruebas. Aunque dichos métodos deberían ser privados, para probarlos con comodidad los pondremos con el ámbito de visibilidad menos restrictivo.

El primer paso que vamos a abordar es crear una matriz de booleanos para indicar qué números están marcados y cuáles no. Escribimos una prueba y, después haremos una implementación obvia (código 3.4).

Código 3.4

Prueba

```
@Test
public void testCreaListaDeNumerosSinMarcar() {
            int tope = 4;
      List<Boolean> l =
CribaDeEratosthenes.CreaListaDeNumerosSinMarcar(tope);

            assertEquals((tope+1), l.size());
            for (Boolean b:l) {
                  assertFalse(b);
            }
}
```

Implementación

```
public static List<Boolean> CreaListaDeNumerosSinMarcar(int i) {
            List<Boolean> lb = new ArrayList<Boolean>();
            for (int c=0; c<=i; c++)
                  lb.add(false);
            return lb;
}
```

En la implementación obvia del código 3.4, se ha incrementado el tope en 1 ya que para que el número 4 aparezca en la lista de marcados, es necesario que la lista tenga 5 elementos (del 0 al 4). Ignoraremos las posiciones 0 y 1 que siempre serán false, ya que no intervienen en el algoritmo.

No vamos a tener en cuenta la posibilidad de recibir como parámetro un valor inferior a dos, ya que los valores incorrectos serán filtrados por el método que llamará al método *CreaListaDeNumerosSinMarcar*. Aun así, no es una mala idea añadirlo, lo cuál te propongo que ejercicio adicional.

El primer paso de nuestro diario ya está implementado (cuadro 3.1).

3.1. Diario de diseño

- ~~Construir una lista con los números desde 2 hasta n en el que ningún elemento está marcado.~~
- Marcar los múltiplos de cada número x de la lista (x*2, x*3, etc.).
- Construir una lista con todos los números no marcados (la lista de números primos).

Continuamos con la funcionalidad para marcar todos los múltiplos de un número dado. Antes de empezar la implementación vamos a reflexionar qué significa esta funcionalidad diseñando algunos ejemplos como los que se muestran a continuación

- Si se calculan los primos de 2, nos e marca ningún número
- Si se calculan los primos de 3, nos e marca ningún número
- Si se calculan los primos de 4 se marca el 4
- Si se calculan los primos de 5 se marca el 4

Con los ejemplos anteriores ya tenemos más claro la funcionalidad que vamos a implementar. Además vamos a utilizar los ejemplos que consideremos interesantes como casos de prueba. Vamos a empezar escribiendo una prueba que marque los valores para calcular los primos de 4, ya que es el primer número en el que hay un cambio en la lista de números marcados (código 3.5). Si usamos el 2 o el 3, al no marcarse ningún número, no habría sido necesario escribir código, por lo que no son valores de prueba adecuados para hacer un ciclo de TDD.

Código 3.5

```
	@Test
	public void testMarcarMultiplosHasta4() {
		List<Boolean>                    l                    =
CribaDeEratosthenes.CreaListaDeNumerosSinMarcar(4);

		CribaDeEratosthenes.MarcarMultiplos(l);

		assertFalse(l.get(2));
		assertFalse(l.get(3));
		assertTrue(l.get(4));
	}
```

La implementación obvia que pasa la prueba anterior se muestra en el código 3.6.

Código 3.6

```
// Código
	public static void MarcarMultiplos(List<Boolean> l) {
		for (int num = 2; num < l.size(); num++) {
			for (int mul = (num*2); mul < l.size(); mul += num) {
				l.set(mul, true);
			}
		}
	}
```

La prueba funciona. Ya tenemos implementada otra característica del diario de diseño.

3.1. Diario de diseño

- ~~Construir una lista con los números desde 2 hasta n en el que ningún elemento está marcado.~~
- ~~Marcar los múltiplos de cada número x de la lista (x*2, x*3, etc.).~~
- Construir una lista con todos los números no marcados (la lista de números primos).

La siguiente característica a implementar será la creación de la lista de números primos, es decir, aquellos que no han sido marcados. Una prueba de esta funcionalidad está en el código 3.7.

Código 3.7

```
	@Test
	public void testCrearListaDePrimosHasta4() {
		List<Boolean> l =
CribaDeEratosthenes.CreaListaDeNumerosSinMarcar(4);
		CribaDeEratosthenes.MarcarMultiplos(l);

		List<Integer> primos =
CribaDeEratosthenes.CreaListaDePrimos(l);

		assertEquals(2, primos.size());
		assertEquals(new Integer(2), primos.get(0));
		assertEquals(new Integer(3), primos.get(1));
	}
```

La implementación obvia que pasa la prueba anterior con éxito está en el código 3.8.

Código 3.8

```
	public static List<Integer> CreaListaDePrimos(List<Boolean> l) {
		List<Integer> lb = new ArrayList<Integer>();
		for (int c = 2; c < l.size();c++) {
			if (!l.get(c)) {
				lb.add(c);
			}
		}
		return lb;
	}
```

Con este código terminamos los pasos del algoritmo. Veamos cómo ponerlo todo junto en la siguiente sección.

Integrando

Los pasos del algoritmo ya están implementados y probados. Ahora es el momento de dichos pasos en un método que calcule la criba de Eratóstenes.

3.1. Diario de diseño

- ~~Construir una lista con los números desde 2 hasta n en el que ningún elemento está marcado.~~
- ~~Marcar los múltiplos de cada número x de la lista (x*2, x*3, etc.).~~
- ~~Construir una lista con todos los números no marcados (la lista de números primos).~~
- Crear el método principal que llame a los tres métodos auxiliares.

Vamos a reutilizar las primeras pruebas que escribimos al principio de este capítulo (código 3.1, 3.2 y 3.3), por lo que ya contamos con un conjunto de pruebas para implementar el método Calcula. La implementación de este método (*implementación obvia*) está en el código 3.9.

Código 3.9

```
// Código
public static List<Integer> Calcula(int i) {
	List<Boolean> lb =  CreaListaDeNumerosSinMarcar(i);
	MarcarMultiplos(lb);
	return CreaListaDePrimos(lb);
}
```

Las pruebas siguen funcionando por lo que ya podemos dar por terminada la implementación. Sin embargo podemos añadir algunas pruebas más por ejemplo la prueba 3.10.

Código 3.10

```
@Test
public void testGeneraPrimosHastaDoce() {
	List<Integer> l = CribaDeEratosthenes.Calcula(12);
	Assert.assertEquals(l, Arrays.asList(2, 3, 5, 7, 11));
}
```

La prueba anterior calcula todos os números primos hasta el 12 (inclusive) y, además, utiliza Arrays.asList para crear la lista de números esperada lo que facilita la comprobación final. Tachamos el último elemento de nuestro diario de diseño, lo que significa que ya hemos terminado.

Consideraciones finales

Una limitación que se ha visto en este ejercicio es cómo esconder los métodos auxiliares para que solo puedan llamarse desde el método principal, pero, a la vez, permitir que se prueben de manera independiente. Todo lo que queríamos probar eran métodos privados, salvo el director que era un método público.

Es posible probar métodos privados en Java, por ejemplo invocarlo mediante introspección o crear una clase hija que orezca llamadas públicas a estos elementos. Estas dos alternativas requieren trabajo adicional y hacen a las pruebas más frágiles. Como no existe la solución perfecta en este caso, como decisión personal, he optado por abrir la visibilidad de los métodos y he documentado que dichos métodos son de uso interno. Con ello he hecho que las pruebas sean lo más sencilla posibles.

En ningún momento de este ejemplo hemos refactorizado el código, porque no hemos visto que fuera necesario, pero esto es muy poco frecuente.

En este ejemplo, hemos implementado el diario de diseño siguiendo los mismos pasos que emplea la Criba de Eratóstenes, pero recuerda que no hay limitación a la hora de elegir la siguiente característica a implementar.

Para reflexionar.

Hemos empezado el ejercicio tomando un camino que, con el tiempo, hemos descubierto que no era el correcto. Los malos olores que hemos encontrado han sido realizar varios *fakes* seguidos y no refactorizar.

Es difícil aplicar TDD para implementar métodos largos porque, como hemos visto al principio, es difícil ir añadiendo fragmentos de código a un método a medida que añadimos más pruebas. En cambio, TDD funciona muy bien descomponiendo el problema en pasos pequeños y encapsulándolos en métodos. El diario de diseño también nos ayuda a ello cómo has visto a lo largo de este capítulo.

Aunque hayamos empezando por un camino incorrecto no ha sido una pérdida de tiempo. Hemos descubierto una manera de trabajar que no nos funcionaba y que nos ha ayudado a buscar otro camino distinto y hemos aprovechado las primeras pruebas. Es muy común aplicando TDD descubrir que nuestros primeros pasos no van por buen camino. TDD no nos indica el camino correcto, pero nos ayuda a detectar rápidamente los caminos incorrectos.

En el ejemplo anterior se dan algunas indicaciones de ejercicios adicionales para realizar. A continuación te proponemos uno más. Es posible optimizar el algoritmo parando cuando ya no queden más números por marcar. Intenta añadir esta mejora al código que se ha visto en este capítulo. ¿Tendría sentido aplicar TDD para implementar esta mejora?. También es posible desarrollar nuevas implementaciones utilizando estructuras más eficientes, como Bitset, o las nuevas funciones lambda y los streams de Java 8.

En un universo alternativo

¿Qué hubiera sucedido si hubiéramos intentado implementar este algoritmo al revés? Es decir, si en vez de escribir el método *Calcula* al final hubiéramos intentado escribirlo al

principio (código 3.11), a partir de las primeras pruebas que hemos visto (por ejemplo en el código 3.1).

Código 3.11

```
// Código
public static List<Integer> Calcula(int i) {
       List<Boolean> lb =  CreaListaDeNumerosSinMarcar(i);
       MarcarMultiplos(lb);
       return CreaListaDePrimos(lb);
}
```

La solución no tiene por qué ser mucho más difícil ni muy distinta a lo que has visto a lo largo de este capítulo. Vamos a comentar a continuación cómo sería el desarrollo.

Para que la prueba pudiera ejecutarse con éxito es necesario implementar los tres métodos que *Calcula* llama. Para poder hacerlo en un único pequeño paso, estas implementaciones podrían ser fakes. Por ejemplo, *CreaListaDeNumerosSinMarcar* podría devolver null, *MarcarMultiplos* sería un método vacío y *CreaListaDePrimos* devolvería el resultado espera por la prueba del método *Calcula*.

A partir de aquí ya podríamos triangular para evolucionar los fakes de los métodos auxiliares por el código en producción.

Capítulo 4.
Dos soluciones para el Mancala

En este capítulo vamos a resolver la cata del juego Mancala. Uno de los primeros problemas que tendremos que afrontar, antes de implementar la lógica de negocio, es cómo modelar la información y qué interfaz externa es la más cómoda para interactuar con el Mancala.

Este capítulo tiene una organización distinta a los capítulos anteriores. Después de exponer el problema que vamos a abordar, veremos dos soluciones. En la primera aplicaremos TDD pero de una manera descuidada. En la segunda aplicaremos TDD mucho mejor y así, tú, lector, podrás comparar ambas estrategias y sacar tus propias conclusiones.

El problema

El Mancala (o Awale o Awir, etc.) es la familia de juegos más antigua que se conoce. Sus doscientas variantes se extendieron por África y parte de Asia. En esta cata vamos a implementar el movimiento de una de estas variantes.

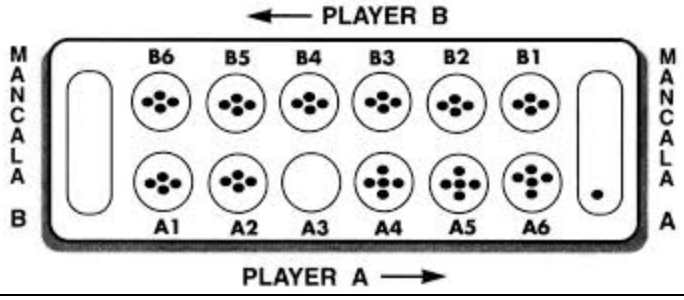
El tablero de juego consta de 6 pozos para cada jugador y dos chozas (una para cada jugador). Cada pozo tendrá cuatro semillas y las chozas estarán vacías.

Imagen 1. Tablero de un Mancala clásico

En cada turno, el jugador selecciona uno de sus 6 pozos (nunca la choza), saca todas las semillas de él y las planta una a una en los pozos consecutivos siguiendo el sentido contrario de las agujas del reloj. No se puede seleccionar un pozo si no hay semillas en él.

En la siguiente imagen acaba de empezar la partida y el jugador A ha seleccionado su pozo 3, por lo que ha puesto una semilla en los pozos 4, 5, 6 y en su choza.

Imagen 2. Distribución de los pozos



Con todo lo anterior la cata ya tiene una dificultad elevada, pero si aún quieres implementar más funcionalidad, añade las siguientes características.

- Si se da una vuelta completa, no se planta ninguna semilla en el pozo original.
- Si la última semilla la plantas en un pozo vacío de tu propiedad, coge todas las semillas del pozo de enfrente y guárdalas en tu choza.
- Cuando le toque mover a un jugador y no tenga semillas en sus pozos la partida termina. El jugador contrario guarda todas sus semillas en su choza y el ganador es el jugador con mayor número de semillas.

A continuación, resolveremos este problema aplicando TDD pero no lo haremos bien. No tomes las siguientes secciones como ejemplo a seguir porque son justo lo contrario, cosas que deberías evitar cuando apliques TDD.
Cuando leas esta primera parte intenta buscar fallos en la manera de aplicar TDD. Al final de esta primera encontrarás una retrospectiva dónde analizaremos estos fallos.

Cada semilla en su pozo

Ya tenemos claro el enunciado, así que no perdamos el tiempo y vamos con nuestra primera prueba. De golpe se nos ocurre representar los pozos y chozas con una clase que gestionen las semillas que contienen, así el código será más sencillo de escribir. Para poder crear esta clase necesitamos una prueba, así que vamos por ello (código 4.1).

Código 4.1.

```
public class TestCell {
	int seeds = 4;

	@Test
	public void testIndicateTheNumberOfSeedsWhenCreatingACell() {
		Cell c = new Cell(seeds);

		assertEquals(seeds, c.getSeeds());
	}
}
```

Y a continuación, en el código 4.2, tenemos nuestra esperada, y esperemos que útil, clase *Cell*.

Código 4.2.

```
public class Cell {

	int seeds;

	public Cell(int seeds) {
		this.seeds = seeds;
	}

	public int getSeeds() {
		return seeds;
	}}
```

El código es lo suficientemente sencillo y rápido para considerarlo un *babystep* y escribirlo de una única vez. De hecho la mayoría del código del cuadro 4.2 puede generarlo Eclipse (u otros IDEs) automáticamente.

Ahora que ya tenemos las celdas para guardar semillas vamos a utilizarlas para crear el tablero. Como el tablero tiene una configuración inicial definida (4 semillas en cada pozo y las chozas vacías) vamos a escribir una prueba que recoja todo lo anterior y que nos empuje a crearlo (código 4.3).

Código 4.3.

```
public class TestBoard {

	@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		Board b = new Board();
		int cellCount = 0;

		for (Cell c: b.getCells()) {
			cellCount++;
			assertEquals(c.getSeeds(), 4);
		}

		assertEquals(12, cellCount);
	}
}
```

Con el código 4.3 comprobamos que una instancia de *Board* tenga 12 pozos (objetos de la clase *Cell*) cada una con 4 semillas. Ahora implementamos la clase *Board* para que la prueba anterior pase con éxito, pero no nos vamos a parar a ver ese código, sino que vamos a ir a la siguiente prueba.

En la siguiente prueba, código 4.4, comprobamos que el tablero también tiene dos chozas con cero semillas.

Código 4.4

```
	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		Board b = new Board();

		for (Cell c: b.getHouses()) {
			assertEquals(c.getSeeds(), 0);
		}

		assertEquals(2, b.getHouses().size());
	}
```

Ahora sí, veamos, en el código 4.5, cuál es el mínimo código de la clase *Board* que permite que las dos pruebas anteriores pasen.

Código 4.5

```
	public class Board {

	List<Cell> cells;
	List<Cell> houses;

	public Board() {
		cells = new ArrayList<Cell>();
		for (int i = 0; i < 12; i++) {
			cells.add(new Cell(4));
		}
		houses = new ArrayList<Cell>();
		for (int i = 0; i < 2; i++) {
			houses.add(new Cell(0));
		}
	}

	public List<Cell> getCells() {
		return cells;
```

```
	}

	public List<Cell> getHouses() {
		return houses;
	}

}
```

Las pruebas de los códigos 4.3 y 4.4 son muy similares. Además, en el constructor de la clase *Board* estamos repitiendo el mismo fragmento de código 2 veces. Todo lo anterior son malos olores que nos animan a refactorizar. El código refactorizado de las pruebas se muestra en el código 4.6. Hemos creado un método assert local *assertAllCellsHasSameSeeds* para evitar el código duplicado.

Código 4.6

```
public class TestBoard {

	Board b;

	@Before
	public void setUp() {
		b = new Board();
	}

	@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		assertAllCellsHasSameSeeds(b.getCells(), 4);
		assertEquals(12, b.getCells().size());
	}

	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertAllCellsHasSameSeeds(b.getHouses(), 0);
		assertEquals(2, b.getHouses().size());
	}


	void assertAllCellsHasSameSeeds(List<Cell> l, int seeds) {
		for (Cell c: l) {
			assertEquals(c.getSeeds(), seeds);
		}
	}
}
```

Y el código refactorizado de la clase *Board* se muestra en el código 4.7. En esta refactorización hemos movido el código duplicado al método privado *createListOfCells*.

Código 4.7

```
public class Board {

	List<Cell> cells;
	List<Cell> houses;

	private List<Cell> createListOfCells(int cellsNumber, int value) {
		List<Cell> cells = new ArrayList<Cell>();
		for (int i = 0; i < cellsNumber; i++) {
			cells.add(new Cell(value));
		}
		return cells;
	}
```

```
	public Board() {
		cells = this.createListOfCells(12, 4);
		houses = this.createListOfCells(2, 0);
	}

	public List<Cell> getCells() {
		return cells;
	}

	public List<Cell> getHouses() {
		return houses;
	}
}
```

Ahora que ya tenemos nuestro tablero, chozas y pozos, vamos a dar el siguiente paso y vamos a implementar la operación de sembrar.

Sembrar para el futuro

Cuando sembramos, desencadenamos varios efectos. Uno de ellos es que el pozo original queda sin semillas. Vamos a escribir una prueba que ponga esto de manifiesto.

Código 4.8

```
@Test
	public void testSeeding4SeedsFromCell0_CellHas0Seeds() {
		b.seed(0);

		assertEquals(0, b.getCells().get(0).getSeeds());
	}
```

En este caso, estamos identificando los pozos por su índice. Uno de los jugadores tendrá el pozo de 0 a 5 y el otro de 6 a 11. Para implementar esta prueba tenemos que añadir el método *seed* a la clase *Board* (código 4.9).

Código 4.9. Clase *Board*

```
	public void seed(int i) {
		this.cells.get(i).removeSeeds();
	}
```

Para que la prueba funcione, tenemos que implementar el método *removeSeeds* en *Cell* para poder quitar las semillas de ese pozo. Su implementación se muestra en el código 4.10

Código 4.10. Clase *Cell*

```
	public void removeSeeds() {
		seeds = 0;
	}
```

La prueba original (código 4.8) nos ha empujado a implementar también un método en la clase *Cell* aunque no hemos escrito ninguna prueba para este método. La prueba del método *removeSeed* es la propia prueba del código 4.8 ya que si el método *removeSeed* de *Cell* no estuviera bien implementado, esta prueba fallaría.

Nuestro siguiente paso será añadir las semillas que hemos quitado al resto de los pozos.

Haced sitio

Otro cambio en el tablero, cuando sembramos es que los pozos (y chozas) adyacentes ganan una semilla. Vamos a escribir una prueba que siembre las semillas del pozo 0 y verifique que el número de semillas de los pozos 1, 2, 3 y 4 se ha incrementado en uno.

Código 4.11. Clase *Cell*

```
	@Test
	public void
	testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		assertEquals(seeds+1, b.getCells().get(1).getSeeds());
		assertEquals(seeds+1, b.getCells().get(2).getSeeds());
		assertEquals(seeds+1, b.getCells().get(3).getSeeds());
		assertEquals(seeds+1, b.getCells().get(4).getSeeds());

	}
```

Con la prueba del código 4.11, triangulamos el método *seed*. Tenemos que añadir código adicional al método *seed* para que esta prueba pase con éxito. Vamos a ver cómo quedaría (código 4.12). Hay que volver a añadir un nuevo método a la clase *Cell*. Este método también se muestra en 4.12

Código 4.12

Método seed de la clase board

```
public void seed(int cellIndex) {
		int seeds = this.cells.get(cellIndex).removeSeeds();
		for (int i = cellIndex+1; i <= seeds; i++) {
			this.cells.get(i).addSeed();
		}

	}
```

Nuevo método para la clase Cell

```
public void addSeed() {
		seeds++;
	}

	public int removeSeeds() {
		int tmp = this.seeds;
		seeds = 0;
		return tmp;
	}
```

También hemos tenido que modificar el método *removeSeed* para que devuelva las semillas que había en el pozo antes de quitarlas.

En este caso, no estamos dando *babysteps*, sino que hemos avanzamos más de lo conveniente. Discutiremos este aspecto en la retrospectiva (más adelante en este capítulo). Ahora, vamos a continuar con nuestra implementación.

Una pausa para refactorizar

Hemos hecho dos ciclos de TDD, que nos han permitido quitar las semillas del pozo y plantarlas en los pozos consecutivos, pero no hemos refactorizado. No vamos a avanzar más sin mejorar la calidad de nuestro código.

Vamos a comenzar por quitar los cuatro *assertEquals* de la prueba del código 4.11. Como las celdas se almacenan en una lista, podemos aprovechar el método *sublist* de la interfaz *java.util.List* y el método *assertAllCellsHasSameSeeds* que creamos al principio del capítulo, para escribir una prueba más clara y compacta en el código 4.13.

Código 4.13. Prueba refactorizada

```
	@Test
	public                                                          void
testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		int expectedSeeds = Board.Default_Seeds+1;
		List<Cell> affectedCells = b.getCells().subList(1, 5);
		assertAllCellsHasSameSeeds(affectedCells, expectedSeeds);
	}
```

También usamos variables locales (*expectedSeeds* y *affectedCells*) para indicar qué papel juega los parámetros de *assertAllCellsHasSameSeeds*. Vamos a continuar refactorizando

Las pruebas que hemos escrito, dependen de que la clase *Board* tenga un método que devuelva una lista de objetos *Cell*. Si esto cambiara, afectaría a todas las pruebas que hemos escrito. Uno de los puntos fuertes que nos da TDD es que hace fácil cambiar el código, sin embargo en este caso estamos poniendo impedimentos al cambio.

Vamos a solucionar esto centralizando en el método *setUp* de la prueba el acceso a la lista de celdas. Si en el futuro decidimos que la clase *Board* no devuelve una lista de objetos *Cell*, podremos incluir el código para crear dicha lista el método *setUp*.

Código 4.14. Un único método para obtener la lista de celdas.

```
public class TestBoard {

	Board b;
	List<Cell> cells;

	@Before
	public void setUp() {
		b = new Board();
		cells = b.getCells();
```

```
	}

	@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		assertAllCellsHasSameSeeds(cells, Board.Default_Seeds);

		assertEquals(12, cells.size());
	}

	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertAllCellsHasSameSeeds(b.getHouses(), 0);

		assertEquals(2, b.getHouses().size());
	}

	@Test
	public void testSeeding4SeedsFromCell0_CellHas0Seeds() {
		b.seed(0);

		assertEquals(0, cells.get(0).getSeeds());
	}

	@Test
	public void
testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		int expectedSeeds = Board.Default_Seeds+1;
		List<Cell> affectedCells = cells.subList(1, 5);
		assertAllCellsHasSameSeeds(affectedCells, expectedSeeds);
	}

	public void assertAllCellsHasSameSeeds(List<Cell> l, int seeds) {
		for (Cell c: l) {
			assertEquals(c.getSeeds(), seeds);
		}
	}
}
```

Terminamos aquí la refactorización y continuamos avanzando en la funcionalidad.

Nada que plantar

¿Qué sucede si en el pozo que hemos elegido no hay ninguna semilla? Parece lógico decidir que no debe pasar nada, es decir, que el siguiente pozo no cambie el número de semillas que contiene. El código 4.15 expresa esto en una prueba.

Código 4.15. Prueba refactorizada

```
@Test
	public void testSeedingEmptyCell_Cell_1_DoesnotChange() {
		cells.get(0).removeSeeds();

		b.seed(0);

		assertEquals(Board.Default_Seeds, cells.get(1).getSeeds());
	}
```

La prueba anterior funciona sin necesidad de escribir código adicional, ya que el código que necesitamos lo escribimos al triangular el método *seed* en el código 4.12. Dejamos esta prueba para completar el conjunto de pruebas de la clase *Board* y buscamos algo nuevo para implementar

De vuelta a la choza

¿Te has dado cuenta de que, hasta ahora, no hemos tenido en cuenta las chozas para nada? Es el momento de que entren en juego.

En el Mancala, vamos plantando cada semilla en un pozo o choza siguiendo el sentido inverso de las agujas del reloj. Teniendo los pozos y chozas en dos listas separadas, cuando llegamos al final de los pozos de un jugador, hemos de ir a la lista de chozas para plantar una semilla en la choza de ese jugador y luego volver a la lista de pozos para confinar plantado las semillas restantes en los pozos del otro jugador.

Dos listas complican el código y no apreciamos ninguna ventaja a tener pozos y chozas por separado, así que vamos a almacenar pozos y chozas en una sola lista. Para ello, modificamos las pruebas que verifican el estado inicial para reflejar este cambio. La prueba original y la nueva prueba están en el código 4.14

Código 4.14

Pruebas que borramos

```
@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		assertAllCellsHasSameSeeds(b.getCells(), 4);

		assertEquals(12, b.getCells().size());
	}

	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertAllCellsHasSameSeeds(b.getHouses(), 0);

		assertEquals(2, b.getHouses().size());
	}
```

Prueba que añadimos

```
	@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		assertAllCellsHasSameSeeds(cells.subList(0, 6),
Board.Default_Seeds);
		assertAllCellsHasSameSeeds(cells.subList(7, 13),
Board.Default_Seeds);

		assertEquals(14, cells.size());
	}

	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertEquals(0, cells.get(6).getSeeds());
		assertEquals(0, cells.get(13).getSeeds());
	}
```

Código 4.15. Pozos y celdas en la misma lista

```
public class Board {

	List<Cell> cells;
	public static final int Default_Seeds = 4;

	List<Cell> createListOfCells(int cellsNumber, int value) {
		List<Cell> cells = new ArrayList<Cell>();
		for (int i = 0; i < cellsNumber; i++) {
			cells.add(new Cell(value));
		}
		return cells;
	}

	public Board() {
		cells = this.createListOfCells(14, Default_Seeds);
		cells.get(6).removeSeeds();
		cells.get(13).removeSeeds();
	}
//…..
}
```

Con este cambio, la primera fila de pozos tendrán los índices de 0 a 5, el índice 6 será una choza, la segunda fila de pozos tendrán los índices de 7 a 12 y la segunda choza será el índice 13. Ya podemos cambiar el código para que pozos y chozas se guarden en la misma lista.

No uses las chozas

Vamos a escribir ahora una prueba que verifique que trabajamos adecuadamente con las chozas. La primera prueba verificará que, si se selecciona una choza, no debe cambiar ninguna semilla de sitio (código 4.16).

Código 4.16.

```
@Test
	public void testSeedingHouseCell_BoardDoesNotChange() {
		int houseFirstPlayer = 6;
		b.seed(houseFirstPlayer);

		assertEquals(Board.Default_Seeds,
cells.get(houseFirstPlayer + 1).getSeeds());
	}
```

La primera vez que hice y ejecuté esta prueba funcionó correctamente cuando no debería haberlo hecho. Tuve que desarrollar varias pruebas para darme cuenta de cuál había sido el error. ¿Eres capaz de encontrar el error en el código 4.12? Recuerda que la prueba del código 4.11 funciona correctamente.

Vamos a explicar el error del código 4.12 a continuación. Como el parámetro *cellIndex* siempre vale 4, en la prueba solo plantábamos desde el pozo 1 hasta la 4. Si quisiéramos sembrar las semillas del pozo 1 en vez de el pozo 0, el código solo plantaba en los pozos 2, 3 y 4. Si intentáramos sembrar las semillas del pozo 5 en adelante, el código no plantaría en ningún pozo ya que el bucle *for* no llegaba a dar ninguna vuelta.

Encontrarme este fallo me hizo pensar de nuevo en algo que ya hemos mencionado. ¿Hemos escrito demasiado código cuando implementamos el bucle de *seed*? ¿No podríamos haber dado pasos más pequeños? Retomaremos estas preguntas en la retrospectiva.

Una vez arreglado el error e implementado el código necesario para que la prueba del código 4.16 pase con éxito, nos queda el código 4.17.

Código 4.17.

```
public void seed(int cellIndex) {
		if (cellIndex == 6)
			return;

		int seeds = this.cells.get(cellIndex).removeSeeds();

		int initialCell = cellIndex+1;
		int finalCell = cellIndex + seeds;

		for (int i = initialCell; i <= finalCell; i++) {
			this.cells.get(i).addSeed();
		}

	}
```

A continuación podríamos escribir una prueba similar para evitar que se utilice la otra choza. No vamos a mostrar dicho código. Ya estamos a punto de acabar con toda la funcionalidad. Vamos a darle un último empujón

De la última a la primera

La última funcionalidad pendiente es que, cuando plante en la última celda (el pozo del segundo jugador) y aún queden semillas por sembrar, se continúe plantando por la primera celda. En otras palabras, después del objeto *Cell* del índice 13 vendría el objeto *Cell* del índice 0. El número de vueltas del bucle viene determinado por las semillas a plantar.

Vamos a definir una prueba que muestre este escenario en el código 4.18.

Código 4.18. Seleccionamos el último pozo y la semillas se plantan en los primeros

```
@Test
	public void testSeedingLasCell_SeedsArePlantedInFirstPits() {
		b.seed(12);

		assertEquals(1, this.cells.get(13).getSeeds());
		assertAllCellsHasSameSeeds(this.cells.subList(0, 3),
Board.Default_Seeds+1);
```

```
	}
```

Solo tenemos que añadir una condición al método *seed* para que, si llega al final vuelva al principio. El resultado se muestra en el código 4.19.

Código 4.19.

```
public void seed(int cellIndex) {
		if (cellIndex == 6)
			return;

		int seeds = this.cells.get(cellIndex).removeSeeds();

		int initialCell = cellIndex+1;
		int finalCell = cellIndex + seeds;

		int index;
		for (int i = initialCell; i <= finalCell; i++) {
			index = i % 14;
			this.cells.get(index).addSeed();
		}

	}
```

Ya hemos terminado con todo lo que teníamos hacer, ya no hay más funcionalidad nueva que implementar.

La última refactorización

Las pruebas anteriores son muy dependientes de detalles de implementación, como el rango de índices necesarios para un grupo de pozos. Por este mismo motivo, nuestras pruebas también son difíciles de entender. Si empezamos a trabajar en otro proyecto y volvemos a este código pasada una semana, ¿entenderemos qué significa *cells.sublist(0, 6)*? ¿Por qué una sublista? ¿Por qué exactamente ese rango?

Podemos corregir esto buscando nuevas abstracciones. Una de ellas es el concepto de jugador. El primer jugador será propietario de algunos de los pozos (del 0 al 6 para incluya su choza) y el segundo jugador será propietario de los demás pozos (del 7 al 13). Vemos como queda esta refactorización en el 4.18.

Código 4.17

Prueba utilizando índices

```
@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		List<Cell> cellsFirstPlayer = cells.subList(0, 6);
		assertAllCellsHasSameSeeds(cellsFirstPlayer,
Board.Default_Seeds);

		List<Cell> cellsSecondPlayer = cells.subList(7, 13);
		assertAllCellsHasSameSeeds(cellsSecondPlayer,
Board.Default_Seeds);

		assertEquals(14, cells.size());
	}
```

Misma prueba utilizando jugadores

```
@Test
	public void testNewBoardHas12CellsWith4SeedsEach() {
		assertAllCellsHasSameSeeds(b.getCellsFirstPlayer(),
Board.Default_Seeds);
		assertAllCellsHasSameSeeds(b.getCellsSecondPlayer(),
Board.Default_Seeds);

		assertEquals(14, cells.size());
	}
```

Podemos hacer lo mismo para las chozas, como se muestra en el código 4.18.

Código 4.18.

Prueba utilizando índices

```
	@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertEquals(0, cells.get(6).getSeeds());
		assertEquals(0, cells.get(13).getSeeds());
	}
```

Mima prueba utilizando jugadores

```
@Test
	public void testNewBoardHas2HousesWithoutSeeds() {
		assertEquals(0, b.getHouseFirstPlayer().getSeeds());
		assertEquals(0, b.getHouseSecondPlayer().getSeeds());
	}
```

Vamos a utilizar esta abstracción para introducir índices relativos a los jugadores. Es decir, ahora tendremos el pozo 0 del primero jugador y el pozo 0 del segundo jugador (que será el índice 7 de la lista de la clase *Board*). De esta manera las pruebas quedarían así (código 4.19).

Código 4.19

Prueba utilizando índices

```
@Test
	public void
testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		int expectedSeeds = Board.Default_Seeds+1;
		List<Cell> affectedCells = cells.subList(1, 5);
		assertAllCellsHasSameSeeds(affectedCells, expectedSeeds);
	}
```

Prueba utilizando jugadores

```
@Test
	public void
testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		int expectedSeeds = Board.Default_Seeds+1;
		List<Cell> affectedCells =
b.getCellsFirstPlayer().subList(1, 5);
		assertAllCellsHasSameSeeds(affectedCells, expectedSeeds);
	}
```

```
	@Test
	public void
testSeeding4SeedsFromCell_7_Cells_8_9_10_11_HasOneMoreSeeds() {
		b.seed(7);

		int expectedSeeds = Board.Default_Seeds+1;
		List<Cell> affectedCells =
b.getCellsSecondPlayer().subList(1, 5);
		assertAllCellsHasSameSeeds(affectedCells, expectedSeeds);
	}
```

Podríamos continuar refactorizando el código. A continuación se exponen algunas ideas de posibles refactorizaciones.

Podríamos añadir un método que devuelva sólo los pozos de un jugador y otro método que devuelva sólo las chozas, pero si quisiéramos escribir una prueba que comenzara plantando en el pozo de un jugador y terminara plantando en el pozo del otro jugador, tendríamos que hacer tres llamadas, una obtener los pozos del primer jugador y comprobar el incremento de semillas, otra para obtener la choza y otra para obtener los pozos del segundo jugador.

Otra alternativa podría ser ocultar todo lo anterior tras un método assert que, por ejemplo, tomara como entrada la celda original y la cantidad de celdas (incluida la choza) que queremos verificar.

Si después de terminar de escribir código con TDD necesitamos una gran refactorización como la que hemos hecho en esta sección, es probable que no hayamos aplicado bien TDD.

A pesar de estas refactorizaciones, ni el proceso que hemos seguido ha sido todo lo sencillo y elegante que debería ser aplicando TDD, ni la solución ha quedado sencilla. Vamos a analizar los fallos y vamos a intentar corregirlos.

Retrospectica de la primera solución

Los problemas que hemos encontrado y que vamos a analizar en esta sección son.

- Exponer detalles de implementación y dependencia de los índices.
- Confundir el concepto de pocas líneas de código con el concepto de *babysteps*.
- Implementar dos clases a la vez con un único conjunto de pruebas.
- No detectar el mal olor de incumplir la ley de Demeter.
- Separación del dominio del problema y de la solución.
- No hemos utilizado un diario de diseño.

Durante todo el desarrollo hemos sido muy dependientes de listas e índices. Hemos necesitado saber, por ejemplo, los índices que correspondían a las chozas y los

pozos. Hemos expuesto detalles de implementación desde el principio (la lista de celdas).

Tampoco hemos codificado en *babysteps* de manera estricta. Hemos adelantado demasiado e implementamos código no cubierto por pruebas. Por ejemplo en la sección "Haced sitio", sólo teníamos que escribir el código para quitar las semillas del pozo 0 y añadirlas en los pozos 1, 2, 3, y 4. Sin embargo escribimos una solución general para cualquier escenario.

¿Esta solución funciona para todos los pozos? ¿Es la solución más adecuada? Hemos necesitado más pruebas para verificar que la implementación del método *seed* funciona en otros escenarios (que ya podemos ver que no funciona porque no tiene en cuenta las chozas). No cuenta solo la cantidad del código sino la complejidad del mismo. Este caso ilustra como unas pocas líneas complejas pueden dar lugar a errores inesperados, como nos pasó, ya que no comprobamos que estuviéramos calculando correctamente el límite del bucle.

Una solución más adecuada hubiese sido empezar con la estrategia *fake* y añadir una segunda prueba para evolucionar el código (estrategia de triangulación). Esta segunda prueba habría detectado el error en el bucle.

Hemos estado trabajando con dos clases simultáneamente, las clases: *Board* y *Cell*. Sin embargo, solo hemos escrito pruebas para la clase *Board* (salvo la primera prueba de la cata) y no hemos escrito pruebas unitarias para la clase *Cell* a pesar de añadir y modificar su código durante la cata. Hemos utilizado las pruebas de *Board* para probar el código de *Cell*. Haber escrito una prueba para *Cell* antes de implementar la prueba para *Board* no hubiese sido la solución más adecuada. Veamos porqué.

Desde el principio, el autor de esta solución tenía un diseño en su mente, con una clase *Board* y una clase *Cell* y ha querido implementarlo aplicando TDD. Empezar a aplicar TDD con un posible diseño en la mente no es malo. TDD nos avisa de si este diseño funciona o no. El problema es cuando nos centramos en nuestra idea y no escuchamos a TDD, y esto es lo que ha pasado.

En las secciones anteriores, TDD nos estaba avisando mediante las pruebas (y el mal olor que veremos continuación) de que un cambio en *Board* implicaba un cambio en *Cell*. Ambas clases están tan ligadas que no tiene sentido considerarlas por separado, por lo que *Cell* sería una clase interna de *Board*.

> *TDD te obliga a escribir todo el código en una única clase para extraerlo, mediante refactorizaciones. TDD es muy útil cuando no tenemos claro qué clases construir, pero si ya sabemos las clases que necesitamos: adelante. Con TDD descubrirás si tu diseño es fácil de usar y probar.*

Por si lo anterior no fuera suficiente, mantener ambas clases separadas incumple la ley de Demeter e introduce un mal olor en nuestro código que nos ha acompañado durante todo el desarrollo. Puedes ver este mal olor, por ejemplo (por

ejemplo en el código 4.8) o en el código 4.2 que repetimos a continuación. Repasemos el código 4.2 dónde también aparece este mal olor.

Código 4.2.

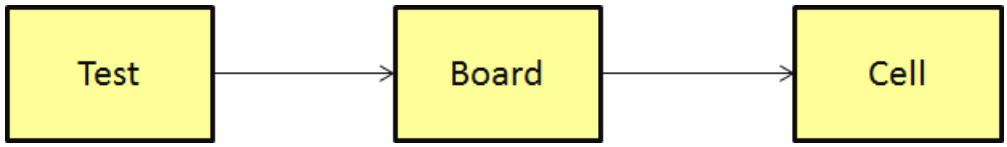
```
@Test
	public void
testSeeding4SeedsFromCell0_Cells_1_2_3_4_HasOneMoreSeeds() {
		b.seed(0);

		assertEquals(seeds+1, b.getCells().get(1).getSeeds());
		assertEquals(seeds+1, b.getCells().get(2).getSeeds());
		assertEquals(seeds+1, b.getCells().get(3).getSeeds());
		assertEquals(seeds+1, b.getCells().get(4).getSeeds());

	}
```

La ley de Demeter nos dice, de manera resumida, que un objeto solo debe comunicarse con sus amigos inmediatos, es decir, con los objetos de los que tiene dependencia directa, como parámetros y atributos. En el código 4.2, el objeto que contiene los métodos de prueba tiene una dependencia del objeto de la clase *Board.* Este objeto, a su vez, depende de objetos de la clase *Cell (imagen 03)*. Sin embargo la prueba no depende de *Cell* por lo que no debería comunicarse con estos objetos, cosa que sí está pasando en el código 4.2.

Imagen 03. Dependencias de las clases



Este mal olor aparece cuando separo la lógica de negocio los datos que dicha lógica de negocios necesita. Es fácil de detectar cuando descubrimos que tenemos que llamar a un método que devuelve un objeto sobre el que llamamos a otro método que devuelve otro objeto, y así varias veces hasta que, al final, llegamos al valor que buscábamos.

Esto aumenta la dependencia de otros objetos, lo que hace al código más difícil de utilizar y de cambiar. Una buena estrategia para evitar este mal olor es mover la lógica a quién tiene los datos. Lo correcto hubiera sido que el propio tablero nos devolviera el número de semillas, por ejemplo con un método *getSeedInCell(int index)*, en lugar del propio objeto *Cell*.

Otro problema que puede parecer trivial, pero no lo es en absoluto, es la separación entre conceptos del dominio del problema y el dominio de la solución. En el enunciado de la cata hablamos de chozas y pozos, pero la clase que las implementan se llama *Cell*.

¿Te has perdido en algún momento de la explicación o te has confundido porque hablamos de objetos *Cell* en vez de pozos (o viceversa)? Si esto sucede esto en un

ejemplo tan sencillo imagínate en un proyecto con varias personas y clientes de verdad. Si cada uno habla en su propia jerga será difícil entenderse y los malentendidos estarán al orden del día, además del tiempo que se pierde traduciendo conceptos que, en verdad, son los mismos. Lo mejor es intentar utilizar los mismos conceptos del problema en el código.

Por último, no hemos utilizado el diario de diseño en ningún momento. No sé si tú, lector, te habrás sentido perdido en algún momento durante las secciones anteriores. Te puedo asegurar que yo sí me he sentido perdido en algún momento y que, a veces, al final de una implementación o una refactorización, he tenido que volver a consultar el enunciado para hacerme una idea de qué quedaba por implementar.

Como ves, hay varias cosas que no han funcionado a pesar de haber aplicado TDD. Esto no es malo, al contrario, hemos aprendido muchos detalles interesantes. Vamos a repetir el ejercicio aprovechando la experiencia que ya tenemos y aplicando mejor las ideas de TDD que vimos en el primer capítulo y en los ejercicios anteriores.

Un nuevo comienzo.

Vamos a repetir la cata Mancala, poniendo en práctica lo que hemos aprendido, evitando los errores que hemos cometido y aplicando mejor los consejos que vimos en el primer capítulo. Por ello, lo primero que hacemos es un diario de diseño con todo lo que queremos implementar en esta cada.

4.1. Diario de diseño

- Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).
- Cada pozo tiene 4 semillas y las chozas están vacías.
- El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- Cuando un jugador planta un pozo, ese pozo queda sin semillas.
- Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Al igual que en la implementación anterior, vamos a comenzar definiendo el estado inicial del Mancala.

Nuestra segunda primera prueba.

Empezamos escribiendo una prueba que verifique el estado inicial del tablero, pero ahora vamos a intentar independizarnos lo máximo posible de la implementación y los atributos del tablero. ¿Cómo podemos hacerlo?

En Java todos los objetos tienen un método *toString* cuya misión es devolver una cadena de texto que represente al objeto. Podemos utilizar esta cadena de texto para representar el estado del Mancala, cómo muestra el código 4.20.

Código 4.20

Prueba.

```
public class TestMancala {
	@Test
	public void testManacalaInicial() {
		String expected = "0, 4, 4, 4, 4, 4, 4\n"
				+ "4, 4, 4, 4, 4, 4, 0";

		Mancala m = new Mancala();
		String mancalaState = m.toString();

		assertEquals(mancalaState, expected);
	}
}
```

Implementación fake

```
public class Mancala {
	@Override
	public String toString() {
		return "0, 4, 4, 4, 4, 4, 4\n"
			+ "4, 4, 4, 4, 4, 4, 0";
	}

}
```

En el código 4.20 hemos utilizado la estrategia *fake* y hemos declarado directamente la cadena de texto que representa el Mancala inicial en el método *toString* para que la prueba pase con éxito.

Utilizando una cadena de texto nos aislamos de los detalles de implementación de la clase Mancala y definimos en una sola prueba el número de casillas, las semillas iniciales y el estado de las chozas.

Ahora podríamos refactorizar para construir esa cadena a partir del estado de los pozos y las chozas, pero vamos a dejarlo así y vamos a ver, en la próxima sección un error muy común cuando se empieza a programar en TDD.

Un error común. Triangular el fake

Vamos a abordar ya la prueba de sembrar y vamos a comenzar, de nuevo, por comprobar que, cuando sembramos, el pozo elegido se queda sin semillas. Escribimos una prueba utilizando la representación del estado del Manca como una cadena de texto en el código 4.21.

Código 4.21.

```
	@Test
	public void testSiembroUnPozo_EsePozoQuedaSinSemillas() {
		String expected = "0, 4, 4, 4, 4, 4, 4\n"
				+ "0, 4, 4, 4, 4, 4, 0";
```

```
            Mancala m = new Mancala();
            m.seed(Player.FIRST_PLAYER, 0);
            String       mancalaState = m.toString();

            assertEquals(mancalaState, expected);
      }
```

A continuación, vamos a escribir el mínimo código posible para que dicha prueba pase (código 4.22).

Código 4.22.

```
public class Mancala {

      String s = "0, 4, 4, 4, 4, 4, 4\n"
                  + "4, 4, 4, 4, 4, 4, 0";


      @Override
      public String toString() {
            return s;
      }

      public void seed(Player firstPlayer, int i) {
            s = "0, 4, 4, 4, 4, 4, 4\n"
                  + "0, 4, 4, 4, 4, 4, 0";

      }
}
```

Aunque la prueba pasa correctamente, el código no ha evolucionado bien. En este caso (código 4.22) hemos triangulado el propio *fake*, en lugar de triangular el código real. La manera de arreglar este error es refactorizar antes y quitar el *fake*, añadiendo un auténtico estado a la clase Mancala.

Refactorizar y avanzar

Volvamos al punto en el que estábamos justo después de terminar el primer ciclo de TDD (código 4.20) y refactoricemos. Es importante que esta refactorización no introduzca apenas código nuevo, ya que si usamos las refactorizaciones para escribir el código que no hemos escrito en el momento de hacer que una prueba pase de rojo a verde estaremos aplicando mal TDD.

En este caso, lo más sencillo parece utilizar una tabla. Cada posición de la tabla representa a un jugador y contiene una segunda tabla con todos los pozos de dicho jugador, incluyendo las chozas. Los ceros representan las chozas y las dos series de 4s los pozos de ambos jugadores con sus semillas iniciales.

Código 4.23.

```
public class Mancala {
      int[] pits = {0, 4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4};
```

```
	@Override
	public String toString() {
		return Arrays.toString(pits);
	}
}
```

Java no genera directamente una cadena con los elementos de un array, así que utilizamos el método *toString* de la clase *Arrays* que ya viene en el API de Java. Sin embargo, este método devuelve una cadena con un formato distinto a la cadena que habíamos escrito en la prueba, por lo que tenemos que modificar la prueba como muestra el código 4.24.

Código 4.24. Cambiamos la cadena de texto para que la prueba pase con éxito

```
@Test
	public void testManacalaInicial() {
		String expected = "[0, 4, 4, 4, 4, 4, 4, "
					+ "0, 4, 4, 4, 4, 4, 4]";

		Mancala m = new Mancala();
		String mancalaState = m.toString();

		assertEquals(mancalaState, expected);
	}
}
```

Ya podemos tachar nuestras primeras tareas del diario de diseño.

4.1. Diario de diseño

- ~~Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).~~
- ~~Cada pozo tiene 4 semillas y las chozas están vacías.~~
- El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- Cuando un jugador planta un pozo, ese pozo queda sin semillas.
- Las y Cuando un jugador planta un pozo, el resto de pozos y chozas gana una …
- Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Triangulando la siembra

Ahora ya podemos escribir una prueba para sembrar semillas y comenzar a triangular la siembra. La prueba que escribimos justo antes de triangular el *fake* está en el código 4.25.

Código 4.25.

```
@Test
public void testSiembroUnPozo_EsePozoQuedaSinSemillas() {
	String expected = "[0, 0, 4, 4, 4, 4, 4, "
			+ "0, 4, 4, 4, 4, 4, 4]";

	Mancala m = new Mancala();
	m.seed(Player.FIRST_PLAYER, 0);
	String	mancalaState = m.toString();

	assertEquals(mancalaState, expected);
}
```

Vamos a comentar algunos detalles de esta prueba. En primer lugar estamos utilizando índices individuales para cada celda, es decir, el primer jugador tendrá sus celdas en los índices de 0 a 5 y el segundo jugador también tendrá sus celdas en esos mismos índices. En el *array*, las celdas del primer jugador serán las primeras (índices del 1 al 7) y las del segundo las segundas (índices del 9 al 14).

Esta prueba no es correcta del todo, ya que además de quedar a cero el pozo elegido, los cuatro pozos siguientes incrementan sus semillas en 1, pero es suficiente para continuar avanzando. Más adelante añadiremos más información a esta prueba.

El código mínimo para la prueba anterior es el código 4.26 (recuerda que el primer pozo del primer jugador está en el índice 1 de la tabla *pits*).

Código 4.26.

```
public void seed(Player player, int pitIndex) {
	this.pits[1] = 0;
}
```

Sin embargo la prueba del código 4.25 no pasa con éxito ya que hemos declarado la tabla *pit* como constante y sus valores no se modifican. Vamos a dejar un momento esta prueba de lado y a refactorizar la creación de la tabla como se muestra en el código 4.27. También vamos a refactorizar la prueba para calcular el índice a partir del parámetro de entrada

Código 4.27.

Refactorización para crear un array modificable.

```
int [] pits;
private final static int PLAYERS = 2;
private final static int PITS = 7;
private final static int INITIALSEEDS = 4;


public Mancala() {
	pits = new int[PLAYERS * PITS];
	Arrays.fill(pits, INITIALSEEDS);
	pits[0] = 0;
	pits[7] = 0;
}
```

Refactorización para calcular el índice a partir del parámetro.

```
public void seed(Player player, int pitIndex) {
	this.pits[pitIndex + 1] = 0;
}
```

Con esta refactorización las dos pruebas que tenemos pasan con éxito y terminamos otra de las tareas de nuestro diario de diseño.

4.1. Diario de diseño

- ~~Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).~~
- ~~Cada pozo tiene 4 semillas y las chozas están vacías.~~
- El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- ~~Cuando un jugador planta un pozo, ese pozo queda sin semillas.~~
- Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Triangulando pruebas y código

Vamos a implementar la funcionalidad de añadir las semillas sembradas en los pozos siguientes. No tiene sentido escribir una nueva prueba, ya que la condición de la prueba 4.25 nunca será cierta porque no estamos comprobando el incremento de los pozos. En este caso es mejor modificar dicha prueba para incorporar la nueva funcionalidad, como hemos hecho en el código 4.28.

Código 4.28. Añadimos que los 4 pozos siguientes tengan 1 semilla más.

```
@Test
	public void
testPrimerJugadorSiembraSuPrimerPozo_EsePozoQuedaSinSemillas_y_LosCuatroP
ozosSiguientesPasanA5() {
		String expected = "[0, 0, 5, 5, 5, 5, 4, " +
                                    "4, 4, 4, 4, 4, 4, 0]";

		m.seed(Player.FIRST_PLAYER, FIRST_PIT);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

La implementación en *babystep* es un *fake* (código 4.29). La prueba pasa correctamente.

Código 4.29.

```
public void seed(Player player, int pitIndex) {
		this.pits[pitIndex + 1] = 0;

		this.pits[pitIndex + 2]++;
		this.pits[pitIndex + 3]++;
		this.pits[pitIndex + 4]++;
		this.pits[pitIndex + 5]++;
	}
```

Las próximas pruebas van a triangular el método *seed*, por lo que vamos a refactorizarlo para que sea más sencillo de modificar. El primer cambio es reemplazar los cuatro incrementos por un bucle. A diferencia de la solución anterior, las cuatro líneas repetidas del código 4.29 me indican cuáles son los índices de inicio y fin correctos para el bucle (código 4.30).

Código 4.30. Añadimos un bucle para icnrementar los pozos.

```
public void seed(Player player, int pitIndex) {

		this.pits[pitIndex + 1] = 0;

		for (int i = pitIndex + 2; i <= (pitIndex + 4); i++) {
			this.pits[i]++;
		}
	}
```

Esta prueba solo funcionará cuando sembremos un pozo que contenga exactamente cuatro semillas, así que no podemos dar esta funcionalidad por concluida. Vamos a seguir triangulando con una nueva prueba, por ejemplo que intente sembrar un pozo sin semillas (código 4.31).

Código 4.31. Sembramos un pozo sin semillas.

```
@Test
	public void
testPrimerJugadorSiembraUnPozoConCeroSemillas_ElMancalaQuedaIgual() {
		String expected = "[0, 0, 5, 5, 5, 5, 4, " +
					"0, 4, 4, 4, 4, 4, 4]";

		m.seed(Player.FIRST_PLAYER, FIRST_PIT);
		m.seed(Player.FIRST_PLAYER, FIRST_PIT);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

La prueba falla, por lo que es el momento de modificar el método *seed* para que solo se siembren las semillas del pozo de origen indicado por *pitInde4.* El resultado se muestra en el código 4.32.

Código 4.32..

```
public void seed(Player player, int pitIndex) {
		int seeds = this.pits[pitIndex + 1];
		this.pits[pitIndex + 1] = 0;

		for (int i = pitIndex + 2; i <= (pitIndex + 4); i++) {
			this.pits[i]++;
		}
	}
```

Vamos a hacer dos refactorizaciones para terminar. En la primera, movemos el código que quita las semillas del pozo a un método auxiliar y, en la segunda, añadimos una nueva variable local que incremente el índice del pozo en 1. El código 4.33 muestra lo que llevamos hasta ahora después de hacer esas refactorizaciones.

Código 4.33. Método *seed* refactorizado

```
public void seed(Player player, int pitIndex) {
		int pit = pitIndex + 1;
		int seeds = removeSeedsFrom(pit);

		for (int i = pit +1; i <= (pit + seeds); i++) {
			this.pits[i]++;
		}
	}

	private int removeSeedsFrom(int pit) {
		int seeds = this.pits[pit];
		this.pits[pit] = 0;

		return seeds;
	}
```

Hasta aquí hemos conseguido que el primer jugador pueda plantar sus pozos, pero este código no funcionará con el segundo jugador.

4.1. Diario de diseño

- ~~Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).~~
- ~~Cada pozo tiene 4 semillas y las chozas están vacías.~~
- ~~El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.~~
- El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.
- ~~Cuando un jugador planta un pozo, ese pozo queda sin semillas.~~
- Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Implantando el segundo jugador

Para completar nuestro código y que soporte al segundo jugador vamos a escribir una prueba que siembre un pozo del segundo jugador (código 4.34).

Código 4.34. Primera prueba para el segundo jugador

```
@Test
	public void testSegundoJugadorSiembraSuPrimerPozoCon_4_Semillas()
{
		String expected = "[0, 4, 4, 4, 4, 4, 4, "
					+ "0, 0, 5, 5, 5, 5, 4]";

		m.seed(Player.SECOND_PLAYER, FIRST_PIT);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

La prueba anterior falla porque el método *seed* siembra el pozo del primer jugador. Para corregir esto, el método debe detectar si queremos sembrar en el pozo del segundo jugador y modificar el índice de la tabla *pits*. En este caso hacemos una implementación obvia que se muestra en el código 4.35.

Código 4.35. Un condicional cambia el índice si es el segundo jugador.

```
public void seed(Player player, int pitIndex) {
             int pit = pitIndex + 1;

             if (player == Player.SECOND_PLAYER) {
                    pit += 7;
             }

             int seeds = removeSeedsFrom(pit);

             for (int i = pit +1; i <= (pit + seeds); i++) {
                    this.pits[i]++;

             }
      }
```

Con el añadido del código 4.34 el segundo jugador ya puede sembrar sus pozos. Como refactorización, vamos a extraer la funcionalidad de calcular el índice correcto a un método auxiliar (código 4.36).

Código 4.36.

```
public void seed(Player player, int pitIndex) {
             int pit = calculatePit(player, pitIndex);
             int seeds = removeSeedsFrom(pit);

             for (int i = pit +1; i <= (pit + seeds); i++) {
                    this.pits[i]++;

             }
      }

      private int calculatePit(Player player, int pitIndex) {
             int pit = pitIndex + 1;

             if (player == Player.SECOND_PLAYER) {
                    pit += PITS;
             }
             return pit;
      }      }
```

Actualizamos el diario de diseño antes de continuar.

4.1. Diario de diseño

- ~~Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).~~
- ~~Cada pozo tiene 4 semillas y las chozas están vacías.~~
- ~~El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.~~
- ~~El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.~~
- ~~Cuando un jugador planta un pozo, ese pozo queda sin semillas.~~

- Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Del final al principio

Ya queda poco por implementar en nuestro diario de diseño. En esta sección vamos a implementar la funcionalidad necesaria para que siga plantando por el principio cuando llegue al final y aún quedan semillas. Esto pasa, por ejemplo, si desde el Mancala inicial, el segundo jugador planta su primer pozo (prueba del código 4.37).

Código 4.37. El segundo jugador planta su último pozo.

```
@Test
	public void
testSiSeLlegaAlUltimoPozoYAunQuedanSemillasSeContinuaPlantandoPorElPrime
rPozo() {
		String expected = "[1, 5, 5, 5, 4, 4, 4, "
					+ "0, 4, 4, 4, 4, 4, 0]";

		m.seed(Player.SECOND_PLAYER,  LAST_PIT);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

La prueba falla ya que el índice de la tabla *pits* se dale de los límites. Como ya vimos en la solución anterior una manera sencilla de calcular el índice correcto, vamos a reutilizarla y la añadirnos como una implementación obvia al método *seed* (código 4.38).

Código 4.38.

```
	public void seed(Player player, int pitIndex) {
		int pit = calculatePit(player, pitIndex);
		int seeds = removeSeedsFrom(pit);

		for (int i = pit +1; i <= (pit + seeds); i++) {
			this.pits[(i % 14)]++;

		}
	}
```

Con esta modificación todas las pruebas pasan correctamente y ya solo nos queda una última funcionalidad en nuestro diario de diseño.

Solo pozos válidos

La última funcionalidad que nos queda en el diario de diseño es verificar que el método *seed* recibe el índice un pozo válido. En este caso el índice es válido si está comprendido entre 0 y 5 ambos inclusive.

4.1. Diario de diseño

- ~~Inicialmente, el tablero tiene 6 pozos por cada jugador, 12 en total y dos chozas por cada jugador (para un total de 14 casillas).~~
- ~~Cada pozo tiene 4 semillas y las chozas están vacías.~~
- ~~El primer jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.~~
- ~~El segundo jugador puede plantar cualquiera de sus pozos que tenga al menos una semilla.~~
- ~~Cuando un jugador planta un pozo, ese pozo queda sin semillas.~~
- ~~Cuando se planta en el último pozo (choza del segundo jugador) y aún quedan semillas, se continúa plantando desde el principio (primer pozo del primer jugador).~~
- Si se intenta sembrar un pozo no válido, el Mancala no cambia.

Para implementar esta funcionalidad en nuestro código, primero escribiríamos una prueba con un índice menor que cero, la haríamos pasar con éxito y luego repetiríamos con una prueba con un índice mayor que 5. Como es una tarea sencilla, vamos a ver ambas pruebas juntas en el código 4.39.

Código 4.39. Pruebas para verificar que no pasa nada con un pozo incorrecto

Índice menor que cero.

```
@Test
	public void testSiSeleciconoUnPozoMenorQueCeroNoPasaNada() {
		String expected = "[0, 4, 4, 4, 4, 4, "
					+ "0, 4, 4, 4, 4, 4, 4]";

		m.seed(Player.FIRST_PLAYER,  -1);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

Índice mayor que cinco.

```
@Test
	public void testSiSeleciconoUnPozoMayorQueElMaximo_NoPasaNada() {
		String expected = "[0, 4, 4, 4, 4, 4, "
					+ "0, 4, 4, 4, 4, 4, 4]";

		m.seed(Player.SECOND_PLAYER,  LAST_PIT + 1);
		String mancalaState = m.toString();

		assertEquals(expected, mancalaState);
	}
```

El código (ya refactorizado) que hace pasar con éxito las pruebas anteriores se muestra en el código 4.40.

Código 4.40.

```
public void seed(Player player, int pitIndex) {
        if (!isValid(pitIndex))
                return;

        int pit = calculatePit(player, pitIndex);
        int seeds = removeSeedsFrom(pit);

        for (int i = pit +1; i <= (pit + seeds); i++) {
                this.pits[i]++;

        }
}

private boolean isValid(int pitIndex) {
        return (pitIndex >= 0) && (pitIndex <= 5);
}
```

Y con esto terminamos la segunda implementación de la cata Mancala. Vamos a resumir las conclusiones de esta segunda solución y, después, compararemos ambas implementaciones.

Retrospectica de la segunda solución

En esta segunda solución todo ha ido mejor que en la primera. Hemos aplicado TDD de manera más correcta y hemos tenido menos dificultades escribiendo pruebas e implementando el código. También hemos encontrado menos errores inesperados. Pero parte de este éxito también se debe a lo que aprendimos implementando la cata Mancala por primera vez. Gracias a la primera solución por ejemplo, hemos separado desde el principio el índice identificaba una celda del índice interno, o hemos utilizado directamente la fórmula para calcular los índices.

No debes desanimarte cuando un fragmento de código no salga a la primera. Un fallo es una oportunidad para mejorar tu experiencia con TDD y para descubrir características importantes del código que estás implementando y que podrá sutilizar en un segundo intento. Si queremos que todo salga bien a la primera y no estamos dispuestos a quitar el código que ha salido mal y volverlo a intentar nos estamos cerrando la puerta a mejorar.

El principal problema principal de esta segunda solución es la dependencia de las cadenas de texto. Aunque usar una cadena de texto nos aísla de los detalles de implementación, también introduce una fragilidad en las pruebas. Un pequeño cambio en la cadena (una coma, un espacio) invalida todas las pruebas. Este no es un motivo de peso para no usar esta estrategia, sino para mejorarla con técnicas y herramientas adicionales, por ejemplo TestText. Puede seguir practicando con esta estrategia implementado la cata *MineSweeper* que puedes encontrar en Internet.

Comparación entre ambas soluciones.

Vamos a comparar ambas soluciones (la primera solución utilizando una lista y la clase *Cell* y la segunda solución utilizando una tabla de enteros). Para ello, hemos calculado el conjunto de métricas de código que se muestra en la tabla 1. El primer número indica el valor medio y el segundo indica el valor máximo

Tabla 1. Métricas de las dos soluciones.

Métrica	Primera solución	Segunda solución
Average Block Depth	0.96 / 2	0.95 / 2
Cyclomatic Complexity	1.13 / 3	1.23 / 3
Lines Of Code Per Method	5.17 / 12	6.29 / 9
Num. of Methods Per Type	5.25 / 11	5.00 / 9
Number of Parameters	0.23 / 2	0.40 / 2
Efferent Couplings	2	2
Lines of Code	143	130
Number of Methods	21	15
Test cases	10 (1 de Cell)	8

Según las métricas de profundidad de bloque y la complejidad ciclomática, ambas soluciones presentan una complejidad similar, aunque la segunda solución tiene una complejidad ciclomática ligeramente mayor.

En cuanto al tamaño la segunda solución tiene menos líneas de código y menos métodos, a pesar de haber refactorizado más que la primera solución. Los métodos de la segunda tienen más código que la primera con una media de 6,29, pero dicho código está distribuido de manera más homogéneo ya que el método más largo de la segunda solución sólo tiene 9 líneas por 12 líneas de la primera solución

Respecto al acoplamiento, ambas soluciones presentan valores similares y bastante bajos.

Estos resultados reflejan que utilizar una clase para encapsular la lógica de gestionar cada uno de los pozos, como la clase *Cell* de la primera solución, no supone una mejora apreciable en la solución final.

Para reflexionar

Un detalle a tener en cuenta es que no hemos escrito ningún comentario en el código, ni en los casos de prueba ni en el código de producción. ¿Los has echado de menos? ¿Crees que hubieran sido necesarios? Si crees que, en el futuro, te costaría entender alguna arte del código de esta cata piensa cómo mejorar su legibilidad con buen refactorización, o con una prueba adicional, antes que con un comentario.

Uno de los principales errores a la hora de aplicar TDD y que hemos cometido en la primera solución es querer ir demasiado rápido escribiendo el código. A veces tenemos el código en la cabeza y queremos volcarlo rápidamente en el entorno de desarrollo. Pero nuestro cerebro no es un compilador y puede tener código que no

funcione y que sea difícil y costoso hacerlo funcionar. Al aplicar TDD de manera más estricta en la segunda solución hemos escrito menos código. TDD nos lleva a buscar la solución más sencilla posible, por lo que escribimos menos código.

Conclusiones

En este capítulo hemos visto dos soluciones distintas a una misma cata. Hemos tenido el coraje para descartar nuestro código y volver a empezar y, como resultado, hemos obtenido una solución mejor: más simple y más pequeña.

Aunque hemos completado el enunciado que planteamos al principio del capítulo, el juego del Mancala no está completo. A continuación tienes una lista de la funcionalidad que falta por implementar para completarlo.

- Si a la hora de sembrar se da una vuelta completa al Mancala, se omite el pozo de origen. Es decir, después de una siembra el pozo de origen está vacío, independientemente de las vueltas que hayamos dado al Mancala sembrando.
- El juego termina cuando el jugador al que le toca mover no puede por no tener semillas.
- Cuando el juego termina porque un jugador no puede mover, el otro jugador coge todas las semillas de sus pozos y las guarda en su choza.
- El ganador es el jugador con más semillas en su choza.

Agradecimientos del capítulo

A Antonio Martínez por inspirar la idea para este capítulo y por sus revisiones de los primeros borradores.

Capítulo 5.
Ejercicio TDD para presentar y usar mocks / doubles, etc.

Capítulo 6.
Cifrado Escítala y Refactoriación

La legibilidad del código es un aspecto clave en TDD ya que vamos añadiendo nuevo código de manera incremental. Para mejorar la legibilidad aplicamos refactorizaciones después de escribir el mínimo código que pasa con éxito una prueba.

Este capítulo tiene una orientación diferente a la de capítulos anterior. Ahora, no empezaremos desde cero sino que partiremos de una solución al problema que, aunque correcta, presenta un buen número de malos olores que reautorizaremos.

El Problema

El cifrado Escítala es una técnica utilizada en la antigua Grecia para comunicar mensajes secretos. Para implementarla, el emisor cogía una tira de cuero, la enrollaban alrededor de una vara y escribían el mensaje en ella a lo largo de la vara.

El receptor del mensaje debía tener una vara de grosor similar para enrollar nuevamente el mensaje y poder leerlo.

El Código

A continuación puedes ver una implementación de la codificación y decodificación utilizando la técnica de la Excítala.

Código 6.1

```
public class Escitala {
    private int caras;
    private String frase;
    private String[][] escitala;
    private int largo;

    public Escitala(int caras, String frase) {
        this.caras = caras;
        this.frase = frase;
    }

    public String encrypt() {
        if (isValid()) {//comprobamos la frase las caras etc
            largo = (frase.length() % caras) == 0 ? frase.length() / caras:
frase.length() / caras + 1; //Si el mensaje dividido entre las caras de la
escitala da resto cero no sobrarían espacios
            escitala = new String[caras][largo]; //las caras representan las
columnas de la escitala
            //el largo representa las filas

            int pivote = 0; //nos servirá de pivote para señalar el caracter
actual en la frase

            for (int columna = 0; columna < largo; columna++) { //recorremos todas
las filas DE CADA COLUMNA
                for (int fila = 0; fila < caras; fila++) { //recorremos todas las
columnas
                    if (pivote < frase.length()) {  //Si no hemos recorrido toda
la frase
                        escitala[fila][columna] =
String.valueOf(frase.charAt(pivote));  //Almacenamos el caracter que toque dentro
del Array
                        pivote++;                          //sumamos uno al pivote
para apuntar al siguiente caracter de la cadena de texto a encrypt
                    } else                              //Si hemos llegado al
final de la frase pero aún quedan datos que cumplimentar en el array
                        escitala[fila][columna] = aleatorio();  //como ya no
quedan letras de la frase generamos un caracter aleatorio para cumplimentar los
elementos del array
                }
            }
            return toText(caras, largo); //llamamos con la cantidad de filas
        }
        return null;
    }

    public String decrypt(String frase2) {
        if (isValid()) {  //comprobamos la frase las caras etc
            largo = (frase.length() % caras) == 0 ? frase.length() / caras:
frase.length() / caras + 1; //Si el mensaje dividido entre las caras de la
escitala da resto cero no sobrarían espacios
            escitala = new String[largo][caras];

            int pivote = 0;  //Nos servirá de pivote para la frase

            for (int columnas = 0; columnas < caras; columnas++) {  //OJO esta vez
la cantidad de columnas lo determinarán las caras
                for (int filas = 0; filas < largo; filas++) {  //OJO esta vez las
la cantidad de filas vendrá determinada por el largo.
                    escitala[filas][columnas] =
String.valueOf(frase2.charAt(pivote)); //vamos escribiendo caracter a caracter en
el array bidimensional
                    pivote++;       //avanzamos el pivote una posición para que
apunte al siguiente caracter del String a convertir.
                }
            }
            return toText(largo, caras); //llamamos con la cantidad de filas,
columnas. OJO hay que pasar como parámetro la variable que indique el máximo de
filas y luego la
```

```
        }                               //que indique el máximo de columnas.
        return null;
    }

    private String aleatorio() {
        int aleatorio = (int) (Math.random() * 100) + 1;  //calculamos un
aleatorio de 1 a 100. Podríamos haber pusto otro rango.

        return String.valueOf((char) aleatorio);  //Devolvemos como String el char
que representa a ese número generado aleatorimente. (Código ASCII)
    }

    private String toText(int f, int c) {
        StringBuilder sb = new StringBuilder();   //Ya que no trabajamos con hilos
creamos un StringBuilder que es más eficiente en estos casos

        for (int filas = 0; filas < f; filas ++) {             //Recorremos el
array
            for (int columnas = 0; columnas < c; columnas++) {
                sb.append(escitala[filas][columnas]);  //y vamos agregando al
StringBuilder caracter a carater
            }
        }
        return sb.toString();  //Devolvemos como String el contenido del
StringBuilder
    }

    private boolean isValid() {
        boolean correcto = false;

        if (caras > 0 && frase != null && frase.length() > caras) {
            correcto = true;
        } else
            System.out.println("Las cara deben ser un número entero positivo
comprendido y la frase no puede ser nula. \n" +
                    ".Además la cantidad de letras de la frase no puede ser menor
al número de caras");
        return correcto;
    }

    public int getCaras() {
        return caras;
    }

    //MÉTODOS GETTERS AND SETTERS TÍPICOS.
    public void setCaras(int caras) {
        if (caras > 0)
            this.caras = caras;
    }

    public String getFrase() {
        return frase;
    }

    public void setFrase(String frase) {
        this.frase = frase;
    }
}
```

Vamos a trabajar con el código anterior a lo largo de este capítulo.

¿Podemos leerlo?

Es difícil trabajar con un código difícil de leer y el código anterior lo es. Las líneas son muy largas y, además los comentarios se incluyen en la misma línea. Hay un número excesivo de comentarios que no facilitan, sino dificultan entender el código.

El primer paso que daremos será eliminar comentarios superfluos (como "recorremos todas las columnas") y reorganizar el formato código para que sea fácil de leer. Haremos eso con cuidado, ya que aún no tenemos pruebas que nos adviertan si hemos introducido algún error.

Código 6.2

```
package escitala;

public class Escitala {
	private int caras;
	private String frase;
	private String[][] escitala;
	private int largo;

	public Escitala(int caras, String frase) {
		this.caras = caras;
		this.frase = frase;
	}

	public String encrypt() {
		if (isValid()) {
			//Si el mensaje dividido entre las caras de
			// la escitala
			// da resto cero no sobrarían espacios
			largo = (frase.length() % caras) == 0 ?
					frase.length() / caras:
						frase.length() / caras + 1;
			//las caras representan las columnas de
			// la escitala
			//el largo representa las filas
			escitala = new String[caras][largo];

			// pivote para señalar el caracter actual
			// en la frase
			int pivote = 0;

			for (int columna = 0; columna < largo; columna++) {
				for (int fila = 0; fila < caras; fila++) {
					if (pivote < frase.length()) {
						escitala[fila][columna] =

	String.valueOf(frase.charAt(pivote));
						pivote++;
					} else
						//como ya no quedan letras de
						//la frase generamos un caracter
						// aleatorio para cumplimentar
						//los elementos del array
						escitala[fila][columna] =
							aleatorio();
				}
			}
			return toText(caras, largo);
		}
		return null;
	}
```

```
	public String decrypt(String frase2) {
		if (isValid()) {
			largo = (frase.length() % caras) == 0 ?
					frase.length() / caras:
						frase.length() / caras + 1;
			escitala = new String[largo][caras];

			int pivote = 0;

			//la cantidad de columnas lo determinarán las caras
			for (int columnas = 0; columnas < caras; columnas++)
{
				for (int filas = 0; filas < largo; filas++) {
					escitala[filas][columnas] =

	String.valueOf(frase2.charAt(pivote));
					pivote++;
				}
			}
			return toText(largo, caras);
		}
		return null;
	}

	private String aleatorio() {
		int aleatorio = (int) (Math.random() * 100) + 1;

		return String.valueOf((char) aleatorio);
	}

	private String toText(int f, int c) {
		StringBuilder sb = new StringBuilder();

		for (int filas = 0; filas < f; filas ++) {
			for (int columnas = 0; columnas < c; columnas++) {
				sb.append(escitala[filas][columnas]);
			}
		}
		return sb.toString();
	}

	private boolean isValid() {
		boolean correcto = false;

		if (caras > 0 && frase != null && frase.length() > caras) {
			correcto = true;
		} else
			System.out.println("Las cara deben ser un número " +
				"entero positivo " +
				"comprendido y la frase no puede ser nula. \n"
+
				".Además la cantidad de letras de la frase no
" +
				"puede ser " +
				"menor al número de caras");
		return correcto;
```

```
	}

	public int getCaras() {
		return caras;
	}

	public void setCaras(int caras) {
		if (caras > 0)
			this.caras = caras;
	}

	public String getFrase() {
		return frase;
	}

	public void setFrase(String frase) {
		this.frase = frase;
	}
}
```

En el código 6.2 hay más líneas de código de las que teníamos en el 6.1. A cambio, hemos mejorado su legibilidad. Ahora podemos hacer una lista de malos olores. Por ejemplo mensajes de texto incrustados en el texto o bloques de código sin llaves o Código duplicado. También podemos ver, leyendo los comentarios, que la mayoría de los nombres parecen inadecuados, ya que hace falta explicar qué hace cada variable, cuando eso debería quedar claro solo por su nombre. Volveremos sobre esto más adelante, porque, sin pruebas, es mejor no modificar el código.

Vamos a crear nuestro diario de diseño para identificar las mejores que vamos a aplicar al código.

Cuadro 6.3 Diario de diseño
• Escribir pruebas. • Evitar el código duplicado en los métodos encript y decript. • Quitar el mensaje del constructor y añadirlo al método encript • Quitar los métodos get / set no utilizados • Nombres más descriptivos para variables y métodos • Quitar mensajes. • Todos los bloques de código entre llaves.

En los próximos apartados veremos cómo corregir estos malos olores, peor antes vamos a escribir un conjunto de pruebas.

Escribiendo Pruebas

¿Por qué tenemos que escribir pruebas si damos por hecho que el código funciona? Las pruebas serán nuestra garantía de que no introducimos errores a la hora de cambiar el código. Asumimos que el código funciona, por lo que comenzamos escribiendo pruebas de aceptación. Una de ellas (utilizando la sintaxis Gerkin) se muestra en el código 6.4

Código 6.4. Escenario.

```
given Un objeto de clase Escitala
  and 10 caras
  and La frase "En un lugar de la Mancha, de cuyo nombre no quiero acordarme"
 when El objeto cifra el texto
 then Obtengo el texto "Ernu n cyna dhoocuea on ,nqr l oudladmiau ebergM rrmaaceoe".
```

La implementación de esta prueba se muestra en el código 6.5.

Código 6.5

```
@Test
public void testEncript() {
    String msg = "En un lugar de la Mancha, de cuyo nombre no quiero acordarme";
    String expt = "Ernu  n cyna dhoocuea  on ,nqr l oudladmiau ebergM rrmaaceoe";
    int caras = 10;
    Escitala escitala = new Escitala(caras, msg);

    String result = escitala.encrypt();

    assertEquals(expt, result);
}
```

Si esta prueba diera error, tendríamos que escribir pruebas para los métodos auxiliares buscando acotar el fragmento de código dónde está el error. En este caso, la prueba 6.5 funciona correctamente, por lo que pasamos a la siguiente prueba.

Vamos a escribir una prueba para el método de desencriptar. Esta prueba es la misma que la anterior cambiando los mensajes. La implementación está en el código 6.6.

Código 6.6

```
@Test
public void testDecript() {
    String msg = "Ernu  n cyna dhoocuea  on ,nqr l oudladmiau ebergM rrmaacec
    String expt = "En un lugar de la Mancha, de cuyo nombre no quiero acordar
    Escitala escitala = new Escitala(10, msg);

    String result = escitala.decrypt(msg);

    assertEquals(expt, result);

}
```

Esta prueba también funciona, por lo que es el momento de empezar a hacer cambios.

Cuadro 6.7. Diario de diseño

- ~~Escribir pruebas.~~
- Evitar el código duplicado en los métodos encript y decript.
- Quitar el mensaje del constructor y añadirlo al método encript.

- Sacar los métodos encript y decript del bloque if.
- Quitar los métodos get / set no utilizados
- Nombres más descriptivos para variables y métodos
- Quitar mensajes de texto.
- Todos los bloques de código entre llaves.
- Excepciones en vez de resultados nulos.

Vamos a actualizar el diario de diseño con los cambios que iremos haciendo a continuación (cuadro 6.7).

Desbloqueando a los métodos principales.

Actualmente el código de los métodos está escondido en un bloque if (código 6.8)

Código 6.8.

```
public String decrypt(String frase2) {
    if (isValid()) {
        largo = (frase.length() % caras) == 0 ?
                frase.length() / caras:
                    frase.length() / caras + 1;
        escitala = new String[largo][caras];

        int pivote = 0;

        //la cantidad de columnas lo determinarán las caras
        for (int columnas = 0; columnas < caras; columnas++) {
            for (int filas = 0; filas < largo; filas++) {
                escitala[filas][columnas] =
                    String.valueOf(frase2.charAt(pivote));
                pivote++;
            }
        }
        return toText(largo, caras);
    }
    return null;
}
```

Tener todo el cuerpo de un método dentro de if lo hace más difícil de leer y entender, por lo que vamos a cambiar la condición del if para sacar el código fuera del bloque. El método *encript* quedaría como se muestra en el código 6.9 y le método *decript* quedaría como se muestra en el código 6.10.

Código 6.9.

```
public String encrypt() {
    if (!isValid()) {
        return null;
    }
    largo = (frase.length() % caras) == 0 ?
            frase.length() / caras:
                frase.length() / caras + 1;
    escitala = new String[caras][largo];
    int pivote = 0;

    for (int columna = 0; columna < largo; columna++) {
        for (int fila = 0; fila < caras; fila++) {
            if (pivote < frase.length()) {
                escitala[fila][columna] =
                    String.valueOf(frase.charAt(pivote));
                pivote++;
            } else
                //como ya no quedan letras de
                //la frase generamos un caracter
                // aleatorio para cumplimentar
                //los elementos del array
                escitala[fila][columna] =
                    aleatorio();
        }
    }
    return toText(caras, largo);
}
```

Ejecutamos las pruebas para comprobar que no hemos introducido ningún error y ya podemos tachar esta tarea en nuestro diario de diseño y continuar con la siguiente tarea.

Cuadro 6.10. Diario de diseño

- ~~Escribir pruebas.~~
- Evitar el código duplicado en los métodos encript y decript.
- Quitar el mensaje del constructor y añadirlo al método encript.
- ~~Sacar los métodos encript y decript del bloque if.~~
- Quitar los métodos get / set no utilizados
- Nombres más descriptivos para variables y métodos
- Quitar mensajes de texto.
- Todos los bloques de código entre llaves.
- Excepciones en vez de resultados nulos.

Cambiando mensajes por excepciones

El mensaje de texto del código actual nos indica que el número de caras no es válido. Sin embargo este código es un componente de otro código por lo que no debe mostrar mensajes directamente. La responsable de enviar mensajes será la interfaz de usuario.

Vamos a cambiar el mensaje por una excepción. El primer paso será escribir una prueba que indique el objetivo que buscamos (código 6.11).

Código 6.11. Prueba.

```
@Test(expected=WrongFacesValueException.class)
public void testEncript_FaceNumberIsBiggerThanMsg() {
    escitala = new Escitala(10, "");
    escitala.encrypt();
}
```

La prueba no se puede ejecutar porque nos falta la clase excepción. Esta clase podemos crearla de manera automática con los asistentes de los IDEs más habituales. Un ejemplo, generado con Eclipse se muestra en el código 6.12.

Código 6.12. Clase excepción generada con Eclipse.

```
public class WrongFacesValueException extends RuntimeException {

    public WrongFacesValueException() {
        // TODO Auto-generated constructor stub
    }

    public WrongFacesValueException(String arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public WrongFacesValueException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public WrongFacesValueException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public WrongFacesValueException(String arg0, Throwable arg1, boolean arg2,
            boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

}
```

Ahora la prueba falla porque, cuando detecta el error, no se lanza la excepción. Vamos a cambiar eso en el código. Vamos a hacer que la excepción la lance el método

público para no dar detalles de métodos privados. La modificación se muestra en el código 6.13.

Código 6.13. Lanza la excepción en lugar de devolver nulo

```
public String encrypt(String mensaje) {
    this.frase = mensaje;
    if (!isValid()) {
        throw new WrongFacesValueException();
    }

    // El resto queda igual
}
```

Ahora, vamos a implementar un mensaje descriptivo en la excepción. De tal manera que, cuando se lance, el mensaje nos indique qué falla y cómo podemos corregirlo. Para ello, empezamos escribiendo una prueba (código 6.14) que nos permita definir la información del mensaje.

Código 6.14. Lanza la excepción en lugar de devolver nulo

```
@Test
public void testMessageContaintsNumberOfFaces() {
    try {
        throw new WrongFacesValueException(6);
    } catch(WrongFacesValueException e) {
        String msg = e.getMessage();
        assertTrue(msg.contains("Caras actuales: 6") );
    }
}
```

La prueba anterior, aunque breve, es difícil de entender. Al final del capítulo la refactorizaremos. El primer paso para implementar esta prueba es crear un constructor que admita un entero. Esa modificación se muestra en el código 6.15.

Código 6.15. Constructor con el número de caras

```
public class WrongFacesValueException extends RuntimeException {

    public WrongFacesValueException(String arg0) {
        super(arg0);
    }


    public WrongFacesValueException(int i) {
        this(String.valueOf(i));
    }

}
```

Hemos quitado el resto de constructores del código 6.12 para que siempre que se cree una excepción se indique el número de caras. Vamos a incluir ahora el mensaje y, para ello, aprovechamos el mensaje del código original. El resultado está en el código 6.16.

Código 6.16.

```
public class WrongFacesValueException extends RuntimeException {

    public WrongFacesValueException(String arg0) {
        super("Caras actuales: " + arg0 +
                ". Las cara deben ser un número " +
                "entero positivo " +
                "comprendido y la frase no puede ser nula. \n" +
                ".Además la cantidad de letras de la frase no " +
                "puede ser " +
                "menor al número de caras");
    }

    public WrongFacesValueException(int i) {
        this(String.valueOf(i));
    }

}
```

Con el código anterior, la prueba del código 6.14 pasa con éxito, pero el código de la clase Escítala da un error, ya que hemos quitado constructores. Para ello hacemos añadimos el número de caras en el momento de lanzar la excepción (código 6.17) y comprobamos que todo vuelve a funcionar.

Código 6.17.

```
if (!isValid()) {
    throw new WrongFacesValueException(this.caras);
}
```

Para terminar vamos a refactorizar el método *isValid* para que ya no muestre el mensaje de error. Además, cambiaremos el nombre de la función para que sea más claro qué se comprueba y cuando es válido. El resultado se muestra en el código 6.18.

Código 6.18.

```
private boolean checkNumberOfFacesAndPhraseIsNotNull() {
    return (caras > 0 && frase != null && frase.length() > caras) ;
}
```

Por último ejecutamos todas las pruebas para verificar que el código sigue funcionando.

Aún quedan más cosas que contar en este capítulo.

Conclusiones

Es erróneo pensar que cuanto más comentarios pongamos a nuestro código más fácil será de entender. Ahora tú mismo puedes comprobarlo comparando el código del principio de este capítulo con el resultado final. ¿Qué código entiendes mejor?

También puedes comprobar lo útiles que nos han resultado las pruebas que escribimos al principio ya que, junto a los ciclos TDD hechos para cada cambio, nos han dado la seguridad de que no estábamos introduciendo nuevos errores. Después de cada cambio, el código ha seguido encriptando y desencriptando correctamente.

Sólo hemos escrito pruebas para los métodos públicos y no para los métodos privados. Vamos a ver una breve justificación. Los métodos privados con internos de la clase por lo que no deberían ser probados, ni tampoco contener una lógica demasiado compleja. Además Java no ayuda a la hora de probar métodos privados ya que hay que dar rodeos (introspección, herencia y delegación, etc.) para poder acceder a ellos.

En el ejemplo que hemos probado los métodos privados a través de los métodos públicos. Un fallo en un método privado desencadena que no se encripte o desencripte correctamente.

Agradecimientos del capítulo

Al usuario Sie7e de Solveet (www.solveet.com) por desarrollar la implementación original que hemos utilizado de base en este capítulo.

Anexo 1.

Test de pruebas

Este test te ayuda a aprender conceptos básicos de la prueba del software y es una continuación del reto de testing publicado en BetaBeers. Si después de hacer este test quieres aprender más, pasa por este enlace y resuelve el reto de testing: http://betabeers.com/test/testing-16/

Preguntas

1. ¿Por cuáles de los siguientes motivos elegirías utilizar algún tipo de double?
    a) Porque estamos haciendo pruebas unitarias
    b) Porque tenemos una dependencia de una clase que aún no existe.
    c) Porque tenemos una dependencia de una clase u es muy lenta.
    d) Todas las anteriores.

2. ¿Qué pruebas deberán ejecutarse en primer lugar?

    a) Las pruebas que busquen los errores más frecuentes
    b) Las pruebas que busquen errores con un mayor impacto económico
    c) Las pruebas indicadas por los usuarios finales
    d) Las prueba sindicadas en el plan de pruebas

3. ¿Cuántas particiones son necesarias para probar un método mediante la técnica de particiones equivalentes?

a) Tantas particiones como parámetros de entrada tenga un método.
b) Tantas particiones como parámetros de entrada y atributos de la clase que le afecten
c) Tantas particiones como posibles valores de entrada
d) Tantas particiones como posibles salidas tenga el método.

4. Actualmente, una de las aplicaciones más habituales de la computación en la nube aplicada a la prueba del software es:

a) Ejecutar las pruebas unitarias en la nube para evitar pausas al escribir código.
b) Contratar recursos para simular múltiples usuarios sobre el sistema bajo prueba.
c) Realizar pruebas A/B
d) Almacenar toda la información sobre la ejecución de pruebas

5. ¿Cuándo debemos probar interacciones en lugar de estados en una prueba unitaria?

a) Siempre que podamos
b) Nunca
c) Cuando el código bajo prueba no genere ningún resultado verificable
d) Cuando el método a probar no devuelva nada

6. El contenido más habitual de un plan de pruebas es.

a) Elementos bajo prueba, criterios de aceptación, entornos de pruebas, responsables y planificación
b) Elementos bajo prueba, criterios de aceptación, entornos de pruebas, responsables, planificación, resultados de las pruebas
c) Iteración, historias de usuario a probar, definición de hecho, entornos de pruebas, responsables, planificación
d) Elementos bajo prueba, criterios de aceptación y entornos de pruebas

7. Un conjunto de pruebas unitarias consume mucho tiempo en su ejecución. ¿Cuál de las siguientes acciones elegiría para aumentar la velocidad?

a) Ejecutar dichas pruebas en un servidor dedicado, por ejemplo un servidor de integración continua
b) Intentar sustituir algún módulo por un doublé / mock /stub
c) Comprar un ordenador más potente
d) Quitar las pruebas que menos importancia tengan

8. Un compañero de trabajo te hace llegar un fragmento de código fuente indicándote que falla. ¿Qué harías?

a) Preguntar cuál es el resultado esperado después de la ejecución de dicho código
b) Ejecutar el código paso a paso con un depurador
c) Escribir varias pruebas unitarias para localizar el error
d) Revisar cuidadosamente el código en busca del error

9. ¿Es recomendable introducir estructuras de control como if o for en las pruebas unitarias?

a) Sí porque así se pueden escribir pruebas más potentes y versátiles que ahorren tiempo
b) No porque dichas estructuras hacen la prueba más difícil de entender.
c) Sí porque con ellas se pueden verificar mejor los resultados de una prueba, con lo que la eficiencia de la prueba aumenta.
d) No, porque una prueba debe ser simple ya que no escribimos pruebas para probar pruebas

10. ¿Cuáles serían los valores límite para probar el siguiente fragmento de código?

```
method(int a, int b) {
        int c = 0;

        if ( a > 5) {
                c++;
        }
        if ( b > 6> {
                c++:
        }
}
```

a) Para a sería -1, 0 y 1 y los mismos para b

b) Para a sería MAX_INTEGER, - MAX_INTEGER y 0 y los mismos para b

c) Para a sería 5 y 6 y para b 6 y 7

d) Para a sería 4, 5 y 6 y para b 5, 6 y 7

Respuestas

Pregunta	Respuesta	Pregunta	Respuesta
1	D	6	A
2	D	7	B
3	D	8	A
4	B	9	D
5	C	10	c

Agradecimientos del capítulo

A Miquel Camps por su estupenda iniciativa BetaBeers y por incluir el reto de testing que le propusimos.

A Antonio Martínez por sus estupendos y valiosos comentarios sobre el reto de testing, los cuáles nos han animado a incluís este anexo.

Desarrollo Dirigido por Pruebas Práctico

Gracias.