Trabajo Práctico Final - Taller de programación 1

Grupo 7
Autores: Savarino Jeremías, Errobidart Facundo
Desarrolladores: Priano Sobenko Bautista, Rojas Agustín
Fecha: 16-11-2022
Taller de Programación 2° Cuatrimestre 2022

Introducción

En este trabajo, se solicitó a los alumnos desarrollar un sistema informático con el objetivo de administrar el funcionamiento diario de un local gastronómico. Para poder realizar este sistema la cátedra proporcionó la narrativa del problema,las funcionalidades deseadas y el documento de Especificación de Requerimientos de Software.

Estos documentos son claves tanto para los desarrolladores del sistema, como para los encargados de testearlo, ya que permite (en la mayoría de los casos) definir claramente las pre y las postcondiciones, y desarrollar una documentación adecuada para cada clase y para cada método, eliminando así la mayor parte de las dudas que pueden llegar a surgir en el desarrollo sistema, y permitiendo detectar la mayoría de los casos posibles que puedan generar problemas en la ejecución del programa.

El foco principal de este trabajo (y de esta materia) es el testing, una tarea esencial en el proceso de desarrollo de software, ya que gracias a él es posible detectar errores e incertezas durante todo el proceso, para luego realizar las correcciones, y obtener así un producto de software consistente y robusto.

Realizar testeos en forma continua para reconocer problemas en forma temprana, permite que se mantenga la consistencia del software. De no testear continuamente , correríamos el riesgo de que, al intentar solucionar uno de estos errores, se generen nuevas fallas que dificultan el desarrollo, y retrasen los tiempos, generando así pérdidas de tiempo y dinero .

Algo importante a considerar es que nunca es posible cubrir todos los errores con un testeo, pero aplicando un conjunto de técnicas se encontrarán gran parte de estos.

En este trabajo, se realizaron las pruebas relativas a Test Unitario de Caja Negra, Caja Blanca, Test de Persistencia de datos, Test de interfaces Gráficas mediante el uso de un robot, y por último, Test de Integración.

Desarrollo

Test de Caja Negra

El Test de Caja Negra es un tipo de prueba muy útil, caracterizado en que el testeador no tiene acceso al código. Si no que se basa en la información dada por parte de la documentación del programa, en la cual se establecen condiciones sobre las funciones, qué parámetros son aceptados, y cuáles son las salidas esperadas.

A partir de estas es que podemos crear pruebas y verificar que una cierta entrada produzca la salida correcta.

La prueba de Caja Negra se basa en el testeo del elemento mínimo del lenguaje utilizado. En nuestro caso, las clases. De esta forma, se propone aislar los errores, de modo que se puedan hallar fácilmente.

Para realizar las pruebas sobre el proyecto se procedió a detectar en un principio los casos de prueba, y luego, utilizando las dependencias de JUnit , se generaron los distintos métodos de prueba, los cuales permiten testear los diferentes métodos de las clases elegidas para este tipo de testing.

Estas clases fueron elegidas en base a la documentación: se escogieron para testear aquellos métodos cuya documentación permitía un correcto análisis de entradas y salidas de los mismos.

No es posible realizar todo el testeo de todas las clases con el Test de Caja Negra, debido a que hay algunos casos donde la documentación es confusa y hay otros casos donde es imposible verificar el cumplimiento de los contratos.

Una vez seleccionados las clases y los métodos a probar, se procedió a planificar y crear los escenarios que permitan utilizar la entrada que mejor se adaptara al entorno de la función seleccionada.

Testeo del método altaMesa( )

Para este testeo se implementó el método altaMesaOk( ), el cual busca verificar la correcta creación de una mesa nueva, y su posterior adición a la lista de mesas que contiene la Empresa.
También se desarrolló el método altaMesaFail( ) el cual contempla el caso de que la mesa ya exista y se encuentre en la lista, y el correcto lanzamiento de la excepcion MesaExistenteException( ).

Escenarios

Nro Escenario	**Descripción**	**Lista Mesas**
Escenario 1	La colección de mesas está vacía	Lista= [ ]
Escenario 2	La colección de mesas no está vacía. La mesa ya se encuentra en la colección.	Lista= [mesa1 = {nroMesa = 1, cantSillas = 3}]

Tabla de particiones

Condición de entrada	Clases válidas	Clases inválidas	Escenario
CrearMesaRequest	mesa { nroMesa != 1 **2.1** }	mesa {nroMesa == 1 **2.2**}	**2**
CrearMesaRequest	mesa {nroMesa > 0 **1.1** ,cantSillas > 0 **1.2**}	masa {nroMesa <= 0 **1.3**, cantSillas <= 0 **1.4**}	**1**

Batería de pruebas

Tipo de clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas	Salida obtenida
Correcta	mesa = { nroMesa = 2, cantSillas = 4 }	Lista=[mesa]	**1.1, 1.2, 2.1**	lista = [mesa]
Correcta	mesa = { nroMesa = 1, cantSillas = 3 }	MesaExistenteException( ).	**2.2**	MesaExistenteException( ).
Incorrecta	mesa = { nroMesa = 1, cantSillas = -10 }	Error	**1.4**	lista = [mesa]
Incorrecta	mesa = { nroMesa = -1, cantSillas = 2	Error	**1.3**	lista[mesa]

	}			

Como se puede observar en la batería de pruebas en algunos casos se obtienen clases inválidas, si bien el contrato del método requiere que la solicitud no sea nula, hay casos dónde se introducen datos inválidos (como un número de mesa o una cantidad de sillas negativa) dónde el proceso finaliza correctamente en lugar de arrojar un error.

Cabe destacar que también se realizaron los testeos de Caja Negra a las clases encargadas de gestionar las mesas, los mozos, las comandas, los operarios y los productos. Donde se detectaron y **reportaron los siguientes errores**:

Clase	**Método**	**Error**	**Corregido**
GestionDeProductos	descontarStock( )	NullPointerException, el producto es nulo	no
GestionDeOperarios	bajaOperario()	No debía lanzar PermisoDenegadoException	si
GestionDeComandas	cargarPedido()	No debería lanzar StockInsuficienteException	no
GestionDeMesas	altaMesa()	Crea mesas con datos inálidos	si
GestionDeMesas	cerrarMesa( )	NullPointerException, la comanda es nula	no

Test de Caja Blanca

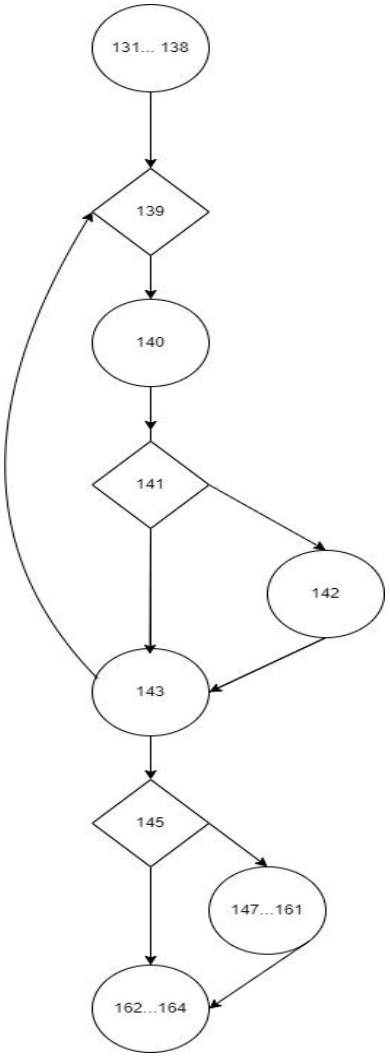
Luego de realizar las pruebas de Caja Negra con JUnit, se realizaron las pruebas de Caja Blanca, para generar una cobertura lo más completa posible, ya que, como se dijo anteriormente, el Test de Caja Negra no siempre ejecuta todas las líneas de código de un módulo, y en algunos casos es imposible testear un método sin conocer su código.

El objetivo de esta técnica de testeo es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa.

En este caso, el método a testear con caja blanca es modificaPromoFija(PromocionDTO promocion) , donde, como su nombre lo indica, se busca modificar la promoción recibida por parámetro. Para ello el método recorre la lista de promociones, y en caso de encontrarla, la modifica.

Para comenzar el proceso de caja blanca, es necesario analizar el método y, a partir de este, generar un grafo, cuya complejidad ciclomática está dada por los “nodos decisión” indicando la cantidad máxima de caminos necesarios para generar una cobertura total.

Grafo Ciclomático:




```
131  public void modificaPromocionFija(PromocionProductoDTO promocion) {
132
133      Set<PromocionFija> promociones = this.getPromocionesFijas();
134
135      Iterator<PromocionFija> it = promociones.iterator();
136      boolean encontrePromo = false;
137      Promocion p = null;
138
139      while(it.hasNext() && !encontrePromo) {
140          p = it.next();
141          if( p.getNombre().equals(promocion.getNombre()) )
142              encontrePromo = true;
143      }
144
145      if(encontrePromo){
146
147          promociones.remove(p);
148
149          PromocionFija promoFija = new PromocionFija(promocion.getNombre(),
150                                                      promocion.getDiasPromo(),
151                                                      promocion.getProducto(),
152                                                      promocion.isDosPorUno(),
153                                                      promocion.isDtoPorCant(),
154                                                      promocion.getDtoPorCantMin(),
155                                                      promocion.getDtoPorCantPrecioU()
156            );
157
158          promociones.add(promoFija);
159          this.empresa.setPromocionesFijas(promociones);
160          persistirPromocionesFijas();
161
162      }
163
164  }
```

Complejidad Ciclomática

= cantidad de regiones cerradas + 1 = 4

Obtuvimos los siguientes caminos utilizando el método simplificado:

- **C1**: 131-139-143-145-164
- **C2**: 131-139-140-141-143-145-164
- **C3**: 131-139-140-141-142-143-145-161-164

Escenarios:

Escenario	Descripción	Parámetro promoción	Lista de promos
1	La promoción se encuentra en la lista	promo1	[promo1, promo2, promo3, promo4]
2	La promoción no se encuentra en la lista	promo5	[promo1, promo2, promo3, promo4]
3	La lista se encuentra vacía	promo1	[ ]

Casos de prueba:

Camino	Escenario	Caso	Promo modificada
C1	**3**	encontrePromo==false promociones.size()==0 p==null	**No**
C2	**2**	encontrePromo=false promociones.size==4 p=promo4	**No**
C3	**1**	encontrePromo=true promociones.size()==4 p=promo1	**Si**

Como se puede observar en el Test de cobertura, basado en el grafo ciclomático y los caminos encontrados, el método de caja blanca fue efectivo, cubriendo el 100% de los casos del método testeado. Se comprobó el correcto funcionamiento del mismo.

Test de Persistencia de datos

El testeo de persistencia de datos es de suma importancia, ya que gracias a él se podrán encontrar y corregir errores en la lectura o en la escritura de archivos, permitiendo al sistema ser capaz de exportar e importar información desde un archivo.

Además de los errores en la escritura y en la lectura de archivos, hay que testear el correcto tratamiento de las distintas excepciones que se pueden lanzar en la persistencia, como por ejemplo FileNotFoundException, la cual hace referencia a que no se pudo abrir el archivo ya

que no se encontró un archivo con el nombre dado, o InvalidFormatException la cual indica que el formato del archivo que se quiso abrir es inválido.

En nuestro caso analizamos únicamente la lectura y la escritura de los datos, debido a que los programadores del código trataron a las excepciones anteriormente mencionadas en la implementación de los métodos de persistencia, en las clases GestionDeMozos y Empresa .

- testCrearArchivo( )

Es el primer método para testear la persistencia, se encarga de comprobar el correcto funcionamiento del método persistirMozos() de la clase GestionDeMozos. El test pasó correctamente, lo que indica que la escritura en archivos es correcta.

-despersistenciaTest()

En este método se analizó el correcto funcionamiento del método cargarMozos( ) en la clase empresa, el cual se encarga de despersistir el archivo que contiene la información de los mozos.

Como se puede notar el test también funcionó correctamente, concluyendo en que la lectura de los archivos es correcta.

De igual manera es posible testear el resto de las lecturas y escrituras de las demás entidades.

Test de GUI para las interfaces gráficas

Debido a que la cantidad de secuencias de comandos que podría utilizar un cliente de la interfaz gráfica, es gigantesca, nos veremos obligados a realizar un corte en las pruebas, probando los caminos que nos parezcan más posibles, y más propensos a arrojar errores.

Las pruebas de unidad para GUIs, nos exigen que al momento de construir las GUI, se las diseñe y codifique pensando en el testeo, de modo tal que no sea necesario modificar el código para probarlo. Por esto, a la hora de diseñar las vistas, se aplicó el patrón MVC, permitiendo testear las reglas de negocio por un lado, y la GUI por otro.
También se buscó la utilización de componentes standards (los JButton, JTextfield, etc) los cuales se pueden considerar como testeados.

En este caso decidimos testear el GUI del Login, utilizando la clase Robot.

Una de las principales clases a crear fue la clase TestUtils la cual nos facilitó implementar los movimientos del robot, y puede ser utilizada en las distintas interfaces gráficas a testear. Esta clase de tipo herramienta encapsula acciones comunes que contienen mensajes a la clase Robot.

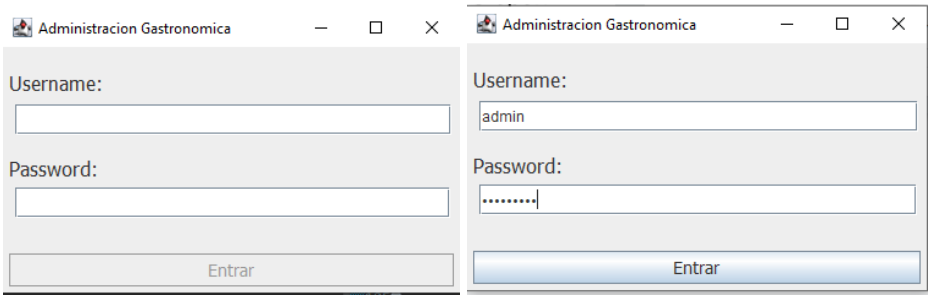
Algunos de los métodos desarrollados en esta clase son:

- getCentro(Component component, Robot robot) el cual nos ayuda a ubicar el centro de una componente. Se utiliza para clickear un boton.
- getComponentForName(String name) el cual nos permite hallar una componente (objeto) con su atributo nombre
- tipeaTexto(String texto, Robot robot) el cual escribe el texto pasado por parametro
- borraJTextField borra todo el texto del JtextField pasado como parámetro

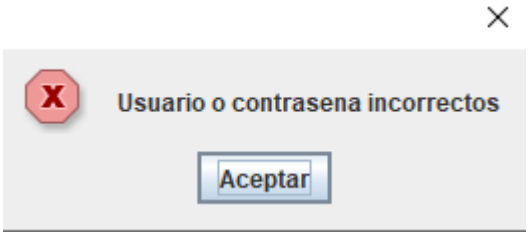
Lo que debimos considerar para este testeo fue:

- Que el botón “Entrar” debe encontrarse deshabilitado, siempre que ambos textField no contengan texto simultáneamente.

  Para testear este caso consideramos 4 casos y desarrollamos los respectivos métodos:
  -Que ambos textField estén en blanco.
  -Que el textFieldContrasena no esté en blanco y el textFieldUsuario esté blanco.
  -Que el textFieldContrasena esté en blanco y el textFieldUsuario no esté blanco.
  -Que ninguno esté en blanco (botón habilitado)



- El segundo aspecto que consideramos fue el Login:
  - En el caso de realizar un correcto inicio de sesión, es decir que tanto el nombre y la contraseña se correspondan con el de un usuario existente.
  - Cuando la contraseña o el nombre de usuario son incorrectos, debe aparecer un cartel advirtiendo esto:



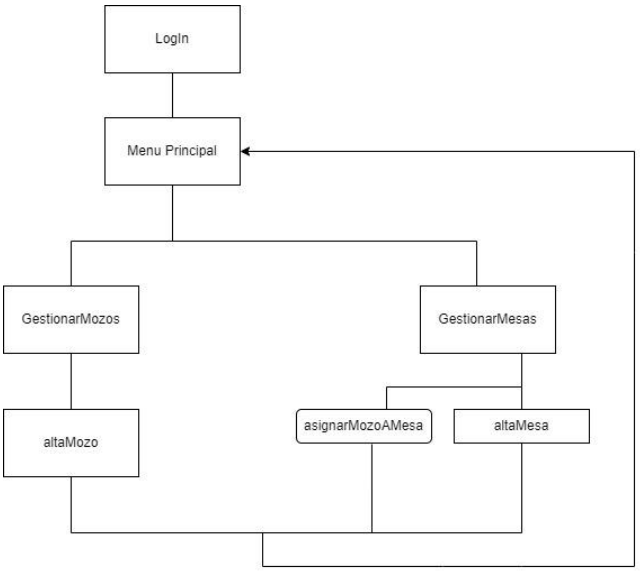
Test de Integración

El propósito principal de las pruebas de integración es probar la interacción entre los distintos componentes de un sistema, suponiendo que cada uno de ellos fue testeado de manera individual. Se buscan encontrar errores de programación entre componentes o errores de interoperabilidad.

Se realizaron dos tests de integración :

- **En el primer test de integración** que se realizó, se probó la interacción entre las clases de la capa de negocio de GestionDeMesas y GestionDeMozos, las cuales fueron testeadas individualmente con los test de Caja Negra.

  Esta es la rama del diagrama de casos de uso correspondiente a este test:



Como se puede observar en el diagrama , a la hora de realizar este test de integración, se ven involucradas diversas clases. Para realizar la asignación de un Mozo a la Mesa, debimos primero instanciar tanto un Operario, como un Mozo y una Mesa.

Los métodos más relevantes de la capa de negocios que se utilizaron son:

*De la clase GestionDeMesas:
-altaMesa
-getMesas
-asignarMozoaMesa ,
*De la clase GestionDeMozos:
- altaMozo( )
- modEstadoMozo( )
- getMozo( )

A continuación se puede ver la clase de Test desarrollada con su cobertura:

```java
import dto.MesaDTO;

public class TestIntegracionMozosMesa {
    private GestionDeMesas gestionDeMesas;
    private GestionDeMozos gestionDeMozos;
    private Empresa empresa;
    private Mozo mozo;
    private Mesa mesa;

    @Before
    public void setUp() {
        this.empresa = Empresa.getEmpresa();
        this.gestionDeMesas = GestionDeMesas.get();
        this.gestionDeMozos = GestionDeMozos.get();
    }

    @Test
    public void test() {
        empresa.setUsuarioLogueado(new Operario("admin","admin", "admin1234", true));
        try {
            gestionDeMesas.altaMesa(new MesaDTO(1004, 4));
        } catch (MesaExistenteException mesaConMozo) {
        }
        mesa = gestionDeMesas.getMesas().stream().filter(m -> m.getNroMesa() == 1004).findFirst().get();


        try {
            gestionDeMozos.altaMozo(new MozoDTO("[TEST] Mozo", "2022-01-01", 10));
        } catch (MozoExistenteException e) {
        }
        mozo = gestionDeMozos.getMozos().stream().filter(m -> m.getNombreCompleto().equals("[TEST] Mozo")).findFirst().get();


        gestionDeMozos.modEstadoMozo(mozo, EstadoMozo.ACTIVO);

        gestionDeMesas.asignarMozoMesa(new MozoDTO(
                mozo.getNombreCompleto(),
                mozo.getFechaNacimiento(),
                mozo.getCantidadHijos()
        ), new MesaDTO(1000, 4));


        Assert.assertEquals(mesa.getMozoAsignado(), mozo);
    }

    @After
    public void tearDown() {
        gestionDeMesas.bajaMesa(1004);
        gestionDeMozos.bajaMozo(mozo);
        empresa.setUsuarioLogueado(null);
    }
}

```

En este test encontramos un grave error, ya que al realizar el assertEquals entre el mozo asignado a la Mesa y el mozo creado, estos son diferentes.

Luego de revisar brevemente el código, pudimos detectar que el error se origina en la asignación del Mozo a la mesa, ya que al intentar asignarlo, en lugar de setear el Mozo creado, se instancia un nuevo Mozo con los datos del que debía ser asignado.

El segundo test de integración realizado comprendió la clase de GestionDeMesas y GestionDeComandas, clases que fueron testeadas individualmente mediante la metodología de Caja Negra.

En este caso se creó una mesa, se abrió una comanda mediante el método abrirComanda() de la clase GestionDeComandas y se agregaron pedidos a la misma mediante el método agregarPedido( ) .

Finalmente se calculó el total de la mesa, se aplicaron las promociones y se cerró la comanda mediante el método totalMesa( ) de la clase GestionDeMesas.

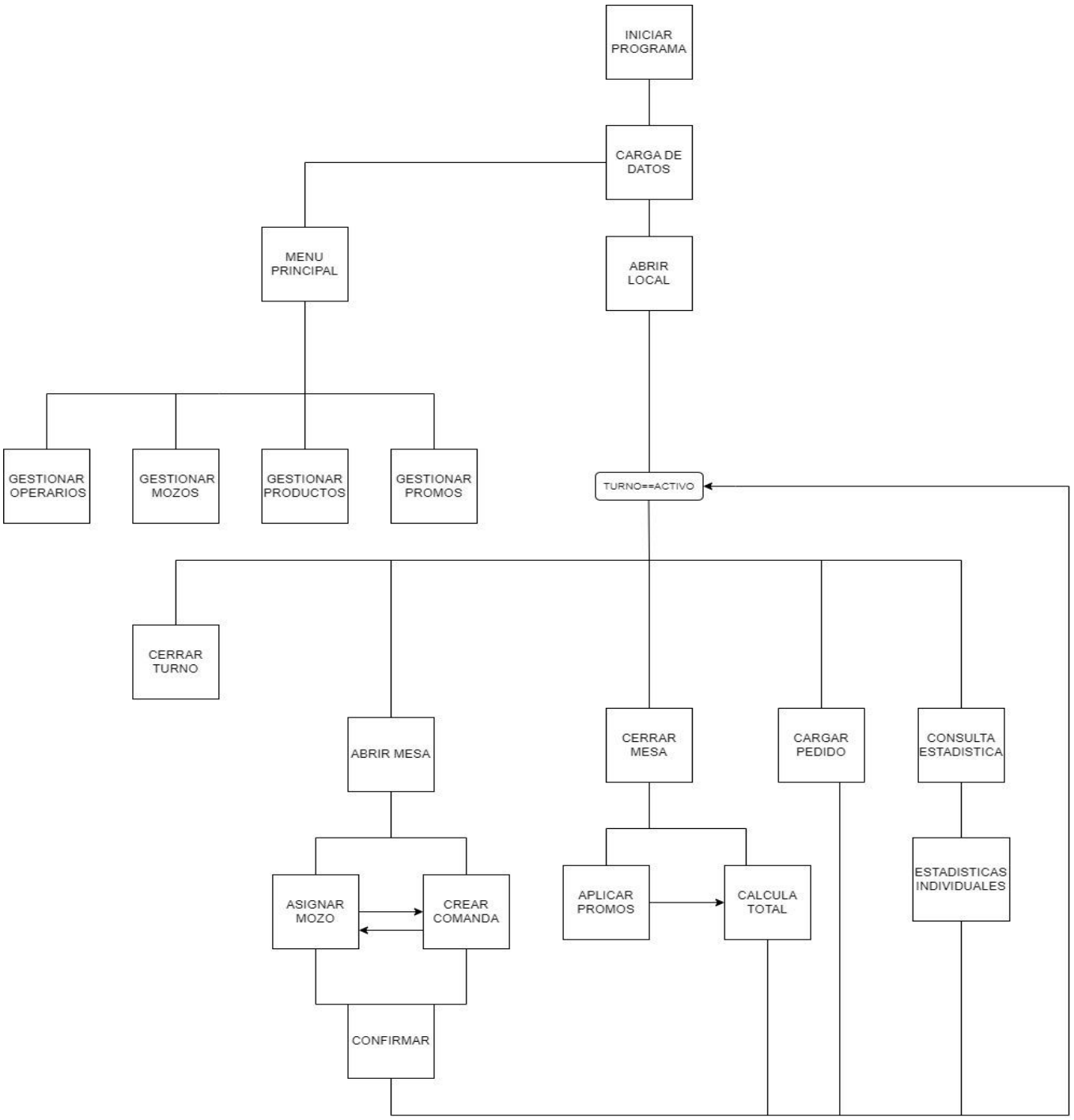
A continuación se puede ver la clase de Test desarrollada con su cobertura:

```java
import enums.Dias;

public class TestIntegracion {
    private GestionDeMesas gestionDeMesas;
    private GestionDeComandas gestionDeComandas;
    private Empresa empresa;
    private Producto prod1, prod2;
    private PromocionFija promo1;
    private PromocionTemporal promo2;

    @Before
    public void setUp() {
        this.empresa = Empresa.getEmpresa();
        this.gestionDeMesas = GestionDeMesas.get();
        this.gestionDeComandas = GestionDeComandas.get();
        this.prod1 = new Producto("Coca cola", 100F, 50F, 10);
        this.prod2 = new Producto("Pepsi", 100F, 50F, 10);
        this.empresa.getProductos().add(prod1);
        this.empresa.getProductos().add(prod2);
        this.promo1 = new PromocionFija("Promo 1 Coca", List.of(Dias.LUNES, Dias.MARTES), prod1, true, false, 0, 0.0);
        this.promo2 = new PromocionTemporal("Promo 2 Coca", List.of(Dias.LUNES, Dias.MARTES), "Efectivo", 25, true);
        this.empresa.getPromocionesFijas().add(promo1);
        this.empresa.getPromocionesTemporales().add(promo2);
    }

    @Test
    public void testAgregarPedidos() throws StockInsuficienteException {
        Mesa mesa = new Mesa(1000, 4);
        Mozo mozo = new Mozo("Mozo", "2022-01-01", 10);
        mesa.setMozoAsignado(mozo);
        gestionDeComandas.abrirComanda(mesa);

        gestionDeComandas.cargarPedido(mesa, new Pedido(prod1, 2));
        gestionDeComandas.cargarPedido(mesa, new Pedido(prod2, 1));

        gestionDeMesas.totalMesa(mesa, "Efectivo");
    }

    @After
    public void tearDown() {
        this.empresa.getProductos().remove(prod1);
        this.empresa.getProductos().remove(prod2);
        this.empresa.getPromocionesFijas().remove(promo1);
        this.empresa.getPromocionesTemporales().remove(promo2);
    }


}
```

En este caso encontramos un error, debido a que algunas promociones no contaban con los días asignados, lo cual generó un NullPointerException debido a que la lista de días de las promociones era nula.

Grafo de casos de uso

Este grafo describe la navegación por la cadena de invocaciones de todo el programa.
Su realización permite al testeador, mediante la observación de la desagregación de invocaciones, descubrir los diferentes datos para elaborar los casos de prueba que se deben poner en el testeo de los métodos que están en el módulo superior. para así, asegurarse que se vayan estimulando la rama que se busca, evitando las redundancias.



Uso de Mock

En este caso no consideramos necesaria la implementación de objetos de tipo Mock, esto debido a que el testeo se realizó con todas las clases a testear ya creadas, las cuales a su vez implementaban métodos de sencillos de testear.

Conclusiones

El desarrollo de los distintos métodos de Testing vistos en la materia, nos permitió obtener información muy valiosa y detectar errores, que de no ser por estas herramientas, no se hubiesen detectado.

En conclusión, la etapa de testing en el desarrollo de Software es un pilar fundamental para asegurar calidad en los productos. Se pudo comprobar también la necesidad de contar con una buena documentación a la hora de la participación en un proceso de desarrollo de Software, ya que , de otra manera, las tareas de testeo pueden dar lugar a resultados difusos, y generar confusiones a la hora de codificar soluciones.

También, es evidente que la etapa de pruebas no debe ser la última del proceso, ya que, si esto sucede, las tareas de testing se vuelven tediosas y corregir estos errores puede incurrir en nuevos problemas que, de encontrarse en períodos avanzados, quizás no tengan solución. Es por esto que el testing debe acompañar en manera periódica al desarrollo del software, para así mejorar la eficacia del manejo de un proyecto.