# INFORME PRÁCTICA 4: PRUEBAS UNITARIAS DE SOFTWARE

## Introducción

El objetivo del documento es explicar cómo a partir de unas pruebas generadas a partir de la documentación de la funcionalidad del código, se escribe el código en sí. Y el uso de FEST para probar interfaces gráficas.

Primero hemos generado los casos de prueba con las técnicas de partición equivalente y AVL y caja negra, es decir casos de prueba basados solo en la especificación del módulo, en sus entradas y salidas, sin tener el código fuente diseñado. Después hemos implementado los casos en JUnit mediante 1-wise y a partir de las ejecuciones de las pruebas vamos implementando el código fuente de la clase Seguro. Aprovechando estos casos de prueba analizamos también la cobertura con la herramienta EclEmma. Luego aplicamos tres pruebas de integración a la clase SegurosGUI con FEST. Por útlimo diseñamos casos de prueba para la clase ListaOrdenada al igual que lo hicimos para Seguro pero adaptandolo a la estructura de datos, implementamos y comprobamos con JUnit y corregimos en el código fuente los errores que detectamos con los tests. En todo el proceso hemos utilizamos las herramientas Git y Maven al igual que en las prácticas anteriores.

## Proceso de pruebas de la clase Seguro

Para diseñar los casos de prueba de la clase Seguro hemos utilizado las técnicas de partición equivalente y AVL. Generando así una tabla por cada método de la clase que vamos a probar con los atributos que influyen en su ejecución.

Método: *Seguro(potecia, cliente, cobertura)*

Para el constructor de la clase, *Seguro()*,evaluamos los tres argumentos de entrada y tomamos sus clase válidas, no válidas y los valores respectivos. Para la potencia, dado que en el constructor no importa el intervalo de las potencia, si no simplemente que la potencia tenga un valor válido (los intervalos se tendrán en cuenta para el precio). Para la cobertura, dado que es un enumerado, en Java no podremos implementar las clases no válidas pero si se indica en la tabla. Para el cliente dado que no es objetivo de estudio, únicamente valoramos que no sea nulo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Clases Válidas | Valores | Clases No Válidas | Valores |
| Potencia | (0, inf) | 1 | <= 0 | 0  -30 |
| Cobertura | TODORIESGO  TERCEROSLUNAS  TERCEROS | 1000  600  400 | ¡{TODORIESGO, TERCEROSLUNAS, TERCEROS}  null | CUARTOS  null |
| Cliente | !null | cliente | null | null |

Método: *precio()*

Ahora para el método *precio()*, no recibe parámetros de entrada pero sí que hay ciertos atributos que se utilizan en su ejecución, por lo que los evaluamos en la tabla. Para la potencia esta vez, sí que hemos dividido las clases válidas en intervalos en función del porcentaje que se aplicaba ya que de ello dependerá, en parte, el precio resultante del seguro. Para la fecha del último siniestro, al igual que en la potencia dividimos en intervalos en función de los precios que se aplican a uno u otro. En ambos atributos, probamos un valor intermedio y un valor para cada límite inferior y superior.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Clases Válidas | Valores | Clases No Válidas | Valores |
| Cobertura | TODORIESGO  TERCEROSLUNAS  TERCEROS | TODORIESGO  TERCEROSLUNAS  TERCEROS | ¡{TODORIESGO, TERCEROSLUNAS, TERCEROS}  null | CUARTOS  null |
| Potencia | (0,90)  [90,110]  >110 | 1  43  89  90  100  110  150 | <= 0 | 0  -30 |
| Siniestralidad | [hoy-1año,hoy]  [hoy-3años,hoy-1año)  < hoy-3años  nunca | hoy-1año  hoy-0.5año  hoy  hoy-3años  hoy-2años  hoy-1año y 1 dia  hoy-3años y 1 dia  null | > hoy | hoy+1dia |
| Minusvalia | true  false | true  false |  |  |

Respecto a la implementación de los casos de prueba en Java, para el constructor y el método *precio()* se hacen en el mismo test ya que para comprobar un precio necesitamos crear previamente un seguro por lo que los comparten. De la misma forma ocurre con los métodos *getFechaUltimoSiniestro()* y *setFechaUltimoSiniestro()* que, para comprobar que se ha establecido bien la fecha indicada con el método *setFechaUltimoSiniestro()* , deberemos obtenerla con el método *getFechaUltimoSiniestro()*, por lo que probamos los dos simultáneamente.

Para los clientes hemos definido dos tipos, un *cliente1* que no tiene minusvalía y un *cliente2* que tiene minusvalía para así poder probar este parámetro también.

Casos de prueba válidos:

* Potencias entre 0 y 89, cobertura y minusvalía sin siniestro

1. (1,cliente1,TERCEROS, sinSiniestro): 400

2. (43,cliente1,TERCEROS\_LUNAS, sinSiniestro): 600

3. (89,cliente1,TODO\_RIESGO, sinSiniestro): 1000

* Potencias entre 90 y 110

4. (90,cliente1,TERCEROS, sinSiniestro): 420

5. (100,cliente1,TERCEROS, sinSiniestro): 420

6. (110,cliente1,TERCEROS, sinSiniestro): 420

* Potencias mayores de 110

7. (150,cliente2,TERCEROS, sinSiniestro): 360

8. (1,cliente1,TERCEROS, hoy): 600

* Con siniestro

9. (1,cliente1,TERCEROS, hoy-0.5año): 600

10. (1,cliente1,TERCEROS, hoy-1año): 600

11. (1,cliente1,TERCEROS, hoy-1año-1dia): 450

12. (1,cliente1,TERCEROS, hoy-2años): 450

13. (1,cliente1,TERCEROS, hoy-3años): 450

14. (1,cliente1,TERCEROS, hoy-3años+1dia): 400

15. (1,cl iente1,TERCEROS, null): 400

En el caso de prueba 2. evaluamos dos valores a la vez, que son la potencia y la cobertura y a partir del caso 3. ya tenemos todas las coberturas probadas. En el caso 7., el último ya tenemos todas las potencias probadas. Dejamos al final el atributo *fechaÚltimoSiniestro*.

Casos de prueba no válidos:

* Cobertura

1. (1,cliente1,CUARTOS): DatoIncorrectoException

2. (1,cliente1,null): DatoIncorrectoException (coberturas comprobadas)

* Potencia

3. (0,cliente1,TERCEROS): DatoIncorrectoException

4. (-30,cliente1,TERCEROS): DatoIncorrectoException (potencias comprobadas)

* Cliente

5. (1,null,TERCEROS): DatoIncorrectoException (cliente comprobado)

En todos los casos deberíamos coger una excepción de tipo *DatoIncorrectoException* y si no el test fallará.

Método: *setFechaUltimoSiniestro(fechaUltimoSiniestro)*

Para el método *setFechaUltimoSiniestro()*, simplemente comprobamos que acepta una fecha válida (anterior al día de hoy u hoy) y que no acepta una fecha inválida (posterior al día de hoy).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Clases Válidas | Valores | Clases No Válidas | Valores |
| Siniestralidad | < hoy | hoy-1dia | > hoy | hoy+1dia |

Casos de prueba válidos:

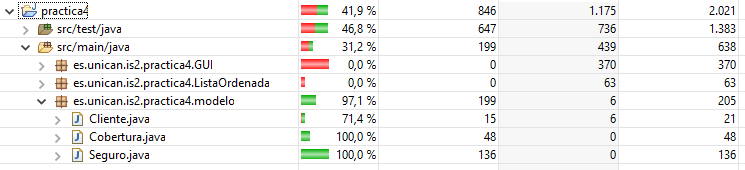
(hoy - 1 dia): hoy - 1 dia

Casos de prueba no válidos:

(hoy + 1 dia): DatoIncorrectoException

**Comprobación de la cobertura**

Por último utilizamos la herramienta EclEmma para analizar la cobertura con las pruebas de caja negra diseñadas con el críterio de condición-decisión. Como resultado obtenemos que se hace una cobertura del 100% de la clase *Seguro* por lo que no es necesario añadir más casos de prueba. A continuación se muestra el resultado de EclEmma al ejecutar los tests:



## Proceso de pruebas de la clase SegurosGUI

Para las pruebas de integración deberemos diseñar un programa utilizando FEST con el que escriba la fecha del siniestro, elija la cobertura, escriba la potencia, marque o no la minusvalía, pulse el botón de “CALCULAR” y compruebe si el precio es el correcto. Para ello diseñamos tres casos de prueba con los que miramos dos casos válidos y uno no válido para comprobar que se informa del error.

(No hemos probado el diseño de la interfaz ya que al no proporcionarse como debería ser, no podemos evaluar si sigue un diseño. Entendemos que no es objetivo de la práctica)

Casos de prueba:

{hace 100 dias, TERCEROS, 60, minusvalia}: 450.0

{hace 2 años, TODO\_RIESGO, 100, no minusvalia}: 1100.0

{hace 10 años, TERCEROS\_LUNAS, -20, no minusvalía}: “Error. Dato introducido incorrecto.”

Al ejecutar los test, encontramos dos problemas. Primero que la fecha con el delimitador “/” lanza un error con FEST por lo que lo cambiamos a “-”. Segundo que al introducir un caso no válido no se informaba del error si no que se indicaba precio 0.0, por lo que cambiamos el catch que manejaba la excepción *DatoIncorrectoException* para que imprimiera un mensaje de error en el cuadro de texto del precio.

## Proceso de pruebas de la clase ListaOrdenada

Primero de todo hemos instalado el archivo .jar mediante Maven para poder utilizar los métodos de la clase sin poder tener acceso a ella.

Creamos una tabla como en el apartado 2. utilizando AVL pero de una manera parcial ya que al tratar con una estructura de datos el enfoque de las pruebas es un poco diferente. Y nos centramos más en el estado de la lista a medida que efectuamos las pruebas. Con esto construimos la siguiente tabla. (Para el formato de la tabla, hemos seguido el visto en clase)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Clases Válidas | Valores | Clases No Válidas | Valores |
| add(e) | e != null  lista vacía  lista con un elemento  lista con elementos | 2 en [] ([2])  4 en [2] ([2,4])  3 en [2,4]: ([2,3,4]) | e == null | null en []: NullPExc  null en [1,2,4]: NullPExc |
| get(i): e | 0 <= i < tamaño lista  lista con un elemento  lista con elementos | i=0 en [2]: (2, [2])  i=1 en [2,4]: (4, [2,4]) | i < 0  i > tamaño  lista vacía | i=-1 en [2,3]: IOOBExc  i=5 en [2,3]: IOOBExc  i=0 en []: IOOBExc |
| remove(i): e | 0 < i < tamaño lista  lsita con un elemento  lista con elementos | i=0 en [4]: (4, [])  i=2 en [2,4,8]: (8, [2,4]) | i < 0  i > tamaño  lista vacía | i=-1 en [2,3]: IOOBExc  i=5 en [2,3]: IOOBExc |
| size(): i | lista vacía  lista con un elemento  lista con elementos | []: 0  [2]: 1  [2,5,8,9]: 4 |  |  |
| clear() | lista vacía  lista con un elemento  lista con elementos | []: []  [6]: []  [2,4,5,7,8]: [] |  |  |

Para evaluar el funcionamiento de la interfaz *IListaOrdenada.java* proporcionada en el fichero .jar hemos creado una serie de test recogidos en *ListaOrdenadaTest.java* donde vamos probando diferentes casos para detectar posibles errores.

Errores detectados:

1. Un error es que en el método *add()* debería mandarse un *NullPointerException* cuando se intenta meter un elemento que es nulo. Esto no lo hace excepto cuando la lista no está vacía, caso en el que si se añade el elemento nulo.
2. El método *clear()* no funciona para ningún caso, puede deberse a que el método está sin completar.
3. El método *remove()* lanza una excepción cuando se intenta eliminar un elemento en una lista con un solo elemento. Además, cuando hay más de un elemento no elimina el elemento indicado, si no el anterior, de ahí que viene la excepción que se comentó antes, puede que elimine el elemento anterior al indicado y cuando este es el primero o único accede fuera de la lista.

Consecuencias de los errores:

1. El método *add()*, cuando se mete un elemento nulo en una lista vacía no lanza excepción, pero cuando se hace en una lista con elementos sí que lanza excepción porque al intentar ordenar los elementos, compara los valores y en el momento de ver el valor del elemento que queremos añadir falla porque intenta acceder a un elemento nulo y lanza.
2. El método *clear()* únicamente clona la lista que se pase como parámetro y finaliza, lo cual no es el comportamiento esperado.
3. En el método *remove()* podemos confirmar que efectivamente elimina el elemento anterior al indicado, ya que hace *lista.remove(i-1)*. Este fallo se daría al eliminar el primer elemento de la lista independientemente del número de elementos que haya en la lista (con los test podríamos llegar a pensar que solo se daba cuando hay un elemento pero el fallo se detecta igualmente).

Solución de los errores:

1. Para el método *add()* añadimos una sentencia if a modo de gestión de errores en el que en caso de que se vaya a añadir un elemento nulo a la lista, se lance la excepción *NullPointerException*.
2. Para el método *clear()* simplemente cambiamos la llamada al método *clone()* de la lista por el método correcto que es *clear()*, de esta forma ya hace el comportamiento esperado.
3. Para el método *remove()* basta con eliminar el ‘-1’ que provocaba el acceso a una posición anterior a la indicada creando en alguna ocasión el lanzamiento de la excepción.

Aclaración: hemos interpretado que no está permitido añadir elementos nulos en la lista, en caso de que si se permitiese, el error estaría en evitar la comparación del elemento nulo con los demás de la lista y colocarlos al inicio, final o no colocarlos en la lista.