Test Driven Development. Introducción y ejercicio

Trabajo sesión seminario 07 de marzo ACG

Curso 2022-2023

El Desarrollo dirigido por pruebas, del inglés Test Driven Development (TDD) es un enfoque para el desarrollo de software en el que se escriben las pruebas antes de escribir el código. Y cuando todas las pruebas pasan, se mejora el código.

La premisa detrás del desarrollo dirigido por pruebas, según Kent Beck, es que todo el código debe ser probado y refactorizado continuamente. Aunque esto es básicamente testing. El punto clave del TDD es que al escribir pruebas primero y esforzarse por mantenerlas fáciles de escribir, estás haciendo tres cosas importantes:

- Estás creando documentación, especificaciones vivas y nunca obsoletas (es decir, documentación).
- Estás (re)diseñando el código para hacerlo y mantenerlo fácilmente comprobable. Y eso lo hace limpio, sin complicaciones y fácil de entender y cambiar.
- Estás creando una red de seguridad para hacer cambios con confianza.

Además, el TDD también consigue:

- Notificación temprana de errores.
- Diagnóstico sencillo de los errores, ya que las pruebas identifican lo que ha fallado.

Existen 3 reglas de Robert C. Martin que son bastante buenas para entender el TDD. Son las siguientes:

- No está permitido escribir código que va a producción (código de negocio) sin tener antes una prueba unitaria que falle.
- No está permitido escribir más código en la prueba unitaria del necesario para que ocurra el fallo.
- No está permitido escribir más código (del que va a producción) del necesario para superar la prueba unitaria.

Si se siguen estas reglas, harás primero, antes de nada, pruebas unitarias pequeñas y escribirás el código justo para superarlas.

A continuación, en la sección 1 se muestra un extracto del primer capítulo de la parte I del libro *Test Driven Development: by Example* de Kent Beck. En la segunda sección, se plantea un ejercicio para la aplicación de TDD.

Sección 1. Extracto del libro Test Driven Development: by Example.

TDD se puede resumir de la siguiente manera:

1. Agregue rápidamente una prueba.
2. Ejecute todas las pruebas y vea que la nueva falla.

3. Haz un pequeño cambio.
4. Ejecute todas las pruebas y vea que todas tengan éxito.
5. Refactorizar para eliminar la duplicación.

Partimos de un software que muestra un informe como el siguiente:

Instrument	Shares	Price	Total
IBM	1000	25	25000
GE	400	100	40000
		Total	65000

Este informe debe cambiarse por la siguiente salida:

Instrument	Shares	Price	Total
IBM	1000	25 USD	25000 USD
Novartis	400	150 CHF	60000 CHF
		Total	65000 USD

¿Qué comportamiento necesitaremos para obtener el nuevo informe? Dicho de otra manera, ¿qué conjunto de pruebas, cuando se superen, demostrarán la presencia de código que estamos seguros calculará correctamente el informe?

- Necesitamos poder sumar cantidades en dos monedas diferentes y convertir el resultado dado en un conjunto de tipos de cambio.
- Necesitamos poder multiplicar una cantidad (precio por acción) por un número (número de acciones) y obtener una cantidad.

Vamos a hacer una lista de tareas pendientes para recordar lo que debemos hacer, para mantenernos enfocados y para decirnos cuándo hemos terminado. Cuando empecemos a trabajar en un elemento, lo pondremos en negrita, **así**. Cuando terminemos un elemento, lo tacharemos, ~~así~~. Cuando pensemos en otra prueba para escribir, la agregaremos a la lista. La lista es la siguiente:

- $5 + 10 CHF = $10 si el cambio es 2:1
- **$5 * 2 = $10**

Primero trabajaremos en la multiplicación. ¿Qué objeto necesitamos primero? Pregunta capciosa. No comenzamos con objetos, comenzamos con pruebas. Lo intentamos otra vez. ¿Qué prueba necesitamos primero? Mirando la lista, esa primera prueba parece complicada. Comience poco a poco. Multiplicación, ¿cuánto de difícil puede ser eso? Trabajaremos en esto primero.

Cuando escribimos una prueba, imaginamos la interfaz perfecta para nuestra operación. Nos estamos contando una historia sobre cómo se verá la operación desde afuera. Nuestra historia no siempre se hará realidad, pero es mejor comenzar con la mejor interfaz de programa de aplicación (API) posible y trabajar hacia atrás que hacer las cosas complicadas, feas y "realistas" desde el principio.

Aquí hay un ejemplo simple de multiplicación:

```
public void testMultiplication() {
   Dollar five= new Dollar(5);
   five.times(2);
   assertEquals(10, five.amount);
}
```

Lo sé, lo sé, campos públicos, números enteros para cantidades monetarias y todo eso. Pequeños pasos. Tomaremos nota del *bad smell* y seguiremos adelante. Tenemos una prueba y queremos pasarla (ir a verde) lo más rápido posible.

- $5 + 10 CHF = $10 si el cambio es 2:1
- **$5 * 2 = $10**
- Hacer que "amount" sea privada
- ¿Redondeo de dinero?

La prueba que acabamos de escribir ni siquiera compila. Esto es bastante fácil de arreglar. ¿Qué es lo mínimo que podemos hacer para compilarlo, incluso si no se ejecuta? Tenemos cuatro errores de compilación:

- No existe la clase `Dollar`
- No existe el constructor
- No existe el método `times(int)`
- No existe el campo `amount`

Vamos a considerar estos errores de uno en uno. Podemos solucionar un error definiendo la clase Dollar:

```
class Dollar
```

Un error solucionado, nos quedan tres errores por delante. Ahora necesitamos el constructor, pero no tiene que hacer nada solo sirve para compilar la prueba:

```
Dollar(int amount) {
}
```

Dos errores por solucionar. Necesitamos una implementación *stub* de times(). Otra vez, haremos únicamente lo necesario para compilar la prueba:

```
void times(int multiplier) {
}
```

Nos queda un error. Finalmente, necesitamos un campo `amount`:

```
int amount;
```

¡Compila! Ahora podemos ejecutar la prueba y ver cómo falla (Figura 1).

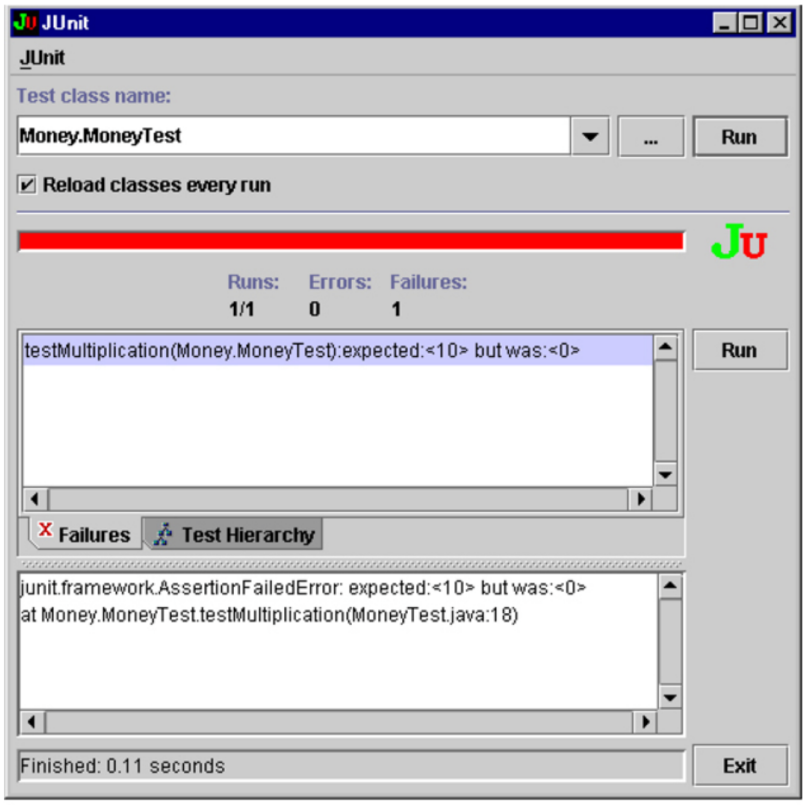

Figura 1. Resultado del test testMultiplication. Barra roja

Estás viendo la temida barra roja. Nuestro marco de prueba (JUnit, en este caso) ejecutó el pequeño fragmento de código con el que comenzamos y comprobó que, aunque esperábamos "10" como resultado, se obtuvo "0". Tristeza.

No, no. El fracaso es progreso. Ahora tenemos una medida concreta de fracaso. Eso es mejor que saber vagamente que estamos fallando. Nuestro problema de programación se ha transformado de "dame varias monedas" a "hacer que esta prueba funcione y luego hacer que el resto de las pruebas funcionen". Mucho más simple. Mucho menor alcance para el miedo. Podemos hacer que esta prueba funcione.

Probablemente no te va a gustar la solución, pero el objetivo en este momento no es obtener la respuesta perfecta sino pasar la prueba.

Aquí está el cambio más pequeño que podría imaginar que haría que nuestra prueba pasara:

```
int amount= 10;
```

Nuestro marco de prueba (JUnit, en este caso) ha ejecutado el pequeño fragmento de código con el que comenzamos, y la Figura 2 muestra el resultado cuando la prueba se ejecuta nuevamente. Ahora tenemos la barra verde.

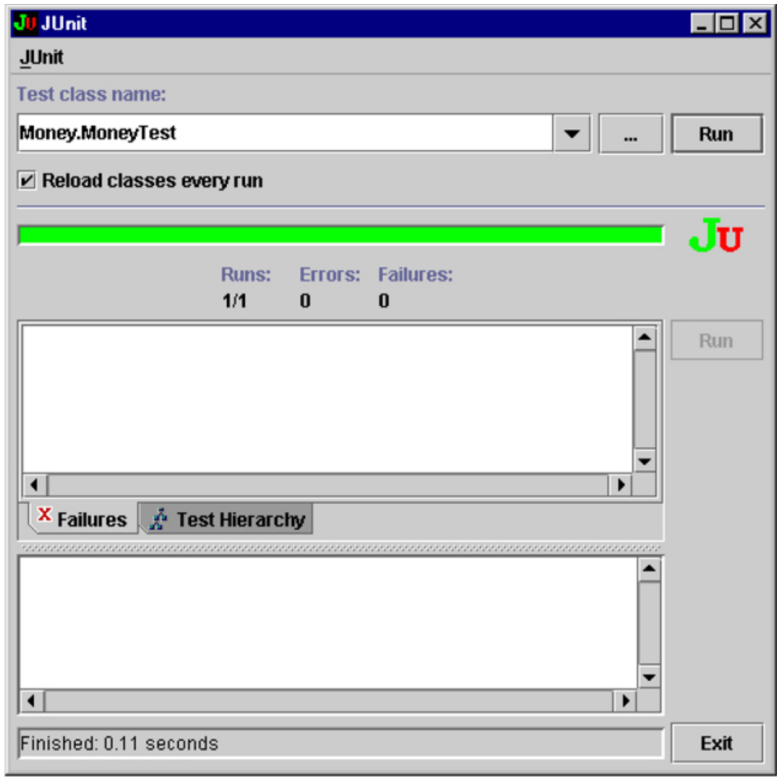

Figura 2. Resultado del test TestMultiplication. Barra verde

¡Oh alegría, oh éxtasis! No tan rápido, chico (o chica) hacker. El ciclo no está completo. Hay muy pocas entradas en el mundo que harán que se apruebe una implementación tan limitada, tan maloliente, tan ingenua. Necesitamos generalizar antes de continuar. Recuerda, el ciclo es el siguiente:

- Añade una pequeña prueba.
- Ejecute todas las pruebas y falle.
- Haz un pequeño cambio.
- Ejecute las pruebas y tenga éxito.
- Refactorizar para eliminar la duplicación.

Dependencia y duplicación

Steve Freeman señaló que el problema con la prueba y el código no es la duplicación. El problema es la dependencia entre el código y la prueba: no puede cambiar uno sin cambiar el otro. Nuestro objetivo es poder escribir otra prueba que "tenga sentido" para nosotros, sin tener que cambiar el código, algo que no es posible con la implementación actual.

La dependencia es el problema clave en el desarrollo de software en todas las escalas. Si tiene detalles de la implementación de SQL de un proveedor dispersos en el código y decide cambiar a otro proveedor, descubrirá que su código depende del proveedor de la base de datos. No puede cambiar la base de datos sin cambiar el código.

Si la dependencia es el problema, la duplicación es el síntoma. La duplicación suele adoptar la forma de lógica duplicada: la misma expresión que aparece en varios lugares del código. Los objetos son excelentes para abstraer la duplicación de la lógica.

A diferencia de la mayoría de los problemas de la vida, en los que la eliminación de los síntomas solo hace que el problema aparezca en otro lugar de peor forma, la eliminación de la duplicación en los programas elimina la dependencia. Por eso aparece la segunda regla en TDD. Al eliminar la duplicación antes de pasar a la siguiente prueba, maximizamos nuestras posibilidades de poder ejecutar la próxima prueba con un solo cambio.

Hemos ejecutado los pasos 1 a 4 del ciclo. Ahora estamos listos para eliminar la duplicación. Pero, ¿dónde está la duplicación? Por lo general, verá una duplicación entre dos piezas de código, pero aquí la duplicación es entre los datos de la prueba y los datos del código. ¿No lo ves? Que tal si escribimos lo siguiente:

```
int amount= 5 * 2;
```

Ese 10 tenía que venir de alguna parte. Hicimos la multiplicación en nuestras cabezas tan rápido que ni nos dimos cuenta. El 5 y el 2 ahora están en dos lugares, y debemos eliminar despiadadamente la duplicación antes de continuar. Las reglas lo dicen.

No hay un solo paso que elimine el 5 y el 2. Pero, ¿qué pasa si movemos la configuración de la cantidad de la inicialización del objeto al método `times()`?

```
int amount;
void times(int multiplier) {
   amount= 5 * 2;
}
```

La prueba aún pasa, la barra permanece verde. La felicidad sigue siendo nuestra.

¿Te parecen pequeños estos pasos? Recuerde, TDD no se trata de dar pequeños pasos, se trata de ser capaz de dar pequeños pasos. ¿Codificaría día a día con pasos tan pequeños? No. Pero cuando las cosas se ponen un poco raras, me alegro de poder hacerlo. Pruebe con pasos diminutos con un ejemplo de su propia elección. Si puede hacer pasos demasiado pequeños, ciertamente puede hacer pasos del tamaño correcto. Si solo das pasos más grandes, nunca sabrás si los pasos más pequeños son apropiados.

Dejando a un lado la actitud defensiva, ¿dónde estábamos? Ah, sí, nos estábamos deshaciendo de la duplicación entre el código de prueba y el código de trabajo. ¿Dónde podemos conseguir un 5? Ese fue el valor pasado al constructor, así que si lo guardamos en la variable cantidad,

```
Dollar(int amount) {
   this.amount= amount;
}
```

ahora podemos usarlo en `times()`:

```
void times(int multiplier) {
```

```
   amount= amount * 2;
}
```

El valor del parámetro “multiplier” es 2, por lo que podemos sustituir el valor por el parámetro:

```
void times(int multiplier) {
   amount= amount * multiplier;
}
```

Para demostrar nuestro conocimiento de la sintaxis de java, queremos utilizar el operador *= (el cual reduce duplicación):

```
void times(int multiplier) {
   amount *= multiplier;
}
```

Actualizamos la lista de tareas:

- $5 + 10 CHF = $10 if rate is 2:1
- ~~$5 * 2 = $10~~
- Make “amount” private
- Dollar side effects?
- Money rounding?

Sección 1. Ejercicio.

El objetivo es implementar una función suma que tome dos números como parámetros y devuelva la suma de ambos números. Plantea una estrategia TDD para la implementación de esta función.