Práctica: Pruebas IDM

Bisiestos

1. Identificar el Software Under Test

```
public class Bisiestos {

    // Devuelve true si año es bisiesto, false en caso contrario
    // Eleva la excepción InvalidParameter si año no es un parámetro válido

    public boolean esBisiesto(int año) throws InvalidParameter {
        //
    }
}
```

2. Identificar los parámetros del SUT

Tenemos una único parámetro: int año, que será el año que vayamos a comprobar.

3. Utilizando la técnica de la modelización de las entradas, crear caracterizaciones basadas en la funcionalidad, y sus particiones en bloques.

Caracterización q1: “Consideraciones para que un año sea bisiesto”.

C1==> años positivos divisibles entre 4 y no entre 100. True/False ==> True.
C2==> años positivos divisibles entre 4, entre 100 y entre 400. True/False ==>True.
C3==> resto de años (negativos,0,positivos no divisibles entre 4, o positivos divisibles entre 4 y entre 100 pero no entre 400). True/False==>False.

4. Elegir valores adecuados de cada bloque que satisfagan alguno de los criterios de cobertura “razonables” vistos en clase, prestando atención a los casos frontera que pueda haber en cada caso a la hora de elegir valores concretos para los tests.

Valores:

C1==>Podríamos tener algunos entre 2016,2020,2024 entre otros.
C2==>Podríamos tener algunos entre 1600,2000,2400 entre otros.
C3==>Podríamos tener por ejemplo -1,1997,0 entre otros.

5. Escribir el código de pruebas con JUnit, antes de escribir el código del SUT.

Hecho.

6. Escribir el código del SUT y mejorarlo hasta que pase todas las pruebas.

Hecho.

7. Utilizad los tests que habéis desarrollado para descubrir fallos en las implementaciones del SUT de otras parejas de compañeros.

Hecho.

Embotelladora

1. Identificar el Software Under Test

```
public class Embotelladora {

    // Parámetros: pequenas: número de botellas en almacén de 1L
    //             grandes : número de botellas en almacén de 5L
    //             total   : número total de litros que hay que embotellar
    // Devuelve:   número de botellas pequeñas necesarias para embotellar el total de líquido, teniendo
    //             en cuenta que hay que  minimizar el número de botellas pequeñas: primero
    //             se rellenan las grandes
    // Eleva la excepción NoSolution si no es posible embotellar todo el líquido

    public int calculaBotellasPequenas(int pequenas, int grandes, int total) throws NoSolution {
        //
    }

}
```

2. Identificar los parámetros del SUT

Tenemos 3 parámetros de tipo int que en este caso son pequenas,grandes y total. Que van a ser los parḿetros con los que vamos a comprobar el test.

3. Utilizando la técnica de la modelización de las entradas, crear caracterizaciones basadas en la funcionalidad, y sus particiones en bloques.

C1.:Cantidad de botellas
- Solo grandes.
-Solo pequeñas.
-Ambas.
-Ninguna.
-Pequeñas negativas.
-Grandes negativas.

C2.: Litros
-Litros negativos.
-Litros positivos.

C3.: Mayor cantidad de botellas
- Iguales botellas grandes y pequeñas.
-Más pequeñas que grandes.
-Más grandes que pequeñas.

C4.: Abastecimiento
-Botellas justas.
-Faltan botellas.
-Se aastece pero sobran botellas.

4. Elegir valores adecuados de cada bloque que satisfagan alguno de los criterios de cobertura "razonables" vistos en clase, prestando atención a los casos frontera que pueda haber en cada caso a la hora de elegir valores concretos para los tests.

C1.:Cantidad de botellas
- Solo grandes ==> 0,5,5
-Solo pequeñas ==> 5,0,5
-Ambas ==> 1,2,10
-Ninguna ==> 0,0,5
-Pequeñas negativas ==> -1,1,5
-Grandes negativas ==> 1,-2,10

C2.: Litros
-Litros negativos ==> 10,0,-10
-Litros positivos ==> 10,10,10

C3.: Mayor cantidad de botellas
- Iguales botellas grandes y pequeñas ==> 3,3,8
-Más pequeñas que grandes ==> 3,1,8
-Más grandes que pequeñas==> 1,3,8

C4.: Abastecimiento
-Botellas justas ==> 0,2,10
-Faltan botellas ==> 3,0,5
-Se aastece pero sobran botellas ==> 4,2,12

5. Escribir el código de pruebas con JUnit, antes de escribir el código del SUT.

Hecho.

6. Escribir el código del SUT y mejorarlo hasta que pase todas las pruebas.

Hecho.

7. Utilizad los tests que habéis desarrollado para descubrir fallos en las implementaciones del SUT de otras parejas de compañeros.

Hecho.

Black Friday

1. Identificar el Software Under Test

```
public class DescuentoBlackFriday {

    // Parámetros: precioOriginal es el precio de un producto marcado
    //            en la etiqueta
    //             porcentajeDescuento es el descuento a aplicar expresado como un porcentaje
    // Devuelve: el precio final teniendo en cuenta que si es black friday (29 de noviembre) se aplica
    //           un descuento de porcentajeDescuento
    // Eleva la excepción InvalidParameter si precioOriginal es negativo

    public double PrecioFinal(double precioOriginal, double porcentajeDbescuento) throws InvalidParameter
        //
    }

}
```

2. Identificar los parámetros del SUT

Tenemos 2 parámetros de tipo double que serán los que vamos a comprobar en el test. Dichos parámetros son precioOriginal y porcentajeDbescuento.

3. Utilizando la técnica de la modelización de las entradas, crear caracterizaciones basadas en la funcionalidad, y sus particiones en bloques.

C1.: Precio original respecto a 0
-Precio negativo.
- Precio 0.
-Precio positivo.

4. Elegir valores adecuados de cada bloque que satisfagan alguno de los criterios de cobertura “razonables” vistos en clase, prestando atención a los casos frontera que pueda haber en cada caso a la hora de elegir valores concretos para los tests.

C1.: Precio original respecto a 0
-Precio negativo ==> -3,0
- Precio 0 ==> 0,0
-Precio positivo ==> 49,50

5. Escribir el código de pruebas con JUnit, antes de escribir el código del SUT.

Hecho.

6. Escribir el código del SUT y mejorarlo hasta que pase todas las pruebas.

Hecho.

7. Utilizad los tests que habéis desarrollado para descubrir fallos en las implementaciones del SUT de otras parejas de compañeros.

Hecho.

Números Romanos

1. Identificar el Software Under Test

```
// Ver https://es.wikipedia.org/wiki/Numeraci%C3%B3n_romana
public class RomanNumeral {
    // Parámetro: s es un número romano
    // Devuelve : el número s en base 10
    // Eleva la excepción InvalidParameter si s no es un número romano

    public int convierte(String s) throws InvalidParameter {
```

2. Identificar los parámetros del SUT

Aquí también un único parámetro que sería el string a convertir.

3. Utilizando la técnica de la modelización de las entradas, crear caracterizaciones basadas en la funcionalidad, y sus particiones en bloques.

Caracterización q1:”Consideraciones para que un número sea romano”

C1==>null, es decir, el string vacío.
C2==>string que sea romano.
C3==>string que no sea romano.

4. Elegir valores adecuados de cada bloque que satisfagan alguno de los criterios de cobertura “razonables” vistos en clase, prestando atención a los casos frontera que pueda haber en cada caso a la hora de elegir valores concretos para los tests.

Valores:

C1==> el string vacío.
C2==>XCIX;MMCM.
C3==>ZKP;IIIIII.

5. Escribir el código de pruebas con JUnit, antes de escribir el código del SUT.

Hecho.

6. Escribir el código del SUT y mejorarlo hasta que pase todas las pruebas.

Hecho.

7. Utilizad los tests que habéis desarrollado para descubrir fallos en las implementaciones del SUT de otras parejas de compañeros.

Hecho.