**MEMORIA PRÁCTICA 4**

Borja Cuevas Cuesta y Adrián García Cubas

1. INTRODUCCIÓN

En este documento, pretendemos dejar constancia de la metodología que hemos seguido a la hora de realizar las pruebas y exponer cómo ha sido el proceso de diseño e implentación de las mismas para los diferentes casos que se nos planteaban. Además, detallaremos su utilidad y cómo nos han ayudado a mejorar/perfeccionar la funcionalidad y calidad de nuestro programa.

El proceso para realizar las pruebas del código de la práctica ha sido:

* **Diseño de casos de prueba** para cada método aplicando, primero, las técnicas de caja negra de partición equivalente y AVL. Para ello, analizábamos para cada paramétro y entrada del método los conjuntos de valores que hacían a la clase válida o no válida (cada conjunto de valores que genera igual respuesta cualitativa se denomina clase de equivalencia) y, aplicando la técnica de partición equivalente, escogíamos un valor interesante de cada clase de equivalencia del método para, combinándolos, diseñar los casos de prueba. Esta combinación de valores de los distintos paramétros para modelar un caso de prueba la hemos realizado cuando todos los valores eran válidos. Para diseñar los casos de prueba no válidos, solamente uno de estos valores debía pertenecer a una clase no válida.

Posteriormente, aplicando la técnica AVL, ampliábamos los casos de prueba obtenidos con valores límite de las clases de equivalencia (lógicamente, para aquellas clases en las que hubiese distintos rangos de valores válidos y no válidos).

Para construir los casos de prueba, hemos aplicado la cobertura 1-wise: cada valor interesante de cada parámetro lo incluimos en al menos un caso de prueba.

* **Implementación de los casos de prueba:**

Con los casos de prueba diseñados en el papel y el código completado, el siguiente paso era implementar, a través de la librería JUnit, los test de los métodos de negocio para comprobar su correcto funcionamiento. En esta implementación, debíamos llevar al código los casos de prueba diseñados en el paso anterior y comprobar que los resultados obtenidos al ejecutar el programa son los mismos que los valores de retorno esperados.

* **Corrección de errores**

Partiendo de que hemos implementado los test correctamente, esta fase consistía en corregir los errores en los casos en los que los valores de retorno obtenidos no coincidieran con los valores de retorno esperados (calculados en el paso uno a mano).

* **Comprobación de la cobertura obtenida**

Con el programa finalizado y funcionando correctamente, la última fase consistía en analizar la cobertura del código (a través de la herramienta Eclemma) con el objetivo de comprobar que, con los casos de prueba diseñados, se daban todas las casuísticas posibles que hacían que se evaluase el código para todo tipo de situaciones distintas (cobertura decisión/condición). De esta manera, nos asegurábamos que nuestro código funcionaba correctamente para cualquier caso.

Las técnicas de prueba aplicadas para las pruebas han sido Partición Equivalente y AVL (Análisis de Valores Límite), siguiendo la estrategia de caja negra, y el análisis de la cobertura de decisión/condición, siguiendo la estrategia de caja blanca. Para las pruebas de integración de las clases SegurosGUI y Seguro, hemos utilizado la herramienta FEST, un conjunto de librerías que permiten la automatización de pruebas de interfaces gráficas Swing (integrada con JUnit).

Las pruebas de caja negra consisten en definir los casos de prueba basándonos en la especificación de cada método en concreto, tratando de cubrir todo el espectro de entradas y salidas posibles. Para ello, hemos empleado las técnicas de Partición Equivalente y AVL, explicadas anteriormente, con las que tratamos de conseguir ejecutar nuestro programa bajo todas las posibles situaciones.

Las pruebas de caja blanca consisten en diseñar casos de prueba conociendo la implementación interna de los métodos para buscar ejecutar el código bajo una situación muy concreta y conseguir así alcanzar la cobertura deseada. Para diseñar las pruebas de caja blanca, nos hemos apoyado en la herramienta Eclemma, de Eclipse, que dividía las instrucciones en ejecutadas, no ejecutadas y parcialmente ejecutadas. De esta forma, éramos capaces de comprobar si habíamos cubierto todos los posibles casos de ejecución con los casos de prueba diseñados para las pruebas de caja negra o debíamos añadir nuevos casos para alcanzar la cobertura deseada.

El criterio de cobertura elegido a la hora de construir los casos de prueba ha sido el criterio 1-wise, en el que, como hemos dicho previamente, cada valor interesante de cada parámetro se incluía al menos en un caso de prueba.

1. PROCESOS DE PRUEBA DE LA CLASE SEGURO

Aplicamos la técnica de caja negra mediante Partición Equivalente y AVL.

* Método constructor: Seguro(potencia: int, tomadorSeguro: Cliente, cobertura: Cobertura)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| potencia | 1. (0, +inf) | 6. (-inf, 0] | 1. 1, 17 | 6. 0  7. -5 |
| cobertura | 2. Terceros  3. Todoriesgo  4. terceroslunas | 7. Null | 2. Terceros  3. Todoriesgo  4. Terceroslunas | 8. null |
| tomadorSeguro | 5. no null | 8. null | 5. Cliente | 9. null |

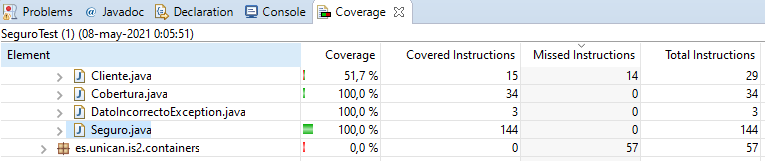
|  |  |
| --- | --- |
| Casos de prueba | |
| Casos válidos | Casos no válidos |
| 1. (1, terceros, cliente): Seguro  2. (17, todoriesgo, cliente): Seguro  3. (1, terceroslunas, cliente): Seguro | 4. (0, terceros, cliente)  5. (-5, todoriesgo, cliente)  6. (1, null, cliente)  7. (1, tercerloslunas, null) |

* Método precio():

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| fechaUltimoSiniestro | 1. [fechaActual, fechaActual – 1año]  2. (fechaActual-1año, fechaActual - 3 años)  3. [fechaActual-3años, -inf) | 12. > fechaActual | 1. fechaActual  2. fechaActual – 50 días  3. fechaActual – 1 año  4. fechaActual – 366 días  5. fechaActual – 500 días  6. fechaActual – 1094 días  7. fechaActual – 1095 días  8. fechaActual – 2000 días  9. null | 23. fechaActual + 1dia  24. fechaActual + 50 días |
| Potencia | 4. [1, 89]  5. [90, 110]  6. [111, +inf) | 13. null  14. (-inf, 0] | 10. 1  11. 65  12. 89  13. 90  14. 105  15. 110  16. 111  17. 725 | 25. 0  26. -5 |
| Cobertura | 7. Terceros  8. Todoriesgo  9. Terceroslunas | 15. null | 18. Terceros  19. Todoriesgo  20. Terceroslunas | 27. Null |
| minusvalía | 10. True  11. False | 16. null | 21. True  22. False | 28. null |

|  |  |
| --- | --- |
| Casos de prueba | |
| Casos válidos | Casos no válidos |
| 1. (fechaActual, 1, teceros, true): 450  2. (fechaActual – 50días, 65, todoriesgo, false): 1200  3. (fechaActual – 1año, 89, todoriesgo, true): 900  4.( fechaActual – 366días, 90, terceroslunas, true): 510  5. fechaActual – 500días, 105, terceros, true): 352.5  6. (fechaActual – 1094dias, 110, terceros, false): 470  7. (fechaActual – 1095dias, 111, todoriesgo, true): 900  8. (fechaActual – 2000días, 725, terceros, false): 480  9. (null, 725, todoriesgo, false): 1200 | 10. (fechaActual + 1dia, 1, terceros, true)  11. (fechaActual + 50días, 50, terceroslunas, false)  12. fechaActual, 0, terceroslunas, false  13. fechaActual, -5, terceroslunas, true  14. fechaActual – 1096dias, 90, null, true  15. fechaActual – 364dias, 110, terceros, null |

Aplicamos el test de cobertura mediante la herramienta Eclemma con el objetivo de alcanzar la cobertura de decisión/condición.



La cobertura alcanzada es del 100%, es decir, tanto las decisiones como las condiciones se evalúan a falso y a verdadero al menos una vez.

Inicialmente, la cobertura no era del 100% debido a que teníamos métodos observadores y “setters” en la clase Seguro que no se utilizaban en ningún momento. Por esta razón, decidimos eliminarlos al conisderarlos innecesarios, dado que en el enunciado de la práctica no se especifica que se puedan cambiar los parámetros del seguro.

También, en la estructura switch en la que se evalúa la cobertura del seguro, teníamos un deault encargado de gestionar los casos en los que la cobertura no era ninguna de las opciones contempladas, pero lo eliminamos también porque, en nuestra implementación, la cobertura no puede tomar un valor distinto.

1. PROCESO DE PRUEBAS DE LA CLASE SEGUROSGUI

Los casos de prueba definidos para la clase SegurosGUI son los mismos que hemos diseñado para el método precio de la clase Seguro (siguiendo las mismas clases de equivalencia), ya que en realidad el método que se está probando es el método precio, solo que ahora se establecen los parámetros del seguro a través de una interfaz gráfica y no directamente en el código.

Para las pruebas de integración de la clase SegurosGUI y Seguro, hemos seleccionamos tres casos de prueba diferentes de los que habíamos diseñado para el método precio() y los hemos implementado con la herramienta FEST (incluida en JUnit). Hemos procurado que los casos de prueba seleccionados contuviesen parámetros pertenecientes a distintas clases de equivalencia para abarcar el máximo espectro de posibles situaciones. Sin embargo, no hemos podido evitar que algunos parámetros se repitan.

Los casos de prueba seleccionados han sido los números 4, 6 y 7.

1. PROCESO DE PRUEBA DE LA CLASE LISTAORDENADA

Para realizar las pruebas de esta clase, primero hemos realizado unas pruebas de caja negra sobre el .jar ejecutable tratando de comprobar si el código funcionaba correctamente sin conocer nosotros su implementación. Para ello, hemos utilizado también las técnicas de Partición Equivalente y AVL.

* Método get(indice: int)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| índice | 1. [0, size-1] | 4. (-inf, 0)  5. [size, +inf)  6. null (imposible en java) | 1. 0  2. 2  3. size-1 | 6. -7  7. -1  8. size  9. size+2 |
| lista | 2. Lista con un elemento  3. Lista con elementos | 7. Lista vacía | 4. [1]  5. [1,2,3,4] | 10. [] |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Casos de prueba | | |
| Casos de prueba válidos | Valor esperado | Casos de prueba no válidos | Valor esperado |
| 1. (0) en [1]  2. (2) en [1,2,3,4]  3. (size-1) en [1,2,3,4] | 1  3  4 | 4. (-7) en [1,2]  5. (-1) en [1,2]  6. (size) en [1,2]  7. (size+2) en [1,2]  8. (0) en [] | IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException |

* Método add(elemento)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| lista | 1. Lista vacía  2. Lista con un elemento  3. Lista con elementos | 5. elemento nulo (imposible en java) | 1. []  2. [1]  3. [2,3,4] |  |
| elemento | 4. (-inf, +inf) |  | 4. 1 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Casos de prueba | | |
| Casos de prueba válidos | Valor esperado | Casos de prueba no válidos | Valor esperado |
| 1. (1) en []  2. (1) en [1]  3. (1) en [2,3,4] | [1]  [1,1]  [1,2,3,4] |  |  |

* Método remove(indice: int)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| índice | 1. [0, size-1] | 4. (-inf, 0)  5. [size, +inf)  6. null (imposible en java) | 1. 0  2. 2  3. size-1 | 6. -7  7. -1  8. size  9. size+2 |
| lista | 2. Lista con un elemento  3. Lista con elementos | 7. Lista vacía | 4. [1]  5. [1,2,3,4] | 10. [] |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Casos de prueba | | |
| Casos de prueba válidos | Valor esperado | Casos de prueba no válidos | Valor esperado |
| 1. (0) en [1]  2. (2) en [1,2,3,4]  3. (size-1) en [1,2,3,4] | []  [1,2,4]  [1,2,3] | 4. (-7) en [1,2,3,4]  5. (-1) en [1,2,3,4]  6. (size] en [1,2,3,4]  7. (size+2) en [1,2,3,4]  8. (0) en [] | IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException  IndexOutOfBoundsException |

* Método size()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| lista | 1. Lista vacía  2. Lista con un elemento  3. Lista con elementos |  | 1. []  2. [1]  3. [1,3,5,7] |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Casos de prueba | | |
| Casos de prueba válidos | Valor esperado | Casos de prueba no válidos | Valor esperado |
| 1. []  2. [1]  3. [1,3,5,7] | 0  1  4 |  |  |

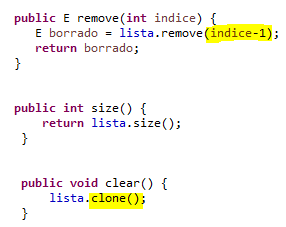
* Método clear()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parámetro | Clases de equivalencia y valores | | | |
| Clases válidas | Clases no válidas | Valores válidos | Valores no válidos |
| lista | 1. Lista vacía  2. Lista con un elemento  3. Lista con elementos |  | 1. []  2. [1]  3. [1,2,3,4] |  |

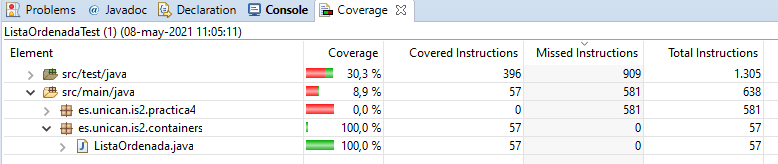
|  |  |  |  |
| --- | --- | --- | --- |
|  | Casos de prueba | | |
| Casos de prueba válidos | Valor esperado | Casos de prueba no válidos | Valor esperado |
| 1. []  2. [1]  3. [1,2,3,4] | []  []  [] |  |  |

Hemos decidido crear un método test por cada método de la clase ListaOrdenada para poder ver de forma más sencilla qué métodos eran los que fallaban y encontrar los errores más rápidamente.

Al ejecutar los test sobre el ejecutable aportado para la práctica, vimos que fallaban los tests clearTest() y removeTest(), y con el código de ListaOrdenada ya en la mano, corregimos las líneas que causaban los errores:



Para realizar ahora las pruebas de caja blanca, utilizamos la herramienta Eclemma para analizar la cobertura siguiendo el criterio de decisión/condición.



En este caso, la cobertura alcanzada también es del 100%. Sin embargo, en un principio había una condición del método add que Eclemma marcaba como parcialmente ejecutada. Esto sucedía porque en los casos de prueba implementados, siempre metíamos los elementos en las listas ordenados, por lo que nunca tenía lugar una reordenación interna.

La sentencia parcialmente ejecutada era la siguiente:

elemento.compareTo(lista.get(indice))>0

Como los elementos siempre se metían a la lista por orden, esta condición siempre era verdadera. Lo solucionamos metiendo algunos elementos en desorden para comprobar que la ordenación interna del método se realizaba correctamente.