Taller 7

Ernesto Pérez – 202112530

Germán Moreno – 202116701

Sofia Velásquez Marín– 202113334

Lo primero y más difícil a la hora de diseñar las pruebas fue entender bien el proyecto, como esta descrito en el PDF es un documento con un alto grado de acoplación y en un principio es confuso entender que tenemos un árbol de Categorías pero que podemos tener más categorías dentro de una categoría (Televisores y Computadores hacen parte de tecnología).

Una vez entendido como está estructurado el programa nos encontramos con otra dificultad a la hora de realizar las pruebas y es que Categoría depende mucho de Almacén y viceversa. Esto implica que las pruebas no se pueden realizar individualmente para cada clase, sino que se deben probar casi que 'en conjunto' lo cual no es lo ideal y en ocasiones dificulta la detección de errores.

Un **ejemplo** de esto es el método agregarProducto de la clase Almacen que, dentro de su implementación hace un llamado al buscarProducto de la clase Categoria y al método agregarProducto de la clase Marca. Asimismo, la clase Almacen solo nos permite ver la Categoria Raiz del arbol con el método darCategoriaRaiz por lo que tenemos que usarlo si queremos ver lo que hay dentro de una clase.

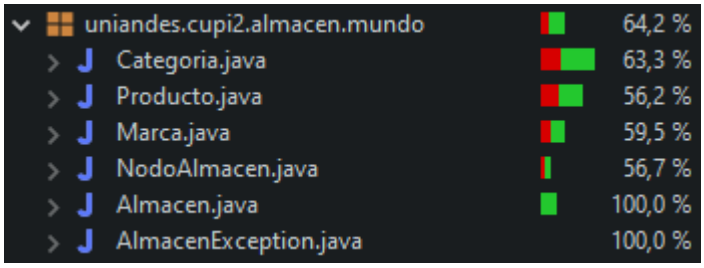
Por esto, a la hora de crear la prueba tenemos que:

1. Agregar el producto bajo un identificador
2. Entrar a la categoría raíz
3. Buscar el producto creado previamente

En esta prueba estamos usando/probando 3 métodos diferentes (darCategoriaRaiz, buscarProducto, darCodigo).

```
@Test
void TestAgregarProducto() throws AlmacenException
{
    this.almacen.agregarProducto("1112", "codigoTest", "nombreTest", "descripcionTest", 18519198);
    assertEquals("codigoTest",this.almacen.darCategoriaRaiz().buscarProducto("codigoTest").darCodigo());
    assertEquals("nombreTest",this.almacen.darCategoriaRaiz().buscarProducto("codigoTest").darNombre());
    assertEquals("descripcionTest",this.almacen.darCategoriaRaiz().buscarProducto("codigoTest").darDescripcion());
    assertEquals(18519198,this.almacen.darCategoriaRaiz().buscarProducto("codigoTest").darPrecio());
}
```

Esto resulta no solo confuso sino poco práctico e ineficiente a la hora de hacer pruebas, pero debido al alto acoplamiento la mayoría de las pruebas tuvo una implementación muy similar a la descrita anteriormente. Por lo que al terminar las pruebas de la clase Almacén se avanzó en las pruebas de las demás clases como se muestra a continuación:



En la imagen se ve que al terminar el 100% de Almacén ya se completó más del 50% para las demás clases sin hacer pruebas directas de sus métodos.

Otra dificultad que se encontró es cuando pasamos a hacer las pruebas de Categoría, esto debido a que cada Categoría hace parte de un árbol dentro de Almacén y para poder usar mock objects lo más fácil era cargar los datos dentro de la misma clase usada para los test de almacén por lo que era necesario realizar pruebas de los métodos de categoría dentro de la clase AlmacenTest.

En adición a esto, otra dificultad fue pensar en todos los posibles casos que se les podía dar a un método como parámetros. Con esto nos referimos a que todos los métodos debían ser probados con casos básicos, casos extremos pero que deberían funcionar y los casos en donde se debería votar la excepción (si el método vota excepción) y revisar que funcione adecuadamente bajo dichos parámetros.

Muchas de las dificultades que se presentaron para la clase Almacen fueron las mismas encontradas en la clase Cateogoria. Notablemente el hecho de que para que se pudiera usar bien esta clase debemos tener el arbol n-ario creado para que se pueda buscar los productos, quitarlos, hacer el preorden y el posorden, etc. Por lo que era necesario acceder a estos métodos desde la clase almacen después de cargar unos datos dados y de ahí probar los métodos de Categoria.

Parte 2

Lo primero a lo que se debe de hacer un análisis es a que es lo que nos exige el 'contrato' que tenga nuestra aplicación estos requerimientos son:

- Visualizar el almacén
- Agregar un nodo al árbol de marcas y categorías.
- Eliminar un nodo del árbol de marcas y categorías.
- Agregar un producto a una marca.
- Eliminar un producto.
- Vender una cantidad dada de un producto.
- Calcular el valor de las ventas de un producto y un nodo del árbol de marcas y categorías.
- Mostrar el preorden y posorden del árbol de marcas y categorías.
- Mostrar los productos en inorden por marca.

Las pruebas diseñadas deben estar enfocadas a que el programa construido cumpla dichos requerimientos.

- **Plan de pruebas:**

Para la clase almacén, identificamos los servicios que debe ofrecer revisando la documentación y es así como concluimos que es primordial que dicha clase sea capaz de realizar una carga de datos, adición o sustracción de elementos y consultar sobre los elementos de la misma. Partiendo del anterior análisis, identificamos que los siguientes métodos son esenciales para la aplicación y debemos verificar su correcto funcionamiento. Luego nos dimos cuenta de que la mayoría de los requerimientos

de la clase almacen son prácticamente métodos que utiliza la clase Categoría; así que, podemos concluir que estos los métodos con el mismo nombre tendrán el mismo comportamiento.

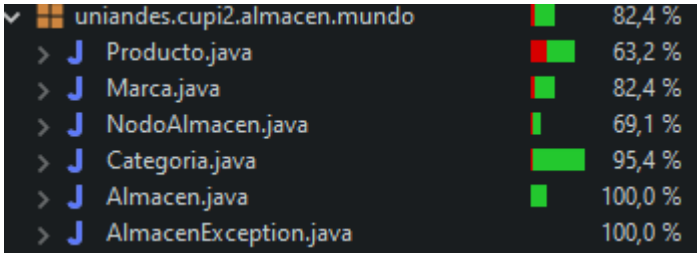
- ◊ Almacen (File pArchivo) : Este método es el encargado de crear una instancia del almacén, es decir, el objeto base que ofrece los servicios de la aplicación. Para esta parte, se hace una carga de datos al recibir un archivo desde el directorio data. En un caso hipotético, podría recibir como parámetro un elemento inexistente. Lo mejor sería probar el funcionamiento de esta clase con el SetUp.
- ◊ agregarProducto(String pIdMarca, String pCodigo, String pNombre, String pDescripcion, double pPrecio) : Debemos verificar que este método si está agregando un producto, lo cual se lograría agregando un producto y posterior la búsqueda al elemento, de tal manera que se garantice su existencia en la aplicación.
- ◊ agregarNodo(String pIdPadre, String pTipo, String pIdentificador, String pNombre) : Este método se encarga de añadir un nuevo nodo. La estructura interna del almacén debería actualizarse cuando se utiliza este método, lo cual se puede lograr con una posterior búsqueda sobre la estructura, sin embargo, entrara en conflicto cuando se trate de agregar un nodo ya existente.
- ◊ eliminarProducto(String pCodigo) / eliminarNodo(String pIdNodo) : Este método se encargará de eliminar un producto/nodo existente en la estructura de la aplicación. Para probar la eficacia, podríamos utilizar un elemento que sabemos que existe e invocar este método; posteriormente, utilizaríamos un AssertNull para garantizar que si fue eliminado.
- ◊ buscarNodo(String pIdNodo) : Para esta parte, consideramos que lo primordial es garantizar que el nodo que retorne sea el mismo que se pasó en ID, así que para verificar debemos comparar el retorno con el nombre correspondiente al ID dado como parámetro.
- ◊ buscarPadre(String pIdNodo) : Nos interesa saber si este método efectivamente está verificando esta información teniendo en cuenta que conocemos la estructura general y los elementos que deberían estar presentes en cada Categoria. Por otro lado, debemos garantizar la aplicación sea capaz de responder cuando un método este siendo utilizado para consultar sobre un nodo inexistente.

➢ Diseño de pruebas:

- ▫ Constructores: Hay que garantizar que la construcción de cada una de las clases funcione, razón por la cual las concentramos en una sola prueba. Si todo se ejecuta correctamente, habremos probado que el método cumple con su función. Se usa BeforeEach para ejecutarse primero y permitir la realización de más pruebas.
- ▫ darLista: Hay métodos que se encargan de retornar las listas, entonces debemos garantizar que si estén cumpliendo con esta función. Lo mejor sería revisar que las listas retornadas no estén vacías, así garantizamos que la operación se realizó con éxito. Esta prueba se limita a utilizar AssertFalse() sobre los métodos darNodos(), darProductos(), darPosorden() y darPreorden() para garantizar que al menos haya un elemento después de la carga de datos.
- ▫ Valor de Ventas: Se implementa con un AssertEquals() para determinar que el precio que posee es el mismo que nosotros conocemos
- ▫ Búsqueda de nodos: Este método utiliza AssertEquals(), dado que ya tenemos conocimiento de cada uno de los nodos y todo se reduce a comparar el nombre del nodo que buscamos y el nombre del nodo retornado.

- Agregar y eliminar nodos/productos: Esta parte se encargará de utilizar las operaciones con nodos/productos y revisar que este efectuando un cambio. Para esto, utilizamos ambos AssertEquals y AssertNull : iniciamos agregando un nuevo nodo/producto para luego utilizar AssertEquals() para buscar el elemento agregado y comparar su nombre con el nombre que le dimos al objeto recién añadido. Posteriormente, debemos deshacer el código para que no se conserven los cambios dentro de la aplicación, asi que eliminamos el nodo/producto y llamamos AssertNull() para llamar la clase y corroborar su inexistencia.
- Venta de productos: Para esta parte, creamos un nuevo producto y le asignamos un número fijo de unidades y posteriormente vendemos una cantidad especifica. Para probar su funcionamiento, utilizamos AssertEquals() sobre el objeto utilizando la cantidad de ventas por producto; es decir, comparamos la cantidad de unidades vendidas que ingresamos con las unidades vendidas del producto.
- Métodos que lanzan excepción: Algunos métodos pueden lanzar excepciones por el hecho de que se están realizando operaciones en situaciones donde no deberían y para manejar los casos de prueba usamos el método AssertThrows().

Al finalizar las pruebas logramos una cobertura bastante alta en el programa



En la imagen se ve que al terminar tenemos el 100% de cobertura para la clase Almacén y el 95,4% de cobertura para la clase Categoria, y se completó más del 60% para las demás clases sin hacer pruebas directas de sus métodos.